| 1 | EDB Postgres Advanced Server (EPAS) | 23 |
| 2 | EDB Postgres Advanced Server release notes | 23 |
| 2.1 | EDB Postgres Advanced Server 16.1 release notes | 23 |
| 3 | Supported platforms | 24 |
| 4 | Fundamentals | 24 |
| 4.1 | EDB Postgres Advanced Server fundamentals | 24 |
| 4.1.1 | Benefits of EDB Postgres Advanced Server | 24 |
| 4.1.2 | Database compatibility for Oracle developers | 25 |
| 4.1.3 | Terminology | 25 |
| 4.2 | SQL fundamentals | 25 |
| 4.2.1 | About the examples | 25 |
| 4.2.2 | Conventions used | 26 |
| 4.2.3 | SQL tutorial | 26 |
| 4.2.3.1 | Sample database | 26 |
| 4.2.3.1.1 | Accessing the sample database | 27 |
| 4.2.3.1.2 | Overview of the sample database | 27 |
| 4.2.3.2 | Creating a new table | 37 |
| 4.2.3.3 | Populating a table With rows | 37 |
| 4.2.3.4 | Querying a table | 38 |
| 4.2.3.5 | Executing joins between tables | 39 |
| 4.2.3.6 | Aggregating data | 41 |
| 4.2.3.7 | Updating a table | 42 |
| 4.2.3.8 | Deleting a table | 43 |
| 4.2.3.9 | The SQL language | 44 |
| 4.2.4 | Advanced concepts | 44 |
| 4.2.4.1 | Views | 44 |
| 4.2.4.2 | Foreign keys | 45 |
| 4.2.4.3 | The ROWNUM pseudo-column | 45 |
| 4.2.4.4 | Synonyms | 46 |
| 4.2.4.5 | Hierarchical queries | 48 |
| 4.2.4.5.1 | Defining the parent/child relationship | 52 |
| 4.2.4.5.2 | Selecting the root nodes | 52 |
| 4.2.4.5.3 | Organization tree in the sample application | 52 |
| 4.2.4.5.4 | Designating the node level | 53 |
| 4.2.4.5.5 | Ordering the siblings | 54 |
| 4.2.4.5.6 | Retrieving the root node with CONNECT_BY_ROOT | 55 |
| 4.2.4.5.7 | Retrieving a path with SYS_CONNECT_BY_PATH | 59 |
| 4.2.4.6 | Multidimensional analysis | 60 |
| 4.2.4.6.1 | ROLLUP extension | 62 |
| 4.2.4.6.2 | CUBE extension | 64 |
| 4.2.4.6.3 | GROUPING SETS extension | 68 |
| 4.2.4.6.4 | GROUPING function | 74 |
| 4.2.4.6.5 | GROUPING_ID function | 76 |
| 4.2.5 | Sample database description | 78 |
| 5 | Planning | 86 |
| 5.1 | Choosing the configuration mode | 87 |
| 5.2 | Deployment options | 87 |
| 5.3 | Planning prerequisites | 87 |
5.3.1 Limitations

5.3.2 Requirements

6 Installing EDB Postgres Advanced Server

6.1 Installing EDB Postgres Advanced Server on Linux x86 (amd64)

6.1.1 Installing EDB Postgres Advanced Server on RHEL 9 or OL 9 x86_64

6.1.2 Installing EDB Postgres Advanced Server on RHEL 8 or OL 8 x86_64

6.1.3 Installing EDB Postgres Advanced Server on AlmaLinux 9 or Rocky Linux 9 x86_64

6.1.4 Installing EDB Postgres Advanced Server on AlmaLinux 8 or Rocky Linux 8 x86_64

6.1.5 Installing EDB Postgres Advanced Server on RHEL 7 or OL 7 x86_64

6.1.6 Installing EDB Postgres Advanced Server on CentOS 7 x86_64

6.1.7 Installing EDB Postgres Advanced Server on SLES 15 x86_64

6.1.8 Installing EDB Postgres Advanced Server on SLES 12 x86_64

6.1.9 Installing EDB Postgres Advanced Server on Ubuntu 22.04 x86_64

6.1.10 Installing EDB Postgres Advanced Server on Ubuntu 20.04 x86_64

6.1.11 Installing EDB Postgres Advanced Server on Ubuntu 18.04 x86_64

6.1.12 Installing EDB Postgres Advanced Server on Debian 11 x86_64

6.1.13 Installing EDB Postgres Advanced Server on Debian 10 x86_64

6.2 Installing EDB Postgres Advanced Server on Linux IBM Power (ppc64le)

6.2.1 Installing EDB Postgres Advanced Server on RHEL 9 ppc64le

6.2.2 Installing EDB Postgres Advanced Server on RHEL 8 ppc64le

6.2.3 Installing EDB Postgres Advanced Server on SLES 15 ppc64le

6.2.4 Installing EDB Postgres Advanced Server on RHEL 7 ppc64le

6.2.5 Installing EDB Postgres Advanced Server on SLES 12 ppc64le

6.3 EDB Postgres Advanced Server installation for Windows

6.3.1 Installing EDB Postgres Advanced Server with the interactive installer

6.3.1.1 Performing a graphical installation on Windows

6.3.1.2 Invoking the graphical installer from the command line

6.3.1.2.1 Performing an unattended installation

6.3.1.2.2 Performing an installation with limited privileges

6.3.1.2.3 Installation command line options

6.3.1.3 Setting cluster preferences during a graphical installation

6.3.1.4 Using StackBuilder Plus

6.3.2 Managing an EDB Postgres Advanced Server installation

6.3.2.1 Starting and stopping EDB Postgres Advanced Server

6.3.2.2 Using the Windows services applet

6.3.2.3 Using pg_ctl to control EDB Postgres Advanced Server

6.3.2.4 Controlling server startup behavior on Windows

6.3.2.5 Configuring EDB Postgres Advanced Server

6.4 Linux installation details

6.4.1 Default component locations

6.4.2 Available native packages

6.4.3 Updating an RPM installation

6.4.4 Installing on Linux using a local repository

6.4.5 Managing an EDB Postgres Advanced Server installation

6.4.5.1 Configuring a package installation

6.4.5.2 Connecting to EDB Postgres Advanced Server with edb-psql

6.4.5.3 Modifying the data directory location

6.4.5.4 Specifying cluster options with INITDBOPTS
8.1.2.4  Query tuning
8.1.2.5  Query tuning/planner method configuration
8.1.2.6  Reporting and logging/what to log
8.1.2.7  Auditing settings
8.1.2.7.1  edb_audit
8.1.2.7.2  edb_audit_directory
8.1.2.7.3  edb_audit_filename
8.1.2.7.4  edb_audit_rotation_day
8.1.2.7.5  edb_audit_rotation_size
8.1.2.7.6  edb_audit_rotation_seconds
8.1.2.7.7  edb_audit_connect
8.1.2.7.8  edb_audit_disconnect
8.1.2.7.9  edb_audit_statement
8.1.2.7.10  edb_audit_tag
8.1.2.7.11  edb_audit_destination
8.1.2.7.12  edb_log_every_bulk_value
8.1.2.8  Client connection defaults/locale and formatting
8.1.2.9  Client connection defaults/statement behavior
8.1.2.10  Client connection defaults/other defaults
8.1.2.11  Compatibility options
8.1.2.12  Customized options
8.1.2.13  Ungrouped configuration parameters
8.1.2.14  Audit archiving parameters
8.1.2.15  Description of parameter attributes
8.2  Throttling CPU and I/O at the process level
8.2.1  EDB Resource Manager key concepts
8.2.2  Working with resource groups
8.2.3  CPU usage throttling
8.2.4  Dirty buffer throttling
8.3  Loading bulk data
8.3.1  EDB*Loader key concepts and compatibility
8.3.2  Overview of data loading methods
8.3.3  EDB*Loader error handling
8.3.4  Building the EDB*Loader control file
8.3.4.1  EDB*Loader control file examples
8.3.5  Invoking EDB*Loader
8.3.5.1  Running EDB*Loader
8.3.5.2  Updating a table with a conventional path load
8.3.5.3  Running a direct path load
8.3.5.4  Running a parallel direct path load
8.3.5.5  Performing remote loading
8.4  Copying a database
8.4.1  EDB Clone Schema key concepts and limitations
8.4.2  Setting up EDB Clone Schema
8.4.3  Copying database objects from a local source to a target
8.4.4  Copying database objects from a remote source
8.4.5  Checking the status of the cloning process
8.4.6  Performing cleanup tasks

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 EDB Postgres Advanced Server security features</td>
<td>263</td>
</tr>
<tr>
<td>9.1 Transparent Data Encryption overview</td>
<td>263</td>
</tr>
<tr>
<td>9.2 Using EDB audit logging</td>
<td>263</td>
</tr>
<tr>
<td>9.2.1 Selecting SQL statements to audit</td>
<td>263</td>
</tr>
<tr>
<td>9.2.2 Enabling audit logging</td>
<td>273</td>
</tr>
<tr>
<td>9.2.3 Using error codes to filter audit logs</td>
<td>276</td>
</tr>
<tr>
<td>9.2.4 Using command tags to filter audit logs</td>
<td>277</td>
</tr>
<tr>
<td>9.2.5 Redacting passwords in audit logs</td>
<td>278</td>
</tr>
<tr>
<td>9.2.6 Archiving audit logs</td>
<td>279</td>
</tr>
<tr>
<td>9.2.7 Auditing objects</td>
<td>281</td>
</tr>
<tr>
<td>9.3 Protecting against SQL injection attacks</td>
<td>284</td>
</tr>
<tr>
<td>9.3.1 SQL/Protect overview</td>
<td>284</td>
</tr>
<tr>
<td>9.3.2 Configuring SQL/Protect</td>
<td>286</td>
</tr>
<tr>
<td>9.3.3 Common maintenance operations</td>
<td>293</td>
</tr>
<tr>
<td>9.3.4 Backing up and restoring a SQL/Protect database</td>
<td>296</td>
</tr>
<tr>
<td>9.4 Generating SSL certificates</td>
<td>300</td>
</tr>
<tr>
<td>9.5 Protecting proprietary source code</td>
<td>302</td>
</tr>
<tr>
<td>9.5.1 EDB*Wrap key concepts</td>
<td>302</td>
</tr>
<tr>
<td>9.5.2 Obfuscating source code</td>
<td>302</td>
</tr>
<tr>
<td>9.6 Managing user profiles</td>
<td>305</td>
</tr>
<tr>
<td>9.6.1 Profile management key concepts</td>
<td>306</td>
</tr>
<tr>
<td>9.6.2 Creating a new profile</td>
<td>306</td>
</tr>
<tr>
<td>9.6.2.1 Creating a password function</td>
<td>308</td>
</tr>
<tr>
<td>9.6.3 Altering a profile</td>
<td>309</td>
</tr>
<tr>
<td>9.6.4 Dropping a profile</td>
<td>310</td>
</tr>
<tr>
<td>9.6.5 Associating a profile with an existing role</td>
<td>311</td>
</tr>
<tr>
<td>9.6.6 Unlocking a locked account</td>
<td>312</td>
</tr>
<tr>
<td>9.6.7 Creating a new role associated with a profile</td>
<td>312</td>
</tr>
<tr>
<td>9.6.8 Backing up profile management functions</td>
<td>314</td>
</tr>
<tr>
<td>9.7 Redacting data</td>
<td>314</td>
</tr>
<tr>
<td>9.7.1 Data redaction key concepts</td>
<td>314</td>
</tr>
<tr>
<td>9.7.2 Creating a data redaction policy</td>
<td>314</td>
</tr>
<tr>
<td>9.7.3 Modifying a data redaction policy</td>
<td>318</td>
</tr>
<tr>
<td>9.7.4 Removing a data redaction policy</td>
<td>320</td>
</tr>
<tr>
<td>9.7.5 Data redaction system catalogs</td>
<td>321</td>
</tr>
<tr>
<td>9.8 Controlling data access (Virtual Private Database)</td>
<td>321</td>
</tr>
<tr>
<td>10 Managing performance</td>
<td>322</td>
</tr>
<tr>
<td>10.1 Using the dynamic runtime instrumentation tools architecture (DRITA)</td>
<td>322</td>
</tr>
<tr>
<td>10.1.1 Taking a snapshot</td>
<td>322</td>
</tr>
<tr>
<td>10.1.2 Using DRITA functions</td>
<td>323</td>
</tr>
<tr>
<td>10.1.3 Performance tuning recommendations</td>
<td>329</td>
</tr>
<tr>
<td>10.1.4 Simulating Statspack AWR reports</td>
<td>331</td>
</tr>
<tr>
<td>10.2 Using Index Advisor</td>
<td>345</td>
</tr>
<tr>
<td>10.2.1 Index Advisor overview</td>
<td>345</td>
</tr>
<tr>
<td>10.2.2 Index Advisor limitations</td>
<td>346</td>
</tr>
<tr>
<td>10.2.3 Configuring the Index Advisor</td>
<td>346</td>
</tr>
<tr>
<td>10.2.4 Using Index Advisor</td>
<td>347</td>
</tr>
<tr>
<td>10.2.5 Reviewing Index Advisor recommendations</td>
<td>350</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>10.3</td>
<td>Using dynamic resource tuning</td>
</tr>
<tr>
<td>10.4</td>
<td>Evaluating wait states</td>
</tr>
<tr>
<td>10.5</td>
<td>Using SQL Profiler</td>
</tr>
<tr>
<td>11</td>
<td>Application programming</td>
</tr>
<tr>
<td>11.1</td>
<td>Working with packages</td>
</tr>
<tr>
<td>11.1.1</td>
<td>Package components</td>
</tr>
<tr>
<td>11.1.2</td>
<td>Viewing packages and package body definition</td>
</tr>
<tr>
<td>11.1.3</td>
<td>Creating packages</td>
</tr>
<tr>
<td>11.1.4</td>
<td>Referencing a package</td>
</tr>
<tr>
<td>11.1.5</td>
<td>Using packages with user-defined types</td>
</tr>
<tr>
<td>11.1.6</td>
<td>Dropping a package</td>
</tr>
<tr>
<td>11.2</td>
<td>Debugging programs</td>
</tr>
<tr>
<td>11.2.1</td>
<td>Configuring the debugger</td>
</tr>
<tr>
<td>11.2.2</td>
<td>Starting the debugger</td>
</tr>
<tr>
<td>11.2.3</td>
<td>Debugger interface overview</td>
</tr>
<tr>
<td>11.2.4</td>
<td>Running the debugger</td>
</tr>
<tr>
<td>11.3</td>
<td>Using enhanced SQL and other miscellaneous features</td>
</tr>
<tr>
<td>11.3.1</td>
<td>Using the COMMENT command</td>
</tr>
<tr>
<td>11.3.2</td>
<td>Configuring logical decoding on standby</td>
</tr>
<tr>
<td>11.3.3</td>
<td>Obtaining version information</td>
</tr>
<tr>
<td>11.4</td>
<td>Including embedded SQL commands</td>
</tr>
<tr>
<td>11.4.1</td>
<td>ECPGPlus overview</td>
</tr>
<tr>
<td>11.4.2</td>
<td>Installing and configuring ECPGPlus</td>
</tr>
<tr>
<td>11.4.3</td>
<td>Using embedded SQL</td>
</tr>
<tr>
<td>11.4.4</td>
<td>Using descriptors</td>
</tr>
<tr>
<td>11.4.5</td>
<td>Building and executing dynamic SQL statements</td>
</tr>
<tr>
<td>11.4.6</td>
<td>Error handling</td>
</tr>
<tr>
<td>11.5</td>
<td>Using the stored procedural language</td>
</tr>
<tr>
<td>11.5.1</td>
<td>Types of SPL programs</td>
</tr>
<tr>
<td>11.5.1.1</td>
<td>SPL block structure overview</td>
</tr>
<tr>
<td>11.5.1.2</td>
<td>Anonymous blocks</td>
</tr>
<tr>
<td>11.5.1.3</td>
<td>Procedures overview</td>
</tr>
<tr>
<td>11.5.1.3.1</td>
<td>Creating a procedure</td>
</tr>
<tr>
<td>11.5.1.3.2</td>
<td>Calling a procedure</td>
</tr>
<tr>
<td>11.5.1.3.3</td>
<td>Deleting a procedure</td>
</tr>
<tr>
<td>11.5.1.4</td>
<td>Functions overview</td>
</tr>
<tr>
<td>11.5.1.4.1</td>
<td>Creating a function</td>
</tr>
<tr>
<td>11.5.1.4.2</td>
<td>Calling a function</td>
</tr>
<tr>
<td>11.5.1.4.3</td>
<td>Deleting a function</td>
</tr>
<tr>
<td>11.5.1.5</td>
<td>Procedure and function parameters</td>
</tr>
<tr>
<td>11.5.1.5.1</td>
<td>Declaring parameters</td>
</tr>
<tr>
<td>11.5.1.5.2</td>
<td>Positional versus named parameter notation</td>
</tr>
<tr>
<td>11.5.1.5.3</td>
<td>Parameter modes</td>
</tr>
<tr>
<td>11.5.1.5.4</td>
<td>Using default values in parameters</td>
</tr>
<tr>
<td>11.5.1.6</td>
<td>Subprograms: subprocedures and subfunctions</td>
</tr>
<tr>
<td>11.5.1.6.1</td>
<td>Creating a subprocedure</td>
</tr>
<tr>
<td>11.5.1.6.2</td>
<td>Creating a subfunction</td>
</tr>
<tr>
<td>11.5.1.6.3</td>
<td>Declaring block relationships</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>11.5.1.6.4</td>
<td>Invoking subprograms</td>
</tr>
<tr>
<td>11.5.1.6.5</td>
<td>Using forward declarations</td>
</tr>
<tr>
<td>11.5.1.6.6</td>
<td>Overloading subprograms</td>
</tr>
<tr>
<td>11.5.1.6.7</td>
<td>Accessing subprogram variables</td>
</tr>
<tr>
<td>11.5.1.7</td>
<td>Compilation errors in procedures and functions</td>
</tr>
<tr>
<td>11.5.1.8</td>
<td>Program security</td>
</tr>
<tr>
<td>11.5.1.8.1</td>
<td>EXECUTE privilege</td>
</tr>
<tr>
<td>11.5.1.8.2</td>
<td>Database object name resolution</td>
</tr>
<tr>
<td>11.5.1.8.3</td>
<td>Database object privileges</td>
</tr>
<tr>
<td>11.5.1.8.4</td>
<td>About definer and invoker rights</td>
</tr>
<tr>
<td>11.5.1.8.5</td>
<td>Security example</td>
</tr>
<tr>
<td>11.5.2</td>
<td>Using variable declarations</td>
</tr>
<tr>
<td>11.5.2.1</td>
<td>Declaring a variable</td>
</tr>
<tr>
<td>11.5.2.2</td>
<td>Using %TYPE in variable declarations</td>
</tr>
<tr>
<td>11.5.2.3</td>
<td>Using %ROWTYPE in record declarations</td>
</tr>
<tr>
<td>11.5.2.4</td>
<td>User-defined record types and record variables</td>
</tr>
<tr>
<td>11.5.3</td>
<td>Working with transactions</td>
</tr>
<tr>
<td>11.5.3.1</td>
<td>About transactions</td>
</tr>
<tr>
<td>11.5.3.2</td>
<td>COMMIT</td>
</tr>
<tr>
<td>11.5.3.3</td>
<td>ROLLBACK</td>
</tr>
<tr>
<td>11.5.3.4</td>
<td>PRAGMA AUTONOMOUS_TRANSACTION</td>
</tr>
<tr>
<td>11.5.4</td>
<td>Using dynamic SQL</td>
</tr>
<tr>
<td>11.5.5</td>
<td>Working with static cursors</td>
</tr>
<tr>
<td>11.5.5.1</td>
<td>Declaring a cursor</td>
</tr>
<tr>
<td>11.5.5.2</td>
<td>Opening a cursor</td>
</tr>
<tr>
<td>11.5.5.3</td>
<td>Fetching rows from a cursor</td>
</tr>
<tr>
<td>11.5.5.4</td>
<td>Closing a cursor</td>
</tr>
<tr>
<td>11.5.5.5</td>
<td>Using %ROWTYPE with cursors</td>
</tr>
<tr>
<td>11.5.5.6</td>
<td>Cursor attributes</td>
</tr>
<tr>
<td>11.5.5.6.1</td>
<td>%ISOPEN</td>
</tr>
<tr>
<td>11.5.5.6.2</td>
<td>%FOUND</td>
</tr>
<tr>
<td>11.5.5.6.3</td>
<td>%NOTFOUND</td>
</tr>
<tr>
<td>11.5.5.6.4</td>
<td>%ROWCOUNT</td>
</tr>
<tr>
<td>11.5.5.6.5</td>
<td>Summary of cursor states and attributes</td>
</tr>
<tr>
<td>11.5.5.7</td>
<td>Using a cursor FOR loop</td>
</tr>
<tr>
<td>11.5.5.8</td>
<td>Declaring parameterized cursors</td>
</tr>
<tr>
<td>11.5.6</td>
<td>Working with REF CURSOR and cursor variables</td>
</tr>
<tr>
<td>11.5.6.1</td>
<td>REF CURSOR overview</td>
</tr>
<tr>
<td>11.5.6.2</td>
<td>Declaring a cursor variable</td>
</tr>
<tr>
<td>11.5.6.2.1</td>
<td>Declaring a SYS_REFCURSOR cursor variable</td>
</tr>
<tr>
<td>11.5.6.2.2</td>
<td>Declaring a user-defined REF CURSOR type variable</td>
</tr>
<tr>
<td>11.5.6.3</td>
<td>Opening a cursor variable</td>
</tr>
<tr>
<td>11.5.6.4</td>
<td>Fetching rows From a cursor variable</td>
</tr>
<tr>
<td>11.5.6.5</td>
<td>Closing a cursor variable</td>
</tr>
<tr>
<td>11.5.6.6</td>
<td>Usage restrictions</td>
</tr>
<tr>
<td>11.5.6.7</td>
<td>Cursor variable examples</td>
</tr>
<tr>
<td>11.5.6.7.1</td>
<td>Returning a REF CURSOR from a function</td>
</tr>
<tr>
<td>11.5.6.7.2</td>
<td>Modularizing cursor operations</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>11.5.6.8</td>
<td>Using dynamic queries with REF CURSOR</td>
</tr>
<tr>
<td>11.5.7</td>
<td>Working with collection types</td>
</tr>
<tr>
<td>11.5.7.1</td>
<td>About collection types</td>
</tr>
<tr>
<td>11.5.7.2</td>
<td>Using associative arrays</td>
</tr>
<tr>
<td>11.5.7.3</td>
<td>Working with nested tables</td>
</tr>
<tr>
<td>11.5.7.4</td>
<td>Using varrays</td>
</tr>
<tr>
<td>11.5.8</td>
<td>Working with collections</td>
</tr>
<tr>
<td>11.5.8.1</td>
<td>Using the TABLE function</td>
</tr>
<tr>
<td>11.5.8.2</td>
<td>Using the MULTISET operators</td>
</tr>
<tr>
<td>11.5.8.3</td>
<td>Using the FORALL statement</td>
</tr>
<tr>
<td>11.5.8.4</td>
<td>Using the BULK COLLECT clause</td>
</tr>
<tr>
<td>11.5.8.4.1</td>
<td>SELECT BULK COLLECT</td>
</tr>
<tr>
<td>11.5.8.4.2</td>
<td>FETCH BULK COLLECT</td>
</tr>
<tr>
<td>11.5.8.4.3</td>
<td>EXECUTE IMMEDIATE BULK COLLECT</td>
</tr>
<tr>
<td>11.5.8.4.4</td>
<td>RETURNING BULK COLLECT</td>
</tr>
<tr>
<td>11.5.8.5</td>
<td>Errors and messages</td>
</tr>
<tr>
<td>11.5.9</td>
<td>Working with triggers</td>
</tr>
<tr>
<td>11.5.9.1</td>
<td>Trigger overview</td>
</tr>
<tr>
<td>11.5.9.2</td>
<td>Types of triggers</td>
</tr>
<tr>
<td>11.5.9.3</td>
<td>Creating triggers</td>
</tr>
<tr>
<td>11.5.9.4</td>
<td>Trigger variables</td>
</tr>
<tr>
<td>11.5.9.5</td>
<td>Transactions and exceptions</td>
</tr>
<tr>
<td>11.5.9.6</td>
<td>Compound triggers</td>
</tr>
<tr>
<td>11.5.9.7</td>
<td>Trigger examples</td>
</tr>
<tr>
<td>11.5.9.7.1</td>
<td>Before statement-level trigger</td>
</tr>
<tr>
<td>11.5.9.7.2</td>
<td>After statement-level trigger</td>
</tr>
<tr>
<td>11.5.9.7.3</td>
<td>Before row-level trigger</td>
</tr>
<tr>
<td>11.5.9.7.4</td>
<td>After row-level trigger</td>
</tr>
<tr>
<td>11.5.9.7.5</td>
<td>INSTEAD OF trigger</td>
</tr>
<tr>
<td>11.5.9.7.6</td>
<td>Compound triggers</td>
</tr>
<tr>
<td>11.5.10</td>
<td>Working with packages</td>
</tr>
<tr>
<td>11.5.11</td>
<td>Using object types and objects</td>
</tr>
<tr>
<td>11.5.11.1</td>
<td>Basic object concepts</td>
</tr>
<tr>
<td>11.5.11.1.1</td>
<td>Attributes</td>
</tr>
<tr>
<td>11.5.11.1.2</td>
<td>Methods</td>
</tr>
<tr>
<td>11.5.11.1.3</td>
<td>Overloading methods</td>
</tr>
<tr>
<td>11.5.11.2</td>
<td>Object type components</td>
</tr>
<tr>
<td>11.5.11.2.1</td>
<td>Object type specification syntax</td>
</tr>
<tr>
<td>11.5.11.2.2</td>
<td>Object type body syntax</td>
</tr>
<tr>
<td>11.5.11.3</td>
<td>Creating object types</td>
</tr>
<tr>
<td>11.5.11.3.1</td>
<td>Member methods</td>
</tr>
<tr>
<td>11.5.11.3.2</td>
<td>Static methods</td>
</tr>
<tr>
<td>11.5.11.3.3</td>
<td>Constructor methods</td>
</tr>
<tr>
<td>11.5.11.4</td>
<td>Creating object instances</td>
</tr>
<tr>
<td>11.5.11.5</td>
<td>Referencing an object</td>
</tr>
<tr>
<td>11.5.11.6</td>
<td>Dropping an object type</td>
</tr>
<tr>
<td>11.6</td>
<td>Using table partitioning</td>
</tr>
<tr>
<td>11.6.1</td>
<td>Oracle table partitioning compatibility summary</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>11.6.2 Selecting a partition type</td>
<td>536</td>
</tr>
<tr>
<td>11.6.2.1 Interval range partitioning</td>
<td>537</td>
</tr>
<tr>
<td>11.6.2.2 Automatic list partitioning</td>
<td>538</td>
</tr>
<tr>
<td>11.6.3 Using partition pruning</td>
<td>538</td>
</tr>
<tr>
<td>11.6.3.1 Example: Partition pruning</td>
<td>540</td>
</tr>
<tr>
<td>11.6.4 Handling stray values in a LIST or RANGE partitioned table</td>
<td>542</td>
</tr>
<tr>
<td>11.6.4.1 Defining a DEFAULT partition</td>
<td>542</td>
</tr>
<tr>
<td>11.6.4.2 Defining a MAXVALUE partition</td>
<td>545</td>
</tr>
<tr>
<td>11.6.5 Specifying multiple partitioning keys in a RANGE partitioned table</td>
<td>546</td>
</tr>
<tr>
<td>11.6.6 Retrieving information about a partitioned table</td>
<td>546</td>
</tr>
<tr>
<td>11.7 Optimizing code</td>
<td>547</td>
</tr>
<tr>
<td>11.7.1 Using optimizer hints</td>
<td>547</td>
</tr>
<tr>
<td>11.7.1.1 About optimizer hints</td>
<td>547</td>
</tr>
<tr>
<td>11.7.1.2 Default optimization modes</td>
<td>548</td>
</tr>
<tr>
<td>11.7.1.3 Access method hints</td>
<td>549</td>
</tr>
<tr>
<td>11.7.1.4 Specifying a join order</td>
<td>554</td>
</tr>
<tr>
<td>11.7.1.5 Joining relations hints</td>
<td>554</td>
</tr>
<tr>
<td>11.7.1.6 Global hints</td>
<td>556</td>
</tr>
<tr>
<td>11.7.1.7 APPEND optimizer hint</td>
<td>558</td>
</tr>
<tr>
<td>11.7.1.8 Parallelism hints</td>
<td>558</td>
</tr>
<tr>
<td>11.7.1.9 Conflicting hints</td>
<td>561</td>
</tr>
<tr>
<td>11.7.2 Optimizing inefficient SQL code</td>
<td>562</td>
</tr>
<tr>
<td>12 Working with Oracle data</td>
<td>562</td>
</tr>
<tr>
<td>12.1 Enhanced compatibility features</td>
<td>562</td>
</tr>
<tr>
<td>12.2 Querying an Oracle server</td>
<td>566</td>
</tr>
<tr>
<td>12.2.1 Calling dblink_ora functions</td>
<td>566</td>
</tr>
<tr>
<td>12.2.2 Connecting to an Oracle database</td>
<td>567</td>
</tr>
<tr>
<td>12.3 Using Open Client Library</td>
<td>567</td>
</tr>
<tr>
<td>12.4 Including embedded SQL commands in C applications (ECPGPlus)</td>
<td>568</td>
</tr>
<tr>
<td>12.5 Loading bulk data (EDB*Loader)</td>
<td>568</td>
</tr>
<tr>
<td>12.6 Protecting proprietary source code (EDB*Wrap)</td>
<td>568</td>
</tr>
<tr>
<td>13 Tools, utilities, and components</td>
<td>568</td>
</tr>
<tr>
<td>13.1 Application programmer tools</td>
<td>568</td>
</tr>
<tr>
<td>13.1.1 Unicode Collation Algorithm</td>
<td>569</td>
</tr>
<tr>
<td>13.1.2 ECPGPlus</td>
<td>574</td>
</tr>
<tr>
<td>13.1.3 libpq C library</td>
<td>574</td>
</tr>
<tr>
<td>13.2 Connectivity tools</td>
<td>581</td>
</tr>
<tr>
<td>13.2.1 Connecting to your database</td>
<td>581</td>
</tr>
<tr>
<td>13.2.2 Connecting to external data sources</td>
<td>582</td>
</tr>
<tr>
<td>13.2.3 Managing your database connections</td>
<td>582</td>
</tr>
<tr>
<td>13.3 Database administrator tools</td>
<td>582</td>
</tr>
<tr>
<td>13.3.1 Tools and utilities overview</td>
<td>582</td>
</tr>
<tr>
<td>13.3.2 EDB clone schema</td>
<td>582</td>
</tr>
<tr>
<td>13.3.3 EDB*Loader</td>
<td>583</td>
</tr>
<tr>
<td>13.3.4 EDB Resource Manager</td>
<td>583</td>
</tr>
<tr>
<td>13.4 Oracle-compatibility tools</td>
<td>583</td>
</tr>
<tr>
<td>14 Reference</td>
<td>583</td>
</tr>
<tr>
<td>14.1 Application programmer reference</td>
<td>583</td>
</tr>
<tr>
<td>Section</td>
<td>Content</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>14.1.1</td>
<td>Table partitioning views reference</td>
</tr>
<tr>
<td>14.1.1.1</td>
<td>ALL_PART_TABLES</td>
</tr>
<tr>
<td>14.1.1.2</td>
<td>ALL_TAB_PARTITIONS</td>
</tr>
<tr>
<td>14.1.1.3</td>
<td>ALL_TAB_SUBPARTITIONS</td>
</tr>
<tr>
<td>14.1.1.4</td>
<td>ALL_PART_KEY_COLUMNS</td>
</tr>
<tr>
<td>14.1.1.5</td>
<td>ALL_SUBPART_KEY_COLUMNS</td>
</tr>
<tr>
<td>14.1.2</td>
<td>System catalog tables</td>
</tr>
<tr>
<td>14.1.3</td>
<td>Language element reference</td>
</tr>
<tr>
<td>14.1.3.1</td>
<td>C-preprocessor directives</td>
</tr>
<tr>
<td>14.1.3.2</td>
<td>Language element reference</td>
</tr>
<tr>
<td>14.1.3.3</td>
<td>The SQLDA structure</td>
</tr>
<tr>
<td>14.1.3.4</td>
<td>Supported C data types</td>
</tr>
<tr>
<td>14.1.3.5</td>
<td>Type codes</td>
</tr>
<tr>
<td>14.1.4</td>
<td>Stored procedural language (SPL) reference</td>
</tr>
<tr>
<td>14.1.4.1</td>
<td>Basic SPL elements</td>
</tr>
<tr>
<td>14.1.4.1.1</td>
<td>Case sensitivity</td>
</tr>
<tr>
<td>14.1.4.1.2</td>
<td>Identifiers</td>
</tr>
<tr>
<td>14.1.4.1.3</td>
<td>Qualifiers</td>
</tr>
<tr>
<td>14.1.4.1.4</td>
<td>Constants</td>
</tr>
<tr>
<td>14.1.4.1.5</td>
<td>User-defined PL/SQL subtypes</td>
</tr>
<tr>
<td>14.1.4.1.6</td>
<td>Character set</td>
</tr>
<tr>
<td>14.1.4.2</td>
<td>Types of programming statements</td>
</tr>
<tr>
<td>14.1.4.2.1</td>
<td>Assignment</td>
</tr>
<tr>
<td>14.1.4.2.2</td>
<td>DELETE</td>
</tr>
<tr>
<td>14.1.4.2.3</td>
<td>INSERT</td>
</tr>
<tr>
<td>14.1.4.2.4</td>
<td>NULL</td>
</tr>
<tr>
<td>14.1.4.2.5</td>
<td>RETURNING INTO</td>
</tr>
<tr>
<td>14.1.4.2.6</td>
<td>SELECT INTO</td>
</tr>
<tr>
<td>14.1.4.2.7</td>
<td>UPDATE</td>
</tr>
<tr>
<td>14.1.4.2.8</td>
<td>Obtaining the result status</td>
</tr>
<tr>
<td>14.1.4.3</td>
<td>Types of control structures</td>
</tr>
<tr>
<td>14.1.4.3.1</td>
<td>IF statement</td>
</tr>
<tr>
<td>14.1.4.3.1.1</td>
<td>IF-THEN</td>
</tr>
<tr>
<td>14.1.4.3.1.2</td>
<td>IF-THEN-ELSE</td>
</tr>
<tr>
<td>14.1.4.3.1.3</td>
<td>IF-THEN-ELSE IF</td>
</tr>
<tr>
<td>14.1.4.3.1.4</td>
<td>IF-THEN-ELSIF-ELSE</td>
</tr>
<tr>
<td>14.1.4.3.2</td>
<td>RETURN statement</td>
</tr>
<tr>
<td>14.1.4.3.3</td>
<td>GOTO statement</td>
</tr>
<tr>
<td>14.1.4.3.4</td>
<td>CASE expression</td>
</tr>
<tr>
<td>14.1.4.3.4.1</td>
<td>Selector CASE expression</td>
</tr>
<tr>
<td>14.1.4.3.4.2</td>
<td>Searched CASE expression</td>
</tr>
<tr>
<td>14.1.4.3.5</td>
<td>CASE statement</td>
</tr>
<tr>
<td>14.1.4.3.5.1</td>
<td>Selector CASE statement</td>
</tr>
<tr>
<td>14.1.4.3.5.2</td>
<td>Searched CASE statement</td>
</tr>
<tr>
<td>14.1.4.3.6</td>
<td>Loops</td>
</tr>
<tr>
<td>14.1.4.3.6.1</td>
<td>LOOP</td>
</tr>
<tr>
<td>14.1.4.3.6.2</td>
<td>EXIT</td>
</tr>
<tr>
<td>14.1.4.3.6.3</td>
<td>CONTINUE</td>
</tr>
</tbody>
</table>

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
| 14.1.4.3.6.4 | WHILE |
| 14.1.4.3.6.5 | FOR (integer variant) |
| 14.1.4.3.7 | Exception handling |
| 14.1.4.3.8 | User-defined exceptions |
| 14.1.4.3.9 | PRAGMA EXCEPTION_INIT |
| 14.1.4.3.10 | RAISE_APPLICATION_ERROR |
| 14.1.4.4 | Collection methods |
| 14.1.4.4.1 | COUNT |
| 14.1.4.4.2 | DELETE |
| 14.1.4.4.3 | EXISTS |
| 14.1.4.4.4 | EXTEND |
| 14.1.4.4.5 | FIRST |
| 14.1.4.4.6 | LAST |
| 14.1.4.4.7 | LIMIT |
| 14.1.4.4.8 | NEXT |
| 14.1.4.4.9 | PRIOR |
| 14.1.4.4.10 | TRIM |
| 14.1.4.5 | EDB Postgres Advanced Server exceptions |
| 14.2 | Database administrator reference |
| 14.2.1 | Audit logging configuration parameters |
| 14.2.2 | Index Advisor components |
| 14.2.3 | Configuration parameters compatible with Oracle databases |
| 14.2.3.1 | edb_redwood_date |
| 14.2.3.2 | edb_redwood_raw_names |
| 14.2.3.3 | edb_redwood_strings |
| 14.2.3.4 | edb_stmt_level_tx |
| 14.2.3.5 | oracle_home |
| 14.2.4 | Summary of configuration parameters |
| 14.2.5 | Audit log files |
| 14.2.6 | EDB*Loader control file parameters |
| 14.2.7 | System catalogs |
| 14.3 | Oracle compatibility reference |
| 14.3.1 | dblink_ora functions and procedures |
| 14.3.1.1 | dblink_ora_connect() |
| 14.3.1.2 | dblink_ora_status() |
| 14.3.1.3 | dblink_ora_disconnect() |
| 14.3.1.4 | dblink_ora_record() |
| 14.3.1.5 | dblink_ora_call() |
| 14.3.1.6 | dblink_ora_exec() |
| 14.3.1.7 | dblink_ora_copy() |
| 14.3.2 | Partitioning commands compatible with Oracle Databases |
| 14.3.2.1 | CREATE TABLE...PARTITION BY |
| 14.3.2.1.1 | Example: PARTITION BY LIST |
| 14.3.2.1.2 | Example: AUTOMATIC LIST PARTITION |
| 14.3.2.1.3 | Example: PARTITION BY RANGE |
| 14.3.2.1.4 | Example: INTERVAL RANGE PARTITION |
| 14.3.2.1.5 | Example: PARTITION BY HASH |
| 14.3.2.1.6 | Example: PARTITION BY HASH...PARTITIONS num... |
14.3.2.1.7 Example: PARTITION BY RANGE, SUBPARTITION BY LIST
14.3.2.1.8 Example: CREATING UNIQUE INDEX on PARTITION TABLE
14.3.2.1.9 Example: CREATING SUBPARTITION TEMPLATE
14.3.2.2 ALTER TABLE...ADD PARTITION
14.3.2.2.1 Example: Adding a partition to a LIST partitioned table
14.3.2.2.2 Example - Adding a partition to a RANGE partitioned table
14.3.2.2.3 Example: Adding a partition with SUBPARTITIONS num...IN PARTITION DESCRIPTION
14.3.2.3 ALTER TABLE...ADD SUBPARTITION
14.3.2.3.1 Example: Adding a subpartition to a LIST/RANGE partitioned table
14.3.2.3.2 Example: Adding a subpartition to a RANGE/LIST partitioned table
14.3.2.4 ALTER TABLE...SPLIT PARTITION
14.3.2.4.1 Example: Splitting a LIST partition
14.3.2.4.2 Example: Splitting a RANGE partition
14.3.2.4.3 Example: Splitting a RANGE/LIST partition
14.3.2.5 ALTER TABLE...SPLIT SUBPARTITION
14.3.2.5.1 Example: Splitting a LIST subpartition
14.3.2.5.2 Example: Splitting a RANGE subpartition
14.3.2.6 ALTER TABLE...EXCHANGE PARTITION
14.3.2.6.1 Example: Exchanging a table for a partition
14.3.2.7 ALTER TABLE...MOVE PARTITION
14.3.2.7.1 Example: Moving a partition to a different tablespace
14.3.2.8 ALTER TABLE...RENAME PARTITION
14.3.2.8.1 Example: Renaming a partition
14.3.2.9 ALTER TABLE...SET INTERVAL
14.3.2.9.1 Example: Setting an interval range partition
14.3.2.10 ALTER TABLE...SET [ PARTITIONING ] AUTOMATIC
14.3.2.10.1 Example: Setting an AUTOMATIC list partition
14.3.2.11 ALTER TABLE...SET SUBPARTITION TEMPLATE
14.3.2.11.1 Example: Setting a SUBPARTITION TEMPLATE
14.3.2.12 DROP TABLE
14.3.2.13 ALTER TABLE...DROP PARTITION
14.3.2.13.1 Example: Deleting a partition
14.3.2.14 ALTER TABLE...DROP SUBPARTITION
14.3.2.14.1 Example: Deleting a subpartition
14.3.2.15 TRUNCATE TABLE
14.3.2.15.1 Example: Emptying a table
14.3.2.16 ALTER TABLE...TRUNCATE PARTITION
14.3.2.16.1 Example: Emptying a partition
14.3.2.17 ALTER TABLE...TRUNCATE SUBPARTITION
14.3.2.17.1 Example: Emptying a subpartition
14.3.2.18 Accessing a PARTITION or SUBPARTITION
14.3.3 Built-in packages
14.3.3.1 Built-in packages
14.3.3.1.1 DBMS_ALERT
14.3.3.1.2 DBMS_AQ
14.3.3.1.2.1 ENQUEUE
14.3.3.1.2.2 DEQUEUE
14.3.3.1.2.3 REGISTER
<p>| 14.3.3.1.2.4 | UNREGISTER                      | 772 |
| 14.3.3.1.3  | DBMS_AQADM                     | 773 |
| 14.3.3.1.3.1 | ALTER_QUEUE                    | 774 |
| 14.3.3.1.3.2 | ALTER_QUEUE_TABLE              | 775 |
| 14.3.3.1.3.3 | CREATE_QUEUE                   | 775 |
| 14.3.3.1.3.4 | CREATE_QUEUE_TABLE             | 776 |
| 14.3.3.1.3.5 | DROP_QUEUE                     | 778 |
| 14.3.3.1.3.6 | DROP_QUEUE_TABLE               | 779 |
| 14.3.3.1.3.7 | PURGE_QUEUE_TABLE              | 779 |
| 14.3.3.1.3.8 | START_QUEUE                    | 780 |
| 14.3.3.1.3.9 | STOP_QUEUE                     | 780 |
| 14.3.3.1.4  | DBMS_CRYPTO                    | 781 |
| 14.3.3.1.4.1 | DECRYPT                       | 782 |
| 14.3.3.1.4.2 | ENCRYPT                       | 783 |
| 14.3.3.1.4.3 | HASH                           | 785 |
| 14.3.3.1.4.4 | MAC                            | 785 |
| 14.3.3.1.4.5 | RANDOMBYTES                    | 786 |
| 14.3.3.1.4.6 | RANDOMINTEGER                  | 787 |
| 14.3.3.1.4.7 | RANDOMNUMBER                   | 787 |
| 14.3.3.1.5  | DBMS_JOB                       | 787 |
| 14.3.3.1.5.1 | BROKEN                        | 788 |
| 14.3.3.1.5.2 | CHANGE                        | 789 |
| 14.3.3.1.5.3 | INTERVAL                      | 789 |
| 14.3.3.1.5.4 | NEXT_DATE                      | 790 |
| 14.3.3.1.5.5 | REMOVE                        | 790 |
| 14.3.3.1.5.6 | RUN                            | 791 |
| 14.3.3.1.5.7 | SUBMIT                        | 791 |
| 14.3.3.1.5.8 | WHAT                          | 792 |
| 14.3.3.1.6  | DBMS_LOB                       | 793 |
| 14.3.3.1.6.1 | APPEND                         | 793 |
| 14.3.3.1.6.2 | COMPARE                        | 794 |
| 14.3.3.1.6.3 | CONVERTTOBLOB                 | 794 |
| 14.3.3.1.6.4 | CONVERTTOCLOB                 | 795 |
| 14.3.3.1.6.5 | COPY                           | 796 |
| 14.3.3.1.6.6 | ERASE                          | 797 |
| 14.3.3.1.6.7 | GET_STORAGE_LIMIT             | 797 |
| 14.3.3.1.6.8 | GETLENGTH                     | 797 |
| 14.3.3.1.6.9 | INSTR                          | 798 |
| 14.3.3.1.6.10 | READ                           | 798 |
| 14.3.3.1.6.11 | SUBSTR                         | 799 |
| 14.3.3.1.6.12 | TRIM                           | 799 |
| 14.3.3.1.6.13 | WRITE                          | 800 |
| 14.3.3.1.6.14 | WRITEAPPEND                   | 800 |
| 14.3.3.1.7  | DBMS_LOCK                      | 800 |
| 14.3.3.1.8  | DBMS_MVIEW                     | 801 |
| 14.3.3.1.8.1 | GET_MV_DEPENDENCIES           | 801 |
| 14.3.3.1.8.2 | REFRESH                        | 802 |
| 14.3.3.1.8.3 | REFRESH_ALL_MVIEWS             | 803 |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE_COLUMN_CHAR</td>
<td>894</td>
</tr>
<tr>
<td>DEFINE_COLUMN_RAW</td>
<td>894</td>
</tr>
<tr>
<td>DEFINE_COLUMN_LONG</td>
<td>896</td>
</tr>
<tr>
<td>DEFINE_COLUMN_ROWID</td>
<td>897</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS</td>
<td>898</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS2</td>
<td>899</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS3</td>
<td>900</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>901</td>
</tr>
<tr>
<td>EXECUTE_AND_FETCH</td>
<td>902</td>
</tr>
<tr>
<td>FETCH_ROWS</td>
<td>904</td>
</tr>
<tr>
<td>IS_OPEN</td>
<td>905</td>
</tr>
<tr>
<td>LAST_ROW_COUNT</td>
<td>905</td>
</tr>
<tr>
<td>OPEN_CURSOR</td>
<td>907</td>
</tr>
<tr>
<td>PARSE</td>
<td>908</td>
</tr>
<tr>
<td>TO_CURSOR_NUMBER</td>
<td>909</td>
</tr>
<tr>
<td>TO_REFCURSOR</td>
<td>910</td>
</tr>
<tr>
<td>DBMS_UTILITY</td>
<td>911</td>
</tr>
<tr>
<td>UTL_ENCODE</td>
<td>924</td>
</tr>
<tr>
<td>BASE64_DECODE</td>
<td>925</td>
</tr>
<tr>
<td>BASE64_ENCODE</td>
<td>926</td>
</tr>
<tr>
<td>MIMEHEADER_DECODE</td>
<td>927</td>
</tr>
<tr>
<td>MIMEHEADER_ENCODE</td>
<td>928</td>
</tr>
<tr>
<td>QUOTED_PRINTABLE_DECODE</td>
<td>928</td>
</tr>
<tr>
<td>QUOTED_PRINTABLE_ENCODE</td>
<td>928</td>
</tr>
<tr>
<td>TEXT_DECODE</td>
<td>929</td>
</tr>
<tr>
<td>TEXT_ENCODE</td>
<td>930</td>
</tr>
<tr>
<td>uudecode</td>
<td>930</td>
</tr>
<tr>
<td>uuencode</td>
<td>931</td>
</tr>
<tr>
<td>UTL_FILE</td>
<td>932</td>
</tr>
<tr>
<td>UTL_HTTP</td>
<td>951</td>
</tr>
<tr>
<td>UTL_MAIL</td>
<td>967</td>
</tr>
<tr>
<td>UTL_RAW</td>
<td>971</td>
</tr>
<tr>
<td>UTL_SMTP</td>
<td>975</td>
</tr>
<tr>
<td>UTL_URL</td>
<td>982</td>
</tr>
<tr>
<td>HTP and HTF</td>
<td>985</td>
</tr>
<tr>
<td>ADDRESS</td>
<td>985</td>
</tr>
<tr>
<td>ANCHOR and ANCHOR2</td>
<td>985</td>
</tr>
<tr>
<td>APPLETOPEN and APPLETCLOSE</td>
<td>986</td>
</tr>
<tr>
<td>AREA</td>
<td>986</td>
</tr>
<tr>
<td>BASE</td>
<td>987</td>
</tr>
<tr>
<td>BASEFONT</td>
<td>987</td>
</tr>
<tr>
<td>BGSOUND</td>
<td>988</td>
</tr>
<tr>
<td>BIG</td>
<td>988</td>
</tr>
<tr>
<td>BLOCKQUOTEOPEN and BLOCKQUOTECLOSE</td>
<td>989</td>
</tr>
<tr>
<td>BODYOPEN and BODYCLOSE</td>
<td>989</td>
</tr>
<tr>
<td>BOLD</td>
<td>990</td>
</tr>
<tr>
<td>BR and NL</td>
<td>990</td>
</tr>
<tr>
<td>TABLECAPTION</td>
<td>991</td>
</tr>
</tbody>
</table>
14.3.3.1.27.62 NOFRAMESOPEN and NOFRAMESCLOSE
14.3.3.1.27.63 OLSTOPEN and OLSTCLOSE
14.3.3.1.27.64 PARA and PARAGRAPH
14.3.3.1.27.65 PARAM
14.3.3.1.27.66 PLAINTEXT
14.3.3.1.27.67 PREOPEN and PRECLOSE
14.3.3.1.27.68 PRINT and PRN
14.3.3.1.27.69 PRINTS and PS
14.3.3.1.27.70 S
14.3.3.1.27.71 SAMPLE
14.3.3.1.27.72 SMALL
14.3.3.1.27.73 STRIKE
14.3.3.1.27.74 STRONG
14.3.3.1.27.75 STYLE
14.3.3.1.27.76 SUB
14.3.3.1.27.77 SUP
14.3.3.1.27.78 TABLEDATA
14.3.3.1.27.79 TABLEHEADER
14.3.3.1.27.80 TABLEOPEN and TABLECLOSE
14.3.3.1.27.81 TABLEROWOPEN and TABLEROWCLOSE
14.3.3.1.27.82 TELETYPE
14.3.3.1.27.83 TITLE
14.3.3.1.27.84 ULISTOPEN and ULISTCLOSE
14.3.3.1.27.85 UNDERLINE
14.3.3.1.27.86 VARIABLE
14.3.3.1.27.87 WBR
14.3.4 Database compatibility for Oracle developers: catalog views
14.3.4.1 Oracle catalog views
14.3.4.1.1 ALL_ALL_TABLES
14.3.4.1.2 ALL_CONS_COLUMNNS
14.3.4.1.3 ALL_CONSTRAINTS
14.3.4.1.4 ALL_COL_PRIVS
14.3.4.1.5 ALL_DB_LINKS
14.3.4.1.6 ALL_DEPENDENCIES
14.3.4.1.7 ALL_DIRECTORIES
14.3.4.1.8 ALL_IND_COLUMNNS
14.3.4.1.9 ALL_INDEXES
14.3.4.1.10 ALL_JOBS
14.3.4.1.11 ALL_OBJECTS
14.3.4.1.12 ALL_PART_KEY_COLUMNNS
14.3.4.1.13 ALL_PART_TABLES
14.3.4.1.14 ALL_POLICIES
14.3.4.1.15 ALL_QUEUES
14.3.4.1.16 ALL_QUEUE_TABLES
14.3.4.1.17 ALL_SEQUENCES
14.3.4.1.18 ALL_SOURCE
14.3.4.1.19 ALL_SUBPART_KEY_COLUMNNS
14.3.4.1.20 ALL_SYNONYMS
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.3.4.1.21</td>
<td>ALL_TAB_COLUMNS</td>
<td>1033</td>
</tr>
<tr>
<td>14.3.4.1.22</td>
<td>ALL_TAB_PARTITIONS</td>
<td>1034</td>
</tr>
<tr>
<td>14.3.4.1.23</td>
<td>ALL_TAB_SUBPARTITIONS</td>
<td>1034</td>
</tr>
<tr>
<td>14.3.4.1.24</td>
<td>ALL_TAB_PRIVS</td>
<td>1035</td>
</tr>
<tr>
<td>14.3.4.1.25</td>
<td>ALL_TABLES</td>
<td>1036</td>
</tr>
<tr>
<td>14.3.4.1.26</td>
<td>ALL_TRIGGERS</td>
<td>1036</td>
</tr>
<tr>
<td>14.3.4.1.27</td>
<td>ALL_TYPES</td>
<td>1036</td>
</tr>
<tr>
<td>14.3.4.1.28</td>
<td>ALL_USERS</td>
<td>1037</td>
</tr>
<tr>
<td>14.3.4.1.29</td>
<td>ALL_VIEW_COLUMNS</td>
<td>1037</td>
</tr>
<tr>
<td>14.3.4.1.30</td>
<td>ALL_VIEWS</td>
<td>1037</td>
</tr>
<tr>
<td>14.3.4.1.31</td>
<td>DBA_ALL_TABLES</td>
<td>1037</td>
</tr>
<tr>
<td>14.3.4.1.32</td>
<td>DBA_CONS_COLUMNS</td>
<td>1038</td>
</tr>
<tr>
<td>14.3.4.1.33</td>
<td>DBA_CONSTRAINTS</td>
<td>1038</td>
</tr>
<tr>
<td>14.3.4.1.34</td>
<td>DBA_COL_PRIVS</td>
<td>1038</td>
</tr>
<tr>
<td>14.3.4.1.35</td>
<td>DBA_DB_LINKS</td>
<td>1039</td>
</tr>
<tr>
<td>14.3.4.1.36</td>
<td>DBA_DIRECTORIES</td>
<td>1039</td>
</tr>
<tr>
<td>14.3.4.1.37</td>
<td>DBA_DEPENDENCIES</td>
<td>1039</td>
</tr>
<tr>
<td>14.3.4.1.38</td>
<td>DBA_IND_COLUMNS</td>
<td>1040</td>
</tr>
<tr>
<td>14.3.4.1.39</td>
<td>DBA_INDEXES</td>
<td>1040</td>
</tr>
<tr>
<td>14.3.4.1.40</td>
<td>DBA_JOBS</td>
<td>1040</td>
</tr>
<tr>
<td>14.3.4.1.41</td>
<td>DBA_OBJECTS</td>
<td>1041</td>
</tr>
<tr>
<td>14.3.4.1.42</td>
<td>DBA_PART_KEY_COLUMNS</td>
<td>1041</td>
</tr>
<tr>
<td>14.3.4.1.43</td>
<td>DBA_PART_TABLES</td>
<td>1041</td>
</tr>
<tr>
<td>14.3.4.1.44</td>
<td>DBA_POLICIES</td>
<td>1042</td>
</tr>
<tr>
<td>14.3.4.1.45</td>
<td>DBA_PRIVS_CAPTURE</td>
<td>1043</td>
</tr>
<tr>
<td>14.3.4.1.46</td>
<td>DBA_PROFILES</td>
<td>1043</td>
</tr>
<tr>
<td>14.3.4.1.47</td>
<td>DBA_QUEUES</td>
<td>1043</td>
</tr>
<tr>
<td>14.3.4.1.48</td>
<td>DBA_QUEUE_TABLES</td>
<td>1043</td>
</tr>
<tr>
<td>14.3.4.1.49</td>
<td>DBA_ROLE_PRIVS</td>
<td>1044</td>
</tr>
<tr>
<td>14.3.4.1.50</td>
<td>DBA_ROLES</td>
<td>1044</td>
</tr>
<tr>
<td>14.3.4.1.51</td>
<td>DBA_SEQUENCES</td>
<td>1044</td>
</tr>
<tr>
<td>14.3.4.1.52</td>
<td>DBA_SOURCE</td>
<td>1045</td>
</tr>
<tr>
<td>14.3.4.1.53</td>
<td>DBA_SUBPART_KEY_COLUMNS</td>
<td>1045</td>
</tr>
<tr>
<td>14.3.4.1.54</td>
<td>DBA_SYNONYMS</td>
<td>1045</td>
</tr>
<tr>
<td>14.3.4.1.55</td>
<td>DBA_TAB_COLUMNS</td>
<td>1045</td>
</tr>
<tr>
<td>14.3.4.1.56</td>
<td>DBA_TAB_PARTITIONS</td>
<td>1046</td>
</tr>
<tr>
<td>14.3.4.1.57</td>
<td>DBA_TAB_SUBPARTITIONS</td>
<td>1046</td>
</tr>
<tr>
<td>14.3.4.1.58</td>
<td>DBA_TAB_PRIVS</td>
<td>1047</td>
</tr>
<tr>
<td>14.3.4.1.59</td>
<td>DBA_TABLES</td>
<td>1048</td>
</tr>
<tr>
<td>14.3.4.1.60</td>
<td>DBA_TRIGGERS</td>
<td>1048</td>
</tr>
<tr>
<td>14.3.4.1.61</td>
<td>DBA_TYPES</td>
<td>1048</td>
</tr>
<tr>
<td>14.3.4.1.62</td>
<td>DBA_USERS</td>
<td>1049</td>
</tr>
<tr>
<td>14.3.4.1.63</td>
<td>DBA_USED_PRIVS</td>
<td>1049</td>
</tr>
<tr>
<td>14.3.4.1.64</td>
<td>DBA_UNUSED_PRIVS</td>
<td>1050</td>
</tr>
<tr>
<td>14.3.4.1.65</td>
<td>DBA_VIEW_COLUMNS</td>
<td>1050</td>
</tr>
<tr>
<td>14.3.4.1.66</td>
<td>DBA_VIEWS</td>
<td>1050</td>
</tr>
<tr>
<td>14.3.4.1.67</td>
<td>USER_ALL_TABLES</td>
<td>1050</td>
</tr>
<tr>
<td>14.3.4.1.68</td>
<td>USER_CONS_COLUMNS</td>
<td>1051</td>
</tr>
<tr>
<td>14.3.4.1.69</td>
<td>USER_CONSTRAINTS</td>
<td>1051</td>
</tr>
<tr>
<td>14.3.4.1.70</td>
<td>USER_COL_PRIVS</td>
<td>1051</td>
</tr>
<tr>
<td>14.3.4.1.71</td>
<td>USER_DB_LINKS</td>
<td>1052</td>
</tr>
<tr>
<td>14.3.4.1.72</td>
<td>USER_DEPENDENCIES</td>
<td>1052</td>
</tr>
<tr>
<td>14.3.4.1.73</td>
<td>USER_INDEXES</td>
<td>1052</td>
</tr>
<tr>
<td>14.3.4.1.74</td>
<td>USER_JOBS</td>
<td>1053</td>
</tr>
<tr>
<td>14.3.4.1.75</td>
<td>USER_OBJECTS</td>
<td>1053</td>
</tr>
<tr>
<td>14.3.4.1.76</td>
<td>USER_PART_TABLES</td>
<td>1054</td>
</tr>
<tr>
<td>14.3.4.1.77</td>
<td>USER_POLICIES</td>
<td>1054</td>
</tr>
<tr>
<td>14.3.4.1.78</td>
<td>USER_QUEUES</td>
<td>1055</td>
</tr>
<tr>
<td>14.3.4.1.79</td>
<td>USER_QUEUE_TABLES</td>
<td>1055</td>
</tr>
<tr>
<td>14.3.4.1.80</td>
<td>USER_ROLE_PRIVS</td>
<td>1055</td>
</tr>
<tr>
<td>14.3.4.1.81</td>
<td>USER_SEQUENCES</td>
<td>1056</td>
</tr>
<tr>
<td>14.3.4.1.82</td>
<td>USER_SOURCE</td>
<td>1056</td>
</tr>
<tr>
<td>14.3.4.1.83</td>
<td>USER_SUBPART_KEY_COLUMNS</td>
<td>1056</td>
</tr>
<tr>
<td>14.3.4.1.84</td>
<td>USER_SYNONYMS</td>
<td>1056</td>
</tr>
<tr>
<td>14.3.4.1.85</td>
<td>USER_TAB_COLUMNS</td>
<td>1057</td>
</tr>
<tr>
<td>14.3.4.1.86</td>
<td>USER_TAB_PARTITIONS</td>
<td>1057</td>
</tr>
<tr>
<td>14.3.4.1.87</td>
<td>USER_TAB_SUBPARTITIONS</td>
<td>1058</td>
</tr>
<tr>
<td>14.3.4.1.88</td>
<td>USER_TAB_PRIVS</td>
<td>1059</td>
</tr>
<tr>
<td>14.3.4.1.89</td>
<td>USER_TABLES</td>
<td>1059</td>
</tr>
<tr>
<td>14.3.4.1.90</td>
<td>USER_TRIGGERS</td>
<td>1059</td>
</tr>
<tr>
<td>14.3.4.1.91</td>
<td>USER_TYPES</td>
<td>1060</td>
</tr>
<tr>
<td>14.3.4.1.92</td>
<td>USER_USERS</td>
<td>1060</td>
</tr>
<tr>
<td>14.3.4.1.93</td>
<td>USER_VIEW_COLUMNS</td>
<td>1060</td>
</tr>
<tr>
<td>14.3.4.1.94</td>
<td>USER_VIEWS</td>
<td>1061</td>
</tr>
<tr>
<td>14.3.4.1.95</td>
<td>V$VERSION</td>
<td>1061</td>
</tr>
<tr>
<td>14.3.4.1.96</td>
<td>PRODUCT_COMPONENT_VERSION</td>
<td>1061</td>
</tr>
<tr>
<td>14.3.4.2</td>
<td>System catalog views</td>
<td>1061</td>
</tr>
<tr>
<td>14.3.4.2.1</td>
<td>PG_USER</td>
<td>1061</td>
</tr>
<tr>
<td>14.3.5</td>
<td>Compatible SQL commands</td>
<td>1062</td>
</tr>
<tr>
<td>14.3.5.1</td>
<td>ALTER CAPTURE PRIVILEGE POLICY</td>
<td>1062</td>
</tr>
<tr>
<td>14.3.5.2</td>
<td>ALTER DIRECTORY</td>
<td>1063</td>
</tr>
<tr>
<td>14.3.5.3</td>
<td>ALTER INDEX</td>
<td>1064</td>
</tr>
<tr>
<td>14.3.5.4</td>
<td>ALTER PROCEDURE</td>
<td>1065</td>
</tr>
<tr>
<td>14.3.5.5</td>
<td>ALTER PROFILE</td>
<td>1066</td>
</tr>
<tr>
<td>14.3.5.6</td>
<td>ALTER QUEUE</td>
<td>1068</td>
</tr>
<tr>
<td>14.3.5.7</td>
<td>ALTER QUEUE TABLE</td>
<td>1070</td>
</tr>
<tr>
<td>14.3.5.8</td>
<td>ALTER ROLE... IDENTIFIED BY</td>
<td>1071</td>
</tr>
<tr>
<td>14.3.5.9</td>
<td>ALTER ROLE: Managing database link and DBMS_RLS privileges</td>
<td>1072</td>
</tr>
<tr>
<td>14.3.5.10</td>
<td>ALTER SEQUENCE</td>
<td>1074</td>
</tr>
<tr>
<td>14.3.5.11</td>
<td>ALTER SESSION</td>
<td>1075</td>
</tr>
<tr>
<td>14.3.5.12</td>
<td>ALTER SYNONYM</td>
<td>1076</td>
</tr>
<tr>
<td>14.3.5.13</td>
<td>ALTER TABLE</td>
<td>1077</td>
</tr>
<tr>
<td>14.3.5.14</td>
<td>ALTER TRIGGER</td>
<td>1080</td>
</tr>
<tr>
<td>14.3.5.15</td>
<td>ALTER TABLESPACE</td>
<td>1081</td>
</tr>
<tr>
<td>14.3.5.16</td>
<td>ALTER USER... IDENTIFIED BY</td>
<td>1082</td>
</tr>
<tr>
<td>14.3.5.17</td>
<td>ALTER USER</td>
<td>ROLE... PROFILE MANAGEMENT CLAUSES</td>
</tr>
<tr>
<td>Section</td>
<td>Command</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------</td>
<td></td>
</tr>
<tr>
<td>14.3.5.18</td>
<td>CALL</td>
<td></td>
</tr>
<tr>
<td>14.3.5.19</td>
<td>COMMENT</td>
<td></td>
</tr>
<tr>
<td>14.3.5.20</td>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td>14.3.5.21</td>
<td>CREATE CAPTURE PRIVILEGES POLICY</td>
<td></td>
</tr>
<tr>
<td>14.3.5.22</td>
<td>CREATE DATABASE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.23</td>
<td>CREATE PUBLIC DATABASE LINK</td>
<td></td>
</tr>
<tr>
<td>14.3.5.24</td>
<td>CREATE DIRECTORY</td>
<td></td>
</tr>
<tr>
<td>14.3.5.25</td>
<td>CREATE FUNCTION</td>
<td></td>
</tr>
<tr>
<td>14.3.5.26</td>
<td>CREATE INDEX</td>
<td></td>
</tr>
<tr>
<td>14.3.5.27</td>
<td>CREATE MATERIALIZED VIEW</td>
<td></td>
</tr>
<tr>
<td>14.3.5.28</td>
<td>CREATE PACKAGE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.29</td>
<td>CREATE PACKAGE BODY</td>
<td></td>
</tr>
<tr>
<td>14.3.5.30</td>
<td>CREATE PROCEDURE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.31</td>
<td>CREATE PROFILE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.32</td>
<td>CREATE QUEUE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.33</td>
<td>CREATE QUEUE TABLE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.34</td>
<td>CREATE ROLE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.35</td>
<td>CREATE SCHEMA</td>
<td></td>
</tr>
<tr>
<td>14.3.5.36</td>
<td>CREATE SEQUENCE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.37</td>
<td>CREATE SYNONYM</td>
<td></td>
</tr>
<tr>
<td>14.3.5.38</td>
<td>CREATE TABLE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.39</td>
<td>CREATE TABLE AS</td>
<td></td>
</tr>
<tr>
<td>14.3.5.40</td>
<td>CREATE TRIGGER</td>
<td></td>
</tr>
<tr>
<td>14.3.5.41</td>
<td>CREATE TYPE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.42</td>
<td>CREATE TYPE BODY</td>
<td></td>
</tr>
<tr>
<td>14.3.5.43</td>
<td>CREATE USER</td>
<td></td>
</tr>
<tr>
<td>14.3.5.44</td>
<td>CREATE USER ROLE... PROFILE MANAGEMENT CLAUSES</td>
<td></td>
</tr>
<tr>
<td>14.3.5.45</td>
<td>CREATE VIEW</td>
<td></td>
</tr>
<tr>
<td>14.3.5.46</td>
<td>DELETE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.47</td>
<td>DROP CAPTURE PRIVILEGES POLICY</td>
<td></td>
</tr>
<tr>
<td>14.3.5.48</td>
<td>DROP DATABASE LINK</td>
<td></td>
</tr>
<tr>
<td>14.3.5.49</td>
<td>DROP DIRECTORY</td>
<td></td>
</tr>
<tr>
<td>14.3.5.50</td>
<td>DROP FUNCTION</td>
<td></td>
</tr>
<tr>
<td>14.3.5.51</td>
<td>DROP INDEX</td>
<td></td>
</tr>
<tr>
<td>14.3.5.52</td>
<td>DROP PACKAGE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.53</td>
<td>DROP PROCEDURE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.54</td>
<td>DROP PROFILE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.55</td>
<td>DROP QUEUE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.56</td>
<td>DROP QUEUE TABLE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.57</td>
<td>DROP SYNONYM</td>
<td></td>
</tr>
<tr>
<td>14.3.5.58</td>
<td>DROP ROLE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.59</td>
<td>DROP SEQUENCE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.60</td>
<td>DROP TABLE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.61</td>
<td>DROP TABLESPACE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.62</td>
<td>DROP TRIGGER</td>
<td></td>
</tr>
<tr>
<td>14.3.5.63</td>
<td>DROP TYPE</td>
<td></td>
</tr>
<tr>
<td>14.3.5.64</td>
<td>DROP USER</td>
<td></td>
</tr>
<tr>
<td>14.3.5.65</td>
<td>DROP VIEW</td>
<td></td>
</tr>
</tbody>
</table>
EDB Postgres Advanced Server (EPAS)

EDB is the only worldwide company to deliver innovative and low-cost, open-source-derived database solutions. These solutions offer commercial quality, ease of use, compatibility, scalability, and performance for small or large-scale enterprises. EDB Postgres Advanced Server (sometimes referred to as EPAS in this documentation) adds extended functionality to open-source PostgreSQL, including:

- Database administration
- Enhanced SQL capabilities
- Database and application security
- Performance monitoring and analysis
- Application development utilities
- Advanced replication (version 14 and higher).

All of these features are available in Postgres mode and Oracle compatibility mode. When you launch EDB Postgres Advanced Server in Oracle compatibility mode, you get additional features that help with Oracle-to-Postgres migrations.

- Oracle-compatible custom data types
- Oracle keywords
- Oracle functions
- Oracle-style catalog views
- Additional compatibility with Oracle MERGE.

EDB also makes available a full suite of tools and utilities that helps you monitor and manage your EDB Postgres Advanced Server deployment.

EDB Postgres Advanced Server release notes

EDB Postgres Advanced Server 16 is built on open-source PostgreSQL 16, which introduces myriad enhancements that enable databases to scale up and scale out in more efficient ways.

The EDB Postgres Advanced Server documentation describes the latest version of EDB Postgres Advanced Server 16 including minor releases and patches. These release notes provide information on what was new in each release.

<table>
<thead>
<tr>
<th>Version</th>
<th>Release date</th>
<th>Upstream merges</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1</td>
<td>09 Nov 2023</td>
<td>16.1</td>
</tr>
</tbody>
</table>

Component certification

This components are included in the EDB Postgres Advanced Server v16.1 release:

- Database Server 16.1
- SQL Protect

Support announcements

Backup and Recovery Tool (BART) incompatibility

The EDB Backup and Recovery Tool (BART) isn’t supported by EDB Postgres Advanced Server or PostgreSQL version 14 and later. We strongly recommend that you move to Barman or PgBackRest as your backup recovery tool. See Moving to a New Backup Tool in the BART documentation for more information.

EDB Postgres Advanced Server 16.1 release notes

Released: 09 Nov 2023

EDB Postgres Advanced Server 16.1 includes the following enhancements and bug fixes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upstream merge</td>
<td>Merged with community PostgreSQL 16 beta. See the PostgreSQL 16 Release Notes for more information.</td>
</tr>
</tbody>
</table>
Feature
Added support for Oracle-compatible `DBMS_PRIVILEGE_CAPTURE` package and the SQL commands to define the capture privilege policy. See the `DBMS_PRIVILEGE_CAPTURE` for more information.

Feature
Added support for two timezone functions: `DBTIMEZONE` and `SESSIONTIMEZONE`. See the `TIMEZONE` functions for more information.

Feature
Added support for the Oracle-compatible `NANVL` function. See the `NANVL` function for more information.

Feature
Added support for the Oracle-compatible `DUMP` SQL function. See the `DUMP` function for more information.

Feature
Added support for the Oracle-compatible `NLS_CHARSET` functions. See the `NLS` functions for more information.

Feature
Added support for few more subprograms of `DBMS_SQL` package to ease the migrations. See the `DBMS_SQL` package for more information.

Feature
Added `spl_check` extension to check and report errors for the EDB SPL language. See the `spl_check` for more information.

Enhancement
Added support for `UTL_FILE` subprograms to ease the migrations. See the `utl_file` for more information.

Enhancement
Transparent data encryption (TDE) now supports the AES key length 256 option to TDE.

Enhancement
Added the `edb_job_scheduler` extension, which provides a background process that does the job scheduling for the `DBMS_SCHEDULER` and `DBMS_JOB` packages. Before, these packages were dependent on pgAgent. With this extension, packages don’t have that dependency. See the `DBMS_SCHEDULER` extension for more information.

Enhancement
Implemented `MULTISET INTERSECT UNIQUE/DISTINCT` and `MULTISET EXCEPT UNIQUE/DISTINCT` now for SPL collection types.

Enhancement
You can now use the `BULK COLLECT` clause with a `FETCH` statement.

Enhancement
Added synonym support for packages.

Enhancement
The `TO_TIMESTAMP_TZ()` function now takes a single argument that converts a string to `TIMESTAMPTZ` in the default format.

Enhancement
The `EXPAND_SQL_TEXT` procedure of the `DBMS_UTILITY` package is now supported. It returns an expanded version of a given SQL query by replacing view references with its definition.

Deprecation
`edb_partition_pruning` GUC parameter is now deprecated.

3  Supported platforms

EDB Postgres Advanced Server supports installations on Linux and Windows platforms.

To learn about the platform support for EDB Postgres Advanced Server, you can refer either to the platform support for EDB Postgres Advanced Server on the Platform Compatibility page on the EDB website or Installing EDB Postgres Advanced Server.

4  Fundamentals

The key concepts of EDB Postgres Advanced Server include its features, benefits, and terminology. The SQL tutorial uses a sample database and explains important database concepts.

4.1  EDB Postgres Advanced Server fundamentals

EDB Postgres Advanced Server provides tools you can use to administer enterprise-scale data solutions. These essential concepts help you to design, implement, and manage EDB Postgres Advanced Server.

4.1.1  Benefits of EDB Postgres Advanced Server

EDB Postgres Advanced Server is a stand-alone proprietary database server that adds extended functionality to the open-source PostgreSQL database.

EDB Postgres Advanced Server is designed to provide enterprise capabilities on top of the features present in community Postgres. These capabilities:

- Help database administrators to maintain and operate EDB Postgres Advanced Server databases
- Include enhanced SQL functionality and other features that add flexibility and convenience
- Extend Postgres security with features designed to limit unauthorized access
- Assist with performance monitoring and analysis
- Include multiple features designed to increase application programmer productivity
- Advanced replication support (version 14 and higher).

EDB Postgres Advanced Server also has a significant amount of Oracle compatibility features which facilitate Oracle to Postgres migrations.

EDB Postgres Advanced Server includes extended functionality that provides compatibility for syntax supported by Oracle applications, as well as compatible procedural logic, data types, system catalog
views and other features that enable EDB’s Oracle compatible connectors, EDB*Plus, EDB*Loader and many other tools and functionality.

Major versions of EDB Postgres Advanced Server are released yearly in line with the major version release of PostgreSQL. EDB Postgres Advanced Server is normally released one to two months after PostgreSQL major version releases.

EDB Postgres Advanced Server is supported on Linux x86-64, Linux on IBM Power and Windows x86-64.

4.1.2 Database compatibility for Oracle developers

EDB Postgres Advanced Server makes Postgres look, feel, and operate more like Oracle, so when you migrate, there is less code to rewrite, and you can be up and running quickly. The Oracle compatibility features allow you to run many applications written for Oracle in EDB Postgres Advanced Server with minimal to no changes.

EDB Postgres Advanced Server provides Oracle compatible:

- Stored procedure language (SPL) when creating database server-side application logic for stored procedures, functions, triggers, and packages
- Data types that are compatible with Oracle databases
- SQL statements that are compatible with Oracle SQL
- System and built-in functions for use in SQL statements and procedural logic
- System catalog views that are compatible with Oracle’s data dictionary
- Additional compatibility with Oracle MERGE.

Further reading

- Compatible SQL syntax, data types, and views in SQL reference.
- Compatibility offered by the procedures and functions that are part of the built-in packages in Built-in packages.
- Compatible tools and utilities (EDB*Plus, EDB*Loader, DRITA, and EDB*Wrap) that are included with an EDB Postgres Advanced Server installation in Tools, utilities, and components.
- For applications written using the Oracle Call Interface (OCI), EDB’s Open Client Library (OCL) provides interoperability with these applications. See EDB OCL Connector.

4.1.3 Terminology

The terminology that follows is important for understanding EDB Postgres Advanced Server's functionality and its components.

Database server (postmaster)

The server process that provides the key functionality that allows you to store and manage data. It manages the database files, accepts connections to the database from client applications, and performs database actions on behalf of the clients. The database server process is called postgres.

Database cluster

A set of on-disk structures that comprise a collection of databases. A cluster is serviced by a single-instance of the database server. A database cluster is stored in the data directory. Don’t store the data directory of a production database on an NFS file system.

Configuration files

You can use the parameters listed in Postgres configuration files to manage deployment preferences, security preferences, connection behaviors, and logging preferences.

4.2 SQL fundamentals

Structured Query Language (SQL) is a standard language for accessing and manipulating databases. These fundamentals provide a starting point for the most common actions for working with SQL, with pointers to more in-depth documentation. The tutorial uses a sample database and explains key concepts.

4.2.1 About the examples

The examples shown in the documentation use the PSQL program. PSQL is a terminal-based command line interface for Postgres. The prompt that normally appears when using PSQL is omitted in these examples to provide extra clarity for the point being shown.
Examples and output from examples are shown in a fixed-width, white font on a dark background.

Consider the following key points related to the examples:

- During installation of the EDB Postgres Advanced Server, you must make selections for configuration and defaults compatible with Oracle databases to reproduce the same results as the examples. You can verify a default compatible configuration by issuing the following commands in PSQL and returning the results shown:

  ```
  SHOW edb_redwood_date;
  edb_redwood_date
  1-
  on
  SHOW datestyle;
  DateStyle
  1-
  Redwood, DMY
  SHOW edb_redwood_strings;
  edb_redwood_strings
  1-
  on
  ```

- The examples use the sample tables, `dept`, `emp`, and `jobhist`, created and loaded when EDB Postgres Advanced Server was installed. The `emp` table is installed with triggers that you must disable to reproduce the same results as the examples. Log in to EDB Postgres Advanced Server as the `enterprisedb` superuser, and disable the triggers by issuing the following command:

  ```
  ALTER TABLE emp DISABLE TRIGGER USER;
  ```

You can later reactivate the triggers on the `emp` table with the following command:

  ```
  ALTER TABLE emp ENABLE TRIGGER USER;
  ```

4.2.2 Conventions used

The conventions used in the documentation described here apply to both Linux and Windows systems.

Directory paths are presented in the Linux format with forward slashes. When working on Windows systems, start the directory path with the drive letter followed by a colon, and substitute back slashes for forward slashes.

Some of this information might apply interchangeably to the PostgreSQL and EDB Postgres Advanced Server database systems. The term `EDB Postgres Advanced Server` is used to refer to EDB Postgres Advanced Server. The term `Postgres` is used to generically refer to both PostgreSQL and EDB Postgres Advanced Server. When a distinction must be made between these two database systems, the specific names—PostgreSQL or EDB Postgres Advanced Server—are used.

The installation directory path of the PostgreSQL or EDB Postgres Advanced Server products is referred to as `POSTGRES_INSTALL_HOME`.

- For EDB Postgres Advanced Server Linux installations performed using the interactive installer for version 10 and earlier, the default is `/opt/edb/asx.x`.
- For EDB Postgres Advanced Server Linux installations performed using an RPM package, the default is `/usr/edb/as<xx>` where `<xx>` is the EDB Postgres Advanced Server version number.
- For EDB Postgres Advanced Server Windows installations, the default is `C:\Program Files\edb\as<xx>`, The product version number is represented by `x.x` or by `xx` for version 10 and later.

4.2.3 SQL tutorial

EDB Postgres Advanced Server is a relational database management system (RDBMS), which means it’s a system for managing data stored in relations. A relation is essentially a mathematical term for a table. Each table is a named collection of rows. Each row of a given table has the same set of named columns, and each column is of a specific data type. Whereas columns have a fixed order in each row, it’s important to remember that SQL doesn’t guarantee the order of the rows in the table in any way (although you can explicitly sort them for display).

Tables are grouped into databases, and a collection of databases managed by a single EDB Postgres Advanced Server instance constitutes a database `cluster`.

This tutorial walks you through the main features of an RDBMS using a sample database.

4.2.3.1 Sample database
Throughout this tutorial, we work with a sample database to help explain database concepts, from basic to advanced.

### 4.2.3.1 Accessing the sample database

When EDB Postgres Advanced Server is installed, a sample database named `edb` is automatically created. This sample database contains the tables and programs used in this tutorial after you execute the script `edb-sample.sql`, located in the `/usr/edb/as15/share` directory.

This script does the following:

- Creates the sample tables and programs in the currently connected database
- Grants all permissions on the tables to the `PUBLIC` group

The tables and programs are created in the first schema of the search path in which the current user has permission to create tables and procedures. You can display the search path by issuing the command:

```sql
SHOW SEARCH_PATH;
```

You can alter the search path using commands in PSQL, a terminal-based front end to Postgres.

### 4.2.3.2 Overview of the sample database

The sample database represents employees in an organization. It contains three types of records: employees, departments, and historical records of employees.

Each employee has an identification number, name, hire date, salary, and manager. Some employees earn a commission in addition to their salary. All employee-related information is stored in the `emp` table.

The sample company is regionally diverse, so the database keeps track of the location of the departments. Each company employee is assigned to a department. Each department is identified by a unique department number and a short name. Each department is associated with one location. All department-related information is stored in the `dept` table.

The company also tracks information about jobs held by the employees. Some employees have been with the company for a long time and have held different positions, received raises, switched departments, and so on. When a change in employee status occurs, the company records the end date of the former position. A new job record is added with the start date and the new job title, department, salary, and the reason for the status change. All employee history is maintained in the `jobhist` table.

The following is an entity relationship diagram of the sample database tables.

![Sample Database Tables](image)

The `edb-sample.sql` script is:

```sql
-- -- Script that creates the 'sample' tables, views, procedures, functions, triggers, etc. -- -- Start new transaction - commit all or nothing -- BEGIN; --
```

Fig. 1: Sample Database Tables
-- Create and load tables used in the documentation examples.
--
-- Create the 'dept' table
--
CREATE TABLE dept 
(
  deptno NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
  loc VARCHAR2(13)
);
--
-- Create the 'emp' table
--
CREATE TABLE emp 
(
  empno NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename VARCHAR2(18),
  job VARCHAR2(9),
  mgr NUMBER(4),
  hiredate DATE,
  sal NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm NUMBER(7,2),
  deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk REFERENCES dept(deptno),
  startdate DATE NOT NULL,
  enddate DATE,
  job VARCHAR2(9),
  sal NUMBER(7,2),
  comm NUMBER(7,2),
  deptno NUMBER(2),
  chgdesc VARCHAR2(88),
  CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
  CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno) REFERENCES emp(empno) ON DELETE CASCADE,
  CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno) REFERENCES dept (deptno)
);
--
-- Create the 'jobhist' table
--
CREATE TABLE jobhist 
(
  empno NUMBER(4) NOT NULL,
  startdate DATE NOT NULL,
  enddate DATE,
  job VARCHAR2(9),
  sal NUMBER(7,2),
  comm NUMBER(7,2),
  deptno NUMBER(2),
  chgdesc VARCHAR2(88),
  CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
  CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno) REFERENCES emp(empno) ON DELETE CASCADE,
  CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno) REFERENCES dept (deptno)
);
--
-- Create the 'salesemp' view
--
CREATE OR REPLACE VIEW salesemp AS
  SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
--
-- Issue PUBLIC grants
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
GRANT ALL ON next_empno TO PUBLIC;

-- Load the 'dept' table

INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');

-- Load the 'emp' table

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NUL,20);  
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7688,'20-FEB-81',1600,300,30);  
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);  
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-81',2975,NUL,20);  
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-81',1250,1400,30);  
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-81',2850,NUL,30);  
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-81',2450,NUL,10);  
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-87',3000,NUL,20);  
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NUL,'17-NOV-81',5080,NUL,10);  
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,NUL,20);  
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1180,NUL,20);  
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NUL,30);  
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3080,NUL,20);  
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NUL,10);  

-- Load the 'jobhist' table

INSERT INTO jobhist VALUES (7369,'17-DEC-80','CLERK',800,NUL,20,'New Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81','SALESMAN',1600,380,38,'New Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81','SALESMAN',1250,580,38,'New Hire');
INSERT INTO jobhist VALUES (7566,'02-APR-81','MANAGER',2975,NUL,20,'New Hire');
INSERT INTO jobhist VALUES (7654,'28-SEP-81','SALESMAN',1250,1400,38,'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81','MANAGER',2850,NUL,30,'New Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81','MANAGER',2450,NUL,10,'New Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','CLERK',1000,NUL,28,'New Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','CLERK',1848,NUL,20,'Raise!');
INSERT INTO jobhist VALUES (7839,'05-MAY-90','ANALYST',3000,NUL,20,'Promoted to Analyst');
INSERT INTO jobhist VALUES (7839,'17-NOV-81','PRESIDENT',5080,NUL,10,'New Hire');
INSERT INTO jobhist VALUES (7844,'08-SEP-81','SALESMAN',1500,0,30,'New Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87','CLERK',1180,NUL,20,'New Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','CLERK',958,NUL,18,'New Hire');
INSERT INTO jobhist VALUES (7900,'15-JAN-83','CLERK',958,NUL,30,'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81','ANALYST',3080,NUL,20,'New Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82','CLERK',1380,NUL,10,'New Hire');

-- Populate statistics table and view
(pg_statistic/pg_stats)

ANALYZE dept;
ANALYZE emp;
-- Procedure that lists all employees' numbers and names from the 'emp' table using a cursor.

CREATE OR REPLACE PROCEDURE list_emp
IS
  v_empno NUMBER(4);
  v_ename VARCHAR2(18);
  CURSOR emp_cur IS
    SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
  OPEN emp_cur;
  DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
  DBMS_OUTPUT.PUT_LINE('----- -------');
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || v_ename);
  END LOOP;
  CLOSE emp_cur;
END;
/

-- Procedure that selects an employee row given the employee number and displays certain columns.

CREATE OR REPLACE PROCEDURE select_emp
(
  p_empno IN NUMBER
) IS
  v_ename emp.ename%TYPE;
  v_hiredate emp.hiredate%TYPE;
  v_sal emp.sal%TYPE;
  v_comm emp.comm%TYPE;
  v_dname dept.dname%TYPE;
  v_disp_date VARCHAR2(10);
BEGIN
  SELECT ename, hiredate, sal, NVL(comm, 0),
         dname INTO v_ename, v_hiredate, v_sal, v_comm,
         v_dname
    FROM emp e, dept d
  WHERE empno = p_empno
    AND e.deptno = d.deptno;
  v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
  DBMS_OUTPUT.PUT_LINE('Number : ' || p_empno);
  DBMS_OUTPUT.PUT_LINE('Name    : ' || v_ename);
  DBMS_OUTPUT.PUT_LINE('Hire Date: ' || v_disp_date);
  DBMS_OUTPUT.PUT_LINE('Salary   : ' || v_sal);
  DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
  DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found!');
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
CREATE OR REPLACE PROCEDURE emp_query
(
  p_deptno IN NUMBER,
  p_empno IN OUT NUMBER,
  p_ename IN OUT VARCHAR2,
  p_job OUT VARCHAR2,
  p_hiredate OUT DATE,
  p_sal OUT NUMBER
)
IS
BEGIN
  SELECT empno, ename, job, hiredate, sal
  INTO p_empno, p_ename, p_job, p_hiredate, p_sal
  FROM emp
  WHERE deptno = p_deptno
  AND (empno = p_empno
  OR ename = UPPER(p_ename));
END;
/
/
--  Procedure to call 'emp_query_caller' with IN and IN OUT parameters. Displays the results received from IN OUT and OUT parameters.
/
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
  v_deptno NUMBER(2);
  v_empno NUMBER(4);
  v_ename VARCHAR2(18);
  v_job VARCHAR2(9);
  v_hiredate DATE;
  v_sal NUMBER;
BEGIN
  v_deptno := 30;
  v_empno := 0;
  v_ename := 'Martin';
  emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
  DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
  DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
  DBMS_OUTPUT.PUT_LINE('Name : ' || v_ename);
  DBMS_OUTPUT.PUT_LINE('Job : ' || v_job);
  DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_hiredate);
  DBMS_OUTPUT.PUT_LINE('Salary : ' || v_sal);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
-- Function to compute yearly compensation based on semimonthly salary.

CREATE OR REPLACE FUNCTION emp_comp
(
    p_sal NUMBER,
    p_comm NUMBER
) RETURN NUMBER IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
/
-- Function that gets the next number from sequence, 'next_empno', and ensures it is not already in use as an employee number.

CREATE OR REPLACE FUNCTION new_empno RETURN NUMBER IS
    v_cnt INTEGER := 1;
    v_new_empno NUMBER;
BEGIN
    WHILE v_cnt > 0 LOOP
        SELECT next_empno.nextval INTO v_new_empno FROM dual;
        SELECT COUNT(*) INTO v_cnt FROM emp WHERE empno = v_new_empno;
        END LOOP;
        RETURN v_new_empno;
END;
/
-- EDB-SPL function that adds a new clerk to table 'emp'. This function uses package 'emp_admin'.

CREATE OR REPLACE FUNCTION hire_clerk
(
    p_ename VARCHAR2,
    p_deptno NUMBER
) RETURN NUMBER IS
    v_empno NUMBER(4);
    v_ename VARCHAR2(10);
    v_job VARCHAR2(9);
    v_mgr NUMBER(4);
    v_hiredate DATE;
    v_sal NUMBER(7,2);
    v_comm NUMBER(7,2);
    v_deptno NUMBER(2);
BEGIN
    v_empno := new_empno;
    INSERT INTO emp (empno, p_ename, 'CLERK', 7782,
    TRUNC(SYSDATE), 950.00, NULL, NULL,
    SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm,
    FROM emp WHERE empno = v_empno;
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Manager    : ' || v_mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
RETURN
v_empno;
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
DBMS_OUTPUT.PUT_LINE(SQLERRM);
DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
DBMS_OUTPUT.PUT_LINE(SQLCODE);
RETURN -1;
END;
/
-- PostgreSQL PL/pgSQL function that adds a new salesman to table 'emp'.
CREATE OR REPLACE FUNCTION hire_salesman (p_ename VARCHAR, p_sal NUMERIC, p_comm NUMERIC) RETURNS NUMERIC AS $$
DECLARE
  v_empno NUMERIC(4);
  v_ename VARCHAR(10);
  v_job VARCHAR(9);
  v_mgr NUMERIC(4);
  v_hiredate DATE;
  v_sal NUMERIC(7,2);
  v_comm NUMERIC(7,2);
  v_deptno NUMERIC(2);
BEGIN
  v_empno := new_empno();
  INSERT INTO emp VALUES (v_empno, p_ename, 'SALESMAN', CURRENT_DATE, p_sal, p_comm, 30);
  SELECT INTO v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno empno, ename, job, mgr, hiredate, sal, comm, deptno FROM emp WHERE empno = v_empno;
  RAISE INFO 'Department : %', v_deptno;
  RAISE INFO 'Employee No: %', v_empno;
  RAISE INFO 'Name : %', v_ename;
  RAISE INFO 'Job : %', v_job;
  RAISE INFO 'Manager : %', v_mgr;
  RAISE INFO 'Hire Date : %', v_hiredate;
  RAISE INFO 'Salary : %', v_sal;
  RAISE INFO 'Commission : %', v_comm;
  RETURN
v_empno;
EXCEPTION
WHEN OTHERS THEN
RAISE INFO 'The following is SQLERRM:';
RAISE INFO 'W', SQLERRM;
RAISE INFO 'The following is SQLSTATE:';
RAISE INFO 'W', SQLSTATE;
RETURN -1;
END
$$ LANGUAGE 'plpgsql';
/
--
-- Rule to INSERT into view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO salesemp
DO INSTEAD
  INSERT INTO emp VALUES (NEW.empno, NEW.ename, 'SALESMAN',
    7698,
    NEW.hiredate, NEW.sal, NEW.comm, 30);
--
-- Rule to UPDATE view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO salesemp
DO INSTEAD
  UPDATE emp SET 
    empno    = NEW.empno,
    ename    = NEW.ename,
    hiredate = NEW.hiredate,
    sal      = NEW.sal,
    comm     = NEW.comm
  WHERE empno = OLD.empno;
--
-- Rule to DELETE from view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO salesemp
DO INSTEAD
  DELETE FROM emp WHERE empno = OLD.empno;
--
-- After statement-level trigger that displays a message after
-- an insert, update, or deletion to the 'emp' table. One
-- message per SQL command is displayed.
--
CREATE OR REPLACE TRIGGER user_audit_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
  v_action VARCHAR2(24); BEGIN
    IF INSERTING THEN
      v_action := ' added employee(s) on
    END IF;
    DBMS_OUTPUT.PUT_LINE('User '||USER||' '||v_action
 || TO_CHAR(SYSDATE,'YYYY-MM-DD'));
  END;
/
--
-- Before row-level trigger that displays employee number
-- and salary of an employee that is about to be added, updated,
-- or deleted in the 'emp' table.
--
CREATE OR REPLACE TRIGGER emp_sal_trig
BEFORE DELETE OR INSERT OR UPDATE ON emp
  FOR EACH ROW
DECLARE
  sal_diff NUMBER;
BEGIN
  IF INSERTING THEN
    DBMS_OUTPUT.PUT_LINE('Inserting employee '||:NEW.empno||'
    DBMS_OUTPUT.PUT_LINE('.New salary: '||:NEW.sal||');
END IF;
IF UPDATING THEN
  sal_diff := :NEW.sal - :OLD.sal;
  DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
  DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
  DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
  DBMS_OUTPUT.PUT_LINE('..Raise : ' || sal_diff);
END IF;

IF DELETING THEN
  DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
  DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
END IF;
/
-- Package specification for the 'emp_admin' package.
CREATE OR REPLACE PACKAGE emp_admin IS
  FUNCTION get_dept_name (p_deptno NUMBER) RETURN VARCHAR2;
  FUNCTION update_emp_sal (p_empno NUMBER, p_raise NUMBER) RETURN NUMBER;
  PROCEDURE hire_emp (p_empno NUMBER, p_ename VARCHAR2, p_job VARCHAR2, p_sal NUMBER, p_hiredate DATE, p_comm NUMBER, p_mgr NUMBER, p_deptno NUMBER);
  PROCEDURE fire_emp (p_empno NUMBER);
END emp_admin;
/
-- Package body for the 'emp_admin' package.
CREATE OR REPLACE PACKAGE BODY emp_admin IS
  -- Function that queries the 'dept' table based on the department name and returns the corresponding department name.
  FUNCTION get_dept_name (p_deptno NUMBER) RETURN VARCHAR2 IS
    v_dname VARCHAR2(14);
    BEGIN
      SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
    RETURN v_dname;
    EXCEPTION
      ...
WHEN NO_DATA_FOUND THEN
  DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
  RETURN '';
END;
--
-- Function that updates an employee's salary based on
-- the
-- employee number and salary increment/decrement
-- passed
-- as IN parameters. Upon successful completion the
-- function
-- returns the new updated
-- salary.
--
FUNCTION update_emp_sal
(
  p_empno IN NUMBER,
  p_raise IN NUMBER
) RETURN NUMBER IS
  v_sal NUMBER := 0;
BEGIN
  SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
  v_sal := v_sal + p_raise;
  UPDATE emp SET sal = v_sal WHERE empno = p_empno;
  RETURN v_sal;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
    RETURN -1;
  WHEN OTHERS
    DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
    DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
    DBMS_OUTPUT.PUT_LINE(SQLCODE);
    RETURN -1;
END;
--
-- Procedure that inserts a new employee record into the 'emp'
-- table.
--
PROCEDURE hire_emp
(
  p_empno NUMBER,
  p_ename VARCHAR2,
  p_job VARCHAR2,
  p_sal NUMBER,
  p_hiredate DATE,
  p_comm NUMBER,
  p_mgr NUMBER,
  p_deptno
) AS
BEGIN
  INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
  VALUES(p_empno, p_ename, p_job, p_sal, p_hiredate, p_comm, p_mgr, p_deptno);
END;
--
-- Procedure that deletes an employee record from the 'emp' table
-- based
-- on the employee
-- number.
--
PROCEDURE fire_emp
(
  p_empno NUMBER
) AS
### 4.2.3.2 Creating a new table

Create a new table by specifying the table name, along with all column names and their types. The following is a simplified version of the `emp` sample table with the minimal information needed to define a table.

```sql
CREATE TABLE emp
(
    empno NUMBER(4),
    ename VARCHAR2(10),
    job VARCHAR2(9),
    mgr NUMBER(4),
    hiredate DATE,
    sal NUMBER(7,2),
    comm NUMBER(7,2),
    deptno NUMBER(2)
);
```

You can enter this code into PSQL with line breaks. PSQL recognizes that the command isn't terminated until the semicolon.

You can use white space (spaces, tabs, and newlines) in SQL commands. You can align the command as you want or even put all the text all on one line.

Two dashes (`--`) introduce comments. Whatever follows them is ignored up to the end of the line. In addition, SQL is not case sensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case.

`VARCHAR2(10)` specifies a data type that can store arbitrary character strings up to 10 characters in length. `NUMBER(7,2)` is a fixed-point number with precision 7 and scale 2. `NUMBER(4)` is an integer number with precision 4 and scale 0.

EDB Postgres Advanced Server supports the usual SQL data types: `INTEGER`, `SMALLINT`, `NUMBER`, `REAL`, `DOUBLE PRECISION`, `CHAR`, `VARCHAR2`, `DATE`, and `TIMESTAMP` as well as various synonyms for these types.

If you don't need a table or want to recreate it, you can remove it using the following command:

```sql
DROP TABLE tablename;
```

### 4.2.3.3 Populating a table With rows

Use the `INSERT` statement to populate a table with rows:

```sql
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
```

All data types use obvious input formats. Constants that aren't simple numeric values usually are surrounded by single quotes ('), as in the example. The `DATE` type is flexible in what it accepts. However, for this tutorial we use the unambiguous format shown here.

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```sql
INSERT INTO emp(empno,ename,job,mgr,hiredate,sal,comm,deptno)
VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
```

You can list the columns in a different order if you want or even omit some columns, for example, if the commission is unknown:

```sql
INSERT INTO emp(empno,ename,job,mgr,hiredate,sal,deptno)
VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,20);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.
4.2.3.4 Querying a table

To retrieve data from a table, you execute a query using an SQL `SELECT` statement. The statement is divided into:

- A select list (the part that lists the columns to return)
- A table list (the part that lists the tables from which to retrieve the data)
- An optional qualification (the part that specifies any restrictions)

The following query lists all columns of all employees in the table in no particular order.

```sql
SELECT * FROM emp;
```

Here, `*` in the select list means all columns. The following is the output from this query.

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80 00:00:00</td>
<td>808.00</td>
<td></td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-FEB-81 00:00:00</td>
<td>1680.00</td>
<td>308.00</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
<td>1259.00</td>
<td>508.00</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
<td>2975.00</td>
<td></td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
<td>1259.00</td>
<td>1408.00</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01-MAY-81 00:00:00</td>
<td>2850.00</td>
<td></td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
<td>2450.00</td>
<td></td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
<td>3080.00</td>
<td></td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>7839</td>
<td>17-NOV-81 00:00:00</td>
<td>5800.00</td>
<td></td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08-SEP-81 00:00:00</td>
<td>1580.00</td>
<td>0.80</td>
</tr>
<tr>
<td>7875</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7788</td>
<td>23-MAY-87 00:00:00</td>
<td>1180.00</td>
<td></td>
</tr>
<tr>
<td>7980</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
<td>958.00</td>
<td></td>
</tr>
<tr>
<td>7982</td>
<td>FORD</td>
<td>ANALYST</td>
<td>7566</td>
<td>03-DEC-81 00:00:00</td>
<td>3080.00</td>
<td></td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23-JAN-82 00:00:00</td>
<td>1380.00</td>
<td></td>
</tr>
</tbody>
</table>

(14 rows)

You can specify any arbitrary expression in the select list. For example, you can do:

```sql
SELECT ename, sal, sal + 24 AS yearly_salary, deptno FROM emp;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>yearly_salary</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>808.00</td>
<td>19200.00</td>
<td>20</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>38400.00</td>
<td>30</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>71400.00</td>
<td>20</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td>68400.00</td>
<td>30</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td>58800.00</td>
<td>10</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
<tr>
<td>KING</td>
<td>5080.00</td>
<td>120000.00</td>
<td>10</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>36000.00</td>
<td>30</td>
</tr>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td>26400.00</td>
<td>20</td>
</tr>
<tr>
<td>JAMES</td>
<td>958.00</td>
<td>22800.00</td>
<td>30</td>
</tr>
<tr>
<td>FORD</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
<tr>
<td>MILLER</td>
<td>1300.00</td>
<td>31200.00</td>
<td>10</td>
</tr>
</tbody>
</table>

Notice how the `AS` clause is used to relabel the output column. The `AS` clause is optional.

You can qualify a query by adding a `WHERE` clause that specifies the rows you want. The `WHERE` clause contains a Boolean (truth value) expression. Only rows for which the Boolean expression is true are returned. The usual Boolean operators (`AND`, `OR`, and `NOT`) are allowed in the qualification. For example, the following retrieves the employees in department 20 with salaries over $1000.00.

```sql
SELECT ename, sal, deptno FROM emp WHERE deptno = 20 AND sal > 1000;
```
### 4.2.3.5 Executing joins between tables

Queries can access multiple tables at once or access the same table in such a way that multiple rows of the table are processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a join query.

#### Executing a join query

For example, suppose you want to list all the employee records together with the name and location of the associated department. To do that, you need to compare the `deptno` column of each row of the `emp` table with the `deptno` column of all rows in the `dept` table. Then select the pairs of rows where these values match. You can accomplish this using the following query:

```sql
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc
FROM emp,
     dept
WHERE emp.deptno = dept.deptno;
```
Observe two things about the result set:

- There's no result row for department 40. That's because there's no matching entry in the `emp` table for department 40, so the join ignores the unmatched rows in the `dept` table. The code that follows shows how to fix this.
- It's more desirable to list the output columns qualified by table name rather than using * or leaving out the qualification as follows:

```sql
SELECT ename, sal, dept.deptno, dname, loc FROM emp, dept WHERE emp.deptno = dept.deptno;
```

Since all the columns had different names (except for `deptno`, which therefore must be qualified), the parser found the table they belong to. However, it's best practice to fully qualify column names in join queries.

You can also write this kind of join queries in this alternative form:

```sql
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM emp
INNER JOIN dept ON emp.deptno = dept.deptno;
```

This syntax is not as commonly used, but we show it here to help you understand the following topics.

### Executing an outer join

In all the above results for joins, no employees were returned that belonged to department 40. As a consequence, the record for department 40 never appears. Next, resolve how to get the department 40 record in the results despite the fact that there are no matching employees. The query must scan the `dept` table and, for each row, find the matching `emp` row. If no matching row is found, we want to substitute some "empty" values for the `emp` table's columns. This kind of query is called an outer join. (The joins you have seen so far are inner joins.) The command looks like this:

```sql
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM emp
LEFT OUTER JOIN dept ON emp.deptno = dept.deptno;
```

This query is called a left outer join. The table mentioned on the left of the join operator has each of its rows in the output at least once. The table on the right has only those rows output that match some row of the left table. When a left-table row is selected for which there is no right-table match, empty (NULL) values are substituted for the right-table columns.

An alternative syntax for an outer join is to use the outer join operator, "(+)", in the join condition in the WHERE clause. The outer join operator is placed after the column name of the table for which you substitute null values for unmatched rows. So for all the rows in the `dept` table that have no matching rows in the `emp` table, EDB Postgres Advanced Server returns null for any select list expressions containing columns of `emp`. Hence you can rewrite the earlier example as:

```sql
SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM dept,
emp
WHERE emp.deptno(+) = dept.deptno;
```
Executing a self join

You can also join a table against itself, which is called a self join. As an example, suppose you want to find the name of each employee and the name of that employee’s manager. You need to compare the `mgr` column of each `emp` row to the `empno` column of all other `emp` rows.

```sql
SELECT e1.ename || ' works for ' || e2.ename AS "Employees and their Managers" FROM emp e1, emp e2 WHERE e1.mgr = e2.empno;
```

<table>
<thead>
<tr>
<th>Employees and their Managers</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORD works for JONES</td>
</tr>
<tr>
<td>SCOTT works for JONES</td>
</tr>
<tr>
<td>WARD works for BLAKE</td>
</tr>
<tr>
<td>TURNER works for BLAKE</td>
</tr>
<tr>
<td>MARTIN works for BLAKE</td>
</tr>
<tr>
<td>JAMES works for BLAKE</td>
</tr>
<tr>
<td>ALLEN works for BLAKE</td>
</tr>
<tr>
<td>MILLER works for CLARK</td>
</tr>
<tr>
<td>ADAMS works for SCOTT</td>
</tr>
<tr>
<td>CLARK works for KING</td>
</tr>
<tr>
<td>BLAKE works for KING</td>
</tr>
<tr>
<td>JONES works for KING</td>
</tr>
<tr>
<td>SMITH works for FORD</td>
</tr>
</tbody>
</table>

(13 rows)

Here, the `emp` table was relabeled as `e1` to represent the employee row in the select list and in the join condition. It was also relabeled as `e2` to represent the matching employee row acting as manager in the select list and in the join condition. You can use these kinds of aliases in other queries to save some typing. For example:

```sql
SELECT e.ename, e.mgr, d.deptno, d.dname, d.loc FROM emp e, dept d WHERE e.deptno = d.deptno;
```

<table>
<thead>
<tr>
<th>name</th>
<th>mgr</th>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MILLER</td>
<td>7782</td>
<td>20</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>CLARK</td>
<td>7839</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>KING</td>
<td></td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>SCOTT</td>
<td>7566</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>JONES</td>
<td>7839</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>SMITH</td>
<td>7902</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>ADAMS</td>
<td>7788</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>FORD</td>
<td>7566</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>WARD</td>
<td>7698</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>TURNER</td>
<td>7698</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>ALLEN</td>
<td>7698</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>BLAKE</td>
<td>7839</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>MARTIN</td>
<td>7698</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>JAMES</td>
<td>7698</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>

(14 rows)

This style of abbreviating is used often.

4.2.3.6 Aggregating data

Like most other relational database products, EDB Postgres Advanced Server supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, some aggregates compute the COUNT, SUM, AVG (average), MAX (maximum), and MIN (minimum) over a set of rows.
As an example, you can find the highest and lowest salaries with the following query:

```
SELECT MAX(sal) highest_salary, MIN(sal) lowest_salary FROM emp;
```

```
+-----------------+---------------------+
<table>
<thead>
<tr>
<th>highest_salary</th>
<th>lowest_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000.00</td>
<td>800.00</td>
</tr>
</tbody>
</table>
+-----------------+---------------------+
(1 row)
```

If you want to find the employee with the largest salary, you might be tempted to try:

```
SELECT ename FROM emp WHERE sal =
MAX(sal);
```

```
ERROR:  aggregates not allowed in WHERE clause
```

This approach doesn't work because you can't use the aggregate function `MAX` in the `WHERE` clause. This restriction exists because the `WHERE` clause determines the rows that go into the aggregation stage. Hence, it has to be evaluated before aggregate functions are computed. However, you can restart the query to accomplish the intended result by using a subquery. The subquery is an independent computation that obtains its own result separately from the outer query.

```
SELECT ename FROM emp WHERE sal =
(SELECT MAX(sal) FROM emp);
```

```
+----------+
<table>
<thead>
<tr>
<th>ename</th>
</tr>
</thead>
<tbody>
<tr>
<td>KING</td>
</tr>
</tbody>
</table>
+----------+
(1 row)
```

Aggregates are also useful in combination with the `GROUP BY` clause. For example, the following query gets the highest salary in each department.

```
SELECT deptno, MAX(sal) FROM emp GROUP BY
deptno;
```

```
+--------+-------+
<table>
<thead>
<tr>
<th>deptno</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5000.00</td>
</tr>
<tr>
<td>20</td>
<td>3000.00</td>
</tr>
<tr>
<td>30</td>
<td>2850.00</td>
</tr>
</tbody>
</table>
+--------+-------+
(3 rows)
```

This query produces one output row per department. Each aggregate result is computed over the rows matching that department. These grouped rows can be filtered using the `HAVING` clause.

```
SELECT deptno, MAX(sal) FROM emp GROUP BY deptno HAVING AVG(sal) >
2000;
```

```
+--------+-------+
<table>
<thead>
<tr>
<th>deptno</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5000.00</td>
</tr>
<tr>
<td>20</td>
<td>3000.00</td>
</tr>
</tbody>
</table>
+--------+-------+
(3 rows)
```

This query gives the same results for only those departments that have an average salary greater than 2000.

Finally, the following query takes into account only the highest paid employees who are analysts in each department.

```
SELECT deptno, MAX(sal) FROM emp WHERE job = 'ANALYST' GROUP BY deptno HAVING AVG(sal) >
2000;
```

```
+--------+-------+
<table>
<thead>
<tr>
<th>deptno</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>3000.00</td>
</tr>
</tbody>
</table>
+--------+-------+
(1 row)
```

There's a subtle distinction between the `WHERE` and `HAVING` clauses. The `WHERE` clause filters out rows before grouping occurs and aggregate functions are applied. The `HAVING` clause applies filters on the results after rows are grouped and aggregate functions are computed for each group.

So, in the previous example, only employees who are analysts are considered. From this subset, the employees are grouped by department. Only those groups where the average salary of analysts in the group is greater than 2000 are in the final result. This is true only of the group for department 20, and the maximum analyst salary in department 20 is 3000.00.

### 4.2.3.7 Updating a table
You can change the column values of existing rows using the `UPDATE` command. For example, the following sequence of commands shows the before and after results of giving everyone who is a manager a 10% raise:

```sql
SELECT ename, sal FROM emp WHERE job = 'MANAGER';
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>JONES</td>
<td>2975.00</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
</tr>
</tbody>
</table>

(3 rows)

```sql
UPDATE emp SET sal = sal * 1.1 WHERE job = 'MANAGER';
```

```sql
SELECT ename, sal FROM emp WHERE job = 'MANAGER';
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>JONES</td>
<td>3272.50</td>
</tr>
<tr>
<td>BLAKE</td>
<td>3135.00</td>
</tr>
<tr>
<td>CLARK</td>
<td>2695.00</td>
</tr>
</tbody>
</table>

(3 rows)

4.2.3.8 Deleting a table

You can remove rows from a table using the `DELETE` command. For example, the following sequence of commands shows the before and after results of deleting all employees in department 20.

```sql
SELECT ename, deptno FROM emp;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>20</td>
</tr>
<tr>
<td>ALLEN</td>
<td>30</td>
</tr>
<tr>
<td>WARD</td>
<td>30</td>
</tr>
<tr>
<td>JONES</td>
<td>20</td>
</tr>
<tr>
<td>MARTIN</td>
<td>30</td>
</tr>
<tr>
<td>BLAKE</td>
<td>30</td>
</tr>
<tr>
<td>CLARK</td>
<td>10</td>
</tr>
<tr>
<td>SCOTT</td>
<td>20</td>
</tr>
<tr>
<td>KING</td>
<td>10</td>
</tr>
<tr>
<td>TURNER</td>
<td>30</td>
</tr>
<tr>
<td>ADAMS</td>
<td>20</td>
</tr>
<tr>
<td>JAMES</td>
<td>30</td>
</tr>
<tr>
<td>FORD</td>
<td>20</td>
</tr>
<tr>
<td>MILLER</td>
<td>10</td>
</tr>
</tbody>
</table>

(14 rows)

```sql
DELETE FROM emp WHERE deptno = 20;
```

```sql
SELECT ename, deptno FROM emp;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLEN</td>
<td>30</td>
</tr>
<tr>
<td>WARD</td>
<td>30</td>
</tr>
<tr>
<td>MARTIN</td>
<td>30</td>
</tr>
<tr>
<td>BLAKE</td>
<td>30</td>
</tr>
<tr>
<td>CLARK</td>
<td>10</td>
</tr>
<tr>
<td>KING</td>
<td>10</td>
</tr>
<tr>
<td>TURNER</td>
<td>30</td>
</tr>
<tr>
<td>JAMES</td>
<td>30</td>
</tr>
<tr>
<td>MILLER</td>
<td>10</td>
</tr>
</tbody>
</table>

(9 rows)

Warning

Be careful when giving a `DELETE` command without a `WHERE` clause such as the following:
This statement removes all rows from the given table, leaving it completely empty. The system doesn’t request confirmation before doing this.

4.2.3.9  The SQL language

EDB Postgres Advanced Server supports SQL language that’s compatible with Oracle syntax as well as syntax and commands for extended functionality. Extended functionality doesn’t provide database compatibility for Oracle or support Oracle-styled applications.

SQL reference provides detailed information about:

- Compatible SQL syntax and language elements
- Data types
- Functions and operators for the built-in data types

4.2.4  Advanced concepts

Advanced SQL features can simplify management and prevent loss or corruption of your data.

4.2.4.1  Views

A view provides a consistent interface that encapsulates details of the structure of your tables that might change as your application evolves. Making liberal use of views is a key aspect of good SQL database design.

Consider the following SELECT command:

```sql
SELECT ename, sal, sal * 24 AS yearly_salary, deptno FROM emp;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>yearly_salary</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td>19200.00</td>
<td>20</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>38400.00</td>
<td>30</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>71400.00</td>
<td>20</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td>68400.00</td>
<td>30</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td>58800.00</td>
<td>10</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
<tr>
<td>KING</td>
<td>5000.00</td>
<td>120000.00</td>
<td>10</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>36000.00</td>
<td>30</td>
</tr>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td>26400.00</td>
<td>20</td>
</tr>
<tr>
<td>JAMES</td>
<td>950.00</td>
<td>22800.00</td>
<td>30</td>
</tr>
<tr>
<td>FORD</td>
<td>3600.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
<tr>
<td>MILLER</td>
<td>1380.00</td>
<td>31200.00</td>
<td>10</td>
</tr>
</tbody>
</table>

(14 rows)

If this is a query that you use repeatedly, a shorthand method of reusing this query without retyping the entire SELECT command each time is to create a view:

```sql
CREATE VIEW employee_pay AS SELECT ename, sal, sal * 24 AS yearly_salary, deptno FROM emp;
```

The view name, `employee_pay`, can now be used like an ordinary table name to perform the query:

```sql
SELECT * FROM employee_pay;
```

<table>
<thead>
<tr>
<th>ename</th>
<th>sal</th>
<th>yearly_salary</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td>19200.00</td>
<td>20</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>38400.00</td>
<td>30</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>JONES</td>
<td>2975.00</td>
<td>71400.00</td>
<td>20</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>30000.00</td>
<td>30</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td>68400.00</td>
<td>30</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td>58800.00</td>
<td>10</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td>72000.00</td>
<td>20</td>
</tr>
</tbody>
</table>

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
You can use views in almost any place that you use a real table. Building views on other views is also common.

4.2.4.2 Foreign keys

A foreign key represents one or more than one column used to establish and enforce a link between data in two database tables for controlling data stored in the foreign key table.

Suppose you want to make sure all employees belong to a valid department, that is, you want to maintain the referential integrity of your data. In simplistic database systems, you can ensure referential integrity by first looking at the `dept` table to check if a matching record exists and then inserting or rejecting the new employee record.

This approach has a number of problems and is inconvenient. EDB Postgres Advanced Server can make it easier for you.

A modified version of the `emp` table presented in Creating a new table is shown here with the addition of a foreign key constraint. The modified `emp` table looks like the following:

```
CREATE TABLE emp
(
    empno NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename VARCHAR2(10),
    job VARCHAR2(9),
    mgr NUMBER(4),
    hiredate DATE,
    sal NUMBER(7,2),
    comm NUMBER(7,2),
    deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk REFERENCES dept(deptno)
);
```

If you try to issue the following `INSERT` command in the sample `emp` table, the foreign key constraint `emp_ref_dept_fk` is meant to ensure that department 50 exists in the `dept` table. Since it doesn't, the command is rejected.

```
INSERT INTO emp VALUES (8000, 'JONES', 'CLERK', 7902, '17-AUG-07', 1200, NULL, 50);
```

ERROR: insert or update on table "emp" violates foreign key constraint "emp_ref_dept_fk"
DETAIL: Key (deptno)=50 is not present in table "dept".

You can finely tune the behavior of foreign keys to your application. Making correct use of foreign keys improves the quality of your database applications. We strongly encourage you to learn more about them.

4.2.4.3 The ROWNUM pseudo-column

`ROWNUM` is a pseudo-column that’s assigned an incremental, unique integer value for each row based on the order the rows were retrieved from a query. Therefore, the first row retrieved has `ROWNUM` of 1, the second row has `ROWNUM` of 2, and so on.

Limiting the number or rows returned

You can use this feature to limit the number of rows retrieved by a query:

```
SELECT empno, ename, job FROM emp WHERE ROWNUM < 5;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
</tr>
</tbody>
</table>
The `ROWNUM` value is assigned to each row before any sorting of the result set takes place. Thus, the result set is returned in the order given by the `ORDER BY` clause. However, the `ROWNUM` values might not be in ascending order:

```
SELECT ROWNUM, empno, ename, job FROM emp WHERE ROWNUM < 5 ORDER BY ename;
```

<table>
<thead>
<tr>
<th>rownum</th>
<th>empno</th>
<th>ename</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>4</td>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
</tr>
<tr>
<td>1</td>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
</tr>
<tr>
<td>3</td>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
</tr>
</tbody>
</table>

(4 rows)

Adding a sequence number to rows in a table

The following example shows how you can add a sequence number to every row in the `jobhist` table. First add a column named `seqno` to the table. Then set `seqno` to `ROWNUM` in the `UPDATE` command.

```
ALTER TABLE jobhist ADD seqno NUMBER(3);
UPDATE jobhist SET seqno = ROWNUM;
```

The following `SELECT` command shows the new `seqno` values:

```
SELECT seqno, empno, TO_CHAR(startdate, 'DD-MON-YY') AS start, job
FROM jobhist;
```

<table>
<thead>
<tr>
<th>seqno</th>
<th>empno</th>
<th>start</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7369</td>
<td>17-DEC-80</td>
<td>CLERK</td>
</tr>
<tr>
<td>2</td>
<td>7499</td>
<td>20-FEB-81</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>3</td>
<td>7521</td>
<td>22-FEB-81</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>4</td>
<td>7566</td>
<td>02-APR-81</td>
<td>MANAGER</td>
</tr>
<tr>
<td>5</td>
<td>7654</td>
<td>28-SEP-81</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>6</td>
<td>7698</td>
<td>01-MAY-81</td>
<td>MANAGER</td>
</tr>
<tr>
<td>7</td>
<td>7782</td>
<td>09-JUN-81</td>
<td>MANAGER</td>
</tr>
<tr>
<td>8</td>
<td>7788</td>
<td>19-APR-87</td>
<td>CLERK</td>
</tr>
<tr>
<td>9</td>
<td>7788</td>
<td>13-APR-88</td>
<td>CLERK</td>
</tr>
<tr>
<td>10</td>
<td>7788</td>
<td>05-MAY-90</td>
<td>ANALYST</td>
</tr>
<tr>
<td>11</td>
<td>7839</td>
<td>17-NOV-81</td>
<td>PRESIDENT</td>
</tr>
<tr>
<td>12</td>
<td>7844</td>
<td>08-SEP-81</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>13</td>
<td>7876</td>
<td>23-MAY-87</td>
<td>CLERK</td>
</tr>
<tr>
<td>14</td>
<td>7900</td>
<td>03-DEC-81</td>
<td>CLERK</td>
</tr>
<tr>
<td>15</td>
<td>7900</td>
<td>15-JAN-83</td>
<td>CLERK</td>
</tr>
<tr>
<td>16</td>
<td>7902</td>
<td>03-DEC-81</td>
<td>ANALYST</td>
</tr>
<tr>
<td>17</td>
<td>7934</td>
<td>23-JAN-82</td>
<td>CLERK</td>
</tr>
</tbody>
</table>

(17 rows)

4.2.4.4 Synonyms

A synonym is an identifier that you can use to reference another database object in a SQL statement. A synonym is useful in cases where a database object normally requires full qualification by schema name to be properly referenced in a SQL statement. A synonym defined for that object simplifies the reference to a single, unqualified name.

EDB Postgres Advanced Server supports synonyms for:

- Tables
- Views
- Materialized views
- Sequences
- Packages
- Procedures
- Functions
- Types
- Objects that you can access through a database link
- Other synonyms
Neither the referenced schema or referenced object must exist when you create the synonym. A synonym can refer to a nonexistent object or schema. A synonym becomes invalid if you drop the referenced object or schema. You must explicitly drop a synonym to remove it.

As with any other schema object, EDB Postgres Advanced Server uses the search path to resolve unqualified synonym names. If you have two synonyms with the same name, an unqualified reference to a synonym resolves to the first synonym with the given name in the search path. If public is in your search path, you can refer to a synonym in that schema without qualifying that name.

When EDB Postgres Advanced Server executes an SQL command, the privileges of the current user are checked against the synonym's underlying database object. If the user doesn't have the proper permissions for that object, the SQL command fails.

Creating a synonym

Use the CREATE SYNONYM command to create a synonym. The syntax is:

```
CREATE [ OR REPLACE ] [ PUBLIC ] SYNONYM
  [ <schema> ] <syn_name>
  FOR <object_schema>.<object_name>[ @<dblink_name> ];
```

Parameters

- **syn_name**: The name of the synonym. A synonym name must be unique in a schema.
- **schema**: Specifies the name of the schema that the synonym resides in. If you don't specify a schema name, the synonym is created in the first existing schema in your search path.
- **object_name**: Specifies the name of the object.
- **object_schema**: Specifies the name of the schema that the object resides in.
- **dblink_name**: Specifies the name of the database link through which you can access a target object.

Include the REPLACE clause to replace an existing synonym definition with a new synonym definition.

Include the PUBLIC clause to create the synonym in the public schema. Compatible with Oracle databases, the CREATE PUBLIC SYNONYM command creates a synonym that resides in the public schema:

```
CREATE [ OR REPLACE ] PUBLIC SYNONYM <syn_name>
  FOR <object_schema>.<object_name> ;
```

This is a shorthand way to write:

```
CREATE [ OR REPLACE ] SYNONYM public.<syn_name>
  FOR <object_schema>.<object_name> ;
```

This example creates a synonym named personnel that refers to the enterprisedb.emp table:

```
CREATE SYNONYM personnel FOR enterprisedb.emp;
```

Unless the synonym is schema qualified in the CREATE SYNONYM command, it's created in the first existing schema in your search path. You can view your search path by executing the following command:

```
SHOW SEARCH_PATH;
```

```
search_path
-----------------------
development,accounting
(1 row)
```
Example

In the example, if a schema named `development` doesn't exist, the synonym are created in the schema named `accounting`.

Now you can reference the `emp` table in the `enterprisedb` schema in any SQL statement (DDL or DML) by using the synonym `personnel`:

```
INSERT INTO personnel VALUES (8142,'ANDERSON','CLERK',7902,'17-DEC-06',1300,NULL,20);
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80 00:00:00</td>
<td>800.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81 00:00:00</td>
<td>1600.00</td>
<td>300.00</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
<td>1250.00</td>
<td>500.00</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
<td>2975.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>21-MAY-81 00:00:00</td>
<td>2850.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
<td>2450.00</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
<td>3800.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td></td>
<td>17-NOV-81 00:00:00</td>
<td>5800.00</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08-SEP-81 00:00:00</td>
<td>1500.00</td>
<td>0.00</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7788</td>
<td>23-MAY-87 00:00:00</td>
<td>1100.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
<td>950.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>7566</td>
<td>03-DEC-81 00:00:00</td>
<td>3000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23-JAN-82 00:00:00</td>
<td>1300.00</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>8142</td>
<td>ANDERSON</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-06 00:00:00</td>
<td>1300.00</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

(15 rows)

Deleting a synonym

To delete a synonym, use the command `DROP SYNONYM`. The syntax is:

```
DROP [PUBLIC] SYNONYM [<schema>.]<syn_name>
```

**Parameters**

- `<syn_name>`
  - `syn_name` is the name of the synonym. A synonym name must be unique in a schema.
  - `<schema>` specifies the name of the schema in which the synonym resides.

Like any other object that can be schema qualified, you can have two synonyms with the same name in your search path. To disambiguate the name of the synonym that you're dropping, include a schema name. Unless a synonym is schema qualified in the `DROP SYNONYM` command, EDB Postgres Advanced Server deletes the first instance of the synonym it finds in your search path.

You can optionally include the `PUBLIC` clause to drop a synonym that resides in the `public` schema. Compatible with Oracle databases, the `DROP PUBLIC SYNONYM` command drops a synonym that resides in the `public` schema:

```
DROP PUBLIC SYNONYM <syn_name>;
```

**Example**

The following example drops the synonym `personnel`:

```
DROP SYNONYM personnel;
```

### 4.2.4.5 Hierarchical queries

A hierarchical query is a type of query that returns the rows of the result set in a hierarchical order based on data forming a parent-child relationship. A hierarchy is typically represented by an inverted...
tree structure. The tree is made up of interconnected nodes. Each node can be connected to none, one, or multiple child nodes. Each node is connected to one parent node except for the top node, which has no parent. This node is the root node. Each tree has exactly one root node.

Nodes that don’t have any children are called leaf nodes. A tree always has at least one leaf node, for example, the trivial case where the tree is made up of a single node. In this case, the node is both the root and the leaf.

In a hierarchical query, the rows of the result set represent the nodes of one or more trees.

Note
A single given row can appear in more than one tree and thus appear more than once in the result set.

Syntax
The hierarchical relationship in a query is described by the CONNECT BY clause, which forms the basis of the order in which rows are returned in the result set. This example shows the context of where the CONNECT BY clause and its associated optional clauses appear in the SELECT command:

```
SELECT <select_list> FROM <table_expression> [ WHERE ...
] [ START WITH <start_expression>
] { CONNECT BY <condition>
} { ORDER SIBLINGS BY <column> [ ASC | DESC
] [, <column2> [ ASC | DESC ] }
... [ GROUP BY ....]
[ HAVING ...
][ <other> ...
]
```

- `<select_list>` is one or more expressions that make up the fields of the result set.
- `<table_expression>` is one or more tables or views from which the rows of the result set originate. `<other>` is any additional legal SELECT command clauses.

**CONNECT BY** specifies the relationship between a parent record and its child records of the hierarchy.

- You can specify the **PRIOR** keyword with only one expression in a condition in the **CONNECT BY** clause. You must specify it with either right or left expression to refer to a parent row in a hierarchical query. For example:

  ```sql
  PRIOR expr = expr
  or
  expr = PRIOR expr
  ```

- If the **CONNECT BY condition** is compound, only one condition requires the **PRIOR** keyword, although there can be multiple **PRIOR** conditions.

- You can use the **PRIOR** keyword to evaluate the parent row in a parent-child relationship in a hierarchical query.

- You can’t use **PRIOR** keyword with unary expressions.

- You can use the **PRIOR** keyword to add a pseudo-column for each record corresponding to an expression in a target list. The pseudo-column represents a parent record’s actual value when referenced in a child query using a **PRIOR** expression.

Use the **CONNECT_BY_ROOT** operator to further enhance the functionality of the **CONNECT BY [PRIOR]** condition. This operator returns not only immediate parent rows but also all ancestors rows in the hierarchy. For more information, see **Retrieving the root node with CONNECT_BY_ROOT**.

Examples
This example uses the **CONNECT BY** clause with **PRIOR** to define a relationship between employees and managers. Using the **LEVEL** value, the employee names are indented to further emphasize the depth in the hierarchy of each row.

```sql
SELECT LPAD(ename,length(ename))*2 +((LEVEL - 1),', ' ) AS employee, empno, mgr, LEVEL
FROM emp
CONNECT BY PRIOR empno = mgr AND PRIOR emp.deptno = emp.deptno;
```

The output from this query is:

```
__OUTPUT__
```
This example adds the `ORDER BY` clause and a compound condition with `AND` operator in a `CONNECT BY` clause to show employee and manager hierarchy:

```sql
SELECT LPAD(ename, length(ename) + 2 * (LEVEL - 1), ' ') AS employee, empno, mgr, LEVEL FROM emp
CONNECT BY PRIOR empno = mgr AND LEVEL <= 3
ORDER BY LEVEL, ename;
```

The output from this query is:

<table>
<thead>
<tr>
<th>employee</th>
<th>empno</th>
<th>mgr</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMS</td>
<td>7876</td>
<td>7788</td>
<td>1</td>
</tr>
<tr>
<td>ALLEN</td>
<td>7499</td>
<td>7698</td>
<td>1</td>
</tr>
<tr>
<td>BLAKE</td>
<td>7698</td>
<td>7839</td>
<td>1</td>
</tr>
<tr>
<td>CLARK</td>
<td>7782</td>
<td>7839</td>
<td>1</td>
</tr>
<tr>
<td>MILLER</td>
<td>7934</td>
<td>7782</td>
<td>1</td>
</tr>
<tr>
<td>SCOTT</td>
<td>7788</td>
<td>7566</td>
<td>2</td>
</tr>
<tr>
<td>ADAMS</td>
<td>7876</td>
<td>7788</td>
<td>2</td>
</tr>
<tr>
<td>MARTIN</td>
<td>7654</td>
<td>7698</td>
<td>2</td>
</tr>
<tr>
<td>JAMES</td>
<td>7900</td>
<td>7698</td>
<td>2</td>
</tr>
<tr>
<td>ALLEN</td>
<td>7499</td>
<td>7698</td>
<td>2</td>
</tr>
<tr>
<td>CLARK</td>
<td>7782</td>
<td>7839</td>
<td>2</td>
</tr>
<tr>
<td>MILLER</td>
<td>7934</td>
<td>7782</td>
<td>3</td>
</tr>
<tr>
<td>TURNER</td>
<td>7844</td>
<td>7698</td>
<td>3</td>
</tr>
<tr>
<td>ADAMS</td>
<td>7876</td>
<td>7788</td>
<td>3</td>
</tr>
<tr>
<td>JAMES</td>
<td>7900</td>
<td>7698</td>
<td>3</td>
</tr>
<tr>
<td>FORD</td>
<td>7902</td>
<td>7566</td>
<td>3</td>
</tr>
<tr>
<td>SMITH</td>
<td>7369</td>
<td>7902</td>
<td>3</td>
</tr>
<tr>
<td>MILLER</td>
<td>7934</td>
<td>7782</td>
<td>4</td>
</tr>
</tbody>
</table>
```

This example adds the `ORDER BY` clause and a compound condition with `AND` operator in a `CONNECT BY` clause to show employee and manager hierarchy:
This example uses the PRIOR keyword to return the employee name and manager name for each employee:

```
SELECT ename, PRIOR ename AS mgr
FROM emp
START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
ORDER BY ename;
```

The output from this query is:

```
__OUTPUT__
ename | mgr
-------+-------
ADAMS  | SCOTT
ALLEN  | BLAKE
BLAKE  | KING
CLARK  | FORD
          | JONES
          | JONES
          | MILLER
          | MILLER
          | SCOTT
          | SMITH
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
          | JAMES
          | MARTIN
          | MILLER
          | MILLER
          | TURNER
          | WARD
          | ADAMS
          | ALLEN
          | FORD
0 rows
```
This example uses a CONNECT BY clause with an optional PRIOR statement to show the level value:

```
SELECT LEVEL
FROM DUAL
CONNECT BY LEVEL <= 5;
```

The output from this query is:

```
 level
-------
 1
 2
 3
 4
 5
```

4.2.4.5.1 Defining the parent/child relationship

To determine the children for any given row:

1. Evaluate `parent_expr` on the given row.

2. Evaluate `child_expr` on any other row resulting from the evaluation of `table_expression`.

3. If `parent_expr = child_expr`, then this row is a child node of the given parent row.

4. Repeat the process for all remaining rows in `table_expression`. All rows that satisfy the equation in step 3 are the children nodes of the given parent row.

**Note**

The evaluation process to determine if a row is a child node occurs on every row returned by `table_expression` before the WHERE clause is applied to `table_expression`.

Iteratively repeating this process, treating each child node found in the prior steps as a parent, constructs an inverted tree of nodes. The process is complete when the final set of child nodes has no children of its own. These are the leaf nodes.

A SELECT command that includes a CONNECT BY clause typically includes the START WITH clause. The START WITH clause determines the rows that are to be the root nodes, that is, the rows that are the initial parent nodes upon which to apply the algorithm.

4.2.4.5.2 Selecting the root nodes

The START WITH clause determines the rows selected by `table_expression` to use as the root nodes. All rows selected by `table_expression` where `start_expression` evaluates to `true` become a root node of a tree. Thus, the number of potential trees in the result set is equal to the number of root nodes. As a consequence, if the START WITH clause is omitted, then every row returned by `table_expression` is a root of its own tree.

4.2.4.5.3 Organization tree in the sample application

To see an example of an organizational tree, consider the `emp` table of the sample application. The rows of the `emp` table form a hierarchy based on the `mgr` column, which contains the employee number of the employee's manager. Each employee has at most one manager. KING is the president of the company. He has no manager, therefore KING's `mgr` column is null. Also, it's possible for an employee to act as a manager for more than one employee. This relationship forms a typical tree-structured hierarchical organization chart:
To form a hierarchical query based on this relationship, the `SELECT` command includes the clause `CONNECT BY PRIOR empno = mgr`. For example, given the company president, KING, with employee number 7839, any employee whose `mgr` column is 7839 reports directly to KING. This relationship is true for JONES, BLAKE, and CLARK, the child nodes of KING. Similarly, for employee JONES, any other employee with `mgr` column equal to 7566 is a child node of JONES. These nodes are SCOTT and FORD in the example.

The top of the organization chart is KING, so there is one root node in this tree. The `START WITH mgr IS NULL` clause selects only KING as the initial root node.

The complete `SELECT` command is:

```sql
SELECT ename, empno, mgr
FROM emp
START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;
```

The rows in the query output traverse each branch from the root to leaf moving in a top-to-bottom, left-to-right order:

<table>
<thead>
<tr>
<th>ename</th>
<th>empno</th>
<th>mgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>KING</td>
<td>7839</td>
<td></td>
</tr>
<tr>
<td>JONES</td>
<td>7566</td>
<td>7839</td>
</tr>
<tr>
<td>SCOTT</td>
<td>7788</td>
<td>7566</td>
</tr>
<tr>
<td>ADAMS</td>
<td>7876</td>
<td>7788</td>
</tr>
<tr>
<td>FORD</td>
<td>7902</td>
<td>7566</td>
</tr>
<tr>
<td>SMITH</td>
<td>7369</td>
<td>7902</td>
</tr>
<tr>
<td>BLAKE</td>
<td>7698</td>
<td>7369</td>
</tr>
<tr>
<td>ALLEN</td>
<td>7499</td>
<td>7698</td>
</tr>
<tr>
<td>WARD</td>
<td>7521</td>
<td>7499</td>
</tr>
<tr>
<td>MARTIN</td>
<td>7654</td>
<td>7698</td>
</tr>
<tr>
<td>TURNER</td>
<td>7844</td>
<td>7654</td>
</tr>
<tr>
<td>JAMES</td>
<td>7900</td>
<td>7698</td>
</tr>
<tr>
<td>CLARK</td>
<td>7782</td>
<td>7654</td>
</tr>
<tr>
<td>MILLER</td>
<td>7934</td>
<td>7782</td>
</tr>
</tbody>
</table>

(14 rows)

### 4.2.4.5.4 Designating the node level

`LEVEL` is a pseudo-column that you can use wherever a column can appear in the `SELECT` command. For each row in the result set, `LEVEL` returns a non-zero integer value designating the depth in the hierarchy of the node represented by this row. The `LEVEL` for root nodes is 1. The `LEVEL` for direct children of root nodes is 2, and so on.

Example
The following query is a modification of the previous query with the addition of the `LEVEL` pseudo-column. In addition, using the `LEVEL` value, the employee names are indented to further emphasize the depth in the hierarchy of each row.

```sql
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr
FROM emp START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr;
```

The output from this query is:

<table>
<thead>
<tr>
<th>level</th>
<th>employee</th>
<th>empno</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KING</td>
<td>7839</td>
</tr>
<tr>
<td>2</td>
<td>JONES</td>
<td>7566</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>7786</td>
</tr>
<tr>
<td>3</td>
<td>ADAMS</td>
<td>7867</td>
</tr>
<tr>
<td>3</td>
<td>FORD</td>
<td>7902</td>
</tr>
<tr>
<td>3</td>
<td>SMITH</td>
<td>7369</td>
</tr>
<tr>
<td>3</td>
<td>BLAKE</td>
<td>7698</td>
</tr>
<tr>
<td>3</td>
<td>ALLEN</td>
<td>7499</td>
</tr>
<tr>
<td>3</td>
<td>WARD</td>
<td>7521</td>
</tr>
<tr>
<td>3</td>
<td>MARTIN</td>
<td>7654</td>
</tr>
<tr>
<td>3</td>
<td>TURNER</td>
<td>7844</td>
</tr>
<tr>
<td>3</td>
<td>JAMES</td>
<td>7900</td>
</tr>
<tr>
<td>2</td>
<td>CLARK</td>
<td>7782</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>7934</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(14 rows)</td>
<td></td>
</tr>
</tbody>
</table>

Nodes that share a common parent and are at the same level are called siblings. For example, in the above output, employees ALLEN, WARD, MARTIN, TURNER, and JAMES are siblings since they are all at level three with parent BLAKE. JONES, BLAKE, and CLARK are siblings since they are at level two and KING is their common parent.

### 4.2.4.5.5 Ordering the siblings

You can order the result set so the siblings appear in ascending or descending order by selected column values using the `ORDER SIBLINGS BY` clause. This is a special case of the `ORDER BY` clause that you can use only with hierarchical queries.

You can further modify the previous query by adding `ORDER SIBLINGS BY ename ASC`.

```sql
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr
FROM emp START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The output from the prior query is now modified so the siblings appear in ascending order by name. Siblings BLAKE, CLARK, and JONES are now alphabetically arranged under KING. Siblings ALLEN, JAMES, MARTIN, TURNER, and WARD are alphabetically arranged under BLAKE, and so on.

<table>
<thead>
<tr>
<th>level</th>
<th>employee</th>
<th>empno</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KING</td>
<td>7839</td>
</tr>
<tr>
<td>2</td>
<td>BLAKE</td>
<td>7698</td>
</tr>
<tr>
<td>3</td>
<td>ALLEN</td>
<td>7499</td>
</tr>
<tr>
<td>3</td>
<td>JAMES</td>
<td>7900</td>
</tr>
<tr>
<td>3</td>
<td>MARTIN</td>
<td>7654</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(14 rows)</td>
<td></td>
</tr>
</tbody>
</table>
This final example adds the `WHERE` clause and starts with three root nodes. After the node tree is constructed, the `WHERE` clause filters out rows in the tree to form the result set.

The output from the query shows three root nodes (level one): **BLAKE**, **CLARK**, and **JONES**. In addition, rows that don't satisfy the `WHERE` clause were eliminated from the output.

```
SELECT LEVEL, LPAD (', ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr
FROM emp WHERE mgr IN (7839, 7782, 7902, 7788)
START WITH ename IN ('BLAKE', 'CLARK', 'JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename
ASC;
```

The output from the query shows three root nodes (level one): **BLAKE**, **CLARK**, and **JONES**. In addition, rows that don't satisfy the `WHERE` clause were eliminated from the output.

```
<table>
<thead>
<tr>
<th>level</th>
<th>employee</th>
<th>empno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BLAKE</td>
<td>7698</td>
</tr>
<tr>
<td>1</td>
<td>CLARK</td>
<td>7782</td>
</tr>
<tr>
<td>2</td>
<td>MILLER</td>
<td>7934</td>
</tr>
<tr>
<td>1</td>
<td>JONES</td>
<td>7566</td>
</tr>
<tr>
<td>3</td>
<td>SMITH</td>
<td>7369</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>7788</td>
</tr>
<tr>
<td>4</td>
<td>ADAMS</td>
<td>7876</td>
</tr>
</tbody>
</table>
```

(14 rows)

### 4.2.4.5.6 Retrieving the root node with CONNECT_BY_ROOT

**CONNECT_BY_ROOT** is a unary operator that you can use to qualify a column to return the column's value of the row considered to be the root node in relation to the current row.

#### Note

A unary operator operates on a single operand. In the case of **CONNECT_BY_ROOT**, it's the column name following the **CONNECT_BY_ROOT** keyword.

#### Syntax

In the `SELECT` list, the **CONNECT_BY_ROOT** operator is shown by the following:

```
SELECT [... ,] CONNECT_BY_ROOT <column> [,...]
FROM <table_expression> ...;
```

Some points to note about the **CONNECT_BY_ROOT** operator:

- You can use the **CONNECT_BY_ROOT** operator in the `SELECT` list, `WHERE` clause, `GROUP BY` clause, `HAVING` clause, `ORDER BY` clause, and `ORDER SIBLINGS BY` clause as long as the `SELECT` command is for a hierarchical query.
- You can't use the **CONNECT_BY_ROOT** operator in the `CONNECT BY` clause or the `START WITH` clause of the hierarchical query.
- You can apply **CONNECT_BY_ROOT** to an expression involving a column. To do so, you must enclose the expression in parentheses.
Examples

The following query shows the use of the `CONNECT_BY_ROOT` operator to return the employee number and employee name of the root node for each employee listed in the result set based on trees starting with employees `BLAKE`, `CLARK`, and `JONES`.

```sql
SELECT
    LEVEL,
    LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno,
    CONNECT_BY_ROOT empno "mgr empno",
    CONNECT_BY_ROOT ename "mgr ename"
FROM
emp
START WITH ename IN ('BLAKE', 'CLARK', 'JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename
ASC;
```

The output from the query shows that all of the root nodes in columns `mgr empno` and `mgr ename` are one of the employees— `BLAKE`, `CLARK`, or `JONES`—listed in the `START WITH` clause.

<table>
<thead>
<tr>
<th>level</th>
<th>employee</th>
<th>empno</th>
<th>mgr</th>
<th>mgr empno</th>
<th>mgr ename</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BLAKE</td>
<td>7698</td>
<td>7839</td>
<td>7698</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ALLEN</td>
<td>7499</td>
<td>7698</td>
<td>7698</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>JAMES</td>
<td>7900</td>
<td>7698</td>
<td>7698</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MARTIN</td>
<td>7654</td>
<td>7698</td>
<td>7698</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TURNER</td>
<td>7844</td>
<td>7698</td>
<td>7698</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>WARD</td>
<td>7521</td>
<td>7698</td>
<td>7698</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>CLARK</td>
<td>7782</td>
<td>7839</td>
<td>7782</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MILLER</td>
<td>7934</td>
<td>7782</td>
<td>7782</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>JONES</td>
<td>7566</td>
<td>7839</td>
<td>7566</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FORD</td>
<td>7982</td>
<td>7566</td>
<td>7566</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SMITH</td>
<td>7369</td>
<td>7982</td>
<td>7566</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SCOTT</td>
<td>7788</td>
<td>7566</td>
<td>7566</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ADAMS</td>
<td>7876</td>
<td>7788</td>
<td>7566</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A similar query but producing only one tree starting with the single, top-level employee where the `mgr` column is null:

```sql
SELECT
    LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno,
    CONNECT_BY_ROOT empno "mgr empno",
    CONNECT_BY_ROOT ename "mgr ename"
FROM
emp
START WITH mgr IS NULL
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename
ASC;
```

In the following output, all of the root nodes in columns `mgr empno` and `mgr ename` indicate `KING` as the root for this query.

<table>
<thead>
<tr>
<th>level</th>
<th>employee</th>
<th>empno</th>
<th>mgr</th>
<th>mgr empno</th>
<th>mgr ename</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KING</td>
<td>7839</td>
<td></td>
<td>7839</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BLAKE</td>
<td>7698</td>
<td>7839</td>
<td>7839</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ALLEN</td>
<td>7499</td>
<td>7698</td>
<td>7839</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>JAMES</td>
<td>7900</td>
<td>7698</td>
<td>7839</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>MARTIN</td>
<td>7654</td>
<td>7698</td>
<td>7839</td>
<td></td>
</tr>
</tbody>
</table>

(13 rows)
By contrast, this example omits the `START WITH` clause, thereby resulting in 14 trees:

```sql
SELECT LEVEL,
       LPAD(' ', 2 * (LEVEL - 1)) || ename "employee", empno,
       CONNECT_BY_ROOT empno "mgr
       ename"
FROM emp
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename
ASC;
```

In the output from the query, each node appears at least once as a root node under the `mgr empno` and `mgr ename` columns, since even the leaf nodes form the top of their own trees.
When applied to an expression that isn’t enclosed in parentheses, the CONNECT_BY_ROOT operator affects only the term ename immediately following it. The subsequent concatenation of || ' manages ' || ename isn’t part of the CONNECT_BY_ROOT operation. Hence the second occurrence of ename results in the value of the currently processed row, while the first occurrence of ename results in the value from the root node.

```
SELECT LEVEL,
 LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno,
mgr,
CONNECT_BY_ROOT ename || ' manages ' || ename "top
mgr/employee"
FROM emp
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename
ASC;
```

Note the values produced under the top mgr/employee column in the output from the query:

<table>
<thead>
<tr>
<th><em>OUTPUT</em></th>
</tr>
</thead>
</table>
| level | employee | empno | mgr | top
|-------------------------------|
| 1 | BLAKE | 7698 | 7839 | BLAKE manages
| 2 | ALLEN | 7499 | 7698 | BLAKE manages
| 2 | JAMES | 7900 | 7698 | BLAKE manages
| 2 | MARTIN | 7654 | 7698 | BLAKE manages
| 2 | TURNER | 7844 | 7698 | BLAKE manages
| 1 | WARD | 7521 | 7698 | BLAKE manages
| 1 | CLARK | 7782 | 7839 | CLARK manages
| 2 | MILLER | 7934 | 7782 | CLARK manages
| 1 | JONES | 7566 | 7839 | JONES manages
| 2 | FORD | 7902 | 7566 | JONES manages
| 3 | SMITH | 7369 | 7902 | JONES manages
| 2 | SCOTT | 7788 | 7566 | JONES manages
| 3 | ADAMS | 7876 | 7788 | JONES manages
|                               |
|                               |
|                               |
|                               |
|                               |
|                               |
|                               |
|                               |
| (13 rows)                     |

This example uses the CONNECT_BY_ROOT operator on an expression enclosed in parentheses:
The values of both \textit{ename} and \textit{empno} are affected by the \textit{CONNECT\_BY\_ROOT} operator. As a result, they return the values from the root node under the \textit{top mgr/empno} column:

\begin{verbatim}
CONNECT\_BY\_ROOT ('Manager ' || ename || ' is emp #' || empno) "top mgr/empno"
FROM emp
START WITH ename IN ('BLAKE','CLARK','JONES')
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename
ASC;
\end{verbatim}

The values of both \textit{ename} and \textit{empno} are affected by the \textit{CONNECT\_BY\_ROOT} operator. As a result, they return the values from the root node under the \textit{top mgr/empno} column:

\begin{verbatim}
SELECT level, ename, SYS_CONNECT\_BY\_PATH (ename, '/')
FROM managers
CONNECT BY PRIOR empno = mgr
START WITH mgr IS NULL
\end{verbatim}

4.2.4.5.7 Retrieving a path with SYS\_CONNECT\_BY\_PATH

\texttt{SYS\_CONNECT\_BY\_PATH} is a function that works in a hierarchical query to retrieve the column values of a specified column that occur between the current node and the root node.

Syntax

The signature of the function is:

\begin{verbatim}
SYS\_CONNECT\_BY\_PATH (<column>, <delimiter>)
\end{verbatim}

The function takes two arguments:

- \textit{column} is the name of a column that resides in a table specified in the hierarchical query that's calling the function.
- \textit{delimiter} is the \texttt{varchar} value that separates each entry in the specified column.

Examples

This example returns a list of employee names and their managers. If the manager has a manager, that name is appended to the result.

\begin{verbatim}
EDB# SELECT level, ename, SYS\_CONNECT\_BY\_PATH(ename, '/')
FROM managers
CONNECT BY PRIOR empno = mgr
START WITH mgr IS NULL
\end{verbatim}
<table>
<thead>
<tr>
<th>level</th>
<th>ename</th>
<th>managers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KING</td>
<td>/KING</td>
</tr>
<tr>
<td>2</td>
<td>BLAKE</td>
<td>/KING/BLAKE</td>
</tr>
<tr>
<td>2</td>
<td>CLARK</td>
<td>/KING/CLARK</td>
</tr>
<tr>
<td>2</td>
<td>JONES</td>
<td>/KING/JONES</td>
</tr>
<tr>
<td>3</td>
<td>ALLEN</td>
<td>/KING/BLAKE/ALLEN</td>
</tr>
<tr>
<td>3</td>
<td>FORD</td>
<td>/KING/JONES/FORD</td>
</tr>
<tr>
<td>3</td>
<td>JAMES</td>
<td>/KING/BLAKE/JAMES</td>
</tr>
<tr>
<td>3</td>
<td>MARTIN</td>
<td>/KING/BLAKE/MARTIN</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>/KING/JONES/SCOTT</td>
</tr>
<tr>
<td>3</td>
<td>TURNER</td>
<td>/KING/BLAKE/TURNER</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>/KING/JONES/SCOTT</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>/KING/JONES/SCOTT</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>/KING/JONES/SCOTT</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>/KING/JONES/SCOTT</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>/KING/JONES/SCOTT</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>/KING/JONES/SCOTT</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>/KING/JONES/SCOTT</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
<tr>
<td>3</td>
<td>SCOTT</td>
<td>/KING/JONES/SCOTT</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>/KING/CLARK/MILLER</td>
</tr>
</tbody>
</table>

In the result set:

- The **level** column displays the number of levels that the query returned.
- The **ename** column displays the employee name.
- The **managers** column contains the hierarchical list of managers.

The EDB Postgres Advanced Server implementation of **SYS_CONNECT_BY_PATH** doesn't support use of:

- **SYS_CONNECT_BY_PATH** inside **CONNECT_BY_PATH**
- **SYS_CONNECT_BY_PATH** inside **SYS_CONNECT_BY_PATH**

### 4.2.4.6 Multidimensional analysis

Multidimensional analysis refers to the process of examining data using various combinations of dimensions. Dimensions are categories used to classify data such as time, geography, a company's departments, product lines, and so forth. This process is commonly used in data warehousing applications. The results associated with a particular set of dimensions are called facts. Facts are typically figures associated with product sales, profits, volumes, counts, and so on.

To obtain these facts according to a set of dimensions in a relational database system, you typically use SQL aggregation. SQL aggregation basically means data is grouped according to certain criteria (dimensions), and the result set consists of aggregates of facts, such as counts, sums, and averages of the data in each group.

#### Aggregating results

The **GROUP BY** clause of the SQL **SELECT** command supports the following extensions that simplify the process of producing aggregate results:

- **ROLLUP extension**
- **CUBE extension**
- **GROUPING SETS extension**

In addition, you can use the **GROUPING** function and the **GROUPING_ID** function in the **SELECT** list or the **HAVING** clause to aid with the interpretation of the results when you use these extensions.

**Note**

The sample **dept** and **emp** tables are used extensively in this discussion to provide usage examples. The following changes were applied to these tables to provide more informative results:

**UPDATE dept**

```
UPDATE dept SET loc = 'BOSTON' WHERE deptno = 20;
```

**INSERT INTO emp**

```
INSERT INTO emp (empno,ename,job,deptno) VALUES (9001,'SMITH','CLERK',40);
INSERT INTO emp (empno,ename,job,deptno) VALUES (9002,'JONES','ANALYST',40);
INSERT INTO emp (empno,ename,job,deptno) VALUES (9003,'ROGERS','MANAGER',40);
```

The following rows from a join of the **emp** and **dept** tables are used:

**SELECT**

```
SELECT loc, dname, job, empno FROM emp e, dept d
WHERE e.deptno = d.deptno;
```
<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>empno</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>ANALYST</td>
<td>9082</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>CLERK</td>
<td>9081</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>MANAGER</td>
<td>9083</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>ANALYST</td>
<td>7788</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>ANALYST</td>
<td>7982</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>CLERK</td>
<td>7369</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>MANAGER</td>
<td>7566</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>CLERK</td>
<td>7980</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>MANAGER</td>
<td>7698</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>SALESMAN</td>
<td>7499</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>SALESMAN</td>
<td>7521</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>SALESMAN</td>
<td>7654</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>SALESMAN</td>
<td>7844</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>CLERK</td>
<td>7934</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>MANAGER</td>
<td>7782</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>PRESIDENT</td>
<td>7839</td>
</tr>
</tbody>
</table>

The `loc`, `dname`, and `job` columns are used for the dimensions of the SQL aggregations used in the examples. The resulting facts of the aggregations are the number of employees obtained by using the `COUNT(*)` function.

### Aggregation example

A basic query grouping the `loc`, `dname`, and `job` columns is given by the following:

```sql
SELECT loc, dname, job, COUNT(*) AS "employees"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc, dname, job
ORDER BY 1, 2, 3;
```

The rows of this result set using the basic `GROUP BY` clause without extensions are referred to as the base aggregate rows.

<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>ANALYST</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>ANALYST</td>
<td>2</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>CLERK</td>
<td>2</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>SALESMAN</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>PRESIDENT</td>
<td>1</td>
</tr>
</tbody>
</table>

(Edb 12 rows)

### Useful extensions

The `ROLLUP` and `CUBE` extensions add to the base aggregate rows by providing additional levels of subtotals to the result set.

The `GROUPING SETS` extension lets you combine different types of groupings into a single result set.
The GROUPING and GROUPING_ID functions help you to interpret the result set.

4.2.4.6.1 ROLLUP extension

The ROLLUP extension produces a hierarchical set of groups with subtotals for each hierarchical group as well as a grand total. The order of the hierarchy is determined by the order of the expressions given in the ROLLUP expression list. The top of the hierarchy is the left-most item in the list. Each successive item proceeding to the right moves down the hierarchy. The right-most item is at the lowest level.

Syntax

The syntax for a single ROLLUP is:

```
ROLLUP ( { expr_1_a | ( expr_1_a_a, expr_1_b ) ... }
[ , expr_2_a | ( expr_2_a_a, expr_2_b ) ... ]
...
)
```

Each expr is an expression that determines the grouping of the result set. If enclosed in parenthesis as ( expr_1_a, expr_1_b, ...), then the combination of values returned by expr_1_a and expr_1_b defines a single grouping level of the hierarchy.

The base level of aggregates returned in the result set is for each unique combination of values returned by the expression list.

In addition, a subtotal is returned for the first item in the list (expr_1 or the combination of ( expr_1_a, expr_1_b, ...), whichever is specified) for each unique value. A subtotal is returned for the second item in the list (expr_2 or the combination of ( expr_2_a, expr_2_b, ...), whichever is specified) for each unique value in each grouping of the first item and so on. Finally, a grand total is returned for the entire result set.

For the subtotal rows, null is returned for the items across which the subtotal is taken.

The ROLLUP extension specified in the context of the GROUP BY clause is shown by the following:

```
SELECT <select_list> FROM ...
GROUP BY [...] ROLLUP ( <expression_list> ) [ , ...
```

The items specified in select_list must either:
- Also appear in the ROLLUP expression_list
- Be aggregate functions such as COUNT, SUM, AVG, MIN, or MAX
- Be constants or functions whose return values are independent of the individual rows in the group (for example, the SYSDATE function)

Use the GROUP BY clause to specify multiple ROLLUP extensions as well as multiple occurrences of other GROUP BY extensions and individual expressions.

Use the ORDER BY clause to display the output in a hierarchical or other meaningful structure. Using this clause guarantees the order of the result set.

The number of grouping levels or totals is \( n + 1 \) where \( n \) represents the number of items in the ROLLUP expression list. A parenthesized list counts as one item.

Examples

The following query produces a rollup based on a hierarchy of columns loc, dname, and then job.

```
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
ORDER BY 1, 2, 3;
```

The result of the query displays a count of the number of employees for each unique combination of loc, dname, and job, as well as subtotals for each unique combination of loc and dname for each unique value of loc. A grand total appears on the last line:

```
<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>ANALYST</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The following query shows the effect of combining items in the `ROLLUP` list in parentheses:

```sql
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, (dname, job))
ORDER BY 1, 2, 3;
```

The output doesn't include subtotals for `loc` and `dname` combinations as in the prior example:

```
<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>ANALYST</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>ANALYST</td>
<td>2</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>CLERK</td>
<td>2</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>SALESMAN</td>
<td>4</td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>PRESIDENT</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>17</td>
</tr>
</tbody>
</table>
```

(16 rows)
If the first two columns in the `ROLLUP` list are enclosed in parentheses, the subtotal levels differ as well:

```sql
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP ((loc, dname), job)
ORDER BY 1, 2, 3;
```

Now there's a subtotal for each unique `loc` and `dname` combination but none for unique values of `loc`:

```
__OUTPUT__
<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>employees</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>ANALYST</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>CLERK</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>MANAGER</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>ANALYST</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>CLERK</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>MANAGER</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>CLERK</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>MANAGER</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>CLERK</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>MANAGER</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>PRESIDENT</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(17 rows)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

4.2.4.6.2 CUBE extension

The `CUBE` extension is similar to the `ROLLUP` extension. However, `ROLLUP` produces groupings and results in a hierarchy based on a left-to-right listing of items in the `ROLLUP` expression list. A `CUBE` produces groupings and subtotals based on every permutation of all items in the `CUBE` expression list. Thus, the result set contains more rows than a `ROLLUP` performed on the same expression list.

**Syntax**

The syntax for a single `CUBE` is:

```
CUBE ( [ ( expr_1 ) | ( expr_1a, expr_1b, ...) ] ...)
```

Each `expr` is an expression that determines the grouping of the result set. If enclosed in parenthesis as `( expr_1a, expr_1b, ... )`, then the combination of values returned by `expr_1a` and `expr_1b` defines a single group.

The base level of aggregates returned in the result set is for each unique combination of values returned by the expression list.

In addition, a subtotal is returned for the first item in the list (`expr_1`) or the combination of `( expr_1a, expr_1b, ... )`, whichever is specified) for each unique value. A subtotal is returned for the second item in the list (`expr_2`) or the combination of `( expr_2a, expr_2b, ... )`, whichever is specified) for each unique value. A subtotal is also returned for each unique combination of the first item and the second item. Similarly, if there's a third item, a subtotal is returned for: 
• Each unique value of the third item
• Each unique value of the third item and first item combination
• Each unique value of the third item and second item combination
• Each unique value of the third item, second item, and first item combination

Finally, a grand total is returned for the entire result set.

For the subtotal rows, null is returned for the items across which the subtotal is taken.

The \texttt{CUBE} extension specified in the context of the \texttt{GROUP BY} clause is shown by the following:

\begin{verbatim}
SELECT <select_list> FROM ...
GROUP BY [...] CUBE { <expression_list> } [, ...]
\end{verbatim}

The items specified in select_list must either:

• Also appear in the \texttt{CUBE expression_list}
• Be aggregate functions such as \texttt{COUNT}, \texttt{SUM}, \texttt{AVG}, \texttt{MIN}, or \texttt{MAX}
• Be constants or functions whose return values are independent of the individual rows in the group (for example, the \texttt{SYSDATE} function)

Use the \texttt{GROUP BY} clause to specify multiple \texttt{CUBE} extensions as well as multiple occurrences of other \texttt{GROUP BY} extensions and individual expressions.

Use the \texttt{ORDER BY} clause to display the output in a meaningful structure and to guarantee the order of the result set.

The number of grouping levels or totals is \(2^n\), where \(n\) represents the number of items in the \texttt{CUBE} expression list. A parenthesized list counts as one item.

\textbf{Examples}

The following query produces a cube based on permutations of columns \texttt{loc}, \texttt{dname}, and \texttt{job}.

\begin{verbatim}
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, dname, job)
ORDER BY 1, 2, 3;
\end{verbatim}

The following is the result of the query. It displays a count of the number of employees for each combination of \texttt{loc}, \texttt{dname}, and \texttt{job}, as well as subtotals for:

• Each combination of \texttt{loc} and \texttt{dname}
• Each combination of \texttt{loc} and \texttt{job}
• Each combination of \texttt{dname} and \texttt{job}
• Each unique value of \texttt{loc}
• Each unique value of \texttt{dname}
• Each unique value of \texttt{job}

A grand total appears on the last line.

\begin{verbatim}
<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>ANALYST</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>ANALYST</td>
<td>2</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>BOSTON</td>
<td>ANALYST</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>BOSTON</td>
<td>CLERK</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>BOSTON</td>
<td>MANAGER</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>
\end{verbatim}
<table>
<thead>
<tr>
<th></th>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>COUNT(*) AS &quot;employees&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td>CHICAGO</td>
<td>NY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SALES</td>
<td>CLERK</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SALES</td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SALES</td>
<td>SALESMAN</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>SALES</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td>CHICAGO</td>
<td>NY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ACCOUNTING</td>
<td>CLERK</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ACCOUNTING</td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ACCOUNTING</td>
<td>PRESIDENT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ACCOUNTING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ACCOUNTING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>ACCOUNTING</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>OPERATIONS</td>
<td>ANALYST</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>OPERATIONS</td>
<td>CLERK</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>OPERATIONS</td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>OPERATIONS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RESEARCH</td>
<td>ANALYST</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>RESEARCH</td>
<td>CLERK</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>RESEARCH</td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>RESEARCH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SALES</td>
<td>CLERK</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SALES</td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SALES</td>
<td>SALESMAN</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>SALES</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>ANALYST</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>CLERK</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>PRESIDENT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>SALESMAN</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

The following query shows the effect of combining items in the `CUBE` list in parentheses:

```sql
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, (dname, job))
ORDER BY 1, 2, 3;
```
The output has no subtotals for permutations involving `loc` and `dname` combinations, `loc` and `job` combinations, for `dname` by itself, or for `job` by itself.

The following query shows another variation whereby the first expression is specified outside of the `CUBE` extension:

```sql
SELECT loc, dname, job,
       COUNT(*) AS "employees"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc, CUBE (dname, job)
ORDER BY 1, 2, 3;
```

In this output, the permutations are performed for `dname` and `job` in each grouping of `loc`:

```
| loc      | dname    | job     |
|-----------------+------------+----------|
| BOSTON   | OPERATIONS | ANALYST  |
| BOSTON   | OPERATIONS | CLERK    |
| BOSTON   | OPERATIONS | MANAGER  |
| BOSTON   | RESEARCH   | ANALYST  |
| BOSTON   | RESEARCH   | CLERK    |
| BOSTON   | RESEARCH   | MANAGER  |
| BOSTON   |            |          |
| CHICAGO  | SALES      | CLERK    |
| CHICAGO  | SALES      | MANAGER  |
| CHICAGO  | SALES      | SALESMAN |
| CHICAGO  |            |          |
| NEW YORK | ACCOUNTING | CLERK    |
| NEW YORK | ACCOUNTING | MANAGER  |
| NEW YORK | ACCOUNTING | PRESIDENT|
| NEW YORK |            |          |
|           | ACCOUNTING | CLERK    |
|           | ACCOUNTING | MANAGER  |
|           | ACCOUNTING | PRESIDENT|
|           | OPERATIONS | ANALYST  |
|           | OPERATIONS | CLERK    |
|           | OPERATIONS | MANAGER  |
|           | RESEARCH   | ANALYST  |
|           | RESEARCH   | CLERK    |
|           | RESEARCH   | MANAGER  |
|           | SALES      | CLERK    |
|           | SALES      | MANAGER  |
|           | SALES      | SALESMAN |
|           |            |          |
```

(28 rows)
4.2.4.6.3 GROUPING SETS extension

The use of the GROUPING SETS extension in the GROUP BY clause provides a means to produce one result set that's the concatenation of multiple results sets based on different groupings. In other words, a UNION ALL operation is performed combining the result sets of multiple groupings into one result set.

A UNION ALL operation, and therefore the GROUPING SETS extension, doesn't eliminate duplicate rows from the result sets that are being combined.

Syntax

The syntax for a single GROUPING SETS extension is:

```sql
GROUPING SETS (
    ( <expr_1> | ( <expr_1a> [, <expr_1b> ] ...) |
        ROLLUP ( <expr_list> ) | CUBE ( <expr_list> ) |
    ) [, ...]
)
```

A GROUPING SETS extension can contain any combination of one or more comma-separated expressions, lists of expressions enclosed in parentheses, ROLLUP extensions, and CUBE extensions.

The GROUPING SETS extension is specified in the context of the GROUP BY clause:

```sql
SELECT <select_list> FROM ... 
GROUP BY [... ,] GROUPING SETS ( <expression_list> ) [ , ...]
```
The items specified in `select_list` must also either:

- Appear in the `GROUPING SETS` expression_list
- Be aggregate functions such as `COUNT`, `SUM`, `AVG`, `MIN`, or `MAX`
- Be constants or functions whose return values are independent of the individual rows in the group (for example, the `SYSDATE` function)

Use the `GROUP BY` clause to specify multiple `GROUPING SETS` extensions as well as multiple occurrences of other `GROUP BY` extensions and individual expressions.

Use the `ORDER BY` clause to display the output in a meaningful structure and to guarantee the order of the result set.

Examples

The following query produces a union of groups given by columns `loc`, `dname`, and `job`:

```sql
SELECT loc, dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY GROUPING SETS (loc, dname, job)
ORDER BY 1, 2, 3;
```

The result is as follows:

```
loc     | dname     | job     |
---------+-----------+---------+
BOSTON   |           |         |
8        |           |         |
CHICAGO  |           |         |
6        |           |         |
NEW YORK |           |         |
3        | ACCOUNTING|         |
3        | OPERATIONS|         |
3        | RESEARCH  |         |
5        | SALES     |         |
6        | ANALYST   |         |
3        | CLERK     |         |
5        | MANAGER   |         |
4        | PRESIDENT |         |
1        | SALESMAN  |         |
4        |           |         |
```

This is equivalent to the following query, which uses the `UNION ALL` operator:

```sql
SELECT loc AS "loc", NULL AS "dname", NULL AS "job", COUNT(*) AS "employees"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc
UNION ALL
SELECT NULL, dname, NULL, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY dname
UNION ALL
SELECT NULL, NULL, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY job
ORDER BY 1, 2, 3;
```

The output from the `UNION ALL` query is the same as the `GROUPING SETS` output:
This example shows how you can use various types of GROUP BY extensions together in a GROUPING SETS expression list:

```sql
SELECT loc, dname, job, COUNT(*) AS "employees"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY GROUPING SETS (loc, ROLLUP (dname, job), CUBE (job, loc))
ORDER BY 1, 2, 3;
```

The following is the output from this query:

<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td></td>
<td>ANALYST</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td>CLERK</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td>MANAGER</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td>CLERK</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td>MANAGER</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td>SALESMAN</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td>CLERK</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td>MANAGER</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td>PRESIDENT</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACCOUNTING</td>
<td>CLERK</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACCOUNTING</td>
<td>MANAGER</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACCOUNTING</td>
<td>PRESIDENT</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACCOUNTING</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OPERATIONS</td>
<td>ANALYST</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OPERATIONS</td>
<td>CLERK</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This example shows how you can use various types of GROUP BY extensions together in a GROUPING SETS expression list:
The output is basically a concatenation of the result sets from GROUP BY loc, GROUP BY ROLLUP (dname, job), and GROUP BY CUBE (job, loc). These individual queries are:

```sql
SELECT loc, NULL AS "dname", NULL AS "job", COUNT(*) AS "employees"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY loc
ORDER BY 1;
```

The result set from the GROUP BY loc clause is:

<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(3 rows)

The following query uses the GROUP BY ROLLUP (dname, job) clause:

```sql
SELECT NULL AS "loc", dname, job, COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (dname, job)
ORDER BY 2, 3;
```

The result set from the GROUP BY ROLLUP (dname, job) clause is:

<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNTING</td>
<td>CLERK</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ACCOUNTING</td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ACCOUNTING</td>
<td>PRESIDENT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ACCOUNTING</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>ANALYST</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

(38 rows)
The following query uses the `GROUP BY CUBE (job, loc)` clause:

```sql
SELECT loc, NULL AS "dname", COUNT(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (job, loc)
ORDER BY 1, 3;
```

The result set from the `GROUP BY CUBE (job, loc)` clause is:

<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td></td>
<td>ANALYST</td>
<td>3</td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td>CLERK</td>
<td>3</td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td>MANAGER</td>
<td>2</td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td>SALESMAN</td>
<td>4</td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td>CLERK</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td>MANAGER</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td>PRESIDENT</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ANALYST</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CLERK</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MANAGER</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRESIDENT</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SALESMAN</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>17</td>
</tr>
</tbody>
</table>

If the previous three queries are combined with the `UNION ALL` operator, a concatenation of the three results sets is produced:

```sql
SELECT loc AS "loc", NULL AS "dname", NULL AS "job", COUNT(*) AS "employees"
FROM emp e, dept d
WHERE e.deptno = d.deptno
```

<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>job</th>
<th>employees</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPERATIONS</td>
<td>CLERK</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>OPERATIONS</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RESEARCH</td>
<td>ANALYST</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RESEARCH</td>
<td>CLERK</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RESEARCH</td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RESEARCH</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>CLERK</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>MANAGER</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>SALESMAN</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>

(18 rows)
GROUP BY loc
UNION ALL
SELECT NULL, dname, job, count(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (dname, job)
UNION ALL
SELECT loc, NULL, job, count(*) AS "employees" FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (job, loc)
ORDER BY 1, 2, 3;

The output is the same as when the GROUP BY GROUPING SETS (loc, ROLLUP (dname, job), CUBE (job, loc)) clause is used.
4.2.4.6.4 GROUPING function

When using the \texttt{ROLLUP}, \texttt{CUBE}, or \texttt{GROUPING SETS} extensions to the \texttt{GROUP BY} clause, it can sometimes be difficult to differentiate between the various levels of subtotals generated by the extensions as well as the base aggregate rows in the result set. The \texttt{GROUPING} function provides a means of making this distinction.

Syntax

The general syntax for use of the \texttt{GROUPING} function is:

\texttt{SELECT [ \langle expr \rangle ...] GROUPING( \langle col_expr \rangle ) [, \langle expr \rangle ]}
\texttt{...}
\texttt{FROM ...}
\texttt{GROUP BY}
\texttt{[...],}
\texttt{( \texttt{ROLLUP} | \texttt{CUBE} | \texttt{GROUPING SETS} ) ( [...],)
\langle col_expr \rangle
[ [...], ] [,
[...], ]} \texttt{) [,
[...], ]}

The \texttt{GROUPING} function takes a single parameter that must be an expression of a dimension column specified in the expression list of a \texttt{ROLLUP}, \texttt{CUBE}, or \texttt{GROUPING SETS} extension of the \texttt{GROUP BY} clause.

The return value of the \texttt{GROUPING} function is either a 0 or 1. In the result set of a query:

- If the column expression specified in the \texttt{GROUPING} function is null because the row represents a subtotal over multiple values of that column, then the \texttt{GROUPING} function returns a value of 1.
- If the row returns results based on a particular value of the column specified in the \texttt{GROUPING} function, then the \texttt{GROUPING} function returns a value of 0.

In the latter case, the column can be null as well as non-null. In any case, it's for a particular value of that column, not a subtotal across multiple values.

Examples

The following query shows how the return values of the \texttt{GROUPING} function correspond to the subtotal lines:

\texttt{SELECT loc, dname, job, COUNT(*) AS "employees",
GROUPING(loc) AS "gf_loc",
GROUPING(dname) AS "gf_dname",
GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
ORDER BY 1, 2, 3;}

\texttt{__OUTPUT__}

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
loc & dname & job & employees & gf_loc & gf_dname \\
\hline
BOSTON & OPERATIONS & ANALYST & 1 & 0 & 0 \\
\hline
BOSTON & OPERATIONS & CLERK & 1 & 0 & 0 \\
\hline
BOSTON & OPERATIONS & MANAGER & 1 & 0 & 0 \\
\hline
\end{tabular}
You can use these indicators as screening criteria for particular subtotals. For example, using the previous query, you can display only those subtotals for `loc` and `dname` combinations by using the `GROUPING` function in a `HAVING` clause:

```sql
SELECT loc, dname, job, COUNT(*) AS "employees",
       GROUPING(loc) AS "gf_loc",
       GROUPING(dname) AS "gf_dname",
       GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
HAVING GROUPING(loc) = 0
       AND GROUPING(dname) = 0
       AND GROUPING(job) = 1
ORDER BY 1, 2;
```

This query produces the following result:

```
BOSTON | OPERATIONS |     | 3 | 0 | 0
BOSTON | RESEARCH   |     | 5 | 0 | 0
BOSTON | RESEARCH   | CLERK | 2 | 0 | 0
BOSTON | RESEARCH   | MANAGER | 1 | 0 | 0
BOSTON | RESEARCH   |     | 8 | 0 | 1
CHICAGO | SALES      | CLERK | 1 | 0 | 0
CHICAGO | SALES      | MANAGER | 1 | 0 | 0
CHICAGO | SALES      | SALESMAN | 4 | 0 | 0
CHICAGO | SALES      |     | 6 | 0 | 0
CHICAGO |     |     | 6 | 0 | 1
NEW YORK | ACCOUNTING | CLERK | 1 | 0 | 0
NEW YORK | ACCOUNTING | MANAGER | 1 | 0 | 0
NEW YORK | ACCOUNTING | PRESIDENT | 1 | 0 | 0
NEW YORK | ACCOUNTING |     | 3 | 0 | 0
NEW YORK |     |     | 3 | 0 | 1
       |     |     | 17 | 1 | 1
(4 rows)
```

You can use the `GROUPING` function to distinguish a subtotal row from a base aggregate row or from certain subtotal rows. These subtotal rows have one item in the expression list that returns null as a result of the column on which the expression is based being null for one or more rows in the table. This is opposed to representing a subtotal over the column.

To illustrate this point, the following row is added to the `emp` table, which provides a row with a null value for the `job` column:

```sql
INSERT INTO emp (empno, ename, deptno)
VALUES (9004, 'PETERS', 40);
```

The following query is issued using a reduced number of rows for clarity:

```sql
SELECT loc, job, COUNT(*) AS "employees",
       GROUPING(loc) AS "gf_loc",
       GROUPING(job) AS "gf_job"
```

You can use these indicators as screening criteria for particular subtotals. For example, using the previous query, you can display only those subtotals for `loc` and `dname` combinations by using the `GROUPING` function in a `HAVING` clause:

```sql
SELECT loc, dname, job, COUNT(*) AS "employees",
       GROUPING(loc) AS "gf_loc",
       GROUPING(dname) AS "gf_dname",
       GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY ROLLUP (loc, dname, job)
HAVING GROUPING(loc) = 0
       AND GROUPING(dname) = 0
       AND GROUPING(job) = 1
ORDER BY 1, 2;
```

This query produces the following result:

```
BOSTON | OPERATIONS |     | 3 | 0 | 0
BOSTON | RESEARCH   |     | 5 | 0 | 0
BOSTON | RESEARCH   | CLERK | 2 | 0 | 0
BOSTON | RESEARCH   | MANAGER | 1 | 0 | 0
BOSTON | RESEARCH   |     | 8 | 0 | 1
CHICAGO | SALES      | CLERK | 1 | 0 | 0
CHICAGO | SALES      | MANAGER | 1 | 0 | 0
CHICAGO | SALES      | SALESMAN | 4 | 0 | 0
CHICAGO | SALES      |     | 6 | 0 | 0
CHICAGO |     |     | 6 | 0 | 1
NEW YORK | ACCOUNTING | CLERK | 1 | 0 | 0
NEW YORK | ACCOUNTING | MANAGER | 1 | 0 | 0
NEW YORK | ACCOUNTING | PRESIDENT | 1 | 0 | 0
NEW YORK | ACCOUNTING |     | 3 | 0 | 0
NEW YORK |     |     | 3 | 0 | 1
       |     |     | 17 | 1 | 1
(4 rows)
```

You can use the `GROUPING` function to distinguish a subtotal row from a base aggregate row or from certain subtotal rows. These subtotal rows have one item in the expression list that returns null as a result of the column on which the expression is based being null for one or more rows in the table. This is opposed to representing a subtotal over the column.

To illustrate this point, the following row is added to the `emp` table, which provides a row with a null value for the `job` column:

```sql
INSERT INTO emp (empno, ename, deptno)
VALUES (9004, 'PETERS', 40);
```
The output contains two rows containing BOSTON in the loc column and spaces in the job column (fourth and fifth entries in the table):

<table>
<thead>
<tr>
<th>loc</th>
<th>job</th>
<th>employees</th>
<th>gf_loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td>ANALYST</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>BOSTON</td>
<td>CLERK</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>BOSTON</td>
<td>MANAGER</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>ANALYST</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CLERK</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>MANAGER</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

The fifth row, where the GROUPING function on the job column (gf_job) returns 1, indicates this is a subtotal over all jobs. The row contains a subtotal value of 9 in the employees column.

In the fourth row, the GROUPING function on the job column as well as on the loc column returns 0. This indicates that this is a base aggregate of all rows, where loc is BOSTON and job is null, which is the row inserted for this example. The employees column contains 1, which is the count of the single such row inserted.

In the ninth row (next to last) the GROUPING function on the job column returns 0, while the GROUPING function on the loc column returns 1. This indicates that this is a subtotal over all locations where the job column is null which, again, is a count of the single row inserted for this example.

4.2.4.6.5 GROUPING_ID function

The GROUPING_ID function provides a simplification of the GROUPING function to determine the subtotal level of a row in the result set from a ROLLBACK, CUBE, or GROUPING SETS extension.

The GROUPING function takes only one column expression and returns an indication of whether a row is a subtotal over all values of the given column. Thus, you might need multiple GROUPING functions to interpret the level of subtotals for queries with multiple grouping columns.

The GROUPING_ID function accepts one or more column expressions that were used in the ROLLBACK, CUBE, or GROUPING SETS extensions. It returns a single integer that you can use to determine over which of these columns a subtotal was aggregated.

Syntax

The general syntax for the GROUPING_ID function is:

```sql
SELECT [ <expr> 
  ..., ]
GROUPING_ID( <col_expr_1> [, <col_expr_2> ] ... ) [ , <expr> ]
[, <expr> ]
...
FROM ...
GROUP BY [...]
{ [ ROLLUP | CUBE | GROUPING SETS ]( [ ... ]
  <col_expr_1> 
  [, <col_expr_2> ] [ , ... ] [ , 
  ... ]
  ) [ , ... ] [ , 
  ... ]
)
```

The GROUPING_ID function takes one or more parameters that must be expressions of dimension columns specified in the expression list of a ROLLUP, CUBE, or GROUPING SETS extension of the GROUP BY clause.
The **GROUPING_ID** function returns an integer value. This value corresponds to the base-10 interpretation of a bit vector consisting of the concatenated 1s and 0s returned by a series of **GROUPING** functions specified in the same left-to-right order as the ordering of the parameters specified in the **GROUPING_ID** function.

### Examples

The following query shows how the returned values of the **GROUPING_ID** function represented in column `gid` correspond to the values returned by two **GROUPING** functions on columns `loc` and `dname`:

```sql
SELECT loc, dname, COUNT(*) AS "employees",
   GROUPING(loc) AS "gf_loc", GROUPING(dname) AS "gf_dname",
   GROUPING_ID(loc, dname) AS "gid"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, dname)
ORDER BY 6, 1, 2;
```

In the following output, note the relationship between a bit vector consisting of the `gf_loc` value and `gf_dname` value compared to the integer given in `gid`:

<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>employees</th>
<th>gf_loc</th>
<th>gf_dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BOSTON</td>
<td>RESEARCH</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BOSTON</td>
<td></td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CHICAGO</td>
<td></td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ACCOUNTING</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>OPERATIONS</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>RESEARCH</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>SALES</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ACCOUNTING</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>OPERATIONS</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RESEARCH</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SALES</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ACCOUNTING</td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

The following table provides specific examples of the **GROUPING_ID** function calculations based on the **GROUPING** function return values for four rows of the output.

<table>
<thead>
<tr>
<th>loc</th>
<th>dname</th>
<th>Bit Vector</th>
<th>GROUPING_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
<td>0 * 2¹ + 0 * 2⁰</td>
<td>0</td>
</tr>
<tr>
<td>BOSTON</td>
<td>null</td>
<td>0 * 2¹ + 1 * 2⁰</td>
<td>1</td>
</tr>
<tr>
<td>null</td>
<td>ACCOUNTING</td>
<td>1 * 2¹ + 0 * 2⁰</td>
<td>2</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>1 * 2¹ + 1 * 2⁰</td>
<td>3</td>
</tr>
</tbody>
</table>

The following table summarizes how the **GROUPING_ID** function return values correspond to the grouping columns over which aggregation occurs.

<table>
<thead>
<tr>
<th>Aggregation by Column</th>
<th>Bit Vector</th>
<th>GROUPING_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc, dname</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>loc</td>
<td>0 1</td>
<td>1</td>
</tr>
<tr>
<td>dname</td>
<td>1 0</td>
<td>2</td>
</tr>
<tr>
<td>Grand Total</td>
<td>1 1</td>
<td>3</td>
</tr>
</tbody>
</table>
To display only those subtotals by `dname`, use the following simplified query with a `HAVING` clause based on the `GROUPING_ID` function:

```sql
SELECT loc, dname, COUNT(*) AS "employees",
       GROUPING(loc) AS "gf_loc", GROUPING(dname) AS "gf_dname",
       GROUPING_ID(loc, dname) AS "gid"
FROM emp e, dept d
WHERE e.deptno = d.deptno
GROUP BY CUBE (loc, dname)
HAVING GROUPING_ID(loc, dname) = 2
ORDER BY 6, 1, 2;
```

The result of the query is:

<table>
<thead>
<tr>
<th><strong>OUTPUT</strong></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>gid</td>
<td>loc</td>
<td>dname</td>
<td>employees</td>
<td>gf_loc</td>
<td>gf_dname</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ACCOUNTING</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OPERATIONS</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RESEARCH</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SALES</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(4 rows)
The following is the `pg-sample.sql` script:

```sql
SET datestyle TO 'iso, dmy';

-- Script that creates the 'sample' tables, views
-- functions, triggers, etc.
-- Start new transaction - commit all or nothing
BEGIN;

-- Create and load tables used in the documentation examples.
-- Create the 'dept' table
CREATE TABLE dept
(
    deptno NUMERIC(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname VARCHAR(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc VARCHAR(13)
);

-- Create the 'emp' table
CREATE TABLE emp
(
    empno NUMERIC(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename VARCHAR(10),
    job VARCHAR(9),
    mgr NUMERIC(4),
    hiredate DATE,
    sal NUMERIC(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm NUMERIC(7,2),
    deptno NUMERIC(2) CONSTRAINT emp_ref_dept_fk REFERENCES dept(deptno)
);

-- Create the 'jobhist' table
CREATE TABLE jobhist
(
    empno NUMERIC(4) NOT NULL,
    startdate TIMESTAMP(0) NOT NULL,
    enddate TIMESTAMP(0),
    job VARCHAR(9),
    sal NUMERIC(7,2),
    comm NUMERIC(7,2),
    deptno NUMERIC(2),
    chgdesc VARCHAR(88),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
    CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
        REFERENCES emp(empno) ON DELETE CASCADE,
    CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
        REFERENCES dept (deptno) ON DELETE SET NULL,
    CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);

-- Create the 'salesemp' view
CREATE OR REPLACE VIEW salesemp
AS
    SELECT empno, ename, hiredate, sal, comm
FROM emp
WHERE job = 'SALESMAN';
```
CREATE SEQUENCE next_empno START WITH 8008 INCREMENT BY 1;
-- Issue PUBLIC grants
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
-- Load the 'dept'
table
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (28,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (38,'SALES','CHICAGO');
INSERT INTO dept VALUES (48,'OPERATIONS','BOSTON');
-- Load the 'emp'
table
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',808,80,20);  
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,1000,30);  
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);  
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-81',2975,NULL,20); 
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-81',1250,1400,30); 
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-81',2850,NULL,30); 
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-81',2450,NULL,10); 
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-87',3000,NULL,20); 
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-81',5000,NULL,10); 
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,8,30);  
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);  
INSERT INTO emp VALUES (7908,'JAMES','CLERK',7698,'03-DEC-81',958,NULL,30); 
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20); 
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10); 
-- Load the 'jobhist'
table
INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',808,NULL,20,'New Hire'); 
INSERT INTO jobhist VALUES (7499,'28-FEB-81',NULL,'SALESMAN',1600,380,38,'New Hire'); 
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,580,38,'New Hire'); 
INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,'New Hire'); 
INSERT INTO jobhist VALUES (7654,'28-SEP-81',NULL,'SALESMAN',1250,1400,38,'New Hire'); 
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2858,NULL,30,'New Hire'); 
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2458,NULL,10,'New Hire'); 
INSERT INTO jobhist VALUES (7788,'19-APR-87', NULL,'CLERK',1000,NULL,20,'New Hire'); 
INSERT INTO jobhist VALUES (7788,'13-APR-88', '04-MAY-89','CLERK',1848,NULL,20,'Raise');  
INSERT INTO jobhist VALUES (7788,'05-MAY-88',NULL,'ANALYST',3000,NULL,20,'Promoted to Analyst'); 
INSERT INTO jobhist VALUES (7839,'17-NOV-81',NULL,'PRESIDENT',5000,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7844, '08-SEP-81', NULL, 'SALESMAN', 1500, 'New Hire');
INSERT INTO jobhist VALUES (7876, '23-MAY-87', NULL, 'CLERK', 1100, 'New Hire');
INSERT INTO jobhist VALUES (7900, '03-DEC-81', '14-JAN-83', 'CLERK', 950, 'New Hire');
INSERT INTO jobhist VALUES (7900, '15-JAN-83', NULL, 'CLERK', 950, 'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902, '03-DEC-81', NULL, 'ANALYST', 3000, 'New Hire');
INSERT INTO jobhist VALUES (7934, '23-JAN-82', NULL, 'CLERK', 1300, 'New Hire');

-- Populate statistics table and view (pg_statistic/pg_stats)
ANALYZE dept;
ANALYZE emp;
ANALYZE jobhist;

-- Function that lists all employees' numbers and names
-- from the 'emp' table using a cursor.
CREATE OR REPLACE FUNCTION list_emp() RETURNS VOID AS $$
DECLARE
  v_empno NUMERIC(4);
  v_ename VARCHAR(10);
  emp_cur CURSOR FOR
  SELECT empno, ename
  FROM emp
  ORDER BY empno;
BEGIN
  OPEN emp_cur;
  RAISE INFO 'EMPNO ENAME'
  '----- -------';
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN NOT FOUND;
    RAISE INFO '% %', v_empno, v_ename;
  END LOOP;
  CLOSE emp_cur;
  RETURN;
END;
$$ LANGUAGE 'plpgsql';

-- Function that selects an employee row given the employee number and displays certain columns.
CREATE OR REPLACE FUNCTION select_emp (p_empno NUMERIC) RETURNS VOID AS $$
DECLARE
  v_ename emp.ename%TYPE;
  v_hiredate emp.hiredate%TYPE;
  v_sal emp.sal%TYPE;
  v_comm emp.comm%TYPE;
  v_dname dept.dname%TYPE;
  v_disp_date VARCHAR(10);
BEGIN
  SELECT v_ename, v_hiredate, v_sal, v_comm,
  v_dname
  ename, hiredate, sal, COALESCE(comm, 0),
  FROM emp e, dept d
  WHERE empno = p_empno
  AND e.deptno =
  d.deptno;
END;
IF NOT FOUND THEN
    RAISE INFO 'Employee % not found',
p_empno;
RETURN;
END IF;

v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
RAISE INFO 'Number : %', p_empno;
RAISE INFO 'Name : %',
v_ename;
RAISE INFO 'Hire Date : %',
v_disp_date;
RAISE INFO 'Salary : %',
v_sal;
RAISE INFO 'Commission: %', v_comm;
RAISE INFO 'Department: %',
v_dname;
RETURN;
END

EXCEPTION
WHEN OTHERS THEN
RAISE INFO 'The following is SQLERRM : %', SQLERRM;
RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
RETURN;
END

$$ LANGUAGE 'plpgsql';

-- A RECORD type used to format the return value of function, 'emp_query'.
-- CREATE TYPE emp_query_type AS
{
    empno NUMERIC,
    ename VARCHAR(10),
    job VARCHAR(9),
hiredate DATE,
sal NUMERIC
};
-- Function that queries the 'emp' table based on department number and employee number or name. Returns employee number and name as INOUT parameters and job, hire date, and salary as OUT parameters. These are returned in the form of a record defined by
-- CREATE OR REPLACE FUNCTION emp_query
-- IN p_deptno NUMERIC,
-- INOUT p_empno NUMERIC,
-- INOUT p_ename VARCHAR,
-- OUT p_job VARCHAR,
-- OUT p_hiredate DATE,
-- OUT p_sal NUMERIC
)-- AS $$
BEGIN
    SELECT INTO
    p_empno, p_ename, p_job, p_hiredate,
    empno, ename, job, hiredate,
sal
FROM
emp
WHERE deptno =
p_deptno
AND (empno =
p_empno
    OR ename = UPPER(p_ename));
END
$$ LANGUAGE 'plpgsql';
-- Function to call 'emp_query_caller' with IN and INOUT parameters. Displays the results received from INOUT and OUT parameters.

CREATE OR REPLACE FUNCTION emp_query_caller() RETURNS VOID AS $$
DECLARE
  v_deptno NUMERIC;
  v_empno NUMERIC;
  v_ename VARCHAR;
  v_rows INTEGER;
  r_emp_query EMP_QUERY_TYPE;
BEGIN
  v_deptno := 30;
  v_empno := 0;
  v_ename := 'Martin';
  r_emp_query := emp_query(v_deptno, v_empno, v_ename);
  RAISE INFO 'Department : %', v_deptno;
  RAISE INFO 'Employee No: %', (r_emp_query).empno;
  RAISE INFO 'Name       : %', (r_emp_query).ename;
  RAISE INFO 'Job        : %', (r_emp_query).job;
  RAISE INFO 'Hire Date  : %', (r_emp_query).hiredate;
  RAISE INFO 'Salary     : %', (r_emp_query).sal;
RETURN;
END
$$ LANGUAGE 'plpgsql';

-- Function to compute yearly compensation based on semimonthly salary.

CREATE OR REPLACE FUNCTION emp_comp (p_sal NUMERIC, p_comm NUMERIC) RETURNS NUMERIC AS $$
BEGIN
  RETURN (p_sal + COALESCE(p_comm, 0)) * 24;
END
$$ LANGUAGE 'plpgsql';

-- Function that gets the next number from sequence, 'next_empno', and ensures it is not already in use as an employee number.

CREATE OR REPLACE FUNCTION new_empno() RETURNS INTEGER AS $$
DECLARE
  v_cnt INTEGER := 1;
  v_new_empno INTEGER;
BEGIN
  WHILE v_cnt > 0 LOOP
    SELECT INTO v_new_empno nextval('next_empno');
    SELECT INTO v_cnt COUNT(*) FROM emp WHERE empno = v_new_empno;
  END LOOP;
RETURN;
$$ LANGUAGE 'plpgsql';
CREATE OR REPLACE FUNCTION hire_clerk
(p_ename VARCHAR,
p_deptno NUMERIC)
RETURNS NUMERIC
AS $$
DECLARE
v_empno NUMERIC(4);
v_ename VARCHAR(10);
v_job VARCHAR(9);
v_mgr NUMERIC(4);
v_hiredate DATE;
v_sal NUMERIC(7,2);
v_comm NUMERIC(7,2);
v_deptno NUMERIC(2);
BEGIN
v_empno := new_empno();
INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK',
7782, CURRENT_DATE, 950.00, NULL, p_deptno);
SELECT INTO v_deptno empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm,
FROM emp WHERE empno = v_empno;
v_deptno;
RAISE INFO 'Department : %',
v_deptno;
RAISE INFO 'Employee No: %',
v_empno;
RAISE INFO 'Name : %',
v_ename;
RAISE INFO 'Job : %',
v_job;
RAISE INFO 'Manager : %',
v_mgr;
RAISE INFO 'Hire Date : %',
v_hiredate;
RAISE INFO 'Salary : %',
v_sal;
RAISE INFO 'Commission : %',
v_comm;
RETURN v_empno;
EXCEPTION
WHEN OTHERS
THEN
RAISE INFO 'The following is SQLERRM : %', SQLERRM;
RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
RETURN -1;
END;
$$ LANGUAGE 'plpgsql';
--
-- Function that adds a new salesman to table 'emp'.
--
CREATE OR REPLACE FUNCTION hire_salesman
(p_ename VARCHAR,
p_sal NUMERIC,
p_comm NUMERIC)
RETURNS NUMERIC
AS $$
DECLARE
v_empno NUMERIC(4);
v_ename VARCHAR(10);
v_job VARCHAR(9);
v_mgr NUMERIC(4);
v_hiredate DATE;
v_sal NUMERIC(7,2);
```sql
v_comm
NUMERIC(7,2);

BEGIN
  v_empno := new_empno();
  INSERT INTO emp VALUES (v_empno, p_ename, 'SALESMAN',
   CURRENT_DATE, p_sal, p_comm,
   30);
  SELECT INTO 
    v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm,
    v_deptno
    empno, ename, job, mgr, hiredate, sal, comm, deptno
  FROM emp WHERE empno =
  v_empno;
  RAISE INFO 'Department : %', v_deptno;
  RAISE INFO 'Employee No: %', v_empno;
  RAISE INFO 'Name       : %', v_ename;
  RAISE INFO 'Job        : %', v_job;
  RAISE INFO 'Manager    : %', v_mgr;
  RAISE INFO 'Hire Date  : %', v_hiredate;
  RAISE INFO 'Salary     : %', v_sal;
  RAISE INFO 'Commission : %', v_comm;
  RETURN v_empno;
END;

EXCEPTION
WHEN OTHERS
THEN
  RAISE INFO 'The following is SQLERRM : %', SQLERRM;
  RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
  RETURN -1;
END;

$$ LANGUAGE 'plpgsql';

CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO
glemp
DO INSTEAD
  INSERT INTO emp VALUES (NEW.empno, NEW.ename, 'SALESMAN',
   7698, NEW.hiredate, NEW.sal, NEW.comm, 30);

CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO
glemp
DO INSTEAD
  UPDATE emp SET empno =
    NEW.empno,
    ename = NEW.ename,
    hiredate =
    NEW.hiredate,
    sal =
    NEW.sal,
    comm = NEW.comm
    WHERE empno = OLD.empno;

CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO
glemp
DO INSTEAD
  DELETE FROM emp WHERE empno =
    OLD.empno;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
CREATE OR REPLACE FUNCTION user_audit_trig() RETURNS TRIGGER
AS $$
DECLARE
    v_action VARCHAR(24);
    v_text TEXT;
BEGIN
    IF TG_OP = 'INSERT' THEN
        v_action := ' added employee(s) on ';
    ELSIF TG_OP = 'UPDATE' THEN
        v_action := ' updated employee(s) on ';
    ELSIF TG_OP = 'DELETE' THEN
        v_action := ' deleted employee(s) on ';
    END IF;
    v_text := 'User ' || USER || ' ' || v_action || CURRENT_DATE;
    RAISE INFO ' %', v_text;
RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';
CREATE TRIGGER user_audit_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH STATEMENT EXECUTE PROCEDURE user_audit_trig();

-- Before row-level trigger that displays employee number
-- and salary of an employee that is about to be added,
-- updated, or deleted in the 'emp'
table.

CREATE OR REPLACE FUNCTION emp_sal_trig() RETURNS TRIGGER
AS $$
DECLARE
    sal_diff NUMERIC(7,2);
BEGIN
    IF TG_OP = 'INSERT' THEN
        RAISE INFO 'Inserting employee %',
        NEW.empno;
        RAISE INFO '..New salary: %', NEW.sal;
        RETURN NEW;
    END IF;
    IF TG_OP = 'UPDATE' THEN
        sal_diff := NEW.sal - OLD.sal;
        RAISE INFO 'Updating employee %',
        OLD.empno;
        RAISE INFO '..Old salary: %', OLD.sal;
        RAISE INFO '..New salary: %', NEW.sal;
        RAISE INFO '..Raise : %',
        sal_diff;
        RETURN NEW;
    END IF;
    IF TG_OP = 'DELETE' THEN
        RAISE INFO 'Deleting employee %',
        OLD.empno;
        RAISE INFO '..Old salary: %', OLD.sal;
    END IF;
END;
$$ LANGUAGE 'plpgsql';
CREATE TRIGGER emp_sal_trig
BEFORE DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW EXECUTE PROCEDURE emp_sal_trig();
COMMIT;

5  Planning

The process for planning the implementation or deployment of EDB Postgres Advanced Server includes several activities:
5.1 Choosing the configuration mode

Moving to the cloud can be a challenge, especially if you’re migrating Oracle applications to Postgres in the cloud.

Whether your goal is to reduce database management costs, increase business agility, jumpstart cloud innovation or modernize your data infrastructure, EDB Postgres Advanced Server is the solution specifically designed to assist your migration. EDB Postgres Advanced Server includes feature-rich tools and enhancements that help you maintain, secure, and operate your database environment.

EDB Postgres Advanced Server also has a significant amount of Oracle compatibility features which facilitate Oracle to Postgres migrations. When configured to run in Oracle mode, EDB Postgres Advanced Server includes extended functionality that provides compatibility for syntax supported by Oracle applications, as well as compatible procedural logic, data types, system catalog views and other features that enable EDB’s Oracle compatible connectors, EDB*Plus, EDB*Loader as well as other functionality.

When you initialize your EDB Postgres Advanced Server cluster using the \texttt{initdb} command, you can choose whether or not to include these compatibility features by specifying a configuration mode.

There are two options for the configuration mode:

- Specify the \texttt{no-redwood-compat} option to create the cluster in Postgres mode. When the cluster is created in PostgreSQL mode, it includes all of the advanced features that help you maintain, secure, and operate your database environment. While some EDB Postgres Advanced Server features compatible with Oracle databases are available with this mode, such as Oracle style packages and collections, we recommend using the EDB Postgres Advanced Server in redwood compatibility mode if you are implementing an Oracle to Postgres migration.

- Specify the \texttt{redwood-like} option to create the cluster in Oracle compatibility mode. This mode enables all of the rich Oracle compatibility features to help you facilitate your Oracle to Postgres migration. These features include Oracle compatible custom data types, keywords, functions, and catalog views. You can find details about these features in Working with Oracle data.

5.2 Deployment options

You can deploy and install EDB products using the following methods:

- BigAnimal is a fully managed database-as-a-service with built-in Oracle compatibility, running in your cloud account and operated by the Postgres experts. BigAnimal makes it easy to set up, manage, and scale your databases. Provision PostgreSQL or EDB Postgres Advanced Server with Oracle compatibility.

- EDB PostgreSQL for Kubernetes is an operator designed by EnterpriseDB to manage PostgreSQL workloads on any supported Kubernetes cluster running in private, public, hybrid, or multi-cloud environments. EDB PostgreSQL for Kubernetes adheres to DevOps principles and concepts such as declarative configuration and immutable infrastructure.

Installation using native packages or installers

5.3 Planning prerequisites

Follow these requirements and considerations before deploying EDB Postgres Advanced Server.

5.3.1 Limitations

The following limitations apply when implementing EDB Postgres Advanced Server:

- EDB recommends you don’t store the \texttt{data} directory of a production database on an NFS file system. If you plan to go against this recommendation, see the 19.2.2.1. NFS section in the PostgreSQL documentation for guidance about configuration.

- The LLVM JIT package is supported only on RHEL or CentOS x86.

5.3.2 Requirements

EDB Postgres Advanced Server has certain hardware and software requirements. PostgreSQL has some hard limits that are important to know about during your planning.

Hardware requirements

The following installation requirements assume that you selected the default options during the installation process. The minimum hardware requirements to install and run EDB Postgres Advanced Server are:

- 1 GHz processor
- 2 GB of RAM
- 512 MB of HDD
Additional disk space is required for data or supporting components.

Software requirements

User privileges

To perform an EDB Postgres Advanced Server installation on a Linux system you need superuser, administrator, or sudo privileges.

To perform an EDB Postgres Advanced Server installation on a Windows system, you need administrator privileges. If you’re installing EDB Postgres Advanced Server on a Windows system that’s configured with User Account Control enabled, you can assume the privileges required to invoke the graphical installer. Right-click the name of the installer, and select Run as administrator from the context menu.

Windows-specific software requirements

Apply the Windows operating system updates before invoking the installer. If the installer encounters errors during the installation process, exit the installation, and ensure that your Windows version is up to date. Then restart the installer.

See the release notes for the features added in EDB Postgres Advanced Server 16.

Hard limits

The following table describes various hard limits of PostgreSQL. However, practical limits such as performance limitations or available disk space might apply before absolute hard limits are reached.

<table>
<thead>
<tr>
<th>Item</th>
<th>Upper limit</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>database size</td>
<td>unlimited</td>
<td></td>
</tr>
<tr>
<td>number of databases</td>
<td>4,294,950,911</td>
<td></td>
</tr>
<tr>
<td>relations per database</td>
<td>1,431,650,303</td>
<td></td>
</tr>
<tr>
<td>relation size</td>
<td>32 TB</td>
<td>with the default BLCKSZ of 8192 bytes</td>
</tr>
<tr>
<td>rows per table</td>
<td>limited by 4,294,967,295 pages</td>
<td></td>
</tr>
<tr>
<td>columns per table</td>
<td>1600</td>
<td>further limited by tuple size fitting on a single page; see note below</td>
</tr>
<tr>
<td>field size</td>
<td>1 GB</td>
<td></td>
</tr>
<tr>
<td>identifier length</td>
<td>63 bytes</td>
<td>can be increased by recompiling PostgreSQL</td>
</tr>
<tr>
<td>indexes per table</td>
<td>unlimited</td>
<td>constrained by maximum relations per database</td>
</tr>
<tr>
<td>columns per index</td>
<td>32</td>
<td>can be increased by recompiling PostgreSQL</td>
</tr>
<tr>
<td>partition keys</td>
<td>32</td>
<td>can be increased by recompiling PostgreSQL</td>
</tr>
</tbody>
</table>

Note

- The maximum number of columns for a table is further reduced as the tuple being stored must fit in a single 8192-byte heap page. For example, excluding the tuple header, a tuple made up of 1600 int columns consumes 6400 bytes and can be stored in a heap page. But a tuple of 1600 bigint columns consumes 12800 bytes and therefore doesn’t fit inside a heap page. Variable-length fields of types such as text, varchar, and char can have their values stored out of line in the table’s TOAST table when the values are large enough to require it. Only an 18-byte pointer must remain inside the tuple in the table’s heap. For shorter length variable-length fields, either a 4-byte or 1-byte field header is used, and the value is stored inside the heap tuple.

- Columns that were dropped from the table also contribute to the maximum column limit. Moreover, although the dropped column values for newly created tuples are internally marked as null in the tuple’s null bitmap, the null bitmap also occupies space.

6 Installing EDB Postgres Advanced Server

Select a link to access the applicable installation instructions:

Linux x86-64 (amd64)

Red Hat Enterprise Linux (RHEL) and derivatives
6.1 Installing EDB Postgres Advanced Server on Linux x86 (amd64)

Operating system-specific install instructions are described in the corresponding documentation:

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9
- RHEL 8
- RHEL 7
- Oracle Linux (OL) 9
- Oracle Linux (OL) 8
- Oracle Linux (OL) 7
- Rocky Linux 9
- Rocky Linux 8
- AlmaLinux 9
- AlmaLinux 8
- CentOS 7

SUSE Linux Enterprise (SLES)

- SLES 15, SLES 12

Debian and derivatives

- Ubuntu 22.04, Ubuntu 20.04, Ubuntu 18.04
- Debian 11, Debian 10

Linux IBM Power (ppc64le)

Red Hat Enterprise Linux (RHEL) and derivatives

- RHEL 9, RHEL 8, RHEL 7

SUSE Linux Enterprise (SLES)

- SLES 15, SLES 12

Windows

- Windows Server 2019
Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

To determine if your repository exists, enter this command:

```
dnf repolist | grep enterprise\db
```

If no output is generated, the repository isn't installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.
3. Select the platform and software that you want to download.
4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

```
```

- If you are also installing PostGIS, enable additional repositories to resolve dependencies:

```
ARCH=$( /bin/\arch ) subscription-manager repos --enable "codeready-builder-for-rhel-9-\$\(ARCH\)-\rpms"
```

**Note**

If you are using a public cloud RHEL image, subscription manager may not be enabled and enabling it may incur unnecessary charges. Equivalent packages may be available under a different name such as codeready-builder-for-rhel-9-rhui-\rpms. Consult the documentation for the RHEL image you are using to determine how to install codeready-builder.
Install the package

```bash
sudo dnf -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```bash
sudo dnf -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Installing the server package creates an operating system user named enterprisedb. The user is assigned a user ID (UID) and a group ID (GID). The user has no default password. Use the `passwd` command to assign a password for the user. The default shell for the user is `bash`, and the user's home directory is `/var/lib/edb/as16`.

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see Initializing the cluster in Postgres mode.

```bash
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```bash
sudo su - enterprisedb
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see Modifying the `pg_hba.conf` file.

```sql
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```sql
# running in psql
CREATE DATABASE hr;
CREATE DATABASE

Connect to the `hr` database inside `psql`:

```
h`
```

```sql
psql (15.2.6, server 15.2.8)
You are now connected to database "hr" as user "enterprisedb".

Create columns to hold department numbers, unique department names, and locations:

```sql
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```
CREATE TABLE

Insert values into the `dept` table:

```sql
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

```sql
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
```

View the table data by selecting the values from the table:

```sql
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>

(2 rows)

6.1.2 Installing EDB Postgres Advanced Server on RHEL 8 or OL 8 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

To determine if your repository exists, enter this command:

```bash
dnf repolist | grep enterprisedb
```

If no output is generated, the repository isn't installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.
3. Select the platform and software that you want to download.
4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

```bash
```

- If you're also installing PostGIS, enable additional repositories to resolve dependencies:

```bash
ARCH=$( /bin/arch ) subscription-manager repos --enable "codeready-builder-for-rhel-8-${ARCH}-rpms"
```

**Note**

If you're using a public cloud RHEL image, `subscription manager` might not be enabled. Enabling it might incur unnecessary charges. Equivalent packages might be available under a different name, such as `codeready-builder-for-rhel-8-rhui-rpms`. To determine how to install `codeready-builder`, consult the documentation for the RHEL image you're using.
Install the package

```
sudo dnf -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```
sudo dnf -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Installing the server package creates an operating system user named enterprisedb. The user is assigned a user ID (UID) and a group ID (GID). The user has no default password. Use the `passwd` command to assign a password for the user. The default shell for the user is `bash`, and the user's home directory is `/var/lib/edb/as16`.

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see Initializing the cluster in Postgres mode.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
```

```
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
```

```
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see Modifying the `pg_hba.conf` file.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```

```
CREATE DATABASE
```

Connect to the `hr` database inside `psql`:

```
\c hr
```

```
psql (15.2.6, server 15.2.8)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```
CREATE TABLE

Insert values into the `dept` table:

```sql
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

```sql
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
```

View the table data by selecting the values from the table:

```sql
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>

(2 rows)

6.1.3 Installing EDB Postgres Advanced Server on AlmaLinux 9 or Rocky Linux 9 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

To determine if your repository exists, enter this command:

```
dnf repolist | grep enterprisedb
```

If no output is generated, the repository isn't installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.
3. Select the platform and software that you want to download.
4. Follow the instructions for setting up the EDB repository.

- Install the EPel repository:
  ```
sudo dnf -y install epel-release
  ```

- Enable additional repositories to resolve dependencies:
  ```
sudo dnf config-manager --set-enabled powertools
  ```

Install the package:

```
sudo dnf -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
To install an individual component:

```bash
sudo dnf -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Installing the server package creates an operating system user named enterprisedb. The user is assigned a user ID (UID) and a group ID (GID). The user has no default password. Use the `passwd` command to assign a password for the user. The default shell for the user is `bash`, and the user’s home directory is `~/var/lib/edb/as16`.

**Initial configuration**

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see Initializing the cluster in Postgres mode.

```bash
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
```

```bash
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```bash
sudo su - enterprisedb
```

```bash
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see Modifying the `pg_hba.conf` file.

```sql
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

**Experiment**

Now you’re ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```sql
# running in psql
CREATE DATABASE hr;
```

```sql
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```bash
\c hr
```

```bash
psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```sql
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```

```sql
CREATE TABLE
```

Insert values into the `dept` table:

```sql
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

```sql
# running in psql
CREATE DATABASE hr;
```

EDB Postgres Advanced Server (EPAS)
Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

| deptno | dname    | loc     |
|--------+----------+---------|
| 10     | ACCOUNTING | NEW YORK |
| 20     | RESEARCH  | DALLAS  |

(2 rows)

### 6.1.4 Installing EDB Postgres Advanced Server on AlmaLinux 8 or Rocky Linux 8 x86_64

#### Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don’t need to perform this step.

  To determine if your repository exists, enter this command:

  ```bash
dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn’t installed.

  To set up the EDB repository:

  1. Go to [EDB repositories](#).
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```bash
  sudo dnf -y install epel-release
  ```

- Enable additional repositories to resolve dependencies:

  ```bash
  sudo dnf config-manager --set-enabled powertools
  ```

#### Install the package

```bash
sudo dnf -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you’re installing. For example, if you’re installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```bash
sudo dnf -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.
Installing the server package creates an operating system user named enterprisedb. The user is assigned a user ID (UID) and a group ID (GID). The user has no default password. Use the `passwd` command to assign a password for the user. The default shell for the user is `bash`, and the user's home directory is `/var/lib/edb/as16`.

### Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see [Initializing the cluster in Postgres mode](#).

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
```

```
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
```

```
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see [Modifying the pg_hba.conf file](#).

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

### Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```

```
CREATE DATABASE
```

Connect to the `hr` database inside `psql`:

```
\c hr
```

```
psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

```
INSERT 0 1
```

```
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
```

```
INSERT 0 1
```
6.1.5 Installing EDB Postgres Advanced Server on RHEL 7 or OL 7 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don’t need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn’t installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```
  ```

- Enable additional repositories to resolve dependencies:

  ```
  subscription-manager repos --enable "rhel-*-optional-rpms" --enable "rhel-*-extras-rpms" --enable "rhel-ha-for-rhel-*-server-rpms"
  ```

Install the package

```
sudo yum -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```
sudo yum -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Installing the server package creates an operating system user named enterprisedb. The user is assigned a user ID (UID) and a group ID (GID). The user has no default password. Use the `passwd` command to assign a password for the user. The default shell for the user is `bash`, and the user’s home directory is `/var/lib/edb/as<xx>`.

Initial configuration
Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see Initializing the cluster in Postgres mode.

```
sudo PGSETUP_INITDB_OPTIONS=-E UTF-8 /usr/edb/as16/bin/edb-as-16-setup initdb
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the `enterprisedb` user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, `enterprisedb`. For more information on changing the authentication, see Modifying the `pg_hba.conf` file.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

**Experiment**

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```

Connect to the `hr` database inside `psql`:

```
\c hr
```

You are now connected to database "hr" as user "enterprisedb".

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
    PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
    varchar(13));
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

```
INSERT 0 1
```

```
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
```

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

```
deptno |   dname    |   loc
--------+------------+----------
        |           |          
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
6.1.6 Installing EDB Postgres Advanced Server on CentOS 7 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

To determine if your repository exists, enter this command:

```
  dnf repolist | grep enterprisedb
```

If no output is generated, the repository isn't installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.
3. Select the platform and software that you want to download.
4. Follow the instructions for setting up the EDB repository.

- Install the EPET repository:

```
```

Install the package

```
  sudo yum -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```
  sudo yum -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Installing the server package creates an operating system user named `enterprisedb`. The user is assigned a user ID (UID) and a group ID (GID). The user has no default password. Use the `passwd` command to assign a password for the user. The default shell for the user is `bash`, and the user's home directory is `/var/lib/edb/as16`.

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see Initializing the cluster in Postgres mode.

```
  sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
  sudo systemctl start edb-as-16
```

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate
connection string.

```
sudo su - enterprisedb
psql edb
```

The server runs with the peer or ident permission by default. You can change the authentication method by modifying the pg_hba.conf file.

Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see [Modifying the pg_hba.conf file](#).

```sql
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

### Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```bash
# running in psql
CREATE DATABASE hr;
CREATE DATABASE
```

Connect to the `hr` database inside `psql`:

```bash
\c hr
```

```
psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```sql
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```

```
CREATE TABLE
```

Insert values into the `dept` table:

```sql
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

```
INSERT 0 1
```

```sql
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```sql
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>
```

---

### 6.1.7 Installing EDB Postgres Advanced Server on SLES 15 x86_64

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Setting up the repository is a one-time task. If you have already set up your repository, you don’t need to perform this step.

To determine if your repository exists, enter this command:

```bash
zypper lr -E | grep enterprisedb
```

If no output is generated, the repository isn’t installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.
3. Select the platform and software that you want to download.
4. Follow the instructions for setting up the EDB repository.

- Activate the required SUSE module:

  ```bash
  sudo SUSEConnect -p PackageHub/15.4/x86_64
  ```

- Refresh the metadata:

  ```bash
  sudo zypper refresh
  ```

Install the package

```bash
sudo zypper -n install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced server you are installing. For example, if you are installing version 16, the package name would be `edb-as16-server`.

To install an individual component:

```bash
sudo zypper -n install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see Initializing the cluster in Postgres mode.

```bash
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the `enterprisedb` user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```bash
sudo su - enterprisedb
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.
Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;

CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr

psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```

```
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT 0 1
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>

(2 rows)

6.1.8 Installing EDB Postgres Advanced Server on SLES 12 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.
To determine if your repository exists, enter this command:

```
zypper lr -E | grep enterprisedb
```

If no output is generated, the repository isn't installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.
3. Select the platform and software that you want to download.
4. Follow the instructions for setting up the EDB repository.

- Activate the required SUSE module:
  ```
sudo SUSEConnect -p PackageHub/12.5/x86_64
sudo SUSEConnect -p sle-sdk/12.5/x86_64
```

- Refresh the metadata:
  ```
sudo zypper refresh
```

Install the package

```
sudo zypper -n install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced server you are installing. For example, if you are installing version 16, the package name would be `edb-as16-server`.

To install an individual component:

```
sudo zypper -n install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see the cluster in Postgres mode.

```
sudo PGSETUP_INITDB_OPTIONS=-E UTF-8 /usr/edb/as16/bin/edb-as-16-setup initdb
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the `enterprisedb` user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, `enterprisedb`. For more information on changing the authentication, see Modifying the `pg_hba.conf` file.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

Experiment
Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```sql
CREATE DATABASE hr;
```

Connect to the `hr` database inside psql:

```sql
\c hr
```

psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".

Create columns to hold department numbers, unique department names, and locations:

```sql
CREATE TABLE public.dept (
    deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc varchar(13));
```

Insert values into the `dept` table:

```sql
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

```sql
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
```

View the table data by selecting the values from the table:

```sql
SELECT * FROM dept;
```

| deptno |   dname    |   loc   |
|--------+------------+---------|
| 10     | ACCOUNTING | NEW YORK|
| 20     | RESEARCH   | DALLAS  |

6.1.9 Installing EDB Postgres Advanced Server on Ubuntu 22.04 x86_64

**Prerequisites**

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don’t need to perform this step.

To determine if your repository exists, enter this command:

```sh
cd /var/lib/apt/lists

apt-cache search enterprisedb
```

If no output is generated, the repository isn’t installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.

3. Select the platform and software that you want to download.

4. Follow the instructions for setting up the EDB repository.

**Install the package**

```bash
sudo apt-get -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```bash
sudo apt-get -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

**Initial configuration**

This section steps you through getting started with your cluster including logging in, ensuring the installation was successful, connecting to your cluster, and creating the user password.

To work in your cluster, log in as the `enterprisedb` user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```bash
sudo su - enterprisedb
```
```
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, `enterprisedb`. For more information on changing the authentication, see [Modifying the `pg_hba.conf` file](#).

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

**Experiment**

Now you’re ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```

```
CREATE DATABASE
```

Connect to the `hr` database inside `psql`:

```
\c hr
```

```
psql (15.2.6, server 15.2.8)  
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```

```
CREATE TABLE
```

```
```
Insert values into the `dept` table:

```sql
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

View the table data by selecting the values from the table:

```sql
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>

(2 rows)

6.1.10 Installing EDB Postgres Advanced Server on Ubuntu 20.04 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don’t need to perform this step.

  To determine if your repository exists, enter this command:

  ```bash
  apt-cache search enterprisedb
  ```

  If no output is generated, the repository isn’t installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

Install the package

```bash
sudo apt-get -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you’re installing. For example, if you’re installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```bash
sudo apt-get -y install <package_name>
```

Where `package_name` can be any of the available packages from the available package list.

Initial configuration
This section steps you through getting started with your cluster including logging in, ensuring the installation was successful, connecting to your cluster, and creating the user password.

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, `enterprisedb`. For more information on changing the authentication, see [Modifying the pg_hba.conf file](#).

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

**Experiment**

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```

Connect to the `hr` database inside `psql`:

```
\c hr
```

```
psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
CREATE TABLE
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

```
INSERT 0 1
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

```
deptno | dname    | loc
--------+------------+----------
10      | ACCOUNTING | NEW YORK
20      | RESEARCH   | DALLAS
(2 rows)
```

**6.1.11 Installing EDB Postgres Advanced Server on Ubuntu 18.04 x86_64**
Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

To determine if your repository exists, enter this command:

```
apt-cache search enterprisedb
```

If no output is generated, the repository isn't installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.
3. Select the platform and software that you want to download.
4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```
sudo apt-get -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Initial configuration

This section steps you through getting started with your cluster including logging in, ensuring the installation was successful, connecting to your cluster, and creating the user password.

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see Modifying the `pg_hba.conf` file.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```
CREATE DATABASE

Connect to the hr database inside psql:

\c hr

psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".

Create columns to hold department numbers, unique department names, and locations:

CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));

CREATE TABLE

Insert values into the dept table:

INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');

INSERT 0 1

INSERT into dept VALUES (20,'RESEARCH','DALLAS');

INSERT 0 1

View the table data by selecting the values from the table:

SELECT * FROM dept;

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>

(2 rows)

6.1.12 Installing EDB Postgres Advanced Server on Debian 11 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

To determine if your repository exists, enter this command:

`apt-cache search enterprisedb`

If no output is generated, the repository isn't installed.

To set up the EDB repository:

1. Go to EDB repositories.

2. Select the button that provides access to the EDB repository.

3. Select the platform and software that you want to download.

4. Follow the instructions for setting up the EDB repository.
Install the package

foisuo apt-get -y install edb-as<xx>-server

Where <xx> is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.

To install an individual component:

foisuo apt-get -y install <package_name>

Where package_name can be any of the available packages from the available package list.

Initial configuration

This section steps you through getting started with your cluster including logging in, ensuring the installation was successful, connecting to your cluster, and creating the user password.

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

foisuo su - enterprisedb

psql edb

The server runs with the peer or ident permission by default. You can change the authentication method by modifying the pg_hba.conf file.

Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see Modifying the pg_hba.conf file.

ALTER ROLE enterprisedb IDENTIFIED BY password;

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named `hr` to hold human resource information.

# running in psql
CREATE DATABASE hr;

Connect to the hr database inside psql:

\c hr

psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".

Create columns to hold department numbers, unique department names, and locations:

CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));

CREATE TABLE

Insert values into the dept table:

INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');

INSERT 0 1
INSERT into dept VALUES (20,'RESEARCH','DALLAS');

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

```
<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>
```

6.1.13 Installing EDB Postgres Advanced Server on Debian 10 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

To determine if your repository exists, enter this command:

```
apt-cache search enterprisedb
```

If no output is generated, the repository isn't installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.
3. Select the platform and software that you want to download.
4. Follow the instructions for setting up the EDB repository.

Install the package

```
sudo apt-get -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```
sudo apt-get -y install <package_name>
```

Where `package_name` can be any of the available packages from the available package list.

Initial configuration

This section steps you through getting started with your cluster including logging in, ensuring the installation was successful, connecting to your cluster, and creating the user password.

To work in your cluster, log in as the `enterprisedb` user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
```
The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, `enterprisedb`. For more information on changing the authentication, see the `Modifying the pg_hba.conf file`.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

**Experiment**

Now you’re ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```bash
# running in psql
CREATE DATABASE hr;

psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".

CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));

INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');

SELECT * FROM dept;
```

```
<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>
(2 rows)
```

---

### 6.2 Installing EDB Postgres Advanced Server on Linux IBM Power (ppc64le)

Operating system-specific install instructions are described in the corresponding documentation:

**Red Hat Enterprise Linux (RHEL)**
6.2.1 Installing EDB Postgres Advanced Server on RHEL 9 ppc64le

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don't need to perform this step.

  To determine if your repository exists, enter this command:

  ```bash
dnf repolist | grep enterprisedb
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.
  2. Select the button that provides access to the EDB repository.
  3. Select the platform and software that you want to download.
  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```bash
  ```

- Refresh the cache:

  ```bash
  sudo dnf makecache
  ```

- If you are also installing PostGIS, enable additional repositories to resolve dependencies:

  ```bash
  ARCH=$( /bin/arch ) subscription-manager repos --enable "codeready-builder-for-rhel-9-${ARCH}-rpms"
  ```

  **Note**

  If you are using a public cloud RHEL image, `subscription-manager` may not be enabled and enabling it may incur unnecessary charges. Equivalent packages may be available under a different name such as `codeready-builder-for-rhel-9-rhui-rpms`. Consult the documentation for the RHEL image you are using to determine how to install `codeready-builder`.

Install the package

```bash
sudo dnf -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is `edb-as16-server`.  

EDB Postgres Advanced Server (EPAS)  
Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
To install an individual component:

```
sudo dnf -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Installing the server package creates an operating system user named `enterprisedb`. The user is assigned a user ID (UID) and a group ID (GID). The user has no default password. Use the `passwd` command to assign a password for the user. The default shell for the user is `bash`, and the user's home directory is `/var/lib/edb/as16`.

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see Initializing the cluster in Postgres mode.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
```

```
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the `enterprisedb` user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
```

```
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, `enterprisedb`. For more information on changing the authentication, see Modifying the `pg_hba.conf` file.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```

Connect to the `hr` database inside `psql`:

```
\c hr
```

```
psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```
6.2.2 Installing EDB Postgres Advanced Server on RHEL 8 ppc64le

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don’t need to perform this step.

  To determine if your repository exists, enter this command:

  ```bash
dnf repolist | grep enterprisedb
  ``

  If no output is generated, the repository isn’t installed.

  To set up the EDB repository:

  1. Go to EDB repositories.

  2. Select the button that provides access to the EDB repository.

  3. Select the platform and software that you want to download.

  4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```bash
  ```

- Refresh the cache:

  ```bash
  sudo dnf makecache
  ```

- If you’re also installing PostGIS, enable additional repositories to resolve dependencies:

  ```bash
  ARCH=$( /bin/arch ) subscription-manager repos --enable "codeready-builder-for-rhel-8-${ARCH}-rpm"s
  ```

  **Note**

  If you’re using a public cloud RHEL image, `subscription_manager` might not be enabled. Enabling it might incur unnecessary charges. Equivalent packages might be available under a different name, such as `codeready-builder-for-rhel-8-rhui-rpms`. To determine how to install `codeready-builder`, consult the documentation for the RHEL image you’re using.

  Install the package
sudo dnf -y install edb-as<xx>-server

Where <xx> is the version of the EDB Postgres Advanced Server you're installing. For example, if you're installing version 16, the package name is edb-as16-server.

To install an individual component:

sudo dnf -y install <package_name>

Where package_name can be any of the available packages from the available package list.

Installing the server package creates an operating system user named enterprisedb. The user is assigned a user ID (UID) and a group ID (GID). The user has no default password. Use the passwd command to assign a password for the user. The default shell for the user is bash, and the user's home directory is /var/lib/edb/as16.

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The edb-as-16-setup script creates a cluster in Oracle-compatible mode with the edb sample database in the cluster. To create a cluster in Postgres mode, see Initializing the cluster in Postgres mode.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the psql command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
psql edb
```

The server runs with the peer or ident permission by default. You can change the authentication method by modifying the pg_hba.conf file.

Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see Modifying the pg_hba.conf file.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use psql to create a database named hr to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```

```
psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```

```
CREATE TABLE
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved. 117
Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');

INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>

(2 rows)

### 6.2.3 Installing EDB Postgres Advanced Server on SLES 15 ppc64le

**Prerequisites**

Before you begin the installation process:

- **Set up the EDB repository.**
  
  Setting up the repository is a one-time task. If you have already set up your repository, you don’t need to perform this step.

  To determine if your repository exists, enter this command:

  ```
  zypper lr -E | grep enterprise\db
  ```

  If no output is generated, the repository isn't installed.

  To set up the EDB repository:

  1. Go to EDB repositories.

  2. Select the button that provides access to the EDB repository.

  3. Select the platform and software that you want to download.

  4. Follow the instructions for setting up the EDB repository.

- **Activate the required SUSE module:**

  ```
  sudo SUSEConnect -p PackageHub/15.4/ppc64le
  ```

- **Refresh the metadata:**

  ```
  sudo zypper refresh
  ```

**Install the package**

```
sudo zypper -n install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced server you are installing. For example, if you are installing version 16, the package name would be `edb-as16-server`.

To install an individual component:
Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see initializing the cluster in Postgres mode.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the `enterprisedb` user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, `enterprisedb`. For more information on changing the authentication, see Modifying the `pg_hba.conf` file.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```

Connect to the `hr` database inside `psql`:

```
\c hr
```

```
psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk
PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc
varchar(13));
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
```

```
INSERT 0 1
```

```
INSERT into dept VALUES (20,'RESEARCH','DALLAS');
```

6.2.4 Installing EDB Postgres Advanced Server on RHEL 7 ppc64le

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don’t need to perform this step.

To determine if your repository exists, enter this command:

```
dnf repolist | grep enterprisedb
```

If no output is generated, the repository isn’t installed.

To set up the EDB repository:

1. Go to EDB repositories.
2. Select the button that provides access to the EDB repository.
3. Select the platform and software that you want to download.
4. Follow the instructions for setting up the EDB repository.

- Install the EPEL repository:

  ```
  
  sudo dnf makecache
  
  subscription-manager repos --enable "rhel-*-optional-rpms" --enable "rhel-*-extras-rpms" --enable "rhel-ha-for-rhel-*-server-rpms"
  ```

Install the package:

```
sudo yum -y install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced Server you’re installing. For example, if you’re installing version 16, the package name is `edb-as16-server`.

To install an individual component:

```
sudo yum -y install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.
Installing the server package creates an operating system user named enterprisedb. The user is assigned a user ID (UID) and a group ID (GID). The user has no default password. Use the `passwd` command to assign a password for the user. The default shell for the user is `bash`, and the user's home directory is `/var/lib/edb/as16`.

### Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.

First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see [Initializing the cluster in Postgres mode](#).

```
sudo PGSETUP_INITDB_OPTIONS=":E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb
sudo systemctl start edb-as-16
```

To work in your cluster, log in as the enterprisedb user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb
psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, enterprisedb. For more information on changing the authentication, see [Modifying the pg_hba.conf file](#).

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

### Experiment

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```
# running in psql
CREATE DATABASE hr;
```

Connect to the `hr` database inside psql:

```
\c hr
```

```
psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
```
6.2.5 Installing EDB Postgres Advanced Server on SLES 12 ppc64le

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

  Setting up the repository is a one-time task. If you have already set up your repository, you don’t need to perform this step.

  To determine if your repository exists, enter this command:

  `zypper lr -E | grep enterprisedb`

  If no output is generated, the repository isn’t installed.

  To set up the EDB repository:

  1. Go to EDB repositories.

  2. Select the button that provides access to the EDB repository.

  3. Select the platform and software that you want to download.

  4. Follow the instructions for setting up the EDB repository.

- Activate the required SUSE module:

  `sudo SUSEConnect -p PackageHub/12.5/ppc64le`
  `sudo SUSEConnect -p sle-sdk/12.5/ppc64le`

- Refresh the metadata:

  `sudo zypper refresh`

Install the package

```
sudo zypper -n install edb-as<xx>-server
```

Where `<xx>` is the version of the EDB Postgres Advanced server you are installing. For example, if you are installing version 16, the package name would be `edb-as16-server`.

To install an individual component:

```
sudo zypper -n install <package_name>
```

Where `<package_name>` can be any of the available packages from the available package list.

Initial configuration

Getting started with your cluster involves logging in, ensuring the installation and initial configuration was successful, connecting to your cluster, and creating the user password.
First, you need to initialize and start the database cluster. The `edb-as-16-setup` script creates a cluster in Oracle-compatible mode with the `edb` sample database in the cluster. To create a cluster in Postgres mode, see Initializing the cluster in Postgres mode.

```
sudo PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as16/bin/edb-as-16-setup initdb

sudo systemctl start edb-as-16
```

To work in your cluster, log in as the `enterprisedb` user. Connect to the database server using the `psql` command-line client. Alternatively, you can use a client of your choice with the appropriate connection string.

```
sudo su - enterprisedb

psql edb
```

The server runs with the `peer` or `ident` permission by default. You can change the authentication method by modifying the `pg_hba.conf` file.

Before changing the authentication method, assign a password to the database superuser, `enterprisedb`. For more information on changing the authentication, see Modifying the `pg_hba.conf` file.

```
ALTER ROLE enterprisedb IDENTIFIED BY password;
```

**Experiment**

Now you're ready to create and connect to a database, create a table, insert data in a table, and view the data from the table.

First, use `psql` to create a database named `hr` to hold human resource information.

```
CREATE DATABASE hr;
```

Connect to the `hr` database inside `psql`:

```
\c hr

psql (15.2.0, server 15.2.0)
You are now connected to database "hr" as user "enterprisedb".
```

Create columns to hold department numbers, unique department names, and locations:

```
CREATE TABLE public.dept (deptno numeric(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY, dname varchar(14) CONSTRAINT dept_dname_uq UNIQUE, loc varchar(13));
```

Insert values into the `dept` table:

```
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
```

View the table data by selecting the values from the table:

```
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
</tbody>
</table>
6.3 EDB Postgres Advanced Server installation for Windows

With Windows, you can:

- Install EDB Postgres Advanced Server using graphical installation options available with the interactive wizard on Windows.
- Manage an EDB Postgres Advanced Server installation.

6.3.1 Installing EDB Postgres Advanced Server with the interactive installer

You can use the EDB Postgres Advanced Server interactive installer to install EDB Postgres Advanced Server on Windows. The interactive installer is available from the Downloads page on the EDB website.

You can invoke the graphical installer in different installation modes to perform an EDB Postgres Advanced Server installation. During the installation, the graphical installer copies a number of temporary files to the location specified by the `TEMP` environment variable. You can optionally specify an alternative location for the temporary files by modifying the value of the `TEMP` environment variable at the command line:

```bash
SET TEMP=<temp_file_location>
```

Where `<temp_file_location>` specifies the alternative location for the temporary files and must match the permissions with the `TEMP` environment variable.

**Note**

If you're invoking the installer to perform a system upgrade, the installer preserves the configuration options specified during the previous installation.

6.3.1.1 Performing a graphical installation on Windows

A graphical installation is a quick and easy way to install EDB Postgres Advanced Server on a Windows system. Use the wizard's screens to specify information about your system and system usage. After you complete the screens, the installer performs an installation based on the selections made during the setup process.

1. Assume administrator privileges, and double-click the `edb-as<xx>-server-xx.x.x-x-windows-x64` executable file, where `<xx>` is the EDB Postgres Advanced Server version number.

   **Note**

   To install EDB Postgres Advanced Server on some versions of Windows, you might need to right-click the file and select **Run as Administrator** from the context menu to invoke the installer with administrator privileges.

2. Select Next. The EnterpriseDB license agreement opens.
3. Carefully review the license agreement before selecting the appropriate option. Select **Next**.

The Installation Directory window opens.

By default, the EDB Postgres Advanced Server installation directory is:

```
C:\Program Files\edb\as<xx>
```

Where `<xx>` is the EDB Postgres Advanced Server version number.

4. Accept the default installation location and select **Next**. Alternatively, select the file browser icon to open the Browse For Folder dialog box to choose a different installation directory.

**Note**

Don't store the `data` directory of a production database on an NFS file system.

The Select Components window opens, which contains a list of optional components that you can install with the EDB Postgres Advanced Server setup wizard. You can omit a component from the EDB Postgres Advanced Server installation by clearing the check box next to its name.
The setup wizard can install the following components while installing EDB Postgres Advanced Server:

- Select **EDB Postgres Advanced Server** to install EDB Postgres Advanced Server.

- Select **StackBuilder Plus** to install that utility. The StackBuilder Plus utility is a graphical tool that can update installed products or download and add supporting modules (and the resulting dependencies) after your EDB Postgres Advanced Server setup and installation completes. See [Using StackBuilder Plus](#) for more information.

- The **Command Line Tools** option installs command line tools and supporting client libraries including these and others:
  - libpq
  - psql
  - EDB*Loader
  - ecpgPlus
  - pg_basebackup, pg_dump, and pg_restore
  - pg_bench

  **Note**

  The command line tools are required if you're installing EDB Postgres Advanced Server or pgAdmin 4.

5. After selecting the components you want to install, select **Next**. The Additional Directories window opens.

By default, the EDB Postgres Advanced Server data files are saved to:

C:\Program Files\edb<xx>\data

Where `<xx>` is the EDB Postgres Advanced Server version number.

The default location of the EDB Postgres Advanced Server **Write-Ahead Log (WAL) Directory** is:

C:\Program Files\edb<xx>\data
EDB Postgres Advanced Server uses write-ahead logs to promote transaction safety and speed transaction processing. When you make a change to a table, the change is stored in shared memory, and a record of the change is written to the write-ahead log. When you perform a COMMIT, EDB Postgres Advanced Server writes contents of the write-ahead log to disk.

6. Accept the default file locations, or use the file browser to select a different location. Select Next.

The EDB Postgres Advanced Server Dialect window opens. The server dialect specifies the compatibility features supported by EDB Postgres Advanced Server.

7. From the list, select a server dialect.

By default, EDB Postgres Advanced Server installs in Compatible with Oracle mode. You can choose between Compatible with Oracle and Compatible with PostgreSQL installation modes.

If you select Compatible with Oracle, the installation includes the following features:

- Data dictionary views that are compatible with Oracle databases.
- Oracle data type conversions.
- Date values displayed in a format compatible with Oracle syntax.
- Support for Oracle-styled concatenation rules. (If you concatenate a string value with a NULL value, the returned value is the value of the string.)
- Schemas (dbo and sys) compatible with Oracle databases added to the SEARCH_PATH.
- Support for the following Oracle built-in packages:

<table>
<thead>
<tr>
<th>Package</th>
<th>Functionality compatible with Oracle databases</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbes_alert</td>
<td>Provides the capability to register for, send, and receive alerts.</td>
</tr>
<tr>
<td>dbes_job</td>
<td>Provides the capability for creating, scheduling, and managing jobs.</td>
</tr>
<tr>
<td>dbes_lob</td>
<td>Provides the capability to manage on large objects.</td>
</tr>
<tr>
<td>dbes_output</td>
<td>Provides the capability to send messages to a message buffer or get messages from the message buffer.</td>
</tr>
<tr>
<td>dbes_pipe</td>
<td>Provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.</td>
</tr>
<tr>
<td>dbes_rls</td>
<td>Enables the implementation of Virtual Private Database on certain EDB Postgres Advanced Server database objects.</td>
</tr>
<tr>
<td>dbes_sql</td>
<td>Provides an application interface to the EDB dynamic SQL functionality.</td>
</tr>
<tr>
<td>dbes_utility</td>
<td>Provides various utility programs.</td>
</tr>
<tr>
<td>dbes_aqadm</td>
<td>Provides supporting procedures for Advanced Queueing functionality.</td>
</tr>
<tr>
<td>dbes_aq</td>
<td>Provides message queuing and processing for EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>dbes_profile</td>
<td>Collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a performance profiling session.</td>
</tr>
<tr>
<td>dbes_random</td>
<td>Provides a number of methods to generate random values.</td>
</tr>
<tr>
<td>dbes_redact</td>
<td>Enables redacting or masking of data that's returned by a query.</td>
</tr>
<tr>
<td>dbes_lock</td>
<td>Provides support for the DBMS_LOCK, SLEEP procedure.</td>
</tr>
<tr>
<td>dbes_schedule</td>
<td>Provides a way to create and manage jobs, programs, and job schedules.</td>
</tr>
</tbody>
</table>
dbms_crypto
Provides functions and procedures to encrypt or decrypt RAW, BLOB, or CLOB data. You can also use \texttt{DBMS_CRYPTO} functions to generate cryptographically strong random values.

dbms_mview
Provides a way to manage and refresh materialized views and their dependencies.

dbms_session
Provides support for the \texttt{DBMS_SESSION.SET_ROLE} procedure.

util_encode
Provides a way to encode and decode data.

util_http
Provides a way to use the HTTP or HTTPS protocol to retrieve information found at a URL.

util_file
Provides the capability to read from and write to files on the operating system’s file system.

util_smtp
Provides the capability to send emails over the simple mail transfer protocol (SMTP).

util_mail
Provides the capability to manage email.

util_url
Provides a way to escape illegal and reserved characters in a URL.

util_raw
Provides a way to manipulate or retrieve the length of raw data types.

This isn’t a comprehensive list of the compatibility features for Oracle included when EDB Postgres Advanced Server is installed in Compatible with Oracle mode. For more information, see the Database compatibility for Oracle developers built-in package.

If you choose to install in Compatible with Oracle mode, the EDB Postgres Advanced Server superuser name is enterprisedb.

If you select \texttt{Compatible with PostgreSQL}, EDB Postgres Advanced Server exhibits compatibility with PostgreSQL version 15. If you choose to install in Compatible with PostgreSQL mode, the default EDB Postgres Advanced Server superuser name is postgres. For detailed information about PostgreSQL functionality, see the PostgreSQL website.

8. After specifying a configuration mode, select \texttt{Next}. The Password window opens.

<table>
<thead>
<tr>
<th>Package</th>
<th>Functionality compatible with Oracle databases</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbms_crypto</td>
<td>Provides functions and procedures to encrypt or decrypt RAW, BLOB, or CLOB data. You can also use \texttt{DBMS_CRYPTO} functions to generate cryptographically strong random values.</td>
</tr>
<tr>
<td>dbms_mview</td>
<td>Provides a way to manage and refresh materialized views and their dependencies.</td>
</tr>
<tr>
<td>dbms_session</td>
<td>Provides support for the \texttt{DBMS_SESSION.SET_ROLE} procedure.</td>
</tr>
<tr>
<td>util_encode</td>
<td>Provides a way to encode and decode data.</td>
</tr>
<tr>
<td>util_http</td>
<td>Provides a way to use the HTTP or HTTPS protocol to retrieve information found at a URL.</td>
</tr>
<tr>
<td>util_file</td>
<td>Provides the capability to read from and write to files on the operating system’s file system.</td>
</tr>
<tr>
<td>util_smtp</td>
<td>Provides the capability to send emails over the simple mail transfer protocol (SMTP).</td>
</tr>
<tr>
<td>util_mail</td>
<td>Provides the capability to manage email.</td>
</tr>
<tr>
<td>util_url</td>
<td>Provides a way to escape illegal and reserved characters in a URL.</td>
</tr>
<tr>
<td>util_raw</td>
<td>Provides a way to manipulate or retrieve the length of raw data types.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server uses the password specified on the Password window for the database superuser. The specified password must confirm to any security policies on the EDB Postgres Advanced Server host.

9. After you enter a password in the \texttt{Password} field and confirm the password in the \texttt{Retype Password} field, select \texttt{Next}. The Additional Configuration window opens.
10. Use the fields on the Additional Configuration window to specify installation details:

- Use the Port field to specify the port number for EDB Postgres Advanced Server to listen to for connection requests from client applications. The default is 5444.
- If the Locale field is set to [Default locale], EDB Postgres Advanced Server uses the system locale as the working locale. From the list, select an alternative locale for EDB Postgres Advanced Server.
- By default, the setup wizard installs corresponding sample data for the server dialect specified by the compatibility mode (Oracle or PostgreSQL). If you don't want to install sample data, clear the check box next to Install sample tables and procedures.

11. After verifying the information on the Additional Configuration window, select Next. The Dynatune Dynamic Tuning: Server Utilization window opens.

The setup wizard helps with performance tuning by way of the Dynatune Dynamic Tuning feature. Dynatune functionality allows EDB Postgres Advanced Server to make optimal use of the system resources available on the host machine on which it's installed.

12. Use the options on the Server Utilization window to set the initial value of the edb_dynamute configuration parameter. The edb_dynamute configuration parameter determines how EDB Postgres Advanced Server allocates system resources.

- Select Development to set the value of edb_dynamute to 33. A low value dedicates the least amount of the host machine's resources to the database server. This selection is a good choice for a development machine.
- Select General Purpose to set the value of edb_dynamute to 66. A mid-range value dedicates a moderate amount of system resources to the database server. This setting is good for an application server with a fixed number of applications running on the same host as EDB Postgres Advanced Server.
- Select Dedicated to set the value of edb_dynamute to 100. A high value dedicates most of the system resources to the database server. This setting is a good choice for a dedicated server host.

(After the installation is complete, you can adjust the value of edb_dynamute by editing the postgresql.conf file, located in the data directory of your EDB Postgres Advanced Server installation. After editing the postgresql.conf file, you must restart the server for your changes to take effect.)

Select the appropriate setting for your system, and select Next. The Dynatune Dynamic Tuning: Workload Profile window opens.
13. Use the options on the Workload Profile window to specify the initial value of the `edb_dynatune_profile` configuration parameter. The `edb_dynatune_profile` parameter controls performance-tuning aspects based on the type of work that the server performs.

- Select Transaction Processing (OLTP systems) to specify an `edb_dynatune_profile` value of `oltp`. Recommended when EDB Postgres Advanced Server is supporting heavy online transaction processing.
- Select General Purpose (OLTP and reporting workloads) to specify an `edb_dynatune_profile` value of `mixed`. Recommended for servers that provide a mix of transaction processing and data reporting.
- Select Reporting (Complex queries or OLAP workloads) to specify an `edb_dynatune_profile` value of `reporting`. Recommended for database servers used for heavy data reporting.

After the installation is complete, you can adjust the value of `edb_dynatune_profile` by editing the `postgresql.conf` file, located in the `data` directory of your EDB Postgres Advanced Server installation. After editing the `postgresql.conf` file, you must restart the server for your changes to take effect.

For more information about `edb_dynatune` and other performance-related topics, see the Dynatune.

14. Select Next.

By default, EDB Postgres Advanced Server is configured to start the service when the system boots. The Pre Installation Summary opens.

The Pre Installation Summary provides an overview of the options specified during the setup process.

15. Review the options before selecting Next. (Use Back to navigate back through the screens and update any options.)

The Ready to Install window confirms that the installer has the information it needs about your configuration preferences to install EDB Postgres Advanced Server.

16. Select Next.
As each supporting module is unpacked and installed, the module's installation is confirmed with a progress bar.

Before the setup wizard completes the EDB Postgres Advanced Server installation, it prompts: **Launch StackBuilder Plus at exit?**

EDB Postgres StackBuilder Plus is included with the installation of EDB Postgres Advanced Server and its core supporting components. StackBuilder Plus is a graphical tool that can update installed products or download and add supporting modules and the resulting dependencies after your EDB Postgres Advanced Server setup and installation completes. See **Using StackBuilder Plus** for more information.
To complete the EDB Postgres Advanced Server installation, clear the StackBuilder Plus check box and select Finish. Alternatively, accept the default and proceed to StackBuilder Plus.

6.3.1.2 Invoking the graphical installer from the command line

The command line options of the EDB Postgres Advanced Server installer offer functionality for Windows systems that reside in situations where a graphical installation might not work because of limited resources or privileges. You can:

- Include the `--mode unattended` option when invoking the installer to perform an installation without user input.
- Invoke the installer with the `--extract-only` option to perform a minimal installation when you don’t hold the privileges required to perform a complete installation.

Not all command line options are suitable for all situations. For a complete reference guide to the command line options, see Reference - Command Line Options.

**Note**

If you're invoking the installer from the command line to perform a system upgrade, the installer ignores command line options and preserves the configuration of the previous installation.

6.3.1.2.1 Performing an unattended installation

To specify that the installer should run without user interaction, include the `--mode unattended` command line option. In unattended mode, the installer uses one of the following sources for configuration parameters:

- Command line options (specified when invoking the installer)
- Parameters specified in an option file
- EDB Postgres Advanced Server installation defaults

You can embed the non-interactive EDB Postgres Advanced Server installer within another application installer; during the installation process, a progress bar allows the user to view the progression of the installation.

You must have administrative privileges to install EDB Postgres Advanced Server using the `--mode unattended` option. If you are using the `--mode unattended` option to install EDB Postgres Advanced Server with a client, the calling client must be invoked with superuser or administrative privileges.

To start the installer in unattended mode, navigate to the directory that contains the executable file, and enter:

```
edb-as<xx>-server-<xx>.<xx>-<xx>-windows-x64.exe --mode unattended --superpassword database_superuser_password --servicepassword system_password
```

Where `<xx>` is the EDB Postgres Advanced Server version number.

When invoking the installer, include the `--servicepassword` option to specify an operating system password for the user installing EDB Postgres Advanced Server.

Use the `--superpassword` option to specify a password that conforms to the password security policies defined on the host; enforced password policies on your system may not accept the default password (enterprisedb).

6.3.1.2.2 Performing an installation with limited privileges

To perform an abbreviated installation of EDB Postgres Advanced Server without access to administrative privileges, invoke the installer from the command line and include the `--extract-only` option. The `--extract-only` option extracts the binary files in an unaltered form, allowing you to experiment with a minimal installation of EDB Postgres Advanced Server.

If you invoke the installer with the `--extract-only` options, you can either manually create a cluster and start the service, or run the installation script. To manually create the cluster, you must:

- Use initdb to initialize the cluster
- Configure the cluster
- Use pg_ctl to start the service

For more information about initdb and pg_ctl, see the PostgreSQL core documentation for `initdb` and `pg_ctl`.

If you include the `--extract-only` option, the installer steps through a shortened form of the setup wizard. During the brief installation process, the installer generates an installation script that can be later used to complete a more complete installation. You must have administrative privileges to invoke the installation script.

The installation script:

- Initializes the database cluster if the cluster is empty.
- Configures the server to start at boot time.
- Establishes initial values for Dynatune (dynamic tuning) variables.
The scripted EDB Postgres Advanced Server installation doesn't create menu shortcuts or provide access to EDB Postgres StackBuilder Plus, and it doesn't make changes to registry files.

To perform a limited installation and generate an installation script:

1. Download and unpack the EDB Postgres Advanced Server installer.

2. Navigate to the directory that contains the installer, and invoke the installer:

   `edb-as<xx>-server-xx.x.x-x-windows.exe --extract-only yes`

   Where `<xx>` is the EDB Postgres Advanced Server version number.

   A dialog box opens, prompting you to choose an installation language.

3. Select a language for the installation, and select OK. The setup wizard opens.

4. Select Next.

   On Windows, the default EDB Postgres Advanced Server installation directory is `C:\Program Files\edb\as15`.

5. Accept the default installation location and select Next to continue to the Ready to Install window. Alternatively, you can select the file browser to choose a different installation directory.
To proceed with the EDB Postgres Advanced Server installation, select **Next**. During the installation, progress bars and popups mark the installation progress. The installer notifies you when the installation is complete.

After completing the minimal installation, you can execute a script to initialize a cluster and start the service. The script is by default located in `C:\Program Files\edb`.

### 6.3.1.2.3 Installation command line options

When invoking the EDB Postgres Advanced Server installer, you can optionally include the following parameters for an EDB Postgres Advanced Server installation on the command line or in a configuration file.

```bash
--create_samples { yes | no }
```

Specifies whether to create the sample tables and procedures for the database dialect specified with the `--databasemode` parameter. The default is `yes`.

```bash
--databasemode { oracle | postgresql }
```

Specifies a database dialect. The default is `oracle`.

```bash
--datadir data_directory
```

Specifies a location for the cluster's data directory. `data_directory` is the name of the directory. Include the complete path to the desired directory.

```bash
--debuglevel { 0 | 1 | 2 | 3 | 4 }
```

Sets the level of detail written to the `debug_log` file (see `--debugtrace`). Higher values produce more detail in a longer trace file. The default is `2`. 
**--debugtrace debug_log**

Helps with troubleshooting installation problems. `debug_log` is the name of the file that contains troubleshooting details.

**--disable-components component_list**

Specifies a list of EDB Postgres Advanced Server components to exclude from the installation. By default, `component_list` contains "" (the empty string). `component_list` is a comma-separated list containing one or more of the following components:

- `dbserver` — EDB Postgres Advanced Server.
- `pgadmin4` — The EDB Postgres pgAdmin 4 provides a powerful graphical interface for database management and monitoring.

**--enable_acledit { 1 | 0 }**

The `--enable_acledit 1` option grants permission to the user specified by the `--serviceaccount` option to access the EDB Postgres Advanced Server binaries and data directory. By default, this option is disabled if `--enable_acledit 0` is specified or if the `--enable_acledit` option is completely omitted.

**Note**

Specifying this option is valid only when installing on Windows. Specify the `--enable_acledit 1` option when a discretionary access control list (DACL) needs to be set for allowing access to objects on which to install EDB Postgres Advanced Server. For information on a DACL, see DACLs and ACEs in the Microsoft documentation.

To perform future operations, such as upgrading EDB Postgres Advanced Server, the service account user specified by the `--serviceaccount` option must have access to the data directory. The `--enable_acledit 1` option provides access to the data directory by the service account user.

**--enable-components component_list**

Although this option is listed when you run the installer with the `--help` option, the `--enable-components` parameter has no effect on the components installed. All components are installed regardless of this setting. To install only specific selected components, use the `--disable-components` parameter to list the components you don’t want to install.

**--extract-only { yes | no }**

Include the `--extract-only` parameter to extract the EDB Postgres Advanced Server binaries without performing a complete installation. Superuser privileges aren’t required for the `--extract-only` option. The default value is `no`.

**--help**

Displays a list of the optional parameters.

**--installer-language { en | ja | zh_CN | zh_TW | ko }**

Specifies an installation language for EDB Postgres Advanced Server.

- `en` specifies English. This is the default.
- `ja` specifies Japanese
- `zh_CN` specifies Chinese Simplified.
- `zh_TW` specifies Traditional Chinese.
- `ko` specifies Korean.

**--install_runtimes { yes | no }**

Specifies whether to install the Microsoft Visual C++ runtime libraries. The default is `yes`.

**--locale locale**

Specifies the locale for the EDB Postgres Advanced Server cluster. By default, the installer uses the locale detected by `initdb`.

**--mode { unattended }**

Specifies to perform an installation that requires no user input during the installation process.

**--optionfile config_file**

Specifies the name of a file that contains the installation configuration parameters. `config_file` must specify the complete path to the configuration parameter file.
--prefix installation_dir/as14

Specifies an installation directory for EDB Postgres Advanced Server. The installer appends a version-specific subdirectory (that is, as14) to the specified directory. The default installation directory is:

C:\Program Files\edb\as14

--serverport port_number

Specifies a listener port number for EDB Postgres Advanced Server.

If you’re installing EDB Postgres Advanced Server in unattended mode and don’t specify a value using the --serverport parameter, the installer uses port 5444 or the first available port after port 5444 as the default listener port.

--server_utilization {33 | 66 | 100}

Specifies a value for the edb_dynatune configuration parameter. The edb_dynatune configuration parameter determines how EDB Postgres Advanced Server allocates system resources.

- A value of 33 is appropriate for a system used for development. A low value dedicates the least amount of the host machine’s resources to the database server.
- A value of 66 is appropriate for an application server with a fixed number of applications. A midrange value dedicates a moderate amount of system resources to the database server. This is the default value.
- A value of 100 is appropriate for a host machine that’s dedicated to running EDB Postgres Advanced Server. A high value dedicates most of the system resources to the database server.

When the installation is complete, you can adjust the value of edb_dynatune by editing the postgresql.conf file, located in the data directory of your EDB Postgres Advanced Server installation. After editing the postgresql.conf file, you must restart the server for the changes to take effect.

--serviceaccount user_account_name

Specifies the name of the user account that owns the server process.

- If --databasemode is set to oracle (the default), the default value of --serviceaccount is enterprisedb.
- If --databasemode is postgresql, the default value of --serviceaccount is postgres.

For security reasons, the --serviceaccount parameter must specify the name of an account that doesn’t hold administrator privileges.

If you specify both the --serviceaccount option and the --enable_acledit option when invoking the installer, the database service and pgAgent uses the same service account. They therefore have the permissions required to access the EDB Postgres Advanced Server binaries and data directory.

Note

If you don’t include the --serviceaccount option when invoking the installer, the NetworkService account owns the database service, and the pgAgent service is owned by either enterprisedb or postgres, depending on the installation mode.

--servicename service_name

Specifies the name of the EDB Postgres Advanced Server service. The default is edb-as-14.

--servicepassword user_password

Specifies the OS system password. If unspecified, the value of --servicepassword defaults to the value of --superpassword.

--superaccount super_user_name

Specifies the user name of the database superuser.

- If --databasemode is set to oracle (the default), the default value of --superaccount is enterprisedb.
- If --databasemode is set to postgresql, the default value of --superaccount is postgres.

--superpassword superuser_password

Specifies the database superuser password. If you’re installing in non-interactive mode, --superpassword defaults to enterprisedb.

--unattendedmodeui { none | minimal | minimalWithDialogs }

Specifies installer behavior during an unattended installation.

Include --unattendedmodeui none if you don’t want to display progress bars during the EDB Postgres Advanced Server installation.

Include --unattendedmodeui minimal to display progress bars during the installation process. This is the default behavior.

Include --unattendedmodeui minimalWithDialogs to display progress bars and report any errors encountered during the installation process in additional dialog boxes.
Retrieves version information about the installer.

Specifies an initial value for the `edb_dynatune_profile` configuration parameter. `edb_dynatune_profile` controls aspects of performance-tuning based on the type of work that the server performs.

- Specify `oltp` if the EDB Postgres Advanced Server installation is used to support heavy online transaction processing workloads. This is the default value.
- Specify `mixed` if EDB Postgres Advanced Server provides a mix of transaction processing and data reporting.
- Specify `reporting` if EDB Postgres Advanced Server is used for heavy data reporting.

After the installation is complete, you can adjust the value of `edb_dynatune_profile` by editing the `postgresql.conf` file, located in the `data` directory of your EDB Postgres Advanced Server installation. After editing the `postgresql.conf` file, you must restart the server for the changes to take effect.

For more information about `edb_dynatune` and other performance-related topics, see Managing performance.

### 6.3.1.3 Setting cluster preferences during a graphical installation

During an installation, the graphical installer invokes the PostgreSQL `initdb` utility to initialize a cluster. If you are using the graphical installer, you can use the `INITDBOPTS` environment variable to specify your `initdb` preferences. Before invoking the graphical installer, set the value of `INITDBOPTS` at the command line, specifying one or more cluster options. For example:

```bash
SET INITDBOPTS=-k -E=UTF-8
```

If you specify values in `INITDBOPTS` that are also provided by the installer (such as the `-D` option, which specifies the installation directory), the value specified in the graphical installer supersedes the value if specified in `INITDBOPTS`.

For more information about using `initdb` cluster configuration options, see the PostgreSQL core documentation. In addition to the cluster configuration options documented in the PostgreSQL core documentation, EDB Postgres Advanced Server supports the following `initdb` options:

- `--no-redwood-compat`
- `--no-redwood-compat`: instructs the server to create the cluster in PostgreSQL mode. When the cluster is created in PostgreSQL mode, the name of the database superuser is `postgres` and the name of the default database is `postgres`. A small subset of Advanced Server features compatible with Oracle databases are available with this mode. However, we recommend using Advanced Server in redwood compatibility mode to have access to all its Oracle compatibility features.

- `--redwood-like`
- `--redwood-like`: instructs the server to use an escape character (an empty string `''`) following the `LIKE` (or PostgreSQL compatible `ILIKE`) operator in a SQL statement that is compatible with Oracle syntax.

- `--icu-short-form`
- `--icu-short-form`: creates a cluster that uses a default ICU (International Components for Unicode) collation for all databases in the cluster. For more information about Unicode collations, see Basic Unicode collation algorithm concepts.

### 6.3.1.4 Using StackBuilder Plus

The StackBuilder Plus utility provides a graphical interface that simplifies the process of updating, downloading, and installing modules that complement your EDB Postgres Advanced Server installation. When you install a module with StackBuilder Plus, StackBuilder Plus resolves any software dependencies.

You can invoke StackBuilder Plus any time after the installation is complete by selecting Apps > StackBuilder Plus. If prompted, enter your system password, and the StackBuilder Plus welcome window opens.
From the list, select your EDB Postgres Advanced Server installation.

StackBuilder Plus requires Internet access. If your installation of EDB Postgres Advanced Server is behind a firewall with restricted internet access, StackBuilder Plus can download program installers through a proxy server. The module provider determines if the module can be accessed through an HTTP proxy or an FTP proxy. Currently, all updates are transferred by way of an HTTP proxy. The FTP proxy information isn't used.

If the selected EDB Postgres Advanced Server installation has restricted Internet access, on the welcome window, select **Proxy Servers** to open the Proxy Servers dialog box.

In the Proxy Servers dialog box, in the **HTTP proxy** field enter the IP address and port number of the proxy server. Currently, all StackBuilder Plus modules are distributed by way of HTTP proxy, and FTP proxy information is ignored. Select **OK**.

The tree control on the StackBuilder Plus module selection window displays a node for each module category.

To review a detailed description of the component, expand a module and select a component name in the tree control. To add components to the installation or to upgrade a component, select the check box to the left of a module name and select **Next**.

StackBuilder Plus confirms the packages selected.
Use the browse icon (…) to the right of the Download directory field to open a file selector and choose an alternative location to store the downloaded installers. Select Next to connect to the server and download the required installation files.

When the download completes, a window opens that confirms the installation files were downloaded and are ready for installation.

To start the installation process, leave the Skip Installation box cleared and select Next. To exit StackBuilder Plus without installing the downloaded files, select the check box, and select Next.

Each downloaded installer has different requirements. As the installers execute, they might prompt you to confirm acceptance of license agreements, enter passwords, and provide configuration information.

During the installation process, you might be prompted by one or more of the installers to restart your system. Select No or Restart Later until all installations are completed. When the last installation has completed, restart the system to apply all of the updates.

You might occasionally encounter packages that don’t install successfully. If a package fails to install, StackBuilder Plus alerts you to the installation error and writes a message to the log file at %TEMP%.

When the installation is complete, StackBuilder Plus alerts you to the success or failure of the installations of the requested packages. If you were prompted by an installer to restart your computer, restart now.
6.3.2 Managing an EDB Postgres Advanced Server installation

Unless otherwise noted, the commands and paths for managing an EDB Postgres Advanced Server assume that you performed an installation with the interactive installer.

6.3.2.1 Starting and stopping EDB Postgres Advanced Server

A service is a program that runs in the background and requires no user interaction. A service has no user interface. You can configure a service to start at boot time or manually on demand. Services are best controlled using the platform-specific operating system service control utility. Many of the EDB Postgres Advanced Server supporting components are services.

The following table lists the names of the services that control EDB Postgres Advanced Server and services that control EDB Postgres Advanced Server supporting components:

<table>
<thead>
<tr>
<th>EDB Postgres Advanced Server component name</th>
<th>Windows service name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDB Postgres Advanced Server</td>
<td>edb-as-14</td>
</tr>
<tr>
<td>pgAgent</td>
<td>EDB Postgres Advanced Server Scheduling Agent</td>
</tr>
<tr>
<td>PgBouncer</td>
<td>edb-pgbouncer-1.14</td>
</tr>
<tr>
<td>Slony</td>
<td>edb-slony-replication-14</td>
</tr>
</tbody>
</table>

You can use the command line or the Windows Services applet to control the EDB Postgres Advanced Server database server and the services of EDB Postgres Advanced Server's supporting components on a Windows host.

6.3.2.2 Using the Windows services applet

The Windows operating system includes a graphical service controller that offers control of EDB Postgres Advanced Server and the services associated with EDB Postgres Advanced Server components. Access the Services utility in the Administrative Tools section of the Windows control panel.
6.3.2.3 Using pg_ctl to control EDB Postgres Advanced Server

You can use the pg_ctl utility to control an EDB Postgres Advanced Server service from the command line on any platform. pg_ctl allows you to start, stop, or restart the EDB Postgres Advanced Server database server, reload the configuration parameters, or display the status of a running server. To invoke the utility, assume the identity of the cluster owner, navigate into the home directory of EDB Postgres Advanced Server, and issue the command:

```bash
./bin/pg_ctl -D <data_directory> <action>
```

Where:

- `data_directory` is the location of the data controlled by the EDB Postgres Advanced Server cluster.
- `action` specifies the action taken by the pg_ctl utility. Specify:
  - `start` to start the service.
  - `stop` to stop the service.
  - `restart` to stop and then start the service.
  - `reload` to send the server a SIGHUP signal, reloading configuration parameters.
  - `status` to discover the current status of the service.

For more information about using the pg_ctl utility or the command-line options available, see the PostgreSQL core documentation.

6.3.2.4 Controlling server startup behavior on Windows

You can use the Windows Services utility to control the startup behavior of the server. Right-click the name of the service you want to update, and select Properties from the context menu.

In the Properties dialog box, from the Startup type list, specify how the EDB Postgres Advanced Server service behaves when the host starts.
6.3.2.5 Configuring EDB Postgres Advanced Server

You can easily update parameters that determine the behavior of EDB Postgres Advanced Server and supporting components by modifying the following configuration files:

- The *postgresql.conf* file determines the initial values of EDB Postgres Advanced Server configuration parameters.
- The *pg_hba.conf* file specifies your preferences for network authentication and authorization.
- The *pg_ident.conf* file maps operating system identities (user names) to EDB Postgres Advanced Server identities (roles) when using `ident`-based authentication.

For more information about modifying the *postgresql.conf* and *pg_hba.conf* files, see Setting parameters.

You can use your editor of choice to open a configuration file. On Windows, you can navigate through the EDB Postgres menu to open a file.

### Setting EDB Postgres Advanced Server environment variables

The graphical installer provides a script that simplifies the task of setting environment variables for Windows users. The script sets the environment variables for your current shell session. When your shell session ends, the environment variables are destroyed. You might want to invoke `pgplus_env.bat` from your system-wide shell startup script, which defines environment variables for each shell session.

The `pgplus_env` script is created during the EDB Postgres Advanced Server installation process and reflects the choices made during installation. To invoke the script, at the command line, enter:

```
C:\Program Files\edb\as14\pgplus_env.bat
```
As the `pgplus_env.bat` script executes, it sets the following environment variables:

```
PATH="C:\Program Files\edb\as14\bin";%PATH%
EDBHOME=C:\Program Files\edb\as14
PGDATA=C:\Program Files\edb\as14\data
PGDATABASE=edb
REM @SET PGUSER=enterprisedb
PGPORT=5444
PGLOCALEDIR=C:\Program Files\edb\as14\share\locale
```

If you used an installer created by EDB to install PostgreSQL, the `pg_env` script performs the same function:

```
C:\Program Files\PostgreSQL\14\pg_env.bat
```

As the `pg_env.bat` script executes on PostgreSQL, it sets the following environment variables:

```
PATH="C:\Program Files\PostgreSQL\14\bin";%PATH%
PGDATA=C:\Program Files\PostgreSQL\14\data
PGDATABASE=postgres
PGUSER=postgres
PGPORT=5432
PGLOCALEDIR=C:\Program Files\PostgreSQL\14\share\locale
```

Connecting to EDB Postgres Advanced Server with `edb-psql`

`edb-psql` is a command-line client application that allows you to execute SQL commands and view the results. To open the `edb-psql` client, make sure the client is in your search path. The executable resides in the `bin` directory under your EDB Postgres Advanced Server installation.

Use the following command and options to start the `edb-psql` client:

```
psql -d edb -U enterprisedb
```

Where:

- `-d` specifies the database to which `edb-psql` connects.
- `-U` specifies the identity of the database user to use for the session.

If you performed an installation with the interactive installer, you can access the `edb-psql` client by selecting `EDB Postgres > EDB-PSQL`. When the client opens, provide connection information for your session.

The `edb-psql` file is a symbolic link to a binary called `psql`, a modified version of the PostgreSQL community `psql`, with added support for EDB Postgres Advanced Server features. For more information about using the command-line client, see the PostgreSQL core documentation.

Connecting to EDB Postgres Advanced Server with the pgAdmin 4 client

`pgAdmin 4` provides a graphical interface that you can use to manage your database and database objects. Dialog boxes and online help simplify tasks such as object creation, role management, and granting or revoking privileges. The tabbed browser panel provides quick access to information about the object currently selected in the `pgAdmin` tree control.

The client is distributed with the graphical installer. To open `pgAdmin`, select `EDB Postgres > pgAdmin4`. The client opens in your default browser.
To connect to the EDB Postgres Advanced Server database server, expand the Servers node of the Browser tree control, and right-click on the EDB Postgres Advanced Server node. From the context menu, select Connect Server. The Connect to Server dialog box opens.

To connect, in the Password field, provide the password associated with the database superuser, and select OK.
When the client connects, you can use the Browser tree control to retrieve information about existing database objects or to create new objects. For more information about using the pgAdmin client, use the Help menu.

6.4 Linux installation details

These additional use cases apply to Linux installations.

6.4.1 Default component locations

The package managers for the various Linux variations install EDB Postgres Advanced Server components in different locations. If you need to access the components after installation, see:

- RHEL/OL/Rocky Linux/AlmaLinux/CentOS/SLES locations
- Debian/Ubuntu locations

RHEL/OL/Rocky Linux/AlmaLinux/CentOS/SLES Locations

The RPM installers place EDB Postgres Advanced Server components in the directories listed in the table.

<table>
<thead>
<tr>
<th>Component</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executables</td>
<td>/usr/edb/as15/bin</td>
</tr>
<tr>
<td>Libraries</td>
<td>/usr/edb/as15/lib</td>
</tr>
<tr>
<td>Cluster configuration files</td>
<td>/var/lib/edb/as15</td>
</tr>
<tr>
<td>Documentation</td>
<td>/usr/edb/as15/share/doc</td>
</tr>
<tr>
<td>Contrib</td>
<td>/usr/edb/as15/share/contrib</td>
</tr>
<tr>
<td>Data</td>
<td>/var/lib/edb/as15/data</td>
</tr>
<tr>
<td>Logs</td>
<td>/var/log/edb/as15</td>
</tr>
<tr>
<td>Lock files</td>
<td>/var/lock/edb/as15</td>
</tr>
<tr>
<td>Backup area</td>
<td>/var/lib/edb/as15/backups</td>
</tr>
<tr>
<td>Templates</td>
<td>/usr/edb/as15/share</td>
</tr>
</tbody>
</table>
### Procedural Languages
- `/usr/edb/as15/lib` or `/usr/edb/as15/lib64`

### Development Headers
- `/usr/edb/as15/include`

### Shared data
- `/usr/edb/as15/share`

### Regression tests
- `/usr/edb/as15/lib/pgxs/src/test/regress`

### SGML Documentation
- `/usr/edb/as15/share/doc`

### Debian/Ubuntu Locations

The Debian package manager places EDB Postgres Advanced Server and supporting components in the directories listed in the table.

<table>
<thead>
<tr>
<th>Component</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td><code>/usr/lib/edb-as/15/</code></td>
</tr>
<tr>
<td>Data and Configuration</td>
<td><code>/var/lib/edb-as/15/main</code></td>
</tr>
<tr>
<td>Directory</td>
<td><code>/etc/edb-as/15/main/</code></td>
</tr>
<tr>
<td>pgAgent</td>
<td><code>/usr/lib/edb-as/15</code></td>
</tr>
<tr>
<td>Pgpool</td>
<td><code>/usr/edb/pgpool4.1/</code></td>
</tr>
<tr>
<td>Postgis</td>
<td><code>/usr/lib/edb-as/15/</code></td>
</tr>
<tr>
<td>PGSMMPD</td>
<td><code>/usr/lib/edb-as/15</code></td>
</tr>
<tr>
<td>Slony Replication</td>
<td><code>/usr/lib/edb-as/15</code></td>
</tr>
<tr>
<td>pgBouncer</td>
<td><code>/usr/edb/pgbouncer1.15/</code></td>
</tr>
<tr>
<td>pgBouncer Configuration Files</td>
<td><code>/etc/edb/pgbouncer1.15/pgbouncer.ini</code></td>
</tr>
<tr>
<td></td>
<td><code>/etc/edb/pgbouncer1.15/userlist.txt</code></td>
</tr>
<tr>
<td>SQL-Profiler</td>
<td><code>/usr/lib/edb-as/15/lib</code></td>
</tr>
<tr>
<td>SQL-Protect</td>
<td><code>/usr/lib/edb-as/15/lib</code></td>
</tr>
<tr>
<td>SSLUTILS</td>
<td><code>/usr/lib/edb-as/15/lib</code></td>
</tr>
<tr>
<td>PL-PERL</td>
<td><code>/usr/lib/edb-as/15/lib</code></td>
</tr>
<tr>
<td>PL-PYTHON</td>
<td><code>/usr/lib/edb-as/15/lib</code></td>
</tr>
<tr>
<td>PLTCL</td>
<td><code>/usr/lib/edb-as/15/lib</code></td>
</tr>
<tr>
<td>EFM</td>
<td><code>/usr/edb/efm-4.1/</code></td>
</tr>
<tr>
<td>JDBC</td>
<td><code>/usr/edb/jdbc</code></td>
</tr>
<tr>
<td>MTK</td>
<td><code>/usr/edb/migrationtoolkit/</code></td>
</tr>
</tbody>
</table>

### 6.4.2 Available native packages

EDB provides a number of native packages in the EDB repository. The packages vary slightly for the various Linux variations. See:

- RHEL/OL/Rocky Linux/AlmaLinux/CentOS/SLES packages
- Debian/Ubuntu packages

#### RHEL/OL/Rocky Linux/AlmaLinux/CentOS/SLES packages

EDB Postgres Advanced Server RPM packages

The tables that follow list the RPM packages that are available from EDB. You can also use the `yum search` or `dnf search` command to access a list of the packages that are currently available from your configured repository. At the command line, assume superuser privileges, and enter:

On RHEL or CentOS 7:

```bash
yum search <package>
```

On RHEL or Rocky Linux or AlmaLinux 8:

```bash
dnf search <package>
```
dnf search <package>

Where `<package>` is the search term that specifies the name or partial name of a package.

### Note

The available package list is subject to change.

<table>
<thead>
<tr>
<th>Package name</th>
<th>Installed by default</th>
<th>Package installs</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb-as16-server</td>
<td>Yes</td>
<td>Installs core components of the EDB Postgres Advanced Server database server.</td>
</tr>
<tr>
<td>edb-as16-server-client</td>
<td>Yes</td>
<td>Client programs and utilities that you can use to access and manage EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>edb-as16-server-contrib</td>
<td>Yes</td>
<td>Installs contributed tools and utilities that are distributed with EDB Postgres Advanced Server. Files for these modules are installed in:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Documentation: /usr/edb/as16/share/doc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Loadable modules: /usr/edb/as16/lib</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Binaries: /usr/edb/as16/bin</td>
</tr>
<tr>
<td>edb-as16-server-core</td>
<td>Yes</td>
<td>Includes the programs needed to create the core functionality behind the EDB Postgres Advanced Server database.</td>
</tr>
<tr>
<td>edb-as16-server-devel</td>
<td>Yes</td>
<td>Installs the header files and libraries needed to compile C or C++ applications that directly interact with an EDB Postgres Advanced Server server and the ecpg or ecpgPlus C preprocessor.</td>
</tr>
<tr>
<td>edb-as16-server-docs</td>
<td>Yes</td>
<td>Installs the readme file.</td>
</tr>
<tr>
<td>edb-as16-server-ebdb-modules</td>
<td>No</td>
<td>Installs supporting modules for EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>edb-as16-server-indexadvisor</td>
<td>No</td>
<td>Installs EDB Postgres Advanced Server's Index Advisor feature. The Index Advisor utility helps to determine the columns to index to improve performance in a given workload.</td>
</tr>
<tr>
<td>edb-as16-server-libs</td>
<td>Yes</td>
<td>Provides the essential shared libraries for any EDB Postgres Advanced Server client program or interface.</td>
</tr>
<tr>
<td>edb-as16-server-lvmjit</td>
<td>No</td>
<td>Contains support for just-in-time (JIT) compiling parts of EDB Postgres Advanced Server's queries. You need to install this package separately.</td>
</tr>
<tr>
<td>edb-as16-server-pdebugger</td>
<td>No</td>
<td>Implements an API for debugging PL/pgSQL functions on EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>edb-as16-server-plperl</td>
<td>Yes</td>
<td>Installs the PL/Perl procedural language for EDB Postgres Advanced Server. The <code>edb-as16-server-plperl</code> package depends on the platform-supplied version of Perl.</td>
</tr>
<tr>
<td>edb-as16-server-python3</td>
<td>Yes</td>
<td>Installs the PL/Python procedural language for EDB Postgres Advanced Server. The PL/Python2 support is no longer available in EDB Postgres Advanced Server version 15 and later.</td>
</tr>
<tr>
<td>edb-as16-server-p Tcl</td>
<td>Yes</td>
<td>Installs the PL/Tcl procedural language for EDB Postgres Advanced Server. The <code>edb-as16-p Tcl</code> package depends on the platform-supplied version of TCL.</td>
</tr>
<tr>
<td>edb-as16-server-sqlprofiler</td>
<td>No</td>
<td>Installs EDB Postgres Advanced Server's SQL Profiler feature. SQL Profiler helps identify and optimize SQL code.</td>
</tr>
<tr>
<td>edb-as16-server-sqlprotect</td>
<td>Yes</td>
<td>Installs EDB Postgres Advanced Server's SQL Protect feature. SQL Protect provides protection against SQL injection attacks.</td>
</tr>
<tr>
<td>edb-as16-server-sslutils</td>
<td>No</td>
<td>Installs functionality that provides SSL support.</td>
</tr>
<tr>
<td>edb-as16-server-cloneschema</td>
<td>No</td>
<td>Installs the EDB Clone Schema extension. For more information about EDB Clone Schema, see <a href="#">EDB clone schema</a>.</td>
</tr>
<tr>
<td>edb-as16-server-parallelclone</td>
<td>No</td>
<td>Installs functionality that supports the EDB Clone Schema extension.</td>
</tr>
<tr>
<td>edb-as16-pgagent</td>
<td>No</td>
<td>Installs pgAgent: Installs pgAgent, a job scheduler for EDB Postgres Advanced Server. Before installing this package, you must install EPEL repository. For detailed information about installing EPEL, see <a href="#">Linux installation troubleshooting</a>.</td>
</tr>
<tr>
<td>edb-as16-edbplus</td>
<td>No</td>
<td>The <code>edb-edbplus</code> package contains the files required to install the EDB<em>Plus command line client. EDB</em>Plus commands are compatible with Oracle's SQL*Plus.</td>
</tr>
<tr>
<td>edb-as16-pgpool41-extensions</td>
<td>No</td>
<td>Creates pgPool extensions required by the server for use with pgpool.</td>
</tr>
<tr>
<td>edb-as16-postgis3</td>
<td>No</td>
<td>Installs POSTGIS meta RPMs.</td>
</tr>
<tr>
<td>edb-as16-postgis3-core</td>
<td>No</td>
<td>Provides support for geographic objects to the PostgreSQL object-relational database. In effect, PostGIS &quot;spatially enables&quot; the PostgreSQL server, allowing it to be used as a backend spatial database for geographic information systems (GIS), much like ESRI's SDE or Oracle's Spatial extension.</td>
</tr>
<tr>
<td>Package name</td>
<td>Installed by default</td>
<td>Package installs</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>edb-as16-postgis3-docs</td>
<td>No</td>
<td>Installs PDF documentation of PostGIS.</td>
</tr>
<tr>
<td>edb-as16-postgis-jdbc</td>
<td>No</td>
<td>Installs the essential jdbc driver for PostGIS.</td>
</tr>
<tr>
<td>edb-as16-postgis3-utils</td>
<td>No</td>
<td>Installs the utilities for PostGIS.</td>
</tr>
<tr>
<td>edb-as16-postgis3-gui</td>
<td>No</td>
<td>Provides a GUI for PostGIS.</td>
</tr>
<tr>
<td>edb-as16-libicu</td>
<td>No</td>
<td>Contains the supporting library files.</td>
</tr>
</tbody>
</table>

The following table lists the packages for EDB Postgres Advanced Server supporting components.

<table>
<thead>
<tr>
<th>Package name</th>
<th>Package installs</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb-pgpool41</td>
<td>Contains the pgPool-II installer. The pgpool-II utility package acts as a middleman between client applications and server database servers. pgpool-II functionality is transparent to client applications. Client applications connect to pgpool-II instead of directly to EDB Postgres Advanced Server, and pgpool-II manages the connection. EDB supports the following pgpool-II features:  - Load balancing  - Connection pooling  - High availability  - Connection limits pgpool-II runs as a service on Linux systems, and isn’t supported on Windows systems.</td>
</tr>
<tr>
<td>edb-jdbc</td>
<td>The edb-jdbc package includes the .jar files needed for Java programs to access an EDB Postgres Advanced Server database.</td>
</tr>
<tr>
<td>edb-migrationtoolkit</td>
<td>The edb-migrationtoolkit package installs Migration Toolkit, facilitating migration to an EDB Postgres Advanced Server database from Oracle, PostgreSQL, MySQL, Sybase, and SQL Server.</td>
</tr>
<tr>
<td>edb-oci</td>
<td>The edb-oci package installs the EDB Open Client library, allowing applications that use the Oracle Call Interface API to connect to an EDB Postgres Advanced Server database.</td>
</tr>
<tr>
<td>edb-oci-devel</td>
<td>Installs the OCI include files. Install this package if you’re developing C/C++ applications that require these files.</td>
</tr>
<tr>
<td>edb-odbc</td>
<td>Installs the driver needed for applications to access an EDB Postgres Advanced Server system by way of ODBC.</td>
</tr>
<tr>
<td>edb-odbc-devel</td>
<td>Installs the ODBC include files. Install this package if you’re developing C/C++ applications that require these files.</td>
</tr>
<tr>
<td>edb-pgbouncer115</td>
<td>Contains PgBouncer (a lightweight connection pooler). This package requires the libevent package.</td>
</tr>
<tr>
<td>ppas-xdb</td>
<td>Contains the xDB installer. xDB provides asynchronous cross-database replication.</td>
</tr>
<tr>
<td>ppas-xdb-console</td>
<td>Provides support for xDB.</td>
</tr>
<tr>
<td>ppas-xdb-libs</td>
<td>Provides support for xDB.</td>
</tr>
<tr>
<td>ppas-xdb-publisher</td>
<td>Provides support for xDB.</td>
</tr>
<tr>
<td>ppas-xdb-subscriber</td>
<td>Provides support for xDB.</td>
</tr>
<tr>
<td>edb-pem</td>
<td>The edb-pem package installs the management tool that efficiently manages, monitors, and tunes large Postgres deployments from a single remote GUI console.</td>
</tr>
<tr>
<td>edb-pem-agent</td>
<td>The edb-pem-agent is an agent component of Postgres Enterprise Manager.</td>
</tr>
<tr>
<td>edb-pem-docs</td>
<td>Contains documentation for various languages, which are in HTML format.</td>
</tr>
<tr>
<td>edb-pem-server</td>
<td>Contains server components of Postgres Enterprise Manager.</td>
</tr>
<tr>
<td>pageman4</td>
<td>Installs all required components to run pageman4 in desktop and web modes. Pgman4 is a management tool for Postgres capable of hosting the Python application and presenting it to the user as a desktop application.</td>
</tr>
<tr>
<td>pageman4-desktop</td>
<td>Installs pageman4 for desktop mode only.</td>
</tr>
<tr>
<td>pageman4-web</td>
<td>Installs pageman4 for web mode only.</td>
</tr>
<tr>
<td>pageman4-server</td>
<td>Installs pageman4’s core server package.</td>
</tr>
<tr>
<td>edb-emf40</td>
<td>Installs EDB Failover Manager that adds fault tolerance to database clusters to minimize downtime when a primary database fails by keeping data online in high availability configurations.</td>
</tr>
<tr>
<td>edb-rs</td>
<td>A Java-based replication framework that provides asynchronous replication across Postgres and EDB Postgres Advanced Server database servers. It supports primary-standby, primary-primary, and hybrid configurations.</td>
</tr>
<tr>
<td>edb-rs-client</td>
<td>A Java-based command-line tool that’s used to configure and operate a replication network by way of different commands by interacting with the EPRS server.</td>
</tr>
<tr>
<td>edb-rs-datavalidator</td>
<td>A Java-based command-line tool that provides row- and column-level data comparison of a source and target database table. The supported RDBMS servers include PostgreSQL, EDB Postgres Advanced Server, Oracle, and MySQL Server.</td>
</tr>
<tr>
<td>edb-rs-libs</td>
<td>Contains certain libraries that are commonly used by EPRS server, EPRS client, and monitoring modules.</td>
</tr>
<tr>
<td>edb-rs-monitor</td>
<td>A Java-based application that provides monitoring capabilities to ensure a smooth functioning of the EPRS replication cluster.</td>
</tr>
<tr>
<td>edb-rs-server</td>
<td>A Java-based replication framework that provides asynchronous replication across Postgres and EDB Postgres Advanced Server database servers. It supports primary-standby, primary-primary, and hybrid configurations.</td>
</tr>
<tr>
<td>edb-bart</td>
<td>Installs the Backup and Recovery Tool (BART) to support online backup and recovery across local and remote PostgreSQL and EDB Postgres Advanced Server servers.</td>
</tr>
<tr>
<td>libevent-edb</td>
<td>Contains supporting library files.</td>
</tr>
</tbody>
</table>
### Debian/Ubuntu packages

#### EDB Postgres Advanced Server Debian packages

The table lists some of the Debian packages that are available from EDB. You can also use the `apt list` command to access a list of the packages that are currently available from your configured repository. At the command line, assume superuser privileges, and enter:

```
apt list edb*
```

**Note**

The available package list is subject to change.

<table>
<thead>
<tr>
<th>Package name</th>
<th>Installed by default</th>
<th>Package installs</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb-as16-server</td>
<td>Yes</td>
<td>Installs core components of the EDB Postgres Advanced Server database server.</td>
</tr>
<tr>
<td>edb-as16-server-client</td>
<td>Yes</td>
<td>Includes client programs and utilities that you can use to access and manage EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>edb-as16-server-core</td>
<td>Yes</td>
<td>Includes the programs needed to create the core functionality behind the EDB Postgres Advanced Server database.</td>
</tr>
<tr>
<td>edb-as16-server-dev</td>
<td>Yes</td>
<td>Contains the header files and libraries needed to compile C or C++ applications that directly interact with an EDB Postgres Advanced Server server and the ecpg or ecpgPlus C preprocessor.</td>
</tr>
<tr>
<td>edb-as16-server-doc</td>
<td>Yes</td>
<td>Installs the readme file.</td>
</tr>
<tr>
<td>edb-as16-server-edb-modules</td>
<td>No</td>
<td>Installs supporting modules for EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>edb-as16-server-indexadvisor</td>
<td>No</td>
<td>Installs EDB Postgres Advanced Server's Index Advisor feature. The Index Advisor utility helps to determine the columns to index to improve performance in a given workload.</td>
</tr>
<tr>
<td>edb-as16-server-pldebugger</td>
<td>No</td>
<td>Implements an API for debugging PL/pgSQL functions on EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>edb-as16-server-plpython3</td>
<td>Yes</td>
<td>Installs the PL/Python procedural language for EDB Postgres Advanced Server. PL/Python2 support is not available for EDB Postgres Advanced Server version 15 and later.</td>
</tr>
<tr>
<td>edb-as16-server-pltcl</td>
<td>Yes</td>
<td>Installs the PL/Tcl procedural language for EDB Postgres Advanced Server. The <code>edb-as16-pltcl</code> package depends on the platform-supplied version of TCL.</td>
</tr>
<tr>
<td>edb-as16-server-sqprofiler</td>
<td>No</td>
<td>Installs EDB Postgres Advanced Server's SQL Profiler feature. SQL Profiler helps identify and optimize SQL code.</td>
</tr>
<tr>
<td>edb-as16-server-sqlprotect</td>
<td>Yes</td>
<td>Installs EDB Postgres Advanced Server's SQL Protect feature. SQL Protect provides protection against SQL injection attacks.</td>
</tr>
<tr>
<td>edb-as16-server-sslutilities</td>
<td>No</td>
<td>Installs functionality that provides SSL support.</td>
</tr>
<tr>
<td>edb-as16-server-cloneschema</td>
<td>No</td>
<td>Installs the EDB Clone Schema extension. For more information about EDB Clone Schema, see <a href="#">EDB Clone Schema</a>.</td>
</tr>
<tr>
<td>edb-as16-server-parallelclone</td>
<td>No</td>
<td>Installs functionality that supports the EDB Clone Schema extension.</td>
</tr>
<tr>
<td>edb-as16-edplus</td>
<td>No</td>
<td>Contains the files required to install the EDB<em>Plus command-line client. EDB</em>Plus commands are compatible with Oracle’s SQL*Plus.</td>
</tr>
<tr>
<td>edb-as16-pgadmin4</td>
<td>No</td>
<td>pgAdmin 4 provides a graphical management interface for EDB Postgres Advanced Server and PostgreSQL databases.</td>
</tr>
<tr>
<td>edb-as16-pgadmin-apache</td>
<td>No</td>
<td>Apache support module for pgAdmin 4.</td>
</tr>
<tr>
<td>edb-as16-pgadmin-common</td>
<td>No</td>
<td>pgAdmin 4 supporting files.</td>
</tr>
<tr>
<td>Package name</td>
<td>Installed by default</td>
<td>Package installs</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>edb-as16-pgadmin4-doc</td>
<td>No</td>
<td>pgAdmin 4 documentation module.</td>
</tr>
<tr>
<td>edb-as16-pgpool41-extensions</td>
<td>No</td>
<td>Creates pgPool extensions required by the server.</td>
</tr>
<tr>
<td>edb-as16-postgis3</td>
<td>No</td>
<td>Installs POSTGIS support for geospatial data.</td>
</tr>
<tr>
<td>edb-as16-postgis3-scripts</td>
<td>No</td>
<td>Installs POSTGIS support for geospatial data.</td>
</tr>
<tr>
<td>edb-as16-postgis3-doc</td>
<td>No</td>
<td>Provides support for POSTGIS.</td>
</tr>
<tr>
<td>edb-as16-postgis3-gui</td>
<td>No</td>
<td>Provides a GUI for POSTGIS.</td>
</tr>
<tr>
<td>edb-as16-postgis-jdbc</td>
<td>No</td>
<td>Provides support for POSTGIS.</td>
</tr>
<tr>
<td>edb-as16-postgis-scripts</td>
<td>No</td>
<td>Provides support for POSTGIS.</td>
</tr>
<tr>
<td>edb-as16-pgagent</td>
<td>No</td>
<td>Installs pgAgent, a job scheduler for EDB Postgres Advanced Server. Before installing this package, you must install the EPEL repository. For detailed information about installing EPEL, see <a href="#">Linux installation troubleshooting</a>.</td>
</tr>
<tr>
<td>edb-as16-hdfs-fdw</td>
<td>No</td>
<td>The Hadoop Data Adapter allows you to query and join data from Hadoop environments with your Postgres or EDB Postgres Advanced Server instances. It's YARN Ready certified with HortonWorks and provides optimizations for performance with predicate pushdown support.</td>
</tr>
<tr>
<td>edb-as16-hdfs-fdw-doc</td>
<td>No</td>
<td>Documentation for the Hadoop Data Adapter.</td>
</tr>
<tr>
<td>edb-as16-mongo-fdw</td>
<td>No</td>
<td>EDB Postgres Advanced Server extension that implements a foreign data wrapper for MongoDB.</td>
</tr>
<tr>
<td>edb-as16-mongo-fdw-doc</td>
<td>No</td>
<td>Documentation for the foreign data wrapper for MongoDB.</td>
</tr>
<tr>
<td>edb-as16-mysql-fdw</td>
<td>No</td>
<td>EDB Postgres Advanced Server extension that implements a foreign data wrapper for MySQL.</td>
</tr>
<tr>
<td>edb-pgpool41</td>
<td>No</td>
<td>Contains the pgPool-II installer. The pgpool-II utility package acts as a middleman between client applications and server database servers. pgpool-II functionality is transparent to client applications. Client applications connect to pgpool-II instead of directly to EDB Postgres Advanced Server, and pgpool-II manages the connection. EDB supports the following pgpool-II features: - Load balancing - Connection pooling - High availability - Connection limits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pgpool-II runs as a service on Linux systems, and isn't supported on Windows systems.</td>
</tr>
<tr>
<td>edb-jdbc</td>
<td>No</td>
<td>Includes the .jar files needed for Java programs to access an EDB Postgres Advanced Server database.</td>
</tr>
<tr>
<td>edb-migrationtoolkit</td>
<td>No</td>
<td>Installs Migration Toolkit, facilitating migration to an EDB Postgres Advanced Server database from Oracle, PostgreSQL, MySQL, Sybase, and SQL Server.</td>
</tr>
<tr>
<td>edb-pgbouncer115</td>
<td>No</td>
<td>PgBouncer, a lightweight connection pooler. This package requires the <a href="#">libevent</a> package.</td>
</tr>
<tr>
<td>edb-efm40</td>
<td>No</td>
<td>Installs EDB Failover Manager, which adds fault tolerance to database clusters to minimize downtime when a primary database fails by keeping data online in high availability configurations.</td>
</tr>
</tbody>
</table>

### 6.4.3 Updating an RPM installation

If you have an existing EDB Postgres Advanced Server RPM installation, you can use yum or dnf to upgrade your repository configuration file and update to a more recent product version.

To update the `edb.repo` file, assume superuser privileges and enter:

- On RHEL or CentOS 7:
  ```bash
  yum upgrade edb-repo
  ```

- On RHEL or Rocky Linux or AlmaLinux 8:
  ```bash
  dnf upgrade edb-repo
  ```

Yum or dnf updates the `edb.repo` file to enable access to the current EDB repository, configured to connect with the credentials specified in your `edb.repo` file. Then, you can use yum or dnf to upgrade all packages whose names include the expression `edb`.

- On RHEL or CentOS 7:
yum upgrade edb*

- On RHEL or Rocky Linux or AlmaLinux 8:

   dnf upgrade edb*

Note

The `yum upgrade` or `dnf upgrade` commands perform an update only between minor releases. To update between major releases, use `pg_upgrade`.

For more information about using yum commands and options, enter `yum --help` at the command line.

For more information about using dnf commands and options, see the `dnf documentation`.

## 6.4.4 Installing on Linux using a local repository

If the server on which you want to install EDB Postgres Advanced Server or the supporting components can’t directly access the EDB repository, you can create a local repository to act as a host for the EDB Postgres Advanced Server native packages. For your network, modify this process from these high-level steps.

To create and use a local repository, you must:

- Use yum or dnf to install the `epel-release`, `yum-utils`, and `createrepo` packages.

  On RHEL or CentOS 7.x:

  `yum install epel-release
yum install yum-utils
yum install createrepo`

  On RHEL or Rocky Linux or AlmaLinux 8.x:

  `dnf install epel-release
dnf install yum-utils
dnf install createrepo`

- Create a directory in which to store the repository:

  `mkdir /srv/repos`

- Copy the RPM installation packages to your local repository. You can download the individual packages or use a tarball to populate the repository. The packages are available from the EDB repository.

- Sync the RPM packages, and create the repository.

  `reposync --edbas15 -p /srv/repos
createrepo /srv/repos`

- Install your preferred webserver on the host that acts as your local repository, and ensure that the repository directory is accessible to the other servers on your network.

- On each isolated database server, configure yum or dnf to pull updates from the mirrored repository on your local network. For example, you might create a repository configuration file called `/etc/yum.repos.d/edb-repo` with connection information that specifies:

  ```
  [edbas15]
  name=EnterprisEDB Postgres Advanced Server 15
  baseurl=https://yum.your_domain.com/edbas15
  enabled=1
  gpgcheck=0
  ```

  After specifying the location and connection information for your local repository, you can use yum or dnf commands to install EDB Postgres Advanced Server and its supporting components on the isolated servers. For example:

  - On RHEL or CentOS 7:

    `yum -y install edb-as15-server`

  - On RHEL or Rocky Linux or AlmaLinux 8:

    `dnf -y install edb-as15-server`
6.4.5 Managing an EDB Postgres Advanced Server installation

Unless otherwise noted, all commands and paths assume that you performed an installation using the native packages.

6.4.5.1 Configuring a package installation

The packages that install the database server component create a unit file on version 7.x or 8.x hosts and service startup scripts.

Creating a database cluster and starting the service

The PostgreSQL `initdb` command creates a database cluster. When installing EDB Postgres Advanced Server with an RPM package, the `initdb` executable is in `/usr/edb/asx.x/bin`. After installing EDB Postgres Advanced Server, you must manually configure the service and invoke `initdb` to create your cluster. When invoking `initdb`, you can:

- Specify environment options on the command line.
- Include the systemd service manager on RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x and use a service configuration file to configure the environment.

For more information, see the `initdb` documentation.

After specifying any options in the service configuration file, you can create the database cluster and start the service. The steps are platform specific.

On RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x

To invoke `initdb` on a RHEL/CentOS 7.x or Rocky Linux/AlmaLinux 8.x system with the options specified in the service configuration file, assume the identity of the operating system superuser:

```
su - root
```

To initialize a cluster with nondefault values, you can use the `PGSETUP_INITDB_OPTIONS` environment variable. You can initialize the cluster using the `edb-as-15-setup` script under `EPAS_Home/bin`.

To invoke `initdb`, export the `PGSETUP_INITDB_OPTIONS` environment variable:

```
PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as15/bin/edb-as-15-setup initdb
```

After creating the cluster, use `systemctl` to start, stop, or restart the service:

```
systemctl { start | stop | restart } edb-as-15
```

On Debian 10.x or Ubuntu 18.04 | 20.04

You can initialize multiple clusters using the bundled scripts. To create a new cluster, assume root privileges, and invoke the bundled script:

```
/usr/bin/epas_createcluster 15 main2
```

To start a new cluster:

```
/usr/bin/epas_ctlcluster 15 main2 start
```

To list all the available clusters:

```
/usr/bin/epas_lscusters
```

Note

The data directory is created under `/var/lib/edb-as/15/main2`, and the configuration directory is created under `/etc/edb-as/15/main/`. 
6.4.5.2 Connecting to EDB Postgres Advanced Server with edb-psql

edb-psql is a command-line client application that allows you to execute SQL commands and view the results. To open the edb-psql client, the client must be in your search path. The executable resides in the `bin` directory under your EDB Postgres Advanced Server installation.

Starting the client

Use the following command and options to start the edb-psql client:

```
edb-psql -d edb -U enterprisedb
```

Where:

- `-d` specifies the database to which edb-psql connects.
- `-U` specifies the identity of the database user to use for the session.

The `edb-psql` executable file is a symbolic link to a binary called `psql`, a modified version of the PostgreSQL community `psql`, with added support for EDB Postgres Advanced Server features. For more information about using the command-line client, see the PostgreSQL core documentation.

Managing authentication on a Debian or Ubuntu host

By default, the server is running with the peer or md5 permission on a Debian or Ubuntu host. You can change the authentication method by modifying the `pg_hba.conf` file, located under `/etc/edb-as/15/main/`.

For more information about modifying the `pg_hba.conf` file, see the PostgreSQL core documentation.

6.4.5.3 Modifying the data directory location

On RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x

On a RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x host, the unit file is named `edb-as-<xx>.service`, where `<xx>` is the EDB Postgres Advanced Server version. It resides in `/usr/lib/systemd/system`. The unit file contains references to the location of the EDB Postgres Advanced Server data directory. Avoid making any modifications directly to the unit file because they might be overwritten during package upgrades.

By default, data files reside under the `/var/lib/edb/as15/data` directory. To use a data directory that resides in a nondefault location:

- Create a copy of the unit file under the `/etc` directory:

  ```
  cp /usr/lib/systemd/system/edb-as-15.service /etc/systemd/system/
  ```

- In the `/etc/systemd/system/edb-as-15.service` file, update the following values with the new location of the data directory:

  ```
  Environment=PGDATA=/var/lib/edb/as15/data
  PIDFile=/var/lib/edb/as15/data/postmaster.pid
  ```

- Go to bin directory and initialize the cluster with the new location:

  ```
  ./edb-as-15-initdb
  ```

- Start the EDB Postgres Advanced Server service:

  ```
  systemctl start edb-as-15
  ```

Configuring SELinux policy to change the data directory location on RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x
By default, the data files reside under the `/var/lib/edb/as15/data` directory. To change the default data directory location depending on individual environment preferences, you must configure the SELinux policy:

- Stop the server:
  
  ```bash
  systemctl stop edb-as-15
  ```

- Check the status of SELinux using the `getenforce` or `sestatus` command:

  ```bash
  # getenforce
  Enforcing
  
  # sestatus
  SELinux status:                 enabled
  SELinuxfs mount:                /sys/fs/selinux
  SELinux root directory:         /etc/selinux
  Loaded policy name:             targeted
  Current mode:                   enforcing
  Mode from config file:          enforcing
  Policy MLS status:              enabled
  Policy deny_unknown status:     allowed
  Max kernel policy version:      31
  ```

- View the SELinux context of the default database location:

  ```bash
  ls -lZ /var/lib/edb/as15/data
  drwx------. enterprisedb enterprisedb unconfined_u:object_r:var_lib_t:s0 log
  ```

- Create a directory for a new location of the database:

  ```bash
  mkdir /opt/edb
  ```

- Move the data directory to `/opt/edb`:

  ```bash
  mv /var/lib/edb/as15/data /opt/edb/
  ```

- Create a file `edb-as-15.service` under `/etc/systemd/system` to include the location of a new data directory:

  ```bash
  .include /lib/systemd/system/edb-as-15.service
  [Service]
  Environment=PGDATA=/opt/edb/data
  PIDFile=/opt/edb/data/postmaster.pid
  ```

- Use the `semanage` utility to set the context mapping for `/opt/edb/`. The mapping is written to the `/etc/selinux/targeted/contexts/files/file.contexts.local` file.

  ```bash
  semanage fcontext --add --equal /var/lib/edb/as15/data /opt/edb
  ```

- Apply the context mapping using the `restorecon` utility:

  ```bash
  restorecon -rv /opt/edb/
  ```

- Reload systemd to modify the service script:

  ```bash
  systemctl daemon-reload
  ```

- With the `/opt/edb` location labeled correctly with the context, start the service:

  ```bash
  systemctl start edb-as-15
  ```

### 6.4.5.4 Specifying cluster options with INITDBOPTS

You can use the `INITDBOPTS` variable to specify your cluster configuration preferences. By default, the `INITDBOPTS` variable is commented out in the service configuration file. Unless you modify it, when you run the service startup script, the new cluster is created in a mode compatible with Oracle databases. Clusters created in this mode contain a database named `edb` and have a database superuser named `enterprisedb`.
Initializing the cluster in Oracle mode

If you initialize the database using Oracle-compatibility mode, the installation includes:

- Data dictionary views compatible with Oracle databases.
- Oracle data type conversions.
- Date values displayed in a format compatible with Oracle syntax.
- Support for Oracle-styled concatenation rules. If you concatenate a string value with a `NULL` value, the returned value is the value of the string.
- Support for the following Oracle built-in packages.

<table>
<thead>
<tr>
<th>Package</th>
<th>Functionality compatible with Oracle databases</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbms_alert</td>
<td>Provides the capability to register for, send, and receive alerts.</td>
</tr>
<tr>
<td>dbms_job</td>
<td>Provides the capability to create, schedule, and manage jobs.</td>
</tr>
<tr>
<td>dbms_lob</td>
<td>Provides the capability to manage on large objects.</td>
</tr>
<tr>
<td>dbms_output</td>
<td>Provides the capability to send messages to a message buffer or get messages from the message buffer.</td>
</tr>
<tr>
<td>dbms_pipe</td>
<td>Provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.</td>
</tr>
<tr>
<td>dbms_rls</td>
<td>Enables the implementation of Virtual Private Database on certain EDB Postgres Advanced Server database objects.</td>
</tr>
<tr>
<td>dbms_sql</td>
<td>Provides an application interface to the EDB dynamic SQL functionality.</td>
</tr>
<tr>
<td>dbms_utility</td>
<td>Provides various utility programs.</td>
</tr>
<tr>
<td>dbms_aq</td>
<td>Provides supporting procedures for Advanced Queueing functionality.</td>
</tr>
<tr>
<td>dbms_profiler</td>
<td>Provides message queuing and processing for EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>dbms_random</td>
<td>Collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a performance profiling session.</td>
</tr>
<tr>
<td>dbms_react</td>
<td>Provides a number of methods to generate random values.</td>
</tr>
<tr>
<td>dbms_lock</td>
<td>Enables redacting or masking data that's returned by a query.</td>
</tr>
<tr>
<td>dbms_schedule</td>
<td>Provides support for the <code>DBMS_LOCK.SLEEP</code> procedure.</td>
</tr>
<tr>
<td>utl_encode</td>
<td>Provides a way to manage and refresh materialized views and their dependencies.</td>
</tr>
<tr>
<td>utl_http</td>
<td>Provides a way to manage and refresh materialized views.</td>
</tr>
<tr>
<td>utl_file</td>
<td>Provides a way to encode and decode data.</td>
</tr>
<tr>
<td>utl_http</td>
<td>Provides a way to encode and decode data.</td>
</tr>
<tr>
<td>utl_file</td>
<td>Provides the capability to read from and write to files on the operating system's file system.</td>
</tr>
<tr>
<td>utl_smtp</td>
<td>Provides the capability to send emails over the Simple Mail Transfer Protocol (SMTP).</td>
</tr>
<tr>
<td>utl_mail</td>
<td>Provides the capability to manage email.</td>
</tr>
<tr>
<td>utl_url</td>
<td>Provides a way to escape illegal and reserved characters in a URL.</td>
</tr>
<tr>
<td>utl_raw</td>
<td>Provides a way to manipulate or retrieve the length of raw data types.</td>
</tr>
</tbody>
</table>

Initializing the cluster in Postgres mode

Clusters created in PostgreSQL mode don't include compatibility features. To create a new cluster in PostgreSQL mode, remove the pound sign (#) in front of the `INITDBOPTS` variable, which enables the `--no-redwood-compat` option. Clusters created in PostgreSQL mode contain a database named `postgres` and have a database superuser named postgres.

You can also specify multiple `initdb` options. For example, the following statement creates a database cluster without compatibility features for Oracle. The cluster contains a database named `postgres` that's owned by a user named alice. The cluster uses UTF-8 encoding.

```
INITDBOPTS="--no-redwood-compat -U alice --locale=en_US.UTF-8"
```

If you initialize the database using `--no-redwood-compat` mode, the installation includes the following packages.

<table>
<thead>
<tr>
<th>Package</th>
<th>Functionality noncompatible with Oracle databases</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbms_agade</td>
<td>Provides supporting procedures for Advanced Queueing functionality.</td>
</tr>
<tr>
<td>dbms_aq</td>
<td>Provides message queuing and processing for EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>edb_bulkload</td>
<td>Provides direct/conventional data loading capability when loading huge amount of data into a database.</td>
</tr>
<tr>
<td>edb_gen</td>
<td>Provides miscellaneous packages to run built-in packages.</td>
</tr>
<tr>
<td>edb_objects</td>
<td>Provides Oracle-compatible objects such as packages and procedures.</td>
</tr>
<tr>
<td>waitstates</td>
<td>Provides monitor session blocking.</td>
</tr>
</tbody>
</table>
In addition to the cluster configuration options documented in the PostgreSQL core documentation, EDB Postgres Advanced Server supports the following `initdb` options:

---

### --no-redwood-compat

Include the `--no-redwood-compat` keywords to create the cluster in PostgreSQL mode. When the cluster is created in PostgreSQL mode, the name of the database superuser is postgres, and the name of the default database is `postgres`. The few EDB Postgres Advanced Server features compatible with Oracle databases are available with this mode. However, we recommend using the EDB Postgres Advanced Server in redwood compatibility mode to use all its features.

### --redwood-like

Include the `--redwood-like` keywords to use an escape character. The character is an empty string ('') and it follows the `LIKE` (or PostgreSQL-compatible `ILIKE`) operator in a SQL statement that's compatible with Oracle syntax.

### --icu-short-form

Include the `--icu-short-form` keywords to create a cluster that uses a default International Components for Unicode (ICU) collation for all databases in the cluster. For more information about Unicode collations, see the [Unicode collation algorithm](#).

For more information about using `initdb` and the available cluster configuration options, see the PostgreSQL core documentation.

You can also view online help for `initdb` by assuming superuser privileges and entering:

```
/path_to_initdb_installation_directory/initdb --help
```

Where `path_to_initdb_installation_directory` specifies the location of the `initdb` binary file.

### 6.4.5.5 Starting and stopping services

A service is a program that runs in the background and doesn't require user interaction. A service has no user interface. You can configure a service to start at boot time or manually on demand. Services are best controlled using the platform-specific operating system service control utility. Many of the EDB Postgres Advanced Server supporting components are services.

#### List of services

The following table lists the names of the services that control EDB Postgres Advanced Server and services that control EDB Postgres Advanced Server supporting components.

<table>
<thead>
<tr>
<th>EDB Postgres Advanced Server component name</th>
<th>Linux service name</th>
<th>Debian service name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDB Postgres Advanced Server</td>
<td>edb-as-15</td>
<td>edb-as@15-main</td>
</tr>
<tr>
<td>pgAgent</td>
<td>edb-pgagent-15</td>
<td>edb-as15-pgagent</td>
</tr>
<tr>
<td>PgBouncer</td>
<td>edb-pgbouncer-1.15</td>
<td>edb-pgbouncer115</td>
</tr>
<tr>
<td>pgPool-II</td>
<td>edb-pgpool-4.1</td>
<td>edb-pgpool41</td>
</tr>
<tr>
<td>Slony</td>
<td>edb-slony-replication-15</td>
<td>edb-as15-slony-replication-1</td>
</tr>
<tr>
<td>EFM</td>
<td>edb-efm-4.0</td>
<td>edb-efm-4.0</td>
</tr>
</tbody>
</table>

You can use the Linux command line to control the EDB Postgres Advanced Server database server and the services of EDB Postgres Advanced Server’s supporting components. The commands that control the EDB Postgres Advanced Server service on a Linux platform are host specific.

#### Controlling a service on RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x

If your installation of EDB Postgres Advanced Server resides on version 7.x | 8.x of RHEL and CentOS, you must use the `systemctl` command to control the EDB Postgres Advanced Server service and supporting components.
The `systemctl` command must be in your search path, and you must invoke it with superuser privileges. To use the command, at the command line, enter:

```
systemctl <action> <service_name>
```

Where:

- `<service_name>` specifies the name of the service.
- `<action>` specifies the action taken by the service command. Specify:
  - `start` to start the service.
  - `stop` to stop the service.
  - `restart` to stop and then start the service.
  - `status` to discover the current status of the service.

Controlling a service on Debian 10.x or Ubuntu 18.04 | 20.04

If your installation of EDB Postgres Advanced Server resides on version 18.04 | 20.04 of Ubuntu, assume superuser privileges and invoke the following commands using bundled scripts to manage the service. Use the following commands to:

- Discover the current status of a service:
  ```
  /usr/edb/as15/bin/epas_ctlcluster 15 main status
  ```
- Stop a service:
  ```
  /usr/edb/as15/bin/epas_ctlcluster 15 main stop
  ```
- Restart a service:
  ```
  /usr/edb/as15/bin/epas_ctlcluster 15 main restart
  ```
- Reload a service:
  ```
  /usr/edb/as15/bin/epas_ctlcluster 15 main reload
  ```
- Control the component services:
  ```
  systemctl restart edb-as@15-main
  ```

Using `pg_ctl` to control EDB Postgres Advanced Server

You can use the `pg_ctl` utility to control an EDB Postgres Advanced Server service from the command line on any platform. `pg_ctl` allows you to:

- Start, stop, or restart the EDB Postgres Advanced Server database server
- Reload the configuration parameters
- Display the status of a running server

To invoke the utility, assume the identity of the cluster owner. In the home directory of EDB Postgres Advanced Server, enter:

```
./bin/pg_ctl -D <data_directory> <action>
```

- `<data_directory>` is the location of the data controlled by the EDB Postgres Advanced Server cluster.
- `<action>` specifies the action taken by the `pg_ctl` utility. Specify:
  - `start` to start the service.
  - `stop` to stop the service.
  - `restart` to stop and then start the service.
  - `reload` to send the server a SIGHUP signal, reloading configuration parameters.
  - `status` to discover the current status of the service.

For more information about using the `pg_ctl` utility or the command-line options available, see the [PostgreSQL core documentation](http://www.postgresql.org).
Choosing between pg_ctl and the service command

You can use the pg_ctl utility to manage the status of an EDB Postgres Advanced Server cluster. However, it’s important to know that pg_ctl doesn’t alert the operating system service controller to changes in the status of a server. We recommend using the service command when possible.

Configuring component services to autostart at system reboot

After installing, configuring, and starting the services of EDB Postgres Advanced Server supporting components on a Linux system, you must manually configure your system to autostart the service when your system restarts. To configure a service to autostart on a Linux system, at the command line, assume superuser privileges, and enter the command.

On a Redhat-compatible Linux system, enter:

```
/sbin/chkconfig <service_name> on
```

Where `service_name` specifies the name of the service.

6.4.5.6 Starting multiple postmasters with different clusters

You can configure EDB Postgres Advanced Server to use multiple postmasters, each with its own database cluster. The steps required are specific to the version of the Linux host.

On RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x

The `edb-as15-server-core` RPM for version 7.x | 8.x contains a unit file that starts the EDB Postgres Advanced Server instance. The file allows you to start multiple services with unique data directories and monitor different ports. You need root access to invoke or modify the script.

This example creates an EDB Postgres Advanced Server installation with two instances. The secondary instance is named `secondary`.

- Make a copy of the default file with the new name. As noted at the top of the file, all modifications must reside under `/etc`. You must pick a name that isn't already used in `/etc/systemd/system`.

  ```
  ```

- Edit the file, changing `PGDATA` to point to the new data directory that you're creating the cluster against.

- Create the target `PGDATA` with the user enterprisedb.

- Run `initdb`, specifying the setup script:

  ```
  /usr/edb/as15/bin/edb-as-15-setup initdb secondary-edb-as-15
  ```

- Edit the `postgresql.conf` file for the new instance, specifying the port, the IP address, TCP/IP settings, and so on.

- Make sure that the new cluster runs after a reboot:

  ```
  systemctl enable secondary-edb-as-15
  ```

- Start the second cluster:

  ```
  systemctl start secondary-edb-as-15
  ```

6.5 Troubleshooting

You can troubleshoot your EDB Postgres Advanced Server installation on Linux or Windows.

6.5.1 Linux installation troubleshooting
On your Linux platform, you can:

- Troubleshoot installations.
- Enable core dumps.

### 6.5.1.1 Installation troubleshooting

#### Difficulty displaying Java-based applications

If you have difficulty displaying Java-based server features (controls or text not displaying correctly or blank windows), upgrading to the latest `libxcb-xlib` libraries corrects the problem on most distributions. See the Java bug database for other possible workarounds.

#### The installation fails to complete due to existing data directory contents

If an installation fails to complete due to existing content in the data directory, the server writes an error message to the server logs:

```
A data directory is neither empty, or a recognisable data directory.
```

If you encounter a similar message, confirm that the data directory is empty. The presence of files, including the system-generated `lost+found` folder, prevents the installation from completing. Either remove the files from the data directory, or specify a different location for the data directory before invoking the installer again to complete the installation.

#### Difficulty installing the EPEL release package

If you have difficulty installing the EPEL release package, you can use the following command to install the `epel-release` package on RHEL/CentOS 7.x and RHEL/Rocky Linux/AlmaLinux 8.x:

```
```

You might need to enable the `extras` repository definition in the `CentOS-Base.repo` file, located in `/etc/yum.repos.d`. If `yum` can't access a repository that contains `epel-release`, an error message appears:

```
No package epel available.
Error: Nothing to do
```

If you receive this error, you can download the EPEL rpm package and install it manually. To manually install EPEL, download the rpm package, assume superuser privileges, navigate into the directory that contains the package, and use this command:

```
yum -y install epel-release
dnf -y install epel-release
```

### 6.5.1.2 Enabling core dumps

You can use core dumps to diagnose or debug errors. A core dump is a file containing a process's address space (memory) when the process terminates unexpectedly. Core dumps can be produced on demand, such as by a debugger, or upon termination.

#### Enabling core dumps on a RHEL or CentOS or Rocky Linux or AlmaLinux host

On RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x, core file creation is disabled by default. To enable the core file generation:

1. Identify the system's current limit using the `ulimit -c` or `ulimit -a` command. A `0` indicates that core file generation is disabled.

```
# ulimit -c
0

# ulimit -a
  core file size       # blocks, -c 0
  data seg size       # kbytes, -d unlimited
  scheduling priority # -p unlimited
  file size           # blocks, -f unlimited
```
pending signals: (-i) 3756
max locked memory: (kbytes, -l) 64
max memory size: (kbytes, -m) unlimited
open files: (-n) 1024
pipe size: (512 bytes, -p) 8
POSIX message queues: (bytes, -q) 819200
real-time priority: (-r) 8
stack size: (kbytes, -s) 8192
cpu time: (seconds, -t) unlimited
max user processes: (-u) 3756
virtual memory: (kbytes, -v) unlimited
file locks: (-x) unlimited

Create a directory to store the core dumps, and modify `kernel.core_pattern` to store the dumps in a specified directory:

```bash
mkdir -p /var/coredumps
chmod a+w /var/coredumps
sysctl kernel.core_pattern=/var/coredumps/core-%e-%p
```

Persist the `kernel.core_pattern` setting across reboots:

```bash
echo 'kernel.core_pattern=/var/coredumps/core-%e-%p' >> /etc/sysctl.conf
```

Enable core dumps in `/etc/security/limits.conf` to allow a user to create core files. Each line describes a limit for a user in the following form:

```bash
<domain>  <type>  <item>  <value>
*       soft    core    unlimited
```

Use `*` to enable the core dump size to unlimited.

Set the limit of core file size to **UNLIMITED**:

```bash
ulimit -c unlimited
```

To set a core limit for the services, add the following setting in `/usr/lib/systemd/system/edb-as-15.service`:

```bash
[Service]
LimitCore=Infinity
```

Reload the service configuration:

```bash
systemctl daemon-reload
```

Modify the global default limit using systemd. Add the following setting in `/etc/systemd/system.conf`:

```bash
DefaultLimitCORE=Infinity
```

Restart systemd:

```bash
systemctl daemon-reexec
```

Stop and then start EDB Postgres Advanced Server:

```bash
systemctl stop edb-as-15
systemctl start edb-as-15
```

Now, the core dumps are enabled. Install the gdb tool and debug packages:

```bash
yum install gdb debuginfo-install edb-as15 edb-as15-server-contrib edb-as15-server edb-as15-libs
```

Replace the path to a core dump file. Then, get a backtrace using the `bt` command to analyze output:
Enabling core dumps on a Debian or Ubuntu host

On Debian 10 or Ubuntu 18 and 20, core file creation is disabled by default. To enable the core file generation:

- Identify the system's current limit using the `ulimit -c` or `ulimit -a` command. 0 indicates that core file generation is disabled.

```bash
# ulimit -c
0
# ulimit -a
```

- Create a new directory to store the core dumps, and modify `kernel.core_pattern` to store the dumps in a specified directory:

```bash
mkdir -p /var/coredumps
chmod a+w /var/coredumps
sysctl kernel.core_pattern=/var/coredumps/core-%e-%p
kernel.core_pattern = /var/coredumps/core-%e-%p
```

- Persist the `kernel.core_pattern` setting across reboots:

```bash
echo 'kernel.core_pattern=/var/coredumps/core-%e-%p' >> /etc/sysctl.conf
```

- To allow a user to create core files, enable core dumps in `/etc/security/limits.conf`. Each line describes a limit for a user in the following form.

```text
<domain>  <type>  <item>  <value>
*       soft    core    unlimited
```

Use * to enable the core dump size to unlimited.

- Set the limit of core file size to `UNLIMITED`:

```bash
ulimit -c unlimited
ulimit -c
unlimited
```

- To set a core limit for the services, add the following setting in `/lib/systemd/system/edb-as@.service`:

```bash
[Service]
LimitCore=Infinity
```

- Reload the service configuration:

```bash
systemctl daemon-reload
```

- Modify the global default limit using systemd. Add the following setting in `/etc/systemd/system.conf`:

```bash
DefaultLimitCORE=Infinity
```
Restart systemd:

```
systemctl daemon-reexec
```

Stop and then start EDB Postgres Advanced Server:

```
systemctl stop edb-as@15.service
systemctl start edb-as@15.service
```

Now, the core dumps are enabled. Install the gdb tool and debug symbols:

```
apt-get install gdb
apt-get install edb-as15 edb-as-contrib edb-as15-server edb-debugger-dbgsym
```

Replace the path to a core dump file. Then get a backtrace using the `bt` command to analyze output:

```
gdb /usr/lib/edb-as/15/bin /var/coredumps/core-edb-postgres-21638
(gdb) bt full
```

Note

- The debug info packages name on a Debian or Ubuntu host can vary and include a `-dbgsym` or `-dbg` suffix. For more information about setting `sources.list` and installing the debug info packages, see the Debian wiki or the Ubuntu wiki.
- The core files can be huge, depending on the memory usage. Enabling the core dumps on a system might fill up its mass storage over time.

### 6.5.2 Windows installation troubleshooting

**Difficulty displaying Java-based applications**

If you have difficulty displaying Java-based server features (controls or text not displaying correctly or blank windows), upgrading to the latest `libxcb-xlib` libraries corrects the problem on most distributions. See the Java bug database for other possible workarounds.

**-mode unattended authentication errors**

Authentication errors from component modules during unattended installations might indicate that the specified values of `--servicepassword` or `--superpassword` are incorrect.

**Errors during an EDB Postgres Advanced Server installation**

If you encounter an error during the installation process, exit the installation, and ensure that your version of Windows is up to date. After applying any outstanding operating system updates, invoke the EDB Postgres Advanced Server installer again.

**The installation fails to complete due to existing data directory contents**

If an installation fails to complete due to existing content in the data directory, the server writes an error message to the server logs:

```
A data directory is neither empty, or a recognisable data directory.
```

If you encounter a similar message, confirm that the data directory is empty. The presence of files, including the system-generated `lost+found` folder, prevents the installation from completing. Either remove the files from the data directory, or specify a different location for the data directory before invoking the installer again to complete the installation.

### 6.6 Uninstalling EDB Postgres Advanced Server

You can uninstall the EDB Postgres Advanced Server on the platform where it's installed.

#### 6.6.1 Uninstalling EDB Postgres Advanced Server on Linux

**Note**
After uninstalling EDB Postgres Advanced Server, the cluster data files remain intact, and the service user persists. You can manually remove the cluster data and service user from the system.

Uninstalling on RHEL/OL/AlmaLinux/Rocky Linux

You can use variations of the `rpm`, `yum`, or `dnf` command to remove installed packages. Removing a package doesn't damage the EDB Postgres Advanced Server data directory.

Include the `-e` option when invoking the `rpm` command to remove an installed package:

```
rpm -e <package_name>
```

Where `package_name` is the name of the package that you want to remove.

You can use the `yum remove` or `dnf remove` command to remove a package installed by yum or dnf. To remove a package, at the command line, assume superuser privileges, and enter the appropriate command.

- On RHEL or CentOS 7:

  ```
yum remove <package_name>
  ```

- On RHEL or Rocky Linux or AlmaLinux 8:

  ```
dnf remove <package_name>
  ```

Where `package_name` is the name of the package that you want to remove.

`yum` and `rpm` don't remove a package that's required by another package. If you attempt to remove a package that satisfies a package dependency, `yum` or `rpm` provides a warning.

**Note**

In RHEL or Rocky Linux or AlmaLinux 8, removing a package also removes all its dependencies that aren't required by other packages. To override this default behavior of RHEL or Rocky Linux or AlmaLinux 8, disable the `clean_requirements_on_remove` parameter in the `/etc/yum.conf` file.

To uninstall EDB Postgres Advanced Server and its dependent packages, use the appropriate command.

- On RHEL or CentOS 7:

  ```
  ```

  Where `<xx>` is the EDB Postgres Advanced Server version number.

- On RHEL or Rocky Linux or AlmaLinux 8:

  ```
  ```

Uninstalling on Debian or Ubuntu

- To uninstall EDB Postgres Advanced Server, invoke the following command. The configuration files and data directory remains intact.

  ```
  apt-get remove edb-as<xx>-server*
  ```

- To uninstall EDB Postgres Advanced Server, configuration files, and data directory, invoke the following command:

  ```
  apt-get purge edb-as<xx>-server*
  ```

6.6.2 Uninstalling EDB Postgres Advanced Server on Windows

**Note**
After uninstalling EDB Postgres Advanced Server, the cluster data files remain intact, and the service user persists. You can manually remove the cluster data and service user from the system.

Using EDB Postgres Advanced Server uninstallers at the command line

The EDB Postgres Advanced Server interactive installer creates an uninstaller that you can use to remove EDB Postgres Advanced Server or components that reside on a Windows host. The uninstaller is created in `C:\Program Files\edb\as15`.

1. Assume superuser privileges and, in the directory that contains the uninstaller, enter:

   `uninstall-edb-as<xx>-server.exe`

   Where `<xx>` is the EDB Postgres Advanced Server version number.

   The uninstaller opens.

2. By default, the installer removes the entire application. If you instead want to select components to remove, select **Individual components**. A dialog box prompts you to select the components you want to remove. Make your selections.

3. Select **Next**.

   If you selected components to remove that depend on EDB Postgres Advanced Server, those components are removed first.

4. To continue, select **Yes**.

   Progress bars are displayed as the software is removed. A confirmation reports when the uninstall process is complete.

7 EDB Postgres Advanced Server upgrade

Upgrading EDB Postgres Advanced Server involves:

- `pg_upgrade` to upgrade from an earlier version of EDB Postgres Advanced Server to the latest version.
- `yum` to perform a minor version upgrade on a Linux host.
- `StackBuilder Plus` to perform a minor version upgrade on a Windows host.
7.1 Upgrade considerations

The `pg_upgrade` executable is distributed with EDB Postgres Advanced Server and is installed as part of the Database Server component. You don't need to further install or configure it. The `pg_upgrade` utility performs an in-place transfer of existing data between EDB Postgres Advanced Server and any later version.

Several factors determine if an in-place upgrade is practical:

- The on-disk representation of user-defined tables must not change between the original version and the upgraded version.
- The on-disk representation of data types must not change between the original version and the upgraded version.
- To upgrade between major versions of EDB Postgres Advanced Server with `pg_upgrade`, both versions must share a common binary representation for each data type. Therefore, you can't use `pg_upgrade` to migrate from a 32-bit to a 64-bit Linux platform.

Before performing a version upgrade, `pg_upgrade` verifies that the old cluster and the new cluster are compatible.

If the upgrade involves a change in the on-disk representation of database objects or data, or if it involves a change in the binary representation of data types, `pg_upgrade` can't perform the upgrade. To upgrade, you have to `pg_dump` the old data and then import that data to the new cluster, as described below.

1. Export all of your data using `pg_dump`.
2. Install the new release.
3. Run `initdb` to create a new cluster.
4. Import your old data.

Consider the following when upgrading EDB Postgres Advanced Server:

- The `pg_upgrade` utility can't upgrade a partitioned table if a foreign key refers to the partitioned table.
- If you perform an upgrade of the EDB Postgres Advanced Server installation, you must rebuild any hash-partitioned table on the upgraded server.
- If you're using an ODBC, JDBC, OCI, or .NET driver to connect to your database applications and upgrading to a new major version of EDB Postgres Advanced Server, upgrade your driver to the latest version when upgrading EDB Postgres Advanced Server.

7.2 Upgrading an installation with pg_upgrade

Note

Review the upgrade considerations prior to performing an upgrade with the `pg_upgrade` utility.

While minor upgrades between versions are fairly simple and require only installing new executables, past major version upgrades were both expensive and time consuming. `pg_upgrade` eases migration between any version of EDB Postgres Advanced Server (version 9.0 or later) and any later release of EDB Postgres Advanced Server that's supported on the same platform.

Using `pg_upgrade` can reduce both the amount of time and the disk space required for many major-version upgrades.

7.2.1 About pg_upgrade

The `pg_upgrade` executable is distributed with EDB Postgres Advanced Server and is installed as part of the Database Server component. You don't need to further install or configure it.

The `pg_upgrade` utility performs an in-place transfer of existing data between EDB Postgres Advanced Server and any later version.

Before performing a version upgrade, `pg_upgrade` verifies that the old cluster and the new cluster are compatible.

When `pg_upgrade` starts, it performs a compatibility check to ensure that all required executables are present and contain the expected version numbers. The verification process also checks the old and new `$PGDATA` directories to ensure that the expected files and subdirectories are in place. If the verification process succeeds, `pg_upgrade` starts the old `postmaster` and runs `pg_dumpall --schema-only` to capture the metadata contained in the old cluster. The script produced by `pg_dumpall` is used in later to re-create all user-defined objects in the new cluster.

The script produced by `pg_dumpall` re-creates only user-defined objects and not system-defined objects. The new cluster already contains the system-defined objects created by the latest version of EDB Postgres Advanced Server.

After extracting the metadata from the old cluster, `pg_upgrade` performs the bookkeeping tasks required to sync the new cluster with the existing data.

1. Copy the old data files to the new location.
2. Copy the metadata files to the new location.
3. Run `pg_restore` to import the metadata into the new cluster.
4. Run `pg_dumpall --schema-only` to capture the metadata contained in the new cluster.
5. Run `pg_restore` to import the metadata into the new cluster.

If you're upgrading to EDB Postgres Advanced Server and installed the `edb_dblink_oci` or `edb_dblink_libpq` extension, drop the extension before performing an upgrade. To drop the extension, connect to the server with the `psql` or PEM client, and invoke the commands:

```
DROP EXTENSION edb_dblink_oci;
DROP EXTENSION edb_dblink_libpq;
```

When you finish upgrading, you can use the `CREATE EXTENSION` command to add the current versions of the extensions to your installation.
7.2.2 Performing an upgrade

To upgrade an earlier version of EDB Postgres Advanced Server to the current version:

1. Install the current version of EDB Postgres Advanced Server. The new installation must contain the same supporting server components as the old installation.
2. Empty the target database or create a new target cluster with `initdb`.
3. To avoid authentication conflicts, place the `pg_hba.conf` file for both databases in `trust` authentication mode.
4. Shut down the old and new EDB Postgres Advanced Server services.
5. Invoke the `pg_upgrade` utility.

When `pg_upgrade` starts, it performs a compatibility check to ensure that all required executables are present and contain the expected version numbers. The verification process also checks the old and new `SPGDATA` directories to ensure that the expected files and subdirectories are in place. If the verification process succeeds, `pg_upgrade` starts the old `postmaster` and runs `pg_dumpall` with `--schema-only` to capture the metadata contained in the old cluster. The script produced by `pg_dumpall` is used in later to re-create all user-defined objects in the new cluster.

The script produced by `pg_dumpall` re-creates only user-defined objects and not system-defined objects. The new cluster already contains the system-defined objects created by the latest version of EDB Postgres Advanced Server.

After extracting the metadata from the old cluster, `pg_upgrade` performs the bookkeeping tasks required to sync the new cluster with the existing data.

If you're upgrading to EDB Postgres Advanced Server and installed the `edb_dblink_libpq` or `edb_dblink_oci` extension, drop the extension before performing an upgrade. To drop the extension, connect to the server with the `psql` or PEM client, and invoke the commands:

```
DROP EXTENSION edb_dblink_libpq;
```

When you finish upgrading, you can use the `CREATE EXTENSION` command to add the current versions of the extensions to your installation.

7.2.2.1 Linking versus copying

When invoking `pg_upgrade`, you can use a command-line option to specify whether to copy or link each table and index in the old cluster to the new cluster.

Linking is much faster because `pg_upgrade` creates a second name (a hard link) for each file in the cluster. Linking also requires no extra workspace because `pg_upgrade` doesn't make a copy of the original data. When linking the old cluster and the new cluster, the old and new clusters share the data. After starting the new cluster, you can no longer use your data with the previous version of EDB Postgres Advanced Server.

If you choose to copy data from the old cluster to the new cluster, `pg_upgrade` still reduces the time required to perform an upgrade compared to the traditional `dump/restore` procedure. `pg_upgrade` uses a file-at-a-time mechanism to copy data files from the old cluster to the new cluster versus the row-by-row mechanism used by `dump/restore`. When you use `pg_upgrade`, you avoid building indexes in the new cluster. Each index is instead copied from the old cluster to the new cluster. Finally, using a `dump/restore` procedure to upgrade requires a lot of workspace to hold the intermediate text-based dump of all of your data, while `pg_upgrade` requires very little extra workspace.

Data that's stored in user-defined tablespaces isn't copied to the new cluster. It stays in the same location in the file system but is copied into a subdirectory whose name shows the version number of the new cluster. To manually relocate files that are stored in a tablespace after upgrading, move the files to the new location and update the symbolic links to point to the files. The symbolic links are located in the `pg_tblspc` directory under your cluster's `data` directory.

7.2.3 Invoking pg_upgrade

When invoking `pg_upgrade`, you must specify the location of the old and new cluster's `PGDATA` and executable (`.bin`) directories, the name of the EDB Postgres Advanced Server superuser, and the ports on which the installations are listening. A typical call to invoke `pg_upgrade` to migrate from EDB Postgres Advanced Server 14 to EDB Postgres Advanced Server 15 takes the form:

```
pg_upgrade
   --old-datadir <path_to_14_data_directory>
   --new-datadir <path_to_15_data_directory>
   --user <superuser_name>
   --old-bindir <path_to_14_bin_directory>
   --new-bindir <path_to_15_bin_directory>
   --old-port <14_port> --new-port <14_port>
```

Where:
Use the `--old-datadir` option to specify the complete path to the `data` directory in the EDB Postgres Advanced Server 14 installation.

Use the `--new-datadir` option to specify the complete path to the `data` directory in the EDB Postgres Advanced Server 15 installation.

Include the `--username` option to specify the name of the EDB Postgres Advanced Server superuser. The superuser name must be the same in both versions of EDB Postgres Advanced Server. By default, when EDB Postgres Advanced Server is installed in Oracle mode, the superuser is named `enterprisedb`. If installed in PostgreSQL mode, the superuser is named `postgres`.

If the EDB Postgres Advanced Server superuser name isn’t the same in both clusters, the clusters won’t pass the `pg_upgrade` consistency check.

Use the `--old-bindir` option to specify the complete path to the `bin` directory in the EDB Postgres Advanced Server 14 installation.

Use the `--new-bindir` option to specify the complete path to the `bin` directory in the EDB Postgres Advanced Server 15 installation.

Include the `--old-port` option to specify the port on which EDB Postgres Advanced Server 14 listens for connections.

Include the `--new-port` option to specify the port on which EDB Postgres Advanced Server 15 listens for connections.

7.2.3.1 Command line options for pg_upgrade

`pg_upgrade` accepts the following command line options. Each option is available in a long form or a short form:

- `-b path_to_old_bin_directory` or `--old-bindir path_to_old_bin_directory`
  Use the `-b` or `--old-bindir` keyword to specify the location of the old cluster's executable directory.

- `-B path_to_new_bin_directory` or `--new-bindir path_to_new_bin_directory`
  Use the `-B` or `--new-bindir` keyword to specify the location of the new cluster's executable directory.

- `-c` or `--check`
  Include the `-c` or `--check` keyword to specify for `pg_upgrade` to perform a consistency check on the old and new cluster without performing a version upgrade.

- `-d path_to_old_data_directory` or `--old-datadir path_to_old_data_directory`
  Use the `-d` or `--old-datadir` keyword to specify the location of the old cluster's `data` directory.

- `-D path_to_new_data_directory` or `--new-datadir path_to_new_data_directory`
  Use the `-D` or `--new-datadir` keyword to specify the location of the new cluster's `data` directory.

Data that's stored in user-defined tablespaces isn't copied to the new cluster. It stays in the same location in the file system but is copied into a subdirectory whose name reflects the version number of the new cluster. To manually relocate files that are stored in a tablespace after upgrading, you must move the files to the new location and update the symbolic links (located in the `pg_tblspc` directory under your cluster's `data` directory) to point to the files.

- `-j` or `--jobs`
Include the `-j` or `--jobs` keyword to specify the number of simultaneous processes or threads to use during the upgrade.

```
-j
```

Include the `-k` or `--link` keyword to create a hard link from the new cluster to the old cluster. See Linking versus copying for more information about using a symbolic link.

```
-k
```

Include the `-o` or `--old-options` options keyword to specify options to pass to the old `postgres` command. Enclose options in single or double quotes to ensure that they're passed as a group.

```
-o options
  --old-options options
```

Use the `-p` or `--old-port` keyword to specify the port number of the EDB Postgres Advanced Server installation that you're upgrading.

```
-p old_port_number
  --old-port old_port_number
```

Include the `-P` or `--new-port` keyword to specify the port number of the new EDB Postgres Advanced Server installation.

```
-P new_port_number
  --new-port new_port_number
```

Include the `-U` or `--username` keyword to specify the name of the EDB Postgres Advanced Server database superuser. The same superuser must exist in both clusters.

```
-U user_name
  --username user_name
```

Use the `-v` or `--version` keyword to display version information for `pg_upgrade`.

```
-v
  --version
```

Use the `-?` or `--help` options to display `pg_upgrade` help information.

```
-?
  -h
  --help
```

### 7.2.4 Upgrading to EDB Postgres Advanced Server

You can use `pg_upgrade` to upgrade from an existing installation of EDB Postgres Advanced Server into the cluster built by the EDB Postgres Advanced Server Installer or into an alternative cluster created using the `initdb` command.

The basic steps to perform an upgrade into an empty cluster created with the `initdb` command are the same as the steps to upgrade into the cluster created by the EDB Postgres Advanced Server Installer. However, you can omit Step 2 - Empty the edb database and substitute the location of the alternative cluster when specifying a target cluster for the upgrade.

If a problem occurs during the upgrade process, you can revert to the previous version. See Reverting to the old cluster for detailed information about this process.
You must be an operating system superuser or Windows Administrator to perform an EDB Postgres Advanced Server upgrade.

**Step 1 - Install the new server**

Install the new version of EDB Postgres Advanced Server, specifying the same non-server components that were installed during the previous EDB Postgres Advanced Server installation. The new cluster and the old cluster must reside in different directories.

**Step 2 - Empty the target database**

The target cluster must not contain any data. You can create an empty cluster using the `initdb` command, or you can empty a database that was created during the installation of EDB Postgres Advanced Server. If you installed EDB Postgres Advanced Server in PostgreSQL mode, the installer creates a single database named `postgres`. Installing EDB Postgres Advanced Server in Oracle mode creates a database named `postgres` and a database named `edb`.

The easiest way to empty the target database is to drop the database and then create a new database. Before invoking the `DROP DATABASE` command, you must disconnect any users and halt any services that are currently using the database.

On Windows, from the Control Panel, go to the Services manager. Select each service in the Services list, and select Stop.

On Linux, open a terminal window, assume superuser privileges, and manually stop each service. For example, invoke the following command to stop the pgAgent service:

```
service edb-pgagent-14 stop
```

After stopping any services that are currently connected to EDB Postgres Advanced Server, you can use the EDB-PSQL command line client to drop and create a database. When the client opens, connect to the `template1` database as the database superuser. If prompted, provide authentication information. Then, use the following command to drop your database:

```
DROP DATABASE <database_name>;
```

Where `database_name` is the name of the database.

Then, create an empty database based on the contents of the `template` database.

```
CREATE DATABASE <database_name>;
```

**Step 3 - Set both servers in trust mode**

During the upgrade process, `pg_upgrade` connects to the old and new servers several times. To make the connection process easier, you can edit the `pg_hba.conf` file, setting the authentication mode to `trust`. To modify the `pg_hba.conf` file, from the Start menu, select EDB Postgres > EDB Postgres Advanced Server > Expert Configuration. Select Edit pg_hba.conf to open the `pg_hba.conf` file.

You must allow trust authentication for the previous EDB Postgres Advanced Server installation and EDB Postgres Advanced Server servers. Edit the `pg_hba.conf` file for both installations of EDB Postgres Advanced Server as shown in the figure.

![pg_hba.conf](image)

After editing each file, save the file, and exit the editor.

If the system is required to maintain `md5` authentication mode during the upgrade process, you can specify user passwords for the database superuser in a password file (`pgpass.conf` on Windows, `.pgpass` on Linux). For more information about configuring a password file, see the PostgreSQL core documentation.

**Step 4 - Stop all component services and servers**


Before you invoke `pg_upgrade`, you must stop any services that belong to the original EDB Postgres Advanced Server installation, EDB Postgres Advanced Server, or the supporting components. Stopping these services ensures that a service doesn't attempt to access either cluster during the upgrade process.

The services in the table are most likely to be running in your installation.

<table>
<thead>
<tr>
<th>Service</th>
<th>On Linux</th>
<th>On Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDB Postgres Advanced Server 9.6</td>
<td>edb-as-9.6</td>
<td>edb-as-9.6</td>
</tr>
<tr>
<td>EDB Postgres Advanced Server 10</td>
<td>edb-as-10</td>
<td>edb-as-10</td>
</tr>
<tr>
<td>EDB Postgres Advanced Server 11</td>
<td>edb-as-11</td>
<td>edb-as-11</td>
</tr>
<tr>
<td>EDB Postgres Advanced Server 12</td>
<td>edb-as-12</td>
<td>edb-as-12</td>
</tr>
<tr>
<td>EDB Postgres Advanced Server 13</td>
<td>edb-as-13</td>
<td>edb-as-13</td>
</tr>
<tr>
<td>EDB Postgres Advanced Server 14</td>
<td>edb-as-14</td>
<td>edb-as-14</td>
</tr>
<tr>
<td>EDB Postgres Advanced Server 15</td>
<td>edb-as-15</td>
<td>edb-as-15</td>
</tr>
<tr>
<td>EDB Postgres Advanced Server 9.6 Scheduling Agent</td>
<td>edb-pgagent-9.6</td>
<td>EDB Postgres Advanced Server 9.6 Scheduling Agent</td>
</tr>
<tr>
<td>Infinite Cache 9.6</td>
<td>edb-icache</td>
<td>N/A</td>
</tr>
<tr>
<td>Infinite Cache 10</td>
<td>edb-icache</td>
<td>N/A</td>
</tr>
<tr>
<td>PgBouncer</td>
<td>Pgbouncer</td>
<td>Pgbouncer</td>
</tr>
<tr>
<td>PgBouncer 1.6</td>
<td>ppas-pgbouncer-1.6 or ppas-pgbouncer16</td>
<td>ppas-pgbouncer-1.6</td>
</tr>
<tr>
<td>PgBouncer 1.7</td>
<td>edb-pgbouncer-1.7</td>
<td>edb-pgbouncer-1.7</td>
</tr>
<tr>
<td>PgPool</td>
<td>ppas-pgpool</td>
<td>N/A</td>
</tr>
<tr>
<td>PgPool 3.4</td>
<td>ppas-pgpool-3.4 or ppas-pgpool14</td>
<td>N/A</td>
</tr>
<tr>
<td>PgPool-II</td>
<td>edb-pgpool-3.5</td>
<td>N/A</td>
</tr>
<tr>
<td>Slony 9.6</td>
<td>edb-slony-replication-9.6</td>
<td>edb-slony-replication-9.6</td>
</tr>
<tr>
<td>xDB Publication Server 9.0</td>
<td>edb-xdbpubserver-90</td>
<td>Publication Service 90</td>
</tr>
<tr>
<td>xDB Publication Server 9.1</td>
<td>edb-xdbpubserver-91</td>
<td>Publication Service 91</td>
</tr>
<tr>
<td>xDB Subscription Server</td>
<td>edb-xdbsubserver-90</td>
<td>Subscription Service 90</td>
</tr>
<tr>
<td>xDB Subscription Server</td>
<td>edb-xdbsubserver-91</td>
<td>Subscription Service 91</td>
</tr>
<tr>
<td>EDB Replication Server v6.x</td>
<td>edb-xdbpubserver</td>
<td>Publication Service for xDB Replication Server</td>
</tr>
<tr>
<td>EDB Subscription Server v6.x</td>
<td>edb-xdbsubserver</td>
<td>Subscription Service for xDB Replication Server</td>
</tr>
</tbody>
</table>

To stop a service on Windows

Open the Services applet. Select each EDB Postgres Advanced Server or supporting component service displayed in the list, and select Stop.

To stop a service on Linux

Open a terminal window and manually stop each service at the command line.

**Step 5 for Linux only - Assume the identity of the cluster owner**

If you're using Linux, assume the identity of the EDB Postgres Advanced Server cluster owner. This example assumes EDB Postgres Advanced Server was installed in the default, compatibility-with-Oracle database mode, assigning `enterprisedb` as the cluster owner. (If installed in compatibility-with-PostgreSQL database mode, `postgres` is the cluster owner.)

```
su - enterprisedb
```

If prompted, enter the EDB Postgres Advanced Server cluster owner password. Then, set the path to include the location of the `pg_upgrade` executable:

```
export PATH=$PATH:/usr/edb/as14/bin
```

During the upgrade process, `pg_upgrade` writes a file to the current working directory of the `enterprisedb` user. You must invoke `pg_upgrade` from a directory where the `enterprisedb` user has write privileges. After performing the previous commands, navigate to a directory in which the `enterprisedb` user has sufficient privileges to write a file.

```
cd /tmp
```

**Step 5 for Windows only - Assume the identity of the cluster owner**
If you're using Windows, open a terminal window, assume the identity of the EDB Postgres Advanced Server cluster owner, and set the path to the `pg_upgrade` executable.

If the `--serviceaccount service_account_user` parameter was specified during the initial installation of EDB Postgres Advanced Server, then `service_account_user` is the EDB Postgres Advanced Server cluster owner. In that case, give this user with the `RUNAS` command:

```
RUNAS /USER:service_account_user "CMD.EXE"
```

During the upgrade process, `pg_upgrade` writes a file to the current working directory of the service account user. You must invoke `pg_upgrade` from a directory where the service account user has write privileges. After performing the previous commands, navigate to a directory in which the service account user has privileges to write a file:

```
cd %TEMP%
```

If you omitted the `--serviceaccount` parameter during the initial installation of EDB Postgres Advanced Server, then the default owner of the EDB Postgres Advanced Server service and the database cluster is `NT AUTHORITY\NetworkService`.

When `NT AUTHORITY\NetworkService` is the service account user, the `RUNAS` command might not be usable. It prompts for a password, and the `NT AUTHORITY\NetworkService` account isn’t assigned a password. Thus, there’s typically a failure with an error message such as “Unable to acquire user password.”

Under this circumstance, you must use the Windows utility program `PsExec` to run `CMD.EXE` as the service account `NT AUTHORITY\NetworkService`.

Obtain the `PsExec` program by downloading `PsTools`, which is available at the Microsoft site.

You can then use the following command to run `CMD.EXE` as `NT AUTHORITY\NetworkService`. Then set the path to the `pg_upgrade` executable:

```
pseexec.exe -u "NT AUTHORITY\NetworkService" CMD.EXE
```

During the upgrade process, `pg_upgrade` writes a file to the current working directory of the service account user. You must invoke `pg_upgrade` from a directory where the service account user has write privileges. After performing the previous commands, navigate to a directory in which the service account user has privileges to write a file:

```
cd %TEMP%
```

### Step 6 - Perform a consistency check

Before attempting an upgrade, perform a consistency check to ensure that the old and new clusters are compatible and properly configured. Include the `--check` option to instruct `pg_upgrade` to perform the consistency check.

This example shows invoking `pg_upgrade` to perform a consistency check on Linux:

```
pg_upgrade -d /var/lib/edb/as13/data
-D /var/lib/edb/as14/data -U enterprisedb
-b /usr/edb/as13/bin -B /usr/edb/as14/bin -p 5444 -P 5445 --check
```

If the command is successful, it returns `<Clusters are compatible>`.

If you’re using Windows, quote any directory names that contain a space:

```
pg_upgrade.exe
-d "C:\Program Files\PostgresPlus13AS\data"
-D "C:\Program Files\edb\as14\data" -U enterprisedb
-b "C:\Program Files\PostgresPlus13AS\bin" -B "C:\Program Files\edb\as14\bin" -p 5444 -P 5445 --check
```

During the consistency checking process, `pg_upgrade` logs any discrepancies that it finds to a file located in the directory from which you invoked `pg_upgrade`. When the consistency check completes, review the file to identify any missing components or upgrade conflicts. Resolve any conflicts before invoking `pg_upgrade` to perform a version upgrade.

If `pg_upgrade` alerts you to a missing component, you can use StackBuilder Plus to add the component that contains the component. Before using StackBuilder Plus, restart the EDB Postgres Advanced Server service. Then, open StackBuilder Plus by selecting from the Start menu `EDB Postgres Advanced Server version > StackBuilder Plus`. Follow the onscreen advice of the StackBuilder Plus wizard to download and install the missing components.

After `pg_upgrade` confirms that the clusters are compatible, you can perform a version upgrade.

### Step 7 - Run pg_upgrade
After confirming that the clusters are compatible, you can invoke `pg_upgrade` to upgrade the old cluster to the new version of EDB Postgres Advanced Server.

On Linux:

```
pg_upgrade -d /var/lib/edb/as13/data
-D /var/lib/edb/as14/data -U enterprisedb
-b /usr/edb/as13/bin -B /usr/edb/as14/bin -p 5444 -P 5445
```

On Windows:

```
pg_upgrade.exe -d "C:\Program Files\PostgresPlus\13AS\data"
-D "C:\Program Files\edb\as14\data" -U enterprisedb
-b "C:\Program Files\PostgresPlus\13AS\bin"
-B "C:\Program Files\edb\as14\bin" -p 5444 -P 5445
```

`pg_upgrade` displays the progress of the upgrade onscreen:

```
$ pg_upgrade -d /var/lib/edb/as13/data -D /var/lib/edb/as14/data -U enterprisedb
-b /usr/edb/as13/bin -B /usr/edb/as14/bin -p 5444 -P 5445
Performing Consistency Checks
------------------------------
Checking current, bin, and data directories          ok
Checking cluster versions                              ok
Checking database user is a superuser                ok
Checking for prepared transactions                   ok
Checking for reg* system OID user data types         ok
Checking for contrib/isn with bigint-passing mismatch ok
Creating catalog dump                                  ok
Checking for presence of required libraries          ok
Checking database user is a superuser                ok
Checking for prepared transactions                   ok

If `pg_upgrade` fails after this point, you must re-initdb the new cluster before continuing. Otherwise, it continues as follows:

```
Performing Upgrade
------------------
Analyzing all rows in the new cluster                ok
Freezing all rows on the new cluster                 ok
Deleting files from new pg_clog                      ok
Copying old pg_clog to new server                    ok
Setting next transaction ID for new cluster          ok
Resetting WAL archives                                ok
Setting frozenxid counters in new cluster            ok
Creating databases in the new cluster                ok
Adding support functions to new cluster              ok
Restoring database schema to new cluster             ok
Removing support functions from new cluster          ok
Copying user relation files                          ok
Setting next OID for new cluster                     ok
Creating script to analyze new cluster               ok
Creating script to delete old cluster                ok

Upgrade Complete
----------------
Optimizer statistics are not transferred by pg_upgrade so, once you start the new server, consider running:
   analyze_new_cluster.sh

Running this script will delete the old cluster's data files:
   delete_old_cluster.sh
```

While `pg_upgrade` runs, it might generate SQL scripts that handle special circumstances that it encountered during your upgrade. For example, if the old cluster contains large objects, you might need to invoke a script that defines the default permissions for the objects in the new cluster. When performing the pre-upgrade consistency check, `pg_upgrade` alerts you to any script that you might need to run manually.

You must invoke the scripts after `pg_upgrade` completes. To invoke the scripts, connect to the new cluster as a database superuser with the EDB-PSQL command-line client, and invoke each script using the `\i` option:

```
\i complete_path_to_script/script.sql
```

It's generally unsafe to access tables referenced in rebuild scripts until the rebuild scripts finish. Accessing the tables might yield incorrect results or poor performance. You can access tables not referenced in rebuild scripts immediately.
If `pg_upgrade` fails to complete the upgrade process, the old cluster is unchanged except that `$PGDATA/global/pg_control` is renamed to `pg_control.old` and each tablespace is renamed to `tablespace.old`. To revert to the pre-invocation state:

1. Delete any tablespace directories created by the new cluster.
2. Rename `$PGDATA/global/pg_control`, removing the `.old` suffix.
3. Rename the old cluster tablespace directory names, removing the `.old` suffix.
4. Remove any database objects from the new cluster that were moved before the upgrade failed.

Then, resolve any upgrade conflicts encountered and try the upgrade again.

When the upgrade is complete, `pg_upgrade` might also recommend vacuuming the new cluster. It provides a script that allows you to delete the old cluster.

**Note**

In case you need to revert to a previous version, before removing the old cluster, make sure that you have a backup of the cluster and that the cluster was upgraded.

**Step 8 - Restore the authentication settings in the pg_hba.conf file**

If you modified the `pg_hba.conf` file to permit `trust` authentication, update the contents of the `pg_hba.conf` file to reflect your preferred authentication settings.

**Step 9 - Move and identify user-defined tablespaces (optional)**

If you have data stored in a user-defined tablespace, you must manually relocate tablespace files after upgrading. Move the files to the new location, and update the symbolic links to point to the files. The symbolic links are located in the `pg_tblspc` directory under your cluster's `data` directory.

### 7.2.5 Upgrading a pgAgent installation

If your existing EDB Postgres Advanced Server installation uses pgAgent, you can use a script provided with the EDB Postgres Advanced Server installer to update pgAgent. The script is named `dbms_job.upgrade.script.sql` and is located in the `/share/contrib/` directory under your EDB Postgres Advanced Server installation.

If you're using `pg_upgrade` to upgrade your installation:

1. Perform the upgrade.
2. Invoke the `dbms_job.upgrade.script.sql` script to update the catalog files. If your existing pgAgent installation was performed with a script, the update converts the installation to an extension.

### 7.2.6 Troubleshooting pg_upgrade

These troubleshooting tips address problems you might encounter when using `pg_upgrade`.

**Upgrade Error - There seems to be a postmaster servicing the cluster**

If `pg_upgrade` reports that a postmaster is servicing the cluster, stop all EDB Postgres Advanced Server services and try the upgrade again.

**Upgrade Error - fe_sendauth: no password supplied**

If `pg_upgrade` reports an authentication error that references a missing password, modify the `pg_hba.conf` files in the old and new cluster to enable `trust` authentication, or configure the system to use a `pgpass.conf` file.

**Upgrade Error - New cluster is not empty; exiting**

If `pg_upgrade` reports that the new cluster isn't empty, empty the new cluster. The target cluster might not contain any user-defined databases.
Upgrade Error - Failed to load library

If the original EDB Postgres Advanced Server cluster included libraries that aren't included in the EDB Postgres Advanced Server cluster, `pg_upgrade` alerts you to the missing component during the consistency check by writing an entry to the `loadable_libraries.txt` file in the directory from which you invoked `pg_upgrade`. Generally, for missing libraries that aren't part of a major component upgrade:

1. Restart the EDB Postgres Advanced Server service.
2. Use StackBuilder Plus to download and install the missing module.
3. Stop the EDB Postgres Advanced Server service.
4. Resume the upgrade process. Invoke `pg_upgrade` to perform consistency checking.
5. After you resolve any remaining problems noted in the consistency checks, invoke `pg_upgrade` to perform the data migration from the old cluster to the new cluster.

### 7.2.7 Reverting to the old cluster

The method you use to revert to a previous cluster varies with the options specified when invoking `pg_upgrade`:

- If you specified the `--check` option when invoking `pg_upgrade`, an upgrade wasn't performed and no modifications were made to the old cluster. You can reuse the old cluster at any time.
- If you included the `--link` option when invoking `pg_upgrade`, the data files are shared between the old and new cluster after the upgrade completes. If you started the server that's servicing the new cluster, the new server wrote to those shared files, and it's unsafe to use the old cluster.
- If you ran `pg_upgrade` without the `--link` specification or haven't started the new server, the old cluster is unchanged except that the `.old` suffix was appended to the `$PGDATA/global/pg_control` and tablespace directories. To reuse the old cluster, delete the tablespace directories created by the new cluster. Remove the `.old` suffix from `$PGDATA/global/pg_control` and the old cluster tablespace directory names. Restart the server that services the old cluster.

### 7.3 Performing a minor version update of an RPM installation

If you used an RPM package to install EDB Postgres Advanced Server or its supporting components, you can use `yum` to perform a minor version upgrade to a more recent version. To review a list of the package updates that are available for your system, open a command line, assume root privileges, and enter the command:

```
yum check-update <package_name>
```

Where `<package_name>` is the search term for which you want to search for updates. You can include wildcard values in the search term. To use `yum update` to install an updated package, use the command:

```
yum update <package_name>
```

Where `<package_name>` is the name of the package you want to update. Include wildcard values in the update command to update multiple related packages with a single command. For example, use the following command to update all packages whose names include the expression `edb`:

```
yum update edb*
```

**Note**

The `yum update` command performs an update only between minor releases. To update between major releases, use `pg_upgrade`.

For more information about using `yum` commands and options, enter `yum --help` at the command line.

**Important**

If upgrading to version 15.4 or later, run `edb_sqlpatch`.

The command might respond that it has a number of patches needing to be applied, for example:

```
* database edb
  0 patches were previously applied to this database.
  58 patches need to be applied to this database.
```

In this case, you need to run `edb_sqlpatch` to patch the system catalog:

```
edb_sqlpatch -af
```
### 7.4 Using StackBuilder Plus to perform a minor version update

Note

StackBuilder Plus is supported only on Windows systems.

The StackBuilder Plus utility provides a graphical interface that simplifies the process of updating, downloading, and installing modules that complement your EDB Postgres Advanced Server installation. When you install a module with StackBuilder Plus, StackBuilder Plus resolves any software dependencies.

1. To invoke StackBuilder Plus at any time after the installation has completed, select **Apps > StackBuilder Plus**. Enter your system password if prompted, and the StackBuilder Plus welcome window opens.

![StackBuilder Plus welcome window](image)

2. Select your EDB Postgres Advanced Server installation.

StackBuilder Plus requires Internet access. If your installation of EDB Postgres Advanced Server is behind a firewall (with restricted Internet access), StackBuilder Plus can download program installers through a proxy server. The module provider determines if the module can be accessed through an HTTP proxy or an FTP proxy. Currently, all updates are transferred by an HTTP proxy, and the FTP proxy information isn’t used.

3. If the selected EDB Postgres Advanced Server installation has restricted Internet access, on the Welcome screen, select **Proxy Servers** to open the Proxy servers dialog box:

![Proxy servers dialog box](image)

4. On the dialog box, enter the IP address and port number of the proxy server in the **HTTP proxy** box. Currently, all StackBuilder Plus modules are distributed by HTTP proxy (FTP proxy information is ignored). Select **OK**.

---

For more information about using `edb_sqlpatch` commands and options, see `edb_sqlpatch`. 
The tree control on the StackBuilder Plus module selection window displays a node for each module category.

5. To add a component to the selected EDB Postgres Advanced Server installation or to upgrade a component, select the box to the left of the module name, and select Next.

6. If prompted, enter your email address and password on the StackBuilder Plus registration window.

StackBuilder Plus confirms the packages selected. The Selected packages dialog box displays the name and version of the installer. Select Next.

When the download completes, a window opens that confirms the installation files were downloaded and are ready for installation.
7. Leave the **Skip Installation** check box cleared and select **Next** to start the installation process. (Select the check box and select **Next** to exit StackBuilder Plus without installing the downloaded files.)

When the upgrade is complete, StackBuilder Plus alerts you to the success or failure of the installation of the requested package. If you were prompted by an installer to restart your computer, restart now.

If the update fails to install, StackBuilder Plus alerts you to the installation error and writes a message to the log file at `%TEMP%`.

8. **Database administration**

EDB Postgres Advanced Server includes features to help you to maintain, secure, and operate EDB Postgres Advanced Server databases.

8.1. **Setting configuration parameters**

The EDB Postgres Advanced Server configuration parameters control various aspects of the database server's behavior and environment such as data file and log file locations, connection, authentication and security settings, resource allocation and consumption, archiving and replication settings, error logging and statistics gathering, optimization and performance tuning, and locale and formatting settings.

Configuration parameters that apply only to EDB Postgres Advanced Server are noted in **Summary of configuration parameters**, which lists all EDB Postgres Advanced Server configuration parameters along with a number of key attributes of the parameters.

You can find more information about configuration parameters in the **PostgreSQL core documentation**.

8.1.1. **Setting configuration parameters**

Set each configuration parameter using a name/value pair. Parameter names aren't case sensitive. The parameter name is typically separated from its value by an optional equals sign (`=`).

This example shows some configuration parameter settings in the `postgresql.conf` file:

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = "$user", public
shared_buffers = 128MB
```

**Types of parameter values**

Parameter values are specified as one of five types:
Some settings specify a memory or time value. Each of these has an implicit unit, which is kilobytes, blocks (typically 8 kilobytes), milliseconds, seconds, or minutes. You can find default units by referencing the system view `pg_settings.unit`. You can specify a different unit explicitly.

Valid memory units are:

- **kB** (kilobytes)
- **MB** (megabytes)
- **GB** (gigabytes)

Valid time units are:

- **ms** (milliseconds)
- **s** (seconds)
- **min** (minutes)
- **h** (hours)
- **d** (days)

The multiplier for memory units is 1024.

Specifying configuration parameter settings

A number of parameter settings are set when the EDB Postgres Advanced Server database product is built. These are read-only parameters, and you can't change their values. A couple of parameters are also permanently set for each database when the database is created. These parameters are read-only and you can't later change them for the database. However, there are a number of ways to specify the configuration parameter settings:

- The initial settings for almost all configurable parameters across the entire database cluster are listed in the `postgresql.conf` configuration file. These settings are put into effect upon database server start or restart. You can override some of these initial parameter settings. All configuration parameters have built-in default settings that are in effect unless you explicitly override them. The settings in `postgresql.conf` file are overridden when the same parameters are included in the `postgresql.auto.conf` file. Use the `ALTER SYSTEM` command to manage the configuration parameters in the `postgresql.auto.conf` file.

- Configuration parameters in the `postgresql.conf` file are overridden when the same parameters are included in the `postgresql.auto.conf` file. Use the `ALTER SYSTEM` command to manage the configuration parameters in the `postgresql.auto.conf` file. You can modify parameter settings in the configuration file while the database server is running. If the configuration file is then reloaded (meaning a SIGHUP signal is issued), for certain parameter types, the changed parameters settings immediately take effect. For some of these parameter types, the new settings are available in a currently running session immediately after the reload. For others, you must start a new session to use the new settings. And for some others, modified settings don't take effect until the database server is stopped and restarted. See the PostgreSQL core documentation for information on how to reload the configuration file.

- You can use the SQL commands `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` to modify certain parameter settings. The modified parameter settings take effect for new sessions after you execute the command. `ALTER DATABASE` affects new sessions connecting to the specified database. `ALTER ROLE` affects new sessions started by the specified role. `ALTER ROLE IN DATABASE` affects new sessions started by the specified role connecting to the specified database. Parameter settings established by these SQL commands remain in effect indefinitely, across database server restarts, overriding settings established by the other methods. You can change parameter settings established using the `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` commands by either:
  - Reissuing these commands with a different parameter value.
  - Issuing these commands using the `SET parameter TO DEFAULT` clause or the `RESET parameter` clause. These clauses change the parameter back to using the setting set by the other methods. See the PostgreSQL core documentation for the syntax of these SQL commands.

- You can make changes for certain parameter settings for the duration of individual sessions using the `PGOPTIONS` environment variable or by using the `SET` command in the EDB-PSQL or PSQL command-line programs. Parameter settings made this way override settings established using any of the methods discussed earlier, but only during that session.

Modifying the postgresql.conf file

The configuration parameters in the `postgresql.conf` file specify server behavior with regard to auditing, authentication, encryption, and other behaviors. On Linux and Windows hosts, the `postgresql.conf` file resides in the `data` directory under your EDB Postgres Advanced Server installation.

Parameters that are preceded by a pound sign (`#`) are set to their default value. To change a parameter value, remove the pound sign and enter a new value. After setting or changing a parameter, you must either `reloads` or `restarts` the server for the new parameter value to take effect.

In the `postgresql.conf` file, some parameters contain comments that indicate `change requires restart`. To view a list of the parameters that require a server restart, use the following query at the psql command line:

```
SELECT name FROM pg_settings WHERE context = 'postmaster';
```
Modifying the pg_hba.conf file

Appropriate authentication methods provide protection and security. Entries in the pg_hba.conf file specify the authentication methods that the server uses with connecting clients. Before connecting to the server, you might need to modify the authentication properties specified in the pg_hba.conf file.

When you invoke the initdb utility to create a cluster, the utility creates a pg_hba.conf file for that cluster that specifies the type of authentication required from connecting clients. You can modify this file. After modifying the authentication settings in the pg_hba.conf file, restart the server and apply the changes. For more information about authentication and modifying the pg_hba.conf file, see the PostgreSQL core documentation.

When the server receives a connection request, it verifies the credentials provided against the authentication settings in the pg_hba.conf file before allowing a connection to a database. To log the pg_hba.conf file entry to authenticate a connection to the server, set the log_connections parameter to ON in the postgresql.conf file.

A record specifies a connection type, database name, user name, client IP address, and the authentication method to authorize a connection upon matching these parameters in the pg_hba.conf file. Once the connection to a server is authorized, you can see the matched line number and the authentication record from the pg_hba.conf file.

This example shows a log detail for a valid pg_hba.conf entry after successful authentication:

```
2020-05-08 18:42:17 IST LOG:  connection received: host=[local]
2020-05-08 18:42:17 IST LOG:  connection authorized: user=u1 database=edb
application_name=psql
2020-05-08 18:42:17 IST DETAIL:  Connection matched pg_hba.conf line 84:
"local   all             all               md5"
```

Obfuscating the LDAP password

When using LDAP for authentication, the LDAP password used to connect to the LDAP server (the ldapbindpasswd password) is stored in the pg_hba.conf file. You can store the password there in an obfuscated form, which can then be de-obfuscated by a loadable module that you supply. The loadable module supplies a hook function that performs the de-obfuscation.

For example, this C-loadable module uses rot13_passphrase as the hook function to de-obfuscate the password from the pg_hba.conf file:

```
#include "postgres.h"
#include <float.h>
#include <stdio.h>
#include "libpq/libpq.h"
#include "libpq/libpq-be.h"
#include "libpq/auth.h"
#include "utils/guc.h"

PG_MODULE_MAGIC;

void _PG_init(void);
void _PG_fini(void);
/* hook function */
static char* rot13_passphrase(char *password);

/* Module load callback */
void _PG_init(void)
{
    ldap_password_hook = rot13_passphrase;
}

void _PG_fini(void)
{
    /* do nothing yet */
}

static char* rot13_passphrase(char *pw)
{
    size_t size = strlen(pw) + 1;
    ```
Add your module to the `shared_preload_libraries` parameter in the `postgresql.conf` file. For example:

```
shared_preload_libraries = '$libdir/ldap_password_func'
```

Restart your server to load the changes in this parameter.

---

## 8.1.2 Configuration parameters by functionality

Configuration parameters can be grouped by function. See [Description of parameter attributes](#) for the list of attributes included in each configuration parameter description.

### 8.1.2.1 Top performance-related parameters

These configuration parameters have the most immediate impact on performance.

#### 8.1.2.1.1 `shared_buffers`

Parameter type: Integer

Default value: 32MB

Range: 128kB to system dependent

Minimum scope of effect: Cluster

When value changes take effect: Restart

Required authorization to activate: EDB Postgres Advanced Server service account

Sets the amount of memory the database server uses for shared memory buffers. The default is typically 32MB but might be less if your kernel settings don't support it, as determined during `initdb`. This setting must be at least 128kB. (Nondefault values of `BLCKSZ` change the minimum.) However, you usually need settings significantly higher than the minimum for good performance.

If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for `shared_buffers` is 25% of the memory in your system. For some workloads, even large settings for `shared_buffers` are effective. However, because EDB Postgres Advanced Server also relies on the operating system cache, allocating more than 40% of RAM to `sharedBuffers` isn't likely to work better than a smaller amount.

On systems with less than 1GB of RAM, a smaller percentage of RAM is appropriate to leave space for the operating system. Fifteen percent of memory is more typical in these situations. On Windows, large values for `shared_buffers` aren't as effective. You might have better results keeping the setting relatively low and using the operating system cache more instead. The useful range for `shared_buffers` on Windows systems is generally from 64MB to 512MB.

Increasing this parameter might cause EDB Postgres Advanced Server to request more System V shared memory than your operating system's default configuration allows. See [Shared Memory and Semaphores in the PostgreSQL core documentation](#) for information on how to adjust those parameters.

#### 8.1.2.1.2 `work_mem`

---

EDB Postgres Advanced Server (EPAS)

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Parameter type: Integer
Default value: 1MB
Range: 64kB to 2097151kB
Minimum scope of effect: Per session
When value changes take effect: Immediate
Required authorization to activate: Session user

Specifies the amount of memory for internal sort operations and hash tables to use before writing to temporary disk files.

For a complex query, several sort or hash operations might run in parallel. Each operation is allowed to use as much memory as this value specifies before it starts to write data into temporary files. Also, several running sessions might perform such operations concurrently. Therefore, the total memory used might be many times the value of `work_mem` keep this information in mind when choosing the value.

Sort operations are used for ORDER BY, DISTINCT, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of IN subqueries.

Reasonable values are typically between 4MB and 64MB, depending on the size of your machine, how many concurrent connections you expect (determined by `max_connections`), and the complexity of your queries.

### 8.1.2.1.3 maintenance_work_mem

Parameter type: Integer
Default value: 16MB
Range: 1024kB to 2097151kB
Minimum scope of effect: Per session
When value changes take effect: Immediate
Required authorization to activate: Session user

Specifies the maximum amount of memory used by maintenance operations such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY. It defaults to 16MB. Since a database session can execute only one of these operations at a time, and an installation normally doesn't have many of them running concurrently, it's safe to set this value significantly larger than `work_mem`. Larger settings might improve performance for vacuuming and for restoring database dumps.

When autovacuum runs, you can allocate up to `autovacuum_max_workers` times, so be careful not to set the default value too high.

A good rule of thumb is to set this to about 5% of system memory but not more than about 512MB. Larger values don't necessarily improve performance.

### 8.1.2.1.4 wal_buffers

Parameter type: Integer
Default value: 64kB
Range: 32kB to system dependent
Minimum scope of effect: Cluster
When value changes take effect: Restart
Required authorization to activate: EDB Postgres Advanced Server service account

The amount of memory used in shared memory for WAL data. Because the data is written out to disk at every transaction commit, the setting must be large enough only to hold the amount of WAL data generated by one typical transaction.

Increasing this parameter might cause EDB Postgres Advanced Server to request more System V shared memory than your operating system's default configuration allows. See Shared Memory and Semaphores in the PostgreSQL core documentation for information on how to adjust those parameters.

Although even this very small setting doesn't always cause a problem, in some situations it can result in extra fsync calls and degrade overall system throughput. Increasing this value to about 1MB can address this problem. On very busy systems, you might need an even higher value, up to a maximum of about 16MB. Like `shared_buffers`, this parameter increases EDB Postgres Advanced
Server's initial shared memory allocation. If increasing it causes an EDB Postgres Advanced Server start failure, increase the operating system limit.

8.1.2.1.5 checkpoint_segments

Deprecated. This parameter isn’t supported by EDB Postgres Advanced Server.

8.1.2.1.6 checkpoint_completion_target

Parameter type: Floating point

Default value: 0.5

Range: 0 to 1

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Specifies the target of checkpoint completion as a fraction of total time between checkpoints. This parameter spreads out the checkpoint writes while the system starts working toward the next checkpoint.

The default of 0.5 aims to finish the checkpoint writes when 50% of the next checkpoint is ready. A value of 0.9 aims to finish the checkpoint writes when 90% of the next checkpoint is done. This value throttles the checkpoint writes over a larger amount of time and avoids spikes of performance bottlenecks.

8.1.2.1.7 checkpoint_timeout

Parameter type: Integer

Default value: 5min

Range: 30s to 3600s

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Maximum time between automatic WAL checkpoints, in seconds. Increasing this parameter can increase the amount of time needed for crash recovery.

Increasing checkpoint_timeout to a larger value, such as 15 minutes, can reduce the I/O load on your system, especially when using large values for shared_buffers.

The downside of making these adjustments to the checkpoint parameters is that your system uses a modest amount of additional disk space and takes longer to recover in the event of a crash. However, for most users, this is worth it for a significant performance improvement.

8.1.2.1.8 max_wal_size

Parameter type: Integer

Default value: 1 GB

Range: 2 to 2147483647

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account
**max_wal_size** specifies the maximum size for the WAL to reach between automatic WAL checkpoints. This is a soft limit. WAL size can exceed **max_wal_size** under special circumstances, such as when under a heavy load, with a failing archive_command, or with a high **wal_keep_segments** setting.

Increasing this parameter can increase the amount of time needed for crash recovery. You can set this parameter in the **postgresql.conf** file or on the server command line.

### 8.1.2.1.9 min_wal_size

**Parameter type:** Integer  
**Default value:** 80 MB  
**Range:** 2 to 2147483647  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Reload  
**Required authorization to activate:** EDB Postgres Advanced Server service account

If WAL disk usage stays below the value specified by **min_wal_size**, old WAL files are recycled for future use at a checkpoint rather than removed. This feature ensures that enough WAL space is reserved to handle spikes in WAL usage, like when running large batch jobs. You can set this parameter in the **postgresql.conf** file or on the server command line.

### 8.1.2.1.10 bgwriter_delay

**Parameter type:** Integer  
**Default value:** 200ms  
**Range:** 10ms to 10000ms  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Reload  
**Required authorization to activate:** EDB Postgres Advanced Server service account

Specifies the delay between activity rounds for the background writer. In each round, the writer issues writes for some number of dirty buffers. (You can control the number of dirty buffers using the **bgwriter_lru_maxpages** and **bgwriter_lru_multiplier** parameters.) It then sleeps for **bgwriter_delay** milliseconds, and repeats.

On many systems, the effective resolution of sleep delays is 10ms. Setting **bgwriter_delay** to a value that isn’t a multiple of 10 might have the same results as setting it to the next higher multiple of 10.

Typically, when tuning **bgwriter_delay**, you reduce it from its default value. This parameter is rarely increased. Saving on power consumption on a system with very low use is one case when you might increase the value.

### 8.1.2.1.11 seq_page_cost

**Parameter type:** Floating point  
**Default value:** 1  
**Range:** 0 to 1.79769e+308  
**Minimum scope of effect:** Per session  
**When value changes take effect:** Immediate  
**Required authorization to activate:** Session user

Sets the planner's estimate of the cost of a disk page fetch that's part of a series of sequential fetches. You can override this value for a particular tablespace by setting the tablespace parameter of the same name. See **ALTER TABLESPACE** in the PostgreSQL core documentation.

The default value assumes very little caching, so it's often a good idea to reduce it. Even if your database is significantly larger than physical memory, you might want to try setting this parameter to something lower than the default value of 1 to see if you get better query plans that way. If your database fits entirely in memory, you can lower this value much more, for example, to 0.1.
8.1.2.1.12 random_page_cost

Parameter type: Floating point

Default value: 4

Range: 0 to 1.79769e+308

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

Sets the planner's estimate of the cost of a nonsequentially fetched disk page. You can override the default for a particular tablespace by setting the tablespace parameter of the same name. See \texttt{ALTER TABLESPACE} in the PostgreSQL core documentation.

Reducing this value relative to \texttt{seq_page_cost} causes the system to prefer index scans. Raising it makes index scans look relatively more expensive. You can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs, which are described by the \texttt{cpu_tuple_cost} and \texttt{cpu_index_tuple_cost} parameters.

The default value assumes very little caching, so it's often a good idea to reduce it. Even if your database is significantly larger than physical memory, you might want to try setting this parameter to 2 (that is, lower than the default) to see whether you get better query plans that way. If your database fits entirely in memory, you can lower this value much more, for example, to 0.1.

Although the system allows it, never set \texttt{random_page_cost} less than \texttt{seq_page_cost}. However, setting them equal or very close to equal makes sense if the database fits mostly or entirely in memory, since in that case there’s no penalty for touching pages out of sequence. Also, in a heavily cached database, lower both values relative to the CPU parameters, since the cost of fetching a page already in RAM is much smaller than it normally is.

8.1.2.1.13 effective_cache_size

Parameter type: Integer

Default value: 128MB

Range: 8kB to 17179869176kB

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

Sets the planner's assumption about the effective size of the disk cache that's available to a single query. This assumption is factored into estimates of the cost of using an index. A higher value makes it more likely index scans are used. A lower value makes it more likely sequential scans are used.

When setting this parameter, consider both EDB Postgres Advanced Server's shared buffers and the portion of the kernel's disk cache that are used for EDB Postgres Advanced Server data files. Also, consider the expected number of concurrent queries on different tables, since they have to share the available space.

This parameter doesn't affect the size of shared memory allocated by EDB Postgres Advanced Server, and it doesn't reserve kernel disk cache. Use it only for estimating.

If this parameter is set too low, the planner might decide not to use an index even when it's helpful to do so. Setting \texttt{effective_cache_size} to 50% of physical memory is a normal, conservative setting. A more aggressive setting is approximately 75% of physical memory.

8.1.2.1.14 synchronous_commit

Parameter type: Boolean

Default value: true

Range: \{true | false\}

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user
Specifies whether a transaction commit waits for WAL records to be written to disk before the command returns a success indication to the client. The safe setting is on, which is the default. When off, there can be a delay between when success is reported to the client and when the transaction is really guaranteed to be safe against a server crash. (The maximum delay is three times wal_writer_delay.)

Unlike fsync, setting this parameter to off does not create any risk of database inconsistency. An operating system or database crash might result in some recent "allegedly committed" transactions being lost. However, the database state is the same as if those transactions aborted cleanly.

So, turning synchronous_commit off can be a useful alternative when performance is more important than exact certainty about the durability of a transaction. See Asynchronous Commit in the PostgreSQL core documentation for information.

You can change this parameter at any time. The behavior for any one transaction is determined by the setting in effect when it commits. It's therefore possible and useful to have some transactions commit synchronously and others asynchronously. For example, to make a single multistatement transaction commit asynchronously when the default is the opposite, issue SET LOCAL synchronous_commit TO OFF in the transaction.

### 8.1.2.1.15  edb_max_spins_per_delay

**Parameter type:** Integer

**Default value:** 1000

**Range:** 10 to 1000

**Minimum scope of effect:** Per cluster

**When value changes take effect:** Restart

**Required authorization to activate:** EDB Postgres Advanced Server service account

Specifies the maximum number of times that a session spins while waiting for a spin lock. If a lock isn't acquired, the session sleeps.

This parameter can be useful for systems that use non-uniform memory access (NUMA) architecture.

### 8.1.2.1.16  pg_prewarm.autoprewarm

**Parameter type:** Boolean

**Default value:** true

**Range:** {true | false}

**Minimum scope of effect:** Cluster

**When value changes take effect:** Restart

**Required authorization to activate:** EDB Postgres Advanced Server service account

Controls whether the database server runs autoprewarm, a background worker process that dumps shared buffers to disk before a shutdown. It then prewarms the shared buffers the next time the server is started, meaning it loads blocks from the disk back into the buffer pool.

The advantage to this parameter is that it shortens the warmup times after the server restarts by loading the data that was dumped to disk before shutdown.

Set pg_prewarm.autoprewarm to on to enable the autoprewarm worker. Set it to off to disable autoprewarm.

Before you can use autoprewarm, you must add $libdir/pg_prewarm to the libraries listed in the shared_preload_libraries configuration parameter of the postgresql.conf file, as this example shows:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/dbms_aq,$libdir/pg_prewarm'
```

After modifying the shared_preload_libraries parameter, restart the database server. After the restart, the autoprewarm background worker launches immediately after the server reaches a consistent state.

The autoprewarm process starts loading blocks that were previously recorded in SPGDATA/autoprewarm.blocks until no free buffer space is left in the buffer pool. In this manner, any new blocks that were loaded either by the recovery process or by the querying clients aren't replaced.

Once the autoprewarm process finishes loading buffers from disk, it periodically dumps shared buffers to disk at the interval specified by the pg_prewarm.autoprewarm_interval parameter. At the next server restart, the autoprewarm process prewarms shared buffers with the blocks that were last dumped to disk.
See `pg_prewarm.autoprewarm_interval` for information on the autoprewarm background worker.

### 8.1.2.17 pg_prewarm.autoprewarm_interval

**Parameter type:** Integer  
**Default value:** 300s  
**Range:** 0s to 2147483s  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Reload  
**Required authorization to activate:** EDB Postgres Advanced Server service account

The minimum number of seconds after which the autoprewarm background worker dumps shared buffers to disk. If set to 0, shared buffers aren't dumped at regular intervals. They're dumped only when you shut down the server.

See the `pg_prewarm.autoprewarm` for information on the autoprewarm background worker.

### 8.1.2  Resource usage/memory

These configuration parameters control resource use pertaining to memory.

**edb_dynatune**

**Parameter type:** Integer  
**Default value:** 0  
**Range:** 0 to 100  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Restart  
**Required authorization to activate:** EDB Postgres Advanced Server service account

Determines how much of the host system’s resources for the database server to use based on the host machine’s total available resources and the intended use of the host machine.

When you first install EDB Postgres Advanced Server, you set `edb_dynatune` according to the use of the host machine on which it was installed, that is, development machine, mixed-use machine, or dedicated server. For most purposes, the database administrator doesn't need to adjust the various configuration parameters in the `postgresql.conf` file to improve performance.

You can set the `edb_dynatune` parameter to any integer value from 0 to 100. A value of 0 turns off the dynamic tuning feature, which leaves the database server resource use under the control of the other configuration parameters in the `postgresql.conf` file.

A low, non-zero value, for example, 1 to 33, dedicates the least amount of the host machine’s resources to the database server. These values are suitable for a development machine where many other applications are being used.

A value in the range of 34 to 66 dedicates a moderate amount of resources to the database server. This setting might be used for a dedicated application server that has a fixed number of other applications running on the same machine as EDB Postgres Advanced Server.

The highest values of 67 to 100 dedicate most of the server’s resources to the database server. Use settings in this range for a host machine that’s dedicated to running EDB Postgres Advanced Server.

After you select a value for `edb_dynatune`, you can further fine-tune database server performance by adjusting the other configuration parameters in the `postgresql.conf` file. Any adjusted setting overrides the corresponding value chosen by `edb_dynatune`. To change the value of a parameter, uncomment the configuration parameter, specify the desired value, and restart the database server.

**edb_dynatune_profile**

**Parameter type:** Enum

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
8.1.2.3 Resource usage/EDB Resource Manager

These configuration parameters control resource use through EDB Resource Manager.

`edb_max_resource_groups`

Parameter type: Integer
Default value: 16
Range: 0 to 65536
Minimum scope of effect: Cluster
When value changes take effect: Restart
Required authorization to activate: EDB Postgres Advanced Server service account

Controls the maximum number of resource groups that EDB Resource Manager can use simultaneously. You can create more resource groups than the value specified by `edb_max_resource_groups`. However, the number of resource groups in active use by processes in these groups can't exceed this value.

Set this parameter large enough to handle the number of groups you expect to maintain.

`edb_resource_group`

Parameter type: String
Default value: none
Range: n/a
Minimum scope of effect: Per session
When value changes take effect: Immediate
Required authorization to activate: Session user

The name of the resource group for EDB Resource Manager to control in the current session according to the group's resource type settings.

If you don't set this parameter, then the current session doesn't use EDB Resource Manager.

8.1.2.4 Query tuning
These configuration parameters are used for optimizer hints.

**enable_hints**

*Parameter type:* Boolean  
*Default value:* true  
*Range:* {true | false}  
*Minimum scope of effect:* Per session  
*When value changes take effect:* Immediate  
*Required authorization to activate:* Session user

Enables optimizer hints embedded in SQL commands. Optimizer hints are ignored when this parameter is off.

### 8.1.2.5 Query tuning/planner method configuration

These configuration parameters are used for planner method configuration.

**edb_enable_pruning**

*Parameter type:* Boolean  
*Default value:* false  
*Range:* {true | false}  
*Minimum scope of effect:* Per session  
*When value changes take effect:* Immediate  
*Required authorization to activate:* Session user

Allows the query planner to **early prune** partitioned tables. Early pruning means that the query planner can "prune" (that is, ignore) partitions that aren't searched in a query before generating query plans. This setting helps improve performance because it prevents generating query plans of partitions that aren't searched.

Conversely, **late pruning** means that the query planner prunes partitions after generating query plans for each partition. The `constraint_exclusion` configuration parameter controls late pruning.

The ability to early prune depends on the nature of the query in the **WHERE** clause. You can use early pruning in only simple queries with constraints like **WHERE column = literal**, for example, **WHERE deptno = 10**.

Don't use early pruning for more complex queries such as **WHERE column = expression**, for example, **WHERE deptno = 10 + 5**.

This parameter is deprecated from version 15 and later. Use `enable_partition_pruning` instead.

### 8.1.2.6 Reporting and logging/what to log

These configuration parameters control reporting and logging.

**trace_hints**

*Parameter type:* Boolean  
*Default value:* false
Use with the optimizer hints feature to provide more detailed information about whether the planner used a hint. Set the `client_min_messages` and `trace_hints` configuration parameters as follows:

```
SET client_min_messages TO info;
SET trace_hints TO true;
```

This example shows how the `SELECT` command with the `NO_INDEX` hint displays the added information produced when you set those configuration parameters:

```
EXPLAIN SELECT /*+ NO_INDEX(accounts accounts_pkey) */ FROM accounts
WHERE aid = 108;
```

```
INFO: [HINTS] Index Scan of [accounts].[accounts_pkey] rejected because of NO_INDEX hint.
INFO: [HINTS] Bitmap Heap Scan of [accounts].[accounts_pkey] rejected because of NO_INDEX hint.
```

**QUERY PLAN**

```
Seq Scan on accounts (cost=0.00..14461.10 rows=1 width=97)
  Filter: (aid = 108)
(2 rows)
```

**edb_log_everyBulk_value**

Parameter type: Boolean

Default value: `false`

Range: `{true | false}`

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Superuser

Bulk processing logs the resulting statements into both the EDB Postgres Advanced Server log file and the EDB Audit log file. However, logging every statement in bulk processing is costly. You can control the bulk processing statements that are logged with the `edb_log_everyBulk_value` configuration parameter.

When this parameter is set to `true`, every statement in bulk processing is logged. During bulk execution, when `edb_log_everyBulk_value` is set to `false`, a log message is recorded once per bulk processing along with the number of rows processed. The duration is emitted once per bulk processing.

### 8.1.2.7 Auditing settings

These configuration parameters are for use with the EDB Postgres Advanced Server database auditing feature.

#### 8.1.2.7.1 edb_audit
Enables or disables database auditing. The values `xml` or `csv` enable database auditing. These values determine the file format in which to capture auditing information. `none` disables database auditing.

### 8.1.2.7.2 `edb_audit_directory`

**Parameter type:** String  
**Default value:** `edb_audit`  
**Range:** n/a  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Reload  
**Required authorization to activate:** EDB Postgres Advanced Server service account

Specifies the directory where the audit log files are created. Specify either an absolute path or a path relative to the EDB Postgres Advanced Server `data` directory.

### 8.1.2.7.3 `edb_audit_filename`  

**Parameter type:** String  
**Default value:** `audit-%Y%m%d_%H%M%S`  
**Range:** n/a  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Reload  
**Required authorization to activate:** EDB Postgres Advanced Server service account

Specifies the file name of the audit file where the auditing information is stored. The default file name is `audit-%Y%m%d_%H%M%S`. The escape sequences, such as `%Y` and `%m`, are replaced by the appropriate current values of the system date and time.

### 8.1.2.7.4 `edb_audit_rotation_day`  

**Parameter type:** String  
**Default value:** `every`  
**Range:** `{none | every | sun | mon | tue | wed | thu | fri | sat ...}`  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Reload  
**Required authorization to activate:** EDB Postgres Advanced Server service account

Specifies the day of the week on which to rotate the audit files. Valid values are `sun`, `mon`, `tue`, `wed`, `thu`, `fri`, `sat`, `every`, and `none`. To disable rotation, set the value to `none`. To
rotate the file every day, set the `edb_audit_rotation_day` value to `every`. To rotate the file on a specific day of the week, set the value to the desired day of the week.

### 8.1.2.7.5 edb_audit_rotation_size

**Parameter type:** Integer  
**Default value:** 0MB  
**Range:** 0MB to 5000MB  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Reload  
**Required authorization to activate:** EDB Postgres Advanced Server service account  

Specifies a file size threshold in megabytes when file rotation occurs. If the parameter is commented out or set to 0, rotating the file based on size doesn’t occur.

### 8.1.2.7.6 edb_audit_rotation_seconds

**Parameter type:** Integer  
**Default value:** 0s  
**Range:** 0s to 2147483647s  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Reload  
**Required authorization to activate:** EDB Postgres Advanced Server service account  

Specifies the rotation time in seconds when a new log file is created. To disable this feature, leave this parameter set to 0.

### 8.1.2.7.7 edb_audit_connect

**Parameter type:** Enum  
**Default value:** `failed`  
**Range:** `{none | failed | all}`  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Reload  
**Required authorization to activate:** EDB Postgres Advanced Server service account  

Enables auditing of database connection attempts by users. To disable auditing of all connection attempts, set `edb_audit_connect` to `none`. To audit all failed connection attempts, set the value to `failed`. To audit all connection attempts, set the value to `all`.

### 8.1.2.7.8 edb_audit_disconnect

**Parameter type:** Enum  
**Default value:** `none`  
**Range:** `{none | all}`  
**Minimum scope of effect:** Cluster
When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Enables auditing of database disconnections by connected users. To enable auditing of disconnections, set the value to all. To disable, set the value to none.

8.1.2.7.9   edb_audit_statement

Parameter type: String

Default value: ddl, error

Range: none | ddl | dml | insert | update | delete | truncate | select | error | create | drop | alter | grant | revoke | rollback | set | all | ( select | update | delete | insert )@groupname) ...

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Specifies auditing of different categories of SQL statements as well as statements related to specific SQL commands.

- To log errors, set the parameter value to error.
- To audit all DDL statements, such as CREATE TABLE and ALTER TABLE, set the parameter value to ddl.
- To audit specific types of DDL statements, the parameter values can include those specific SQL commands (create, drop, or alter). In addition, you can specify the object type following the command, such as create table, create view, and drop role.
- To audit all modification statements, such as INSERT, UPDATE, DELETE, or TRUNCATE, set edb_audit_statement to dml.
- To audit specific types of DML statements, the parameter values can include the specific SQL commands insert, update, delete, or truncate. Include parameter values select, grant, revoke, or rollback to audit statements regarding those SQL commands.
- To audit SET statements, include the parameter value SET.
- To audit every statement, set the value to all.
- To disable this feature, set the value to none.

The per-object level auditing audits the operations permitted by object privileges, such as SELECT, UPDATE, DELETE, and INSERT statements, including (@) and excluding (-) groups on a given table. To audit a specific type of object, specify the name of the object group to audit. The edb_audit_statement parameter can include the specific SQL commands (select, update, delete, or insert) associated with a group name with (@) include and (-) exclude symbol.

8.1.2.7.10   edb_audit_tag

Parameter type: String

Default value: none

Minimum scope of effect: Session

When value changes take effect: Immediate

Required authorization to activate: User

Specifies a string value to include in the audit log when the edb_audit parameter is set to csv or xnl.

8.1.2.7.11   edb_audit_destination

Parameter type: Enum

Default value: file

Range: (file | syslog)

Minimum scope of effect: Cluster

When value changes take effect: Reload
Required authorization to activate: EDB Postgres Advanced Server service account

Specifies whether to record the audit log information in the directory as given by the `edb_audit_directory` parameter or to the directory and file managed by the `syslog` process. Set to `file` to use the directory specified by `edb_audit_directory`.

Set to `syslog` to use the syslog process and its location as configured in the `/etc/syslog.conf` file. The `syslog` setting is valid only for EDB Postgres Advanced Server running on a Linux host. It isn't supported on Windows systems.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>In recent Linux versions, <code>syslog</code> was replaced with <code>rsyslog</code>, and the configuration file is in <code>/etc/rsyslog.conf</code>.</td>
</tr>
</tbody>
</table>

8.1.2.7.12  `edb_log_every_bulk_value`

For information on `edb_log_every_bulk_value`, see `edb_log_every_bulk_value`.

8.1.2.8  Client connection defaults/locale and formatting

These configuration parameters affect locale and formatting.

`icu_short_form`

Parameter type: String
Default value: `none`
Range: n/a
Minimum scope of effect: Database
When value changes take effect: n/a
Required authorization to activate: n/a

Contains the default ICU short-form name assigned to a database or to the EDB Postgres Advanced Server instance. See Unicode collation algorithm for general information about the ICU short form and the Unicode collation algorithm.

Set this configuration parameter either when:

- Using the `CREATE DATABASE` command with the `ICU_SHORT_FORM` parameter. In this case, set the specified short-form name. It appears in the `icu_short_form` configuration parameter when connected to this database.
- Creating an EDB Postgres Advanced Server instance with the `initdb` command used with the `--icu_short_form` option. In this case, set the specified short-form name. It appears in the `icu_short_form` configuration parameter when connected to a database in that EDB Postgres Advanced Server instance. The database doesn't override it with its own `ICU_SHORT_FORM` parameter and a different short form.

Once you set this parameter, you can't change it.

8.1.2.9  Client connection defaults/statement behavior

These configuration parameters affect statement behavior.

`default_heap_fillfactor`

Parameter type: Integer
Default value: 100
Range: 10 to 100
Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

Sets the fill factor for a table when the `FILLFACTOR` storage parameter is omitted from a `CREATE TABLE` command.

The fill factor for a table is a percentage from 10 to 100, where 100 is complete packing. When you specify a smaller fill factor, `INSERT` operations pack table pages only to the indicated percentage. The remaining space on each page is reserved for updating rows on that page. This approach gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page.

For a table whose entries are never updated, complete packing is the best choice. In heavily updated tables, use smaller fill factors.

ebd_data_redaction

Parameter type: Boolean

Default value: true

Range: \{true | false\}

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

Data redaction is the support of policies to limit the exposure of certain sensitive data to certain users by altering the displayed information.

When set to `TRUE`, the data redaction is applied to all users except for superusers and the table owner:

- Superusers and table owner bypass data redaction.
- All other users get the redaction policy applied and see the reformatted data.

When set to `FALSE`, the following occurs:

- Superusers and table owner still bypass data redaction.
- All other users get an error.

For information on data redaction, see EDB Postgres Advanced Server Security Features.

8.1.2.10 Client connection defaults/other defaults

These parameters set miscellaneous client connection defaults.

oracle_home

Parameter type: String

Default value: none

Range: n/a

Minimum scope of effect: Cluster

When value changes take effect: Restart

Required authorization to activate: EDB Postgres Advanced Server service account

Before creating a link to an Oracle server, you must direct EDB Postgres Advanced Server to the correct Oracle home directory. Set the `LD_LIBRARY_PATH` environment variable on Linux or `PATH` on Windows to the `lib` directory of the Oracle client installation directory.

Alternatively, you can set the value of the `oracle_home` configuration parameter in the `postgresql.conf` file. The value specified in the `oracle_home` configuration parameter overrides the
LD_LIBRARY_PATH environment variable in Linux and PATH environment variable in Windows.

Note

The oracle_home configuration parameter must provide the correct path to the Oracle client, that is, OCI library.

To set the oracle_home configuration parameter in the postgresql.conf file, add the following line:

```
oracle_home = '<lib_directory>'
```

<lib_directory> is the name of the oracle_home path to the Oracle client installation directory that contains libcintsh.so in Linux and oci.dll in Windows.

After setting the oracle_home configuration parameter, you must restart the server for the changes to take effect. Restart the server:

- On Linux, using the systemctl command or pg_ctl services
- On Windows, from the Windows Services console

odbc_lib_path

Parameter type: String
Default value: none
Range: n/a
Minimum scope of effect: Cluster
When value changes take effect: Restart
Required authorization to activate: EDB Postgres Advanced Server service account

If you're using an ODBC driver manager and if it's installed in a nonstandard location, specify the location by setting the odbc_lib_path configuration parameter in the postgresql.conf file:

```
odbc_lib_path = 'complete_path_to_libodbc.so'
```

The configuration file must include the complete pathname to the driver manager shared library, which is typically libodbc.so.

8.1.2.11 Compatibility options

These configuration parameters control various database compatibility features.

edb_redwood_date

Parameter type: Boolean
Default value: false
Range: {true | false}
Minimum scope of effect: Per session
When value changes take effect: Immediate
Required authorization to activate: Session user

Translates DATE to TIMESTAMP when DATE appears as the data type of a column in the commands and the table definition is stored in the database. A time component is stored in the column along with the date.

If edb_redwood_date is set to FALSE, the column's data type in a CREATE TABLE or ALTER TABLE command remains as a native PostgreSQL DATE data type and is stored as such in the database. The PostgreSQL DATE data type stores only the date without a time component in the column.
Regardless of the setting of `edb_redwood_date`, when `DATE` appears as a data type in any other context, it's always internally translated to a `TIMESTAMP`. It can thus handle a time component if present. Examples of these contexts include:

- The data type of a variable in an SPL declaration section
- The data type of a formal parameter in an SPL procedure or SPL function
- The return type of an SPL function

**edb_redwood_greatest_least**

Parameter type: Boolean

Default value: `true`

Range: `{true | false}`

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

The `GREATEST` function returns the parameter with the greatest value from its list of parameters. The `LEAST` function returns the parameter with the least value from its list of parameters.

When `edb_redwood_greatest_least` is set to `TRUE`, the `GREATEST` and `LEAST` functions return null when at least one of the parameters is null.

```
SET edb_redwood_greatest_least TO on;
SELECT GREATEST(1, 2, NULL, 3);
```

```
greatest
----------
3
(1 row)
```

When `edb_redwood_greatest_least` is set to `FALSE`, null parameters are ignored except when all parameters are null. In that case, the functions return null.

```
SET edb_redwood_greatest_least TO off;
SELECT GREATEST(1, 2, NULL, 3);
```

```
greatest
----------
3
(1 row)
```

```
SELECT GREATEST(NULL, NULL, NULL);
```

```
greatest
----------
(1 row)
```

**edb_redwood_raw_names**

Parameter type: Boolean

Default value: `false`

Range: `{true | false}`

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user
When `edb_redwood_raw_names` is set to `FALSE`, database object names such as table names, column names, trigger names, program names, and user names appear in uppercase letters when viewed from Redwood catalogs (that is, system catalogs prefixed by `ALL_`, `DBA_`, or `USER_`). In addition, quotation marks enclose names that were created with enclosing quotation marks.

When `edb_redwood_raw_names` is set to `TRUE`, the database object names are displayed as they're stored in the PostgreSQL system catalogs when viewed from the Redwood catalogs. Names created without quotation marks around them appear in lower case as expected in PostgreSQL. Names created enclosed by quotation marks appear as they were created but without the quotation marks.

For example, the following user name is created, and then a session is started with that user:

```
CREATE USER reduser IDENTIFIED BY password;
```

Password for user reduser:

You are now connected to database "edb" as user "reduser".

When connected to the database as `reduser`, the following tables are created:

```
CREATE TABLE all_lower (col INTEGER);
CREATE TABLE all_upper (col INTEGER);
CREATE TABLE "Mixed_Case" ("Col" INTEGER);
```

When viewed from the Redwood catalog `USER_TABLES`, with `edb_redwood_raw_names` set to the default value `FALSE`, the names appear in upper case. The exception is the `Mixed_Case` name, which appears as created and also enclosed by quotation marks.

<table>
<thead>
<tr>
<th>schema_name</th>
<th>table_name</th>
<th>tablespace_name</th>
<th>status</th>
<th>temporary</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDUSER</td>
<td>ALL_LOWER</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
<tr>
<td>REDUSER</td>
<td>ALL_UPPER</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
<tr>
<td>REDUSER</td>
<td>&quot;Mixed_Case&quot;</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
</tbody>
</table>

(3 rows)

When viewed with `edb_redwood_raw_names` set to `TRUE`, the names appear in lower case except for the `Mixed_Case` name, which appears as created but without quotation marks.

```
edb=> SET edb_redwood_raw_names TO true;
set
```

```
edb=> SELECT * FROM USER_TABLES;

```

<table>
<thead>
<tr>
<th>schema_name</th>
<th>table_name</th>
<th>tablespace_name</th>
<th>status</th>
<th>temporary</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduser</td>
<td>all_lower</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
<tr>
<td>reduser</td>
<td>all_upper</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
<tr>
<td>reduser</td>
<td>Mixed_Case</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
</tbody>
</table>

(3 rows)

These names now match the case when viewed from the PostgreSQL `pg_tables` catalog:

```
edb=> SELECT schemaname, tablename, tableowner FROM pg_tables
WHERE tableowner = 'reduser';
```

<table>
<thead>
<tr>
<th>schemaname</th>
<th>tablename</th>
<th>tableowner</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduser</td>
<td>all_lower</td>
<td>reduser</td>
</tr>
<tr>
<td>reduser</td>
<td>all_upper</td>
<td>reduser</td>
</tr>
<tr>
<td>reduser</td>
<td>Mixed_Case</td>
<td>reduser</td>
</tr>
</tbody>
</table>

(3 rows)

`edb_redwood_strings`

**Parameter type:** Boolean

**Default value:** `false`

**Range:** `{true | false}`

**Minimum scope of effect:** Per session

**When value changes take effect:** Immediate
If the `edb_redwood_strings` parameter is set to `TRUE`, when a string is concatenated with a null variable or null column, the result is the original string. If `edb_redwood_strings` is set to `FALSE`, the native PostgreSQL behavior is maintained, which is the concatenation of a string with a null variable or null column that gives a null result.

This example shows the difference. The sample application contains a table of employees. This table has a column named `comm` that's null for most employees. The following query is run with `edb_redwood_string` set to `FALSE`. Concatenating a null column with non-empty strings produces a final result of null, so only employees that have a commission appear in the query result. The output line for all other employees is null.

```
SET edb_redwood_strings TO off;
SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' || TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;
```

```
EMPLOYEE COMPENSATION
---------------------
ALLEN  1,600.00     300.00
WARD   1,250.00     500.00
MARTIN 1,250.00   1,400.00

TURNER 1,500.00     .00
(14 rows)
```

The following is the same query executed when `edb_redwood_strings` is set to `TRUE`. Here, the value of a null column is treated as an empty string. Concatenating an empty string with a non-empty string produces the non-empty string.

```
SET edb_redwood_strings TO on;
SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' || TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;
```

```
EMPLOYEE COMPENSATION
---------------------
SMITH  800.00
ALLEN  1,600.00 300.00
WARD  1,250.00 500.00
JONES 2,975.00
MARTIN 1,250.00 1,400.00
BLAKE 2,850.00
CLARK 2,450.00
SCOTT 3,800.00
KING  5,000.00
TURNER 1,500.00 .00
ADAMS 1,100.00
JAMES 950.00
FORD 3,000.00
MILLER 1,300.00
(14 rows)
```

The `edb_redwood_strings` parameter is a Boolean type.

- **Default value:** `false`
- **Range:** `{true | false}`
- **Minimum scope of effect:** Per session
- **When value changes take effect:** Immediate
The term statement-level transaction isolation describes the behavior in which a runtime error occurs in a SQL command, and all the updates on the database caused by that single command are rolled back. For example, if a single UPDATE command successfully updates five rows, but an attempt to update a sixth row results in an exception, the updates to all six rows made by this UPDATE command are rolled back. The effects of prior SQL commands that haven't yet been committed or rolled back are pending until a COMMIT or ROLLBACK command is executed.

In EDB Postgres Advanced Server, if an exception occurs while executing a SQL command, all the updates on the database since the start of the transaction are rolled back. In addition, the transaction is left in an aborted state, and either a COMMIT or ROLLBACK command must be issued before another transaction can start.

If \texttt{edb_stmt_level_tx} is set to \texttt{TRUE}, then an exception doesn't roll back prior uncommitted database updates. If \texttt{edb_stmt_level_tx} is set to \texttt{FALSE}, then an exception rolls back uncommitted database updates.

\begin{verbatim}
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO off;

INSERT INTO emp (empno,ename,deptno) VALUES (9801, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9802, 'JONES', 80);
ERROR: insert or update on table "emp" violates foreign key constraint "emp_ref_dept_fk"
DETAIL: Key (deptno)=8 is not present in table "dept"

COMMIT;
SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno
-------+-------+--------
(0 rows)
\end{verbatim}

In this example, with \texttt{edb_stmt_level_tx} set to \texttt{TRUE}, the first INSERT command wasn't rolled back after the error on the second INSERT command. At this point, the first INSERT command can either be committed or rolled back.

\begin{verbatim}
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

INSERT INTO emp (empno,ename,deptno) VALUES (9801, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9802, 'JONES', 80);
ERROR: insert or update on table "emp" violates foreign key constraint "emp_ref_dept_fk"
DETAIL: Key (deptno)=8 is not present in table "dept"

SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno
-------+-------+--------
9801 | JONES | 40
(1 row)
\end{verbatim}

If a ROLLBACK command is issued instead of the COMMIT command, the insert of employee number 9801 is rolled back as well.

\texttt{db_dialect}

Parameter type: \texttt{Enum}

Default value: \texttt{postgres}

Range: \texttt{(postgres | redwood)}
Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

In addition to the native PostgreSQL system catalog `pg_catalog`, EDB Postgres Advanced Server contains an extended catalog view. This is the `sys` catalog for the expanded catalog view. The `db_dialect` parameter controls the order in which these catalogs are searched for name resolution.

When set to `postgres`, the namespace precedence is `pg_catalog` and then `sys`, giving the PostgreSQL catalog the highest precedence. When set to `redwood`, the namespace precedence is `sys` and then `pg_catalog`, giving the expanded catalog views the highest precedence.

default_with_rowids

Parameter type: Boolean

Default value: false

Range: {true | false}

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

When set to `on`, `CREATE TABLE` includes a `ROWID` column in newly created tables, which you can then reference in SQL commands. In earlier versions of EDB Postgres Advanced Server, `ROWIDs` were mapped to `OIDs`. With EDB Postgres Advanced Server version 12 and later, the `ROWID` is an autoincrementing value based on a sequence that starts with 1. It's assigned to each row of a table created with the `ROWIDs` option. By default, a unique index is created on a `ROWID` column.

The `ALTER` and `DROP` operations are restricted on a `ROWID` column.

To restore a database with `ROWIDs` with EDB Postgres Advanced Server 11 or an earlier version, you must perform the following:

1. Take a dump of the tables by specifying the `--convert-oids-to-rowids` option.
2. Drop the tables, and then perform the upgrade.
3. After the upgrade is successful, restore the dump into a new cluster that contains the dumped tables into a target database.

optimizer_mode

Parameter type: Enum

Default value: choose

Range: {choose | ALL_ROWS | FIRST_ROWS | FIRST_ROWS_10 | FIRST_ROWS_100 | FIRST_ROWS_1000}

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

Sets the default optimization mode for analyzing optimizer hints.

The following table shows the possible values.

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL_ROWS</td>
<td>Optimizes for retrieval of all rows of the result set.</td>
</tr>
<tr>
<td>CHOOSE</td>
<td>Does no default optimization based on assumed number of rows to retrieve from the result set. This is the default.</td>
</tr>
<tr>
<td>FIRST_ROWS</td>
<td>Optimizes for retrieval of only the first row of the result set.</td>
</tr>
<tr>
<td>FIRST_ROWS_10</td>
<td>Optimizes for retrieval of the first 10 rows of the results set.</td>
</tr>
<tr>
<td>FIRST_ROWS_100</td>
<td>Optimizes for retrieval of the first 100 rows of the result set.</td>
</tr>
</tbody>
</table>
### 8.1.2.12 Customized options

#### custom_variable_classes

The `custom_variable_classes` parameter was deprecated in EDB Postgres Advanced Server 9.2. Parameters that previously depended on this parameter no longer require its support. In previous releases of EDB Postgres Advanced Server, `custom_variable_classes` was required by parameters not normally known to be added by add-on modules, such as procedural languages.

#### dbms_alert.max_alerts

**Parameter type:** Integer  
**Default value:** 100  
**Range:** 0 to 500  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Restart  
**Required authorization to activate:** EDB Postgres Advanced Server service account  

Specifies the maximum number of concurrent alerts allowed on a system using the `DBMS_ALERTS` package.

#### dbms_pipe.total_message_buffer

**Parameter type:** Integer  
**Default value:** 30 Kb  
**Range:** 30 Kb to 256 Kb  
**Minimum scope of effect:** Cluster  
**When value changes take effect:** Restart  
**Required authorization to activate:** EDB Postgres Advanced Server service account  

Specifies the total size of the buffer used for the `DBMS_PIPE` package.

#### index_advisor.enabled

**Parameter type:** Boolean  
**Default value:** true  
**Range:** {true | false}  
**Minimum scope of effect:** Per session  
**When value changes take effect:** Immediate  
**Required authorization to activate:** Session user
Temporarily suspends Index Advisor in an EDB-PSQL or PSQL session. To use this configuration parameter, the Index Advisor plugin `index_advisor` must be loaded in the EDB-PSQL or PSQL session.

You can load the Index Advisor plugin as follows:

```
$ psql -d edb -U enterprisedb
Password for user enterprisedb:
psql (14.0.0)
Type "help" for help.
edb=# LOAD 'index_advisor';
LOAD
```

Use the `SET` command to change the parameter setting to control whether Index Advisor generates an alternative query plan:

```
edb=# SET index_advisor.enabled TO off;
SET
edb=# EXPLAIN SELECT * FROM t WHERE a < 10000;

QUERY PLAN
-------------------------------------------------------
Seq Scan on t (cost=0.00..1693.00 rows=9864 width=8)
Filter: (a < 10000)
(2 rows)
```

```
edb=# SET index_advisor.enabled TO on;
SET
edb=# EXPLAIN SELECT * FROM t WHERE a < 10000;

QUERY PLAN
-------------------------------------------------------
Seq Scan on t (cost=0.00..1693.00 rows=9864 width=8)
Filter: (a < 10000)
Result (cost=0.00..327.88 rows=9864 width=8)
One-Time Filter: '==[ HYPOTHETICAL PLAN ]==':text
  -> Index Scan using "hypothetical-index":1" on t (cost=0.00..327.88 rows=9864 width=8)
Index Cond: (a < 10000)
(6 rows)
```

db_sql_protect.enabled

Parameter type: Boolean

Default value: `false`

Range: `{true | false}`

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Controls whether SQL/Protect is actively monitoring protected roles by analyzing SQL statements issued by those roles and reacting according to the setting of `edb_sql_protect.level`. When you're ready to begin monitoring with SQL/Protect, set this parameter to `on`.

db_sql_protect.level

Parameter type: Enum

Default value: `passive`

Range: `{learn | passive | active}`
Sets the action taken by SQL/Protect when a SQL statement is issued by a protected role. You can set this parameter to one of the following values to use learn mode, passive mode, or active mode:

- **learn**: Tracks the activities of protected roles and records the relations used by the roles. Use this value when initially configuring SQL/Protect so the expected behaviors of the protected applications are learned.
- **passive**: Issues warnings if protected roles are breaking the defined rules but doesn’t stop any SQL statements from executing. This is the next step after SQL/Protect learns the expected behavior of the protected roles. This essentially behaves in intrusion detection mode. You can run it in production if you monitor it.
- **active**: Stops all invalid statements for a protected role. This behavior acts as a SQL firewall that prevents dangerous queries from running. This is particularly effective against early penetration testing when the attacker is trying to determine the vulnerability point and the type of database behind the application. Not only does SQL/Protect close those vulnerability points, it tracks the blocked queries. This behavior can alert administrators before the attacker finds an alternative method of penetrating the system.

If you’re using SQL/Protect for the first time, set `edb_sql_protect.level` to `learn`.

### `edb_sql_protect.max_protected_relations`

**Parameter type:** Integer

**Default value:** 1024

**Range:** 1 to 2147483647

**Minimum scope of effect:** Cluster

**When value changes take effect:** Restart

**Required authorization to activate:** EDB Postgres Advanced Server service account

Sets the maximum number of relations that can be protected per role. The total number of protected relations for the server is the number of protected relations times the number of protected roles. Every protected relation consumes space in shared memory. The space for the maximum possible protected relations is reserved during database server startup.

If the server is started when `edb_sql_protect.max_protected_relations` is set to a value outside of the valid range (for example, a value of 2,147,483,648), then a warning message is logged in the database server log file:

```
2014-07-18 16:04:12 EDT WARNING: invalid value for parameter "edb_sql_protect.max_protected_relations": "2147483648"
```

The database server starts successfully but with `edb_sql_protect.max_protected_relations` set to the default value of 1024.

Although the upper range for the parameter is listed as the maximum value for an integer data type, the practical setting depends on how much shared memory is available and the parameter value used during database server startup.

As long as the space required can be reserved in shared memory, the value is acceptable. If the value is such that the space in shared memory can’t be reserved, the database server startup fails with an error message like the following:

```
```

In this case, reduce the parameter value until you can start the database server successfully.

### `edb_sql_protect.max_protected_roles`

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Parameter type: Integer

Default value: 64

Range: 1 to 2147483647

Minimum scope of effect: Cluster

When value changes take effect: Restart

Required authorization to activate: EDB Postgres Advanced Server service account

Sets the maximum number of roles that can be protected.

Every protected role consumes space in shared memory. The server reserves space for the number of protected roles times the number of protected relations (`edb_sql_protect.max_protected_relations`). The space for the maximum possible protected roles is reserved during database server startup.

If the database server is started when `edb_sql_protect.max_protected_roles` is set to a value outside of the valid range (for example, a value of 2,147,483,648), then a warning message is logged in the database server log file:

```
2014-07-18 16:04:12 EDT WARNING: invalid value for parameter
"edb_sql_protect.max_protected_roles": "2147483648"
```

The database server starts successfully but with `edb_sql_protect.max_protected_roles` set to the default value of 64.

Although the upper range for the parameter is listed as the maximum value for an integer data type, the practical setting depends on how much shared memory is available and the parameter value used during database server startup.

As long as the space required can be reserved in shared memory, the value is acceptable. If the value is such that the space in shared memory can't be reserved, the database server startup fails with an error message such as the following:

```
2014-07-18 15:22:17 EDT FATAL: could not map anonymous shared
memory: Cannot allocate memory
2014-07-18 15:22:17 EDT HINT: This error usually means that
PostgreSQL's request for a shared memory segment exceeded available memory,
swap space or huge pages. To reduce the request size (currently
2070118400 bytes), reduce PostgreSQL's shared memory usage, perhaps by
reducing shared_buffers or max_connections.
```

In this cases, reduce the parameter value until you can start the database server successfully.

`edb_sql_protect.max_queries_to_save`

Parameter type: Integer

Default value: 5000

Range: 100 to 2147483647

Minimum scope of effect: Cluster

When value changes take effect: Restart

Required authorization to activate: EDB Postgres Advanced Server service account

Sets the maximum number of offending queries to save in the view `edb_sql_protect_queries`.

Every saved query consumes space in shared memory. The space for the maximum possible queries that can be saved is reserved during database server startup.

If the database server is started when `edb_sql_protect.max_queries_to_save` is set to a value outside of the valid range (for example, a value of 10), then a warning message is logged in the database server log file:

```
2014-07-18 13:05:31 EDT WARNING: 10 is outside the valid range
```
The database server starts successfully but with `edb_sql_protect.max_queries_to_save` set to the default value of 5000.

Although the upper range for the parameter is listed as the maximum value for an integer data type, the practical setting depends on how much shared memory is available and the parameter value used during database server startup.

As long as the space required can be reserved in shared memory, the value is acceptable. If the value is such that the space in shared memory can't be reserved, the database server startup fails with an error message like the following:

```
Cannot allocate memory
```

In this case, reduce the parameter value until you can start the database server successfully.

---

**edb_wait_states.directory**

Parameter type: String

Default value: `edb_wait_states`

Range: n/a

Minimum scope of effect: Cluster

When value changes take effect: Restart

Required authorization to activate: EDB Postgres Advanced Server service account

Sets the directory path where the EDB wait states log files are stored. Use a full, absolute path, not a relative path. However, the default setting is `edb_wait_states`, which makes `$PGDATA/edb_wait_states` the default directory location. See **EDB Wait States** for more information.

---

**edb_wait_states.retention_period**

Parameter type: Integer

Default value: 604800s

Range: 86400s to 2147483647s

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Sets the time to wait before deleting the log files for EDB wait states. The default is 604,800 seconds, which is 7 days. See **EDB Wait States** for more information.

---

**edb_wait_states.sampling_interval**

Parameter type: Integer

Default value: 1s

Range: 1s to 2147483647s

---
Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Sets the timing interval between two sampling cycles for EDB wait states. The default setting is 1 second. See EDB Wait States for more information.

edbldr.empty_csv_field

Parameter type: Enum

Default value: NULL

Range: {NULL | empty_string | pgsql}

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

Use the edbldr.empty_csv_field parameter to specify how EDB*Loader treats an empty string. The table shows the valid values for the edbldr.empty_csv_field parameter.

<table>
<thead>
<tr>
<th>Parameter setting</th>
<th>EDB*Loader behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>An empty field is treated as NULL.</td>
</tr>
<tr>
<td>empty_string</td>
<td>An empty field is treated as a string of length zero.</td>
</tr>
<tr>
<td>pgsql</td>
<td>An empty field is treated as a NULL if it doesn’t contain quotes and as an empty string if it contains quotes.</td>
</tr>
</tbody>
</table>

For more information about the edbldr.empty_csv_field parameter in EDB*Loader, see Tools, utilities, and components.

utl_encode.uudecode_redwood

Parameter type: Boolean

Default value: false

Range: {true | false}

Minimum scope of effect: Per session

When value changes take effect: Immediate

Required authorization to activate: Session user

When set to TRUE, EDB Postgres Advanced Server’s UTL_ENCODE.UUENCODE function can decode uuencoded data that was created by the Oracle implementation of the UTL_ENCODE.UUENCODE function.

When set to FALSE, EDB Postgres Advanced Server’s UTL_ENCODE.UUENCODE function can decode uuencoded data created by EDB Postgres Advanced Server’s UTL_ENCODE.UUENCODE function.

utl_file.umask

Parameter type: String

Default value: 0077

Range: Octal digits for umask settings

Minimum scope of effect: Per session
The `utl_file.umask` parameter sets the file mode creation mask in a manner similar to the Linux `umask` command. This is for use only within the EDB Postgres Advanced Server `UTL_FILE` package.

**Note**

The `utl_file.umask` parameter isn't supported on Windows systems.

The value specified for `utl_file.umask` is a three- or four-character octal string that's valid for the Linux `umask` command. The setting determines the permissions on files created by the `UTL_FILE` functions and procedures.

The following shows the results of the default `utl_file.umask` setting of 0077. All permissions are denied on users belonging to the `enterprisedb` group as well as all other users. Only the user `enterprisedb` has read and write permissions on the file.

```
-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile
```

### 8.1.2.13 Ungrouped configuration parameters

These configuration parameters apply only to EDB Postgres Advanced Server and are for a specific, limited purpose.

**nls_length_semantics**

**Parameter type:** Enum

**Default value:** `byte`

**Range:** `{byte | char}`

**Minimum scope of effect:** Per session

**When value changes take effect:** Immediate

**Required authorization to activate:** Superuser

This parameter has no effect in EDB Postgres Advanced Server. For example, this form of the `ALTER SESSION` command is accepted in EDB Postgres Advanced Server without throwing a syntax error. However, it doesn't alter the session environment.

```
ALTER SESSION SET nls_length_semantics = char;
```

**Note**

Since setting this parameter has no effect on the server environment, it doesn't appear in the system view `pg_settings`.

**query_rewrite_enabled**

**Parameter type:** Enum

**Default value:** `false`

**Range:** `{true | false | force}`

**Minimum scope of effect:** Per session

**When value changes take effect:** Immediate

**Required authorization to activate:** Session user

This parameter has no effect in EDB Postgres Advanced Server. For example, this form of the `ALTER SESSION` command is accepted in EDB Postgres Advanced Server without throwing a syntax error. However, it doesn't alter the session environment.
query_rewrite_integrity

Parameter type: Enum
Default value: enforced
Range: {enforced | trusted | stale_tolerated}
Minimum scope of effect: Per session
When value changes take effect: Immediate
Required authorization to activate: Superuser

This parameter has no effect in EDB Postgres Advanced Server. For example, this form of the ALTER SESSION command is accepted in EDB Postgres Advanced Server without throwing a syntax error. However, it doesn’t alter the session environment.

Note
Since setting this parameter has no effect on the server environment, it doesn’t appear in the system view pg_settings.

timed_statistics

Parameter type: Boolean
Default value: true
Range: {true | false}
Minimum scope of effect: Per session
When value changes take effect: Immediate
Required authorization to activate: Session user

Controls collecting timing data for the Dynamic Runtime Instrumentation Tools Architecture (DRITA) feature. When set to on, timing data is collected.

Note
When EDB Postgres Advanced Server is installed, the postgresql.conf file contains an explicit entry that sets timed_statistics to off. If this entry is commented out and the configuration file is reloaded, timed statistics collection uses the default value, which is on.

8.1.2.14 Audit archiving parameters

These configuration parameters are used by the EDB Postgres Advanced Server database audit archiving feature.

edb_audit_archiver

Parameter type: Enum
Default value: false
**edb_audit_archiver_timeout**

Parameter type: Integer  
Default value: 300s  
Range: 30s to 1d  
Minimum scope of effect: Cluster  
When value changes take effect: Reload  
Required authorization to activate: EDB Postgres Advanced Server service account  

Enforces a timeout in seconds when a database attempts to archive a log file.

**edb_audit_archiver_filename_prefix**

Parameter type: String  
Default value: audit-  
Range: n/a  
Minimum scope of effect: Cluster  
When value changes take effect: Reload  
Required authorization to activate: EDB Postgres Advanced Server service account  

Specifies the file name of an audit log file that needs to be archived. The file name must align with the `edb_audit_filename` parameter. The audit files with `edb_audit_archiver_filename_prefix` in the `edb_audit_directory` are eligible for compression or expiration.

**edb_audit_archiver_compress_time_limit**

Parameter type: Integer  
Default value: -1  
Allowed value: 0, -1, or any positive number value in seconds  
Minimum scope of effect: Cluster  
When value changes take effect: Reload  
Required authorization to activate: EDB Postgres Advanced Server service account  

Specifies the time in seconds after which audit logs are eligible for compression. The possible values to set this parameter are:

- **0**: Compression starts as soon as the log file isn’t a current file.
- **-1**: Compression of the log file on a timely basis doesn’t occur.
**edb_audit_archiver_compress_size_limit**

Parameter type: Integer

Default value: -1

Allowed value: 0, -1, or any positive number value in megabytes

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Specifies a file size threshold in megabytes, after which audit logs are eligible for compression. If the parameter is set to -1, no compression of the log file occurs based on size.

**edb_audit_archiver_compress_command**

Parameter type: String

Default value: gzip %p

Range: n/a

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Specifies the command to execute compressing of the audit log files. The default value for edb_audit_archiver_compress_command is gzip %p. The gzip provides a standard method of compressing files. The %p in the string is replaced by the path name of the file to archive.

**edb_audit_archiver_compress_suffix**

Parameter type: String

Default value: .gz

Range: n/a

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

Specifies the file name of an already compressed log file. The file name must align with edb_audit_archiver_compress_command. The default file name is .gz.

**edb_audit_archiver_expire_time_limit**

Parameter type: Integer

Default value: -1

Allowed value: 0, -1, or any positive number value in seconds

Minimum scope of effect: Cluster

When value changes take effect: Reload

Required authorization to activate: EDB Postgres Advanced Server service account

EDB Postgres Advanced Server (EPAS)

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Specifies the time in seconds after which audit logs are eligible to expire. The possible values to set this parameter are:

- 0: Expiration starts as soon as the log file isn't a current file.
- -1: Expiration of the log file on a timely basis doesn't occur.

**edb_audit_archiver_expire_size_limit**

**Parameter type:** Integer

**Default value:** -1

**Allowed values:** 0, -1, or any positive number value in megabytes

**Minimum scope of effect:** Cluster

**When value changes take effect:** Reload

**Required authorization to activate:** EDB Postgres Advanced Server service account

Specifies a file size threshold in megabytes, after which audit logs are eligible to expire. If the parameter is set to -1, no expiration of a log file based on size occurs.

**edb_audit_archiver_expire_command**

**Parameter type:** String

**Default value:** "

**Range:** n/a

**Minimum scope of effect:** Cluster

**When value changes take effect:** Reload

**Required authorization to activate:** EDB Postgres Advanced Server service account

Specifies the command to execute on an expired audit log file before removal.

**edb_audit_archiver_sort_file**

**Parameter type:** String

**Default value:** mtime

**Range:** n/a

**Minimum scope of effect:** Cluster

**When value changes take effect:** Reload

**Required authorization to activate:** EDB Postgres Advanced Server service account

Identifies the oldest log file to sort alphabetically or based on mtime.

- mtime sorts files based on file modification time.
- alphabetic sorts files alphabetically based on the file name.

**8.1.2.15 Description of parameter attributes**

The description of each group of parameters includes this list of attributes:

- **Parameter type:** Type of values the parameter can accept. See Setting configuration parameters for a discussion of parameter type values.
EDB Resource Manager is an EDB Postgres Advanced Server feature that lets you control the use of operating system resources used by EDB Postgres Advanced Server processes. This capability allows you to protect the system from processes that might uncontrollably overuse and monopolize certain system resources.

### 8.2.1 EDB Resource Manager key concepts

You use EDB Resource Manager to control the use of operating system resources used by EDB Postgres Advanced Server processes.

Some key points about using EDB Resource Manager are:

- The basic component of EDB Resource Manager is a resource group. A resource group is a named, global group. It's available to all databases in an EDB Postgres Advanced Server instance, and you can define various resource usage limits on it. EDB Postgres Advanced Server processes that are assigned as members of a given resource group are then controlled by EDB Resource Manager. This configuration keeps the aggregate resource use of all processes in the group near the limits defined on the group.
- Data definition language commands are used to create, alter, and drop resource groups. Only a database user with superuser privileges can use these commands.
- Resource type parameters define the desired aggregate consumption level of all processes belonging to a resource group. You use different resource type parameters for the different types of system resources currently supported by EDB Resource Manager.
- You can create multiple resource groups, each with different settings for its resource type parameters, which defines different consumption levels for each resource group.
- EDB Resource Manager throttles processes in a resource group to keep resource consumption near the limits defined by the resource type parameters. If multiple resource type parameters have defined settings in a resource group, the actual resource consumption might be significantly lower for certain resource types than their defined resource type parameter settings. This lower consumption happens because EDB Resource Manager throttles processes, attempting to keep all resources with defined resource type settings within their defined limits.
- The definitions of available resource groups and their resource type settings are stored in a shared global system catalog. Thus, all databases in a given EDB Postgres Advanced Server Instance can use resource groups.
- The `edb_max_resource_groups` configuration parameter sets the maximum number of resource groups that can be active at the same time as running processes. The default setting is 16 resource groups. Changes to this parameter take effect when you restart the database server.
- Use the `SET edb_resource_group TO group_name` command to assign the current process to a specified resource group. Use the `RESET edb_resource_group` command or `SET edb_resource_group to DEFAULT` to remove the current process from a resource group.
- You can assign a default resource group to a role using the `ALTER ROLE ... SET edb_resource_group` command to a database using the `ALTER DATABASE ... SET edb_resource_group`. You can assign the entire database server instance a default resource group by setting the parameter in the `postgresql.conf` file.
- To include resource groups in a backup file of the database server instance, use the `pg_dumpall` backup utility with default settings. That is, don't specify any of the `--globals-only, --roles-only`, or `--tablespaces-only` options.

### 8.2.2 Working with resource groups

Use these data definition language commands to create and manage resource groups.

**Creating a resource group**

Use the `CREATE RESOURCE GROUP` command to create a new resource group.

```sql
CREATE RESOURCE GROUP <group_name>;
```

**Description**

The `CREATE RESOURCE GROUP` command creates a resource group with the specified name. You can then define resource limits on the group with the `ALTER RESOURCE GROUP` command. The resource group is accessible from all databases in the EDB Postgres Advanced Server Instance.
To use the `CREATE RESOURCE GROUP` command, you must have superuser privileges.

**Parameters**

`group_name`  
The name of the resource group.

**Example**

This example creates three resource groups named `resgrp_a`, `resgrp_b`, and `resgrp_c`:

```sql
   edb=# CREATE RESOURCE GROUP resgrp_a;
   CREATE RESOURCE GROUP
   edb=# CREATE RESOURCE GROUP resgrp_b;
   CREATE RESOURCE GROUP
   edb=# CREATE RESOURCE GROUP resgrp_c;
   CREATE RESOURCE GROUP
```

This query shows the entries for the resource groups in the `edb_resource_group` catalog:

```sql
   edb=# SELECT * FROM edb_resource_group;
   rgrpname | rgrpcpuratelimit | rgrpdirtyratelimit
   ----------+------------------+-------------------
   resgrp_a  | 0 | 0
   resgrp_b  | 0 | 0
   resgrp_c  | 0 | 0
   (3 rows)
```

**Modifying a resource group**

Use the `ALTER RESOURCE GROUP` command to change the attributes of an existing resource group. The command syntax comes in three forms.

This form renames the resource group:

```sql
   ALTER RESOURCE GROUP <group_name> RENAME TO <new_name>;
```

This form assigns a resource type to the resource group:

```sql
   ALTER RESOURCE GROUP <group_name> SET <resource_type> { TO | = } { <value> | DEFAULT }
```

This form resets the assignment of a resource type to its default in the group:

```sql
   ALTER RESOURCE GROUP <group_name> RESET <resource_type>;
```

**Description**

The `ALTER RESOURCE GROUP` command changes certain attributes of an existing resource group.

The form with the `RENAME TO` clause assigns a new name to an existing resource group.

The form with the `SET resource_type TO` clause assigns the specified literal value to a resource type. Or, when you specify `DEFAULT`, it resets the resource type. Resetting a resource type means that the resource group has no defined limit on that resource type.

The form with the `RESET resource_type` clause resets the resource type for the group.

To use the `ALTER RESOURCE GROUP` command, you must have superuser privileges.

**Parameters**
**group_name**

The name of the resource group to alter.

**new_name**

The new name to assign to the resource group.

**resource_type**

Specifies the type of resource to which to set a usage value.

**value | DEFAULT**

When value is specified, the literal value to assign to resource_type. Specify DEFAULT to reset the assignment of resource_type for the resource group.

**Example**

These examples show the use of the `ALTER RESOURCE GROUP` command:

```sql
edb# ALTER RESOURCE GROUP resgrp_a RENAME TO newgrp;
ALTER RESOURCE GROUP
edb# ALTER RESOURCE GROUP resgrp_b SET cpu_rate_limit = .5;
ALTER RESOURCE GROUP
edb# ALTER RESOURCE GROUP resgrp_b SET dirty_rate_limit = 6144;
ALTER RESOURCE GROUP
edb# ALTER RESOURCE GROUP resgrp_c RESET cpu_rate_limit;
ALTER RESOURCE GROUP
```

This query shows the results of the `ALTER RESOURCE GROUP` commands to the entries in the `edb_resource_group` catalog:

```
edb# SELECT * FROM edb_resource_group;

  rgrpname | rgrpcpuratelimit | rgrpdirtyratelimit
----------+-----------------+-------------------
   newgrp  |      0          |        0          
 resgrp_b |    0.5          |      6144         
 resgrp_c |      0          |        0          
(3 rows)
```

**Removing a resource group**

Use the `DROP RESOURCE GROUP` command to remove a resource group.

```
DROP RESOURCE GROUP [ IF EXISTS ]
<group_name>;
```

**Description**

The `DROP RESOURCE GROUP` command removes a resource group with the specified name.

To use the `DROP RESOURCE GROUP` command, you must have superuser privileges.

**Parameters**

**group_name**

The name of the resource group to remove.

**IF EXISTS**
Don't throw an error if the resource group doesn't exist. Instead, issue a notice.

Example

This example removes the resource group newgrp:

```
edb=# DROP RESOURCE GROUP newgrp;
DROP RESOURCE GROUP
```

Assigning a process to a resource group

Use the `SET edb_resource_group TO group_name` command to assign the current process to a specified resource group:

```
edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;

edb_resource_group
--------------------
resgrp_b
(1 row)
```

The resource type settings of the group take effect on the current process immediately. If you use the command to change the resource group assigned to the current process, the resource type settings of the newly assigned group take effect immediately.

You can include processes in a resource group by default by assigning a default resource group to roles, databases, or an entire database server instance.

You can assign a default resource group to a role using the `ALTER ROLE ... SET` command. For more information about the `ALTER ROLE` command, see the PostgreSQL core documentation.

You can assign a default resource group to a database by using the `ALTER DATABASE ... SET` command. For more information about the `ALTER DATABASE` command, see the PostgreSQL core documentation.

You can assign the entire database server instance a default resource group by setting the `edb_resource_group` configuration parameter in the `postgresql.conf` file:

```
# - EDB Resource Manager
#
edb_max_resource_groups = 16           # 0-65536 (change requires
restart)
edb_resource_group = 'resgrp_b'
```

If you change `edb_resource_group` in the `postgresql.conf` file, reload the configuration file to make it take effect on the database server instance.

Removing a process from a resource group

Set `edb_resource_group` to `DEFAULT` or use `RESET edb_resource_group` to remove the current process from a resource group:

```
edb=# SET edb_resource_group TO DEFAULT;
SET
edb=# SHOW edb_resource_group;

edb_resource_group
--------------------
(1 row)
```

To remove a default resource group from a role, use the `ALTER ROLE ... RESET` form of the `ALTER ROLE` command.

To remove a default resource group from a database, use the `ALTER DATABASE ... RESET` form of the `ALTER DATABASE` command.

To remove a default resource group from the database server instance, set the `edb_resource_group` configuration parameter to an empty string in the `postgresql.conf` file. Then, reload the configuration file.

Monitoring processes in resource groups
After you create resource groups, you can get the number of processes actively using these resource groups from the view `edb_all_resource_groups`.

The following are the columns in `edb_all_resource_groups`:

- **group_name**: Name of the resource group.
- **active_processes**: Number of active processes in the resource group.
- **cpu_rate_limit**: The value of the CPU rate limit resource type assigned to the resource group.
- **per_process_cpu_rate_limit**: The CPU rate limit that applies to an individual active process in the resource group.
- **dirty_rate_limit**: The value of the dirty rate limit resource type assigned to the resource group.
- **per_process_dirty_rate_limit**: The dirty rate limit that applies to an individual active process in the resource group.

**Note**

Columns `per_process_cpu_rate_limit` and `per_process_dirty_rate_limit` don't show the actual resource consumption used by the processes. They indicate how EDB Resource Manager sets the resource limit for an individual process based on the number of active processes in the resource group.

This example shows `edb_all_resource_groups` when resource group `resgrp_a` contains no active processes, resource group `resgrp_b` contains two active processes, and resource group `resgrp_c` contains one active process:

```sql
<table>
<thead>
<tr>
<th>group_name</th>
<th>active_processes</th>
<th>cpu_rate_limit</th>
<th>per_process_cpu_rate_limit</th>
<th>dirty_rate_limit</th>
<th>per_process_dirty_rate_limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>resgrp_a</td>
<td>0</td>
<td>0.5</td>
<td></td>
<td>12288</td>
<td></td>
</tr>
<tr>
<td>resgrp_b</td>
<td>2</td>
<td>0.4</td>
<td>0.195694289022895</td>
<td>6144</td>
<td></td>
</tr>
<tr>
<td>resgrp_c</td>
<td>1</td>
<td>0.3</td>
<td>0.292342129631091</td>
<td>3072</td>
<td></td>
</tr>
</tbody>
</table>
```

The CPU rate limit and dirty rate limit settings that are assigned to these resource groups are:

```sql
<table>
<thead>
<tr>
<th>rgrpname</th>
<th>rgrpcpuuratelim</th>
<th>rgrpdirturatelim</th>
</tr>
</thead>
<tbody>
<tr>
<td>resgrp_a</td>
<td>0.5</td>
<td>12288</td>
</tr>
<tr>
<td>resgrp_b</td>
<td>0.4</td>
<td>6144</td>
</tr>
<tr>
<td>resgrp_c</td>
<td>0.3</td>
<td>3072</td>
</tr>
</tbody>
</table>
```

In the `edb_all_resource_groups` view, the `per_process_cpu_rate_limit` and `per_process_dirty_rate_limit` values are roughly the corresponding CPU rate limit and dirty rate limit divided by the number of active processes.

### 8.2.3 CPU usage throttling

EDB Resource Manager uses CPU throttling to keep the aggregate CPU usage of all processes in the group within the limit specified by the `cpu_rate_limit` parameter. A process in the group might be interrupted and put into sleep mode for a short time to maintain the defined limit. When and how such interruptions occur is defined by a proprietary algorithm used by EDB Resource Manager.

To control CPU use of a resource group, set the `cpu_rate_limit` resource type parameter:

- Set the `cpu_rate_limit` parameter to the fraction of CPU time over wall-clock time to which the combined, simultaneous CPU usage of all processes in the group must not exceed. The value assigned to `cpu_rate_limit` is typically less than or equal to 1.

- On multicore systems, you can apply the `cpu_rate_limit` to more than one CPU core by setting it to greater than 1. For example, if `cpu_rate_limit` is set to 2.0, you use 100% of two CPUs. The valid range of the `cpu_rate_limit` parameter is 0 to 1.67772e+07. A setting of 0 means no CPU rate limit was set for the resource group.
When the value is multiplied by 100, you can also interpret the \texttt{cpu_rate_limit} as the CPU usage percentage for a resource group.

Setting the CPU rate limit for a resource group

Use the \texttt{ALTER RESOURCE GROUP} command with the \texttt{SET cpu_rate_limit} clause to set the CPU rate limit for a resource group.

In this example, the CPU usage limit is set to 50% for \texttt{resgrp\_a}, 40% for \texttt{resgrp\_b}, and 30% for \texttt{resgrp\_c}. This means that the combined CPU usage of all processes assigned to \texttt{resgrp\_a} is maintained at approximately 50%. Similarly, for all processes in \texttt{resgrp\_b}, the combined CPU usage is kept to approximately 40%, and so on.

```
edb=# ALTER RESOURCE GROUP resgrp_a SET cpu_rate_limit TO .5;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_b SET cpu_rate_limit TO .4;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_c SET cpu_rate_limit TO .3;
ALTER RESOURCE GROUP
```

This query shows the settings of \texttt{cpu_rate_limit} in the catalog:

```
<table>
<thead>
<tr>
<th>rgrpname</th>
<th>rgrpcpuratelimit</th>
</tr>
</thead>
<tbody>
<tr>
<td>resgrp_a</td>
<td>0.5</td>
</tr>
<tr>
<td>resgrp_b</td>
<td>0.4</td>
</tr>
<tr>
<td>resgrp_c</td>
<td>0.3</td>
</tr>
</tbody>
</table>
```

Changing the \texttt{cpu_rate_limit} of a resource group affects new processes that are assigned to the group. It also immediately affects any currently running processes that are members of the group. That is, if the \texttt{cpu_rate_limit} is changed from 0.5 to 0.3, currently running processes in the group are throttled downward so that the aggregate group CPU usage is near 30% instead of 50%.

To show the effect of setting the CPU rate limit for resource groups, the following \texttt{psql} command-line examples use a CPU-intensive calculation of 20000 factorial (multiplication of 20000 * 19999 * 19998, and so on) performed by the query \texttt{SELECT 20000!}.

The resource groups with the CPU rate limit settings shown in the previous query are used in these examples.

Example: Single process in a single group

This example shows that the current process is set to use resource group \texttt{resgrp\_b}. The factorial calculation then starts.

```
edb=# SET edb_resource_group TO resgrp\_b;
SET
edb=# SHOW edb_resource_group;

-----------
resgrp\_b
(1 row)
edb=# SELECT 20000!;
```

In a second session, the Linux \texttt{top} command is used to display the CPU usage under the \texttt{%CPU} column. Because the \texttt{top} command output periodically changes, it represents a snapshot at an arbitrary point in time:

```
$ top
top - 16:37:03 up 4:15, 7 users, load average: 0.49, 0.39, 0.38
Tasks: 202 total, 1 running, 201 sleeping, 0 stopped, 0 zombie
Cpu(s): 42.7%us, 2.3%sy, 0.0%ni, 55.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 1025624k total, 791160k used, 234464k free, 23480k buffers
Swap: 183420k total, 13404k used, 90016k free, 373584k cached

   PID USER      PR NI VIRT  RES  SHR S %CPU %MEM     TIME+ COMMAND
28915 enterpri  20   0 105m 2000 4212 S  39.9  8.6  3:36.98 edb-postgres
1833 root      20   0 171m  77m  2960 S  7.8  5.0  3:43.96 Xorg
3840 user      20   0 278m  22m  14m S  2.2  5.1  3:41.72 knotify4
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The row where `edb-postgres` appears under the `COMMAND` column shows the `psql` session performing the factorial calculation. The CPU usage of the session shown under the `%CPU` column is 39.9, which is close to the 40% CPU limit set for resource group `resgrp_b`.

By contrast, if the `psql` session is removed from the resource group and the factorial calculation is performed again, the CPU usage is much higher.

```
edb=# SET edb_resource_group TO DEFAULT;
SET edb_resource_group;
edb_resource_group
---------------------
(1 row)
edb=# SELECT 20000!;
```

Under the `%CPU` column for `edb-postgres`, the CPU usage is now 93.6, which is significantly higher than the 39.9 when the process was part of the resource group:

```
$ top
```

$ top
```
Tasks: 202 total, 5 running, 197 sleeping, 0 stopped, 0 zombie
Cpu(s): 96.7%us, 3.3%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%
Mem: 1025624k total, 791228k used, 234396k free, 23560k buffers
Swap: 103420k total, 13404k used, 90016k free, 373508k cached
```

Example: Multiple processes in a single group

As stated previously, the CPU rate limit applies to the aggregate of all processes in the resource group. This concept is shown in the following example.

The factorial calculation is performed simultaneously in two separate `psql` sessions, each of which was added to resource group `resgrp_b` that has `cpu_rate_limit` set to .4 (CPU usage of 40%).

Session 1

```
edb=# SET edb_resource_group TO resgrp_b;
SET edb_resource_group;
edb_resource_group
---------------------
resgrp_b
(1 row)
edb=# SELECT 20000!;
```

Session 2

```
edb=# SET edb_resource_group TO resgrp_b;
SET edb_resource_group;
edb_resource_group
---------------------
resgrp_b
(1 row)
```
A third session monitors the CPU usage:

```
$ top
```

```
top - 16:53:03 up 4:31, 7 users, load average: 0.31, 0.19, 0.27
Tasks: 202 total, 1 running, 201 sleeping, 0 stopped, 0 zombie
Cpu(s): 41.2%us, 3.0%sy, 0.0%ni, 55.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0
Mem: 1025624k total, 792020k used, 233604k free, 23844k buffers
Swap: 103420k total, 13404k used, 90016k free, 373508k cached

     PID USER      PR  NI  VIRT  RES  SHR  S  %CPU %MEM    TIME+  COMMAND
29857  enterpri  20   0 195m  47m  33m  S   19.9  0.5  0:57.35  edb-postgres
28915  enterpri  20   0 195m  59m  42m  S   19.6  0.6  5:35.49  edb-postgres
3040   user      20   0 278m   2m  14m  S    1.0  2.2  3:54.99  knotify4
1033   root      20   0 171m  78m  29m  S    0.3  7.8  3:55.71  Xorg
```

Two new processes named `edb-postgres` have %CPU values of 19.9 and 19.6. The sum is close to the 40% CPU usage set for resource group `resgrp_b`.

This command sequence displays the sum of all `edb-postgres` processes sampled over half-second time intervals. This example shows how the total CPU usage of the processes in the resource group changes over time as EDB Resource Manager throttles the processes to keep the total resource group CPU usage near 40%.

```
$ while [[ $1 -eq 1 ]]; do top -d0.5 -b -n2 | grep edb-postgres | awk '{ SUM += $9} END { print SUM / 2 }'; done
37.2
39.1
38.9
38.3
44.7
39.2
42.5
39.1
39.2
39.2
41
42.85
46.1
```

Example: Multiple processes in multiple groups

In this example, two additional `psql` sessions are used along with the previous two sessions. The third and fourth sessions perform the same factorial calculation in resource group `resgrp_c` with a `cpu_rate_limit` of .3 (30% CPU usage).

```
Session 3
```

```
edb=# SET edb_resource_group TO resgrp_c;
SET
edb=# SHOW edb_resource_group;
```

```
edb_resource_group
----------------------
resgrp_c
(1 row)
edb=# SELECT 200001;
```

```
Session 4
```

```
edb=# SET edb_resource_group TO resgrp_c;
SET
edb=# SHOW edb_resource_group;
```
The `top` command displays the following output:

```
$ top
```

```
top - 17:45:09 up 5:23, 8 users, load average: 0.47, 0.17, 0.26
Tasks: 203 total, 4 running, 199 sleeping, 0 stopped, 0 zombie
Cpu(s): 78.2%us, 0.0%sy, 0.0%nI, 29.8%id, 8.0%wa, 0.0%ni, 0.0%si, 0.8
Mem: 102564k total, 806140k used, 219484k free, 252296k buffers
Swap: 383420k total, 13480k used, 90016k free, 374892k cached
```

```
PID USER     PR NI VIRT RES   SHR   S  %CPU %MEM TIME+   COMMAND
29857 enterpri 20 0  195m  4820 3324 S 19.9 0.5  4:25.02 edb-postgres
28915 enterpri 20 0  195m  5900 4212 R 19.6 0.6  9:07.50 edb-postgres
29023 enterpri 20 0  195m  4744 3248 R 16.3 0.5  4:01.73 edb-postgres
11019 enterpri 20 0  195m  4120 2764 R 15.3 0.4  0:04.92 edb-postgres
2907  user     20 0  98.7m 12m 9112 S  1.3 1.2  0:56.54 vmware-user-lo
3040  user     20 0  278m  22m  14m S  1.3 2.2  4:38.73 knotify4
```

The two resource groups in use have CPU usage limits of 40% and 30%. The sum of the %CPU column for the first two `edb-postgres` processes is 39.5 (approximately 40%, which is the limit for `resgrp_b`). The sum of the %CPU column for the third and fourth `edb-postgres` processes is 31.6 (approximately 30%, which is the limit for `resgrp_c`).

The sum of the CPU usage limits of the two resource groups to which these processes belong is 70%. The following output shows that the sum of the four processes borders around 70%:

```
$ while [[ 1 -eq 1 ]]; do top -d0.5 -b -n2 | grep edb-postgres | awk '{ SUM += $9} END { print SUM / 2 }'; done
```

```
61.8
76.4
63.6
95.9
68.55
71.25
74.85
76.9
72.4
65.9
74.9
68.25
```

By contrast, if three sessions are processing, where two sessions remain in `resgrp_b` but the third session doesn't belong to any resource group, the `top` command shows the following output:

```
$ top
```

```
top - 17:24:55 up 5:03, 7 users, load average: 1.00, 0.41, 0.38
Tasks: 199 total, 3 running, 196 sleeping, 0 stopped, 0 zombie
Cpu(s): 99.7%us, 0.3%sy, 0.0%nI, 0.0%id, 0.0%wa, 0.0%ni, 0.0%si, 0.0
Mem: 1025624k total, 797692k used, 227932k free, 24724k buffers
Swap: 383420k total, 13480k used, 90016k free, 374892k cached
```

```
PID USER     PR NI VIRT RES   SHR   S  %CPU %MEM TIME+   COMMAND
29023 enterpri 20 0  195m  4744 3248 R  58.6 0.5  2:53.75 edb-postgres
28915 enterpri 20 0  195m  5900 4212 S  18.9 0.6  7:58.45 edb-postgres
29857 enterpri 20 0  195m  4820 3324 S  18.9 0.5  3:14.85 edb-postgres
1033 root     20 0  174m  81m 2960 S   1.7 8.2  4:26.50 Xorg
3040 user     20 0  278m  22m  14m S   1.2 2.2  4:21.20 knotify4
```

The second and third `edb-postgres` processes belonging to the resource group where the CPU usage is limited to 40% have a total CPU usage of 37.8. However, the first `edb-postgres` process has a 58.6% CPU usage, as it isn't within a resource group. It basically uses the remaining available CPU resources on the system.

Likewise, the following output shows the sum of all three sessions is around 95%, since one of the sessions has no set limit on its CPU usage:

```
$ while [[ 1 -eq 1 ]]; do top -d0.5 -b -n2 | grep edb-postgres | awk '{ SUM += $9} END { print SUM / 2 }'; done
```

```
96
90.35
92.55
96.4
94.1
```
8.2.4 Dirty buffer throttling

EDB Resource Manager uses dirty buffer throttling to keep the aggregate shared buffer writing rate of all processes in the group near the limit specified by the `dirty_rate_limit` parameter. A process in the group might be interrupted and put into sleep mode for a short time to maintain the defined limit. When and how such interruptions occur is defined by a proprietary algorithm used by EDB Resource Manager.

To control writing to shared buffers, set the `dirty_rate_limit` resource type parameter.

- Set the `dirty_rate_limit` parameter to the number of kilobytes per second for the combined rate at which all the processes in the group write to, or “dirty”, the shared buffers. An example setting is 3072 kilobytes per seconds.
- The valid range of the `dirty_rate_limit` parameter is 0 to 1.67772e+07. A setting of 0 means no dirty rate limit was set for the resource group.

### Setting the dirty rate limit for a resource group

Use the `ALTER RESOURCE GROUP` command with the `SET dirty_rate_limit` clause to set the dirty rate limit for a resource group.

In this example, the dirty rate limit is set to 12288 kilobytes per second for `resgrp_a`, 6144 kilobytes per second for `resgrp_b`, and 3072 kilobytes per second for `resgrp_c`. This means that the combined writing rate to the shared buffer of all processes assigned to `resgrp_a` is maintained at approximately 12288 kilobytes per second. Similarly, for all processes in `resgrp_b`, the combined writing rate to the shared buffer is kept to approximately 6144 kilobytes per second, and so on.

```sql
edb=# ALTER RESOURCE GROUP resgrp_a SET dirty_rate_limit TO 12288;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_b SET dirty_rate_limit TO 6144;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_c SET dirty_rate_limit TO 3072;
ALTER RESOURCE GROUP
```

This query shows the settings of `dirty_rate_limit` in the catalog:

```sql
edb=# SELECT rgrpname, rgrpdirtyratelimit FROM edb_resource_group;

<table>
<thead>
<tr>
<th>rgrpname</th>
<th>rgrpdirtyratelimit</th>
</tr>
</thead>
<tbody>
<tr>
<td>resgrp_a</td>
<td>12288</td>
</tr>
<tr>
<td>resgrp_b</td>
<td>6144</td>
</tr>
<tr>
<td>resgrp_c</td>
<td>3072</td>
</tr>
</tbody>
</table>
(3 rows)
```

### Changing the dirty rate limit

Changing the `dirty_rate_limit` of a resource group affects new processes that are assigned to the group. Any currently running processes that are members of the group are also immediately affected by the change. That is, if the `dirty_rate_limit` is changed from 12288 to 3072, currently running processes in the group are throttled downward so that the aggregate group dirty rate is near 3072 kilobytes per second instead of 12288 kilobytes per second.

To show the effect of setting the dirty rate limit for resource groups, the examples use the following table for intensive I/O operations:
The `FILLFACTOR = 10` clause results in `INSERT` commands packing rows up to only 10% per page. The result is a larger sampling of dirty shared blocks for the purpose of these examples.

Displaying the number of dirty buffers

The `pg_stat_statements` module is used to display the number of shared buffer blocks that are dirtied by a SQL command and the amount of time the command took to execute. This information is used to calculate the actual kilobytes per second writing rate for the SQL command and thus compare it to the dirty rate limit set for a resource group.

To use the `pg_stat_statements` module:

1. In the `postgresql.conf` file, add `$libdir/pg_stat_statements` to the `shared_preload_libraries` configuration parameter:

   ```sql
   shared_preload_libraries = 'libdir/dbms_pipe,$libdir/edb_gen,$libdir/pg_stat_statements'
   ``

2. Restart the database server.

3. Use the `CREATE EXTENSION` command to finish creating the `pg_stat_statements` module:

   ```sql
   edb=# CREATE EXTENSION pg_stat_statements SCHEMA public;
   ``

The `pg_stat_statements_reset()` function clears out the `pg_stat_statements` view for clarity of each example.

The resource groups with the dirty rate limit settings shown in the previous query are used in these examples.

Example: Single process in a single group

This sequence of commands creates table `t1`. The current process is set to use resource group `resgrp_b`. The `pg_stat_statements` view is cleared out by running the `pg_stat_statements_reset()` function.

The `INSERT` command then generates a series of integers from 1 to 10,000 to populate the table and dirty approximately 10,000 blocks:

```sql
edb=# CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
```

```sql
edb=# SET edb_resource_group TO resgrp_b;
```

```sql
edb=# SHOW edb_resource_group;
```

```sql
edb_resource_group
-------------------------
resgrp_b
(1 row)
```

```sql
edb=# SELECT pg_stat_statements_reset();
```

```sql
pg_stat_statements_reset
-----------------------------
(1 row)
```

```sql
edb=# INSERT INTO t1 VALUES (generate_series (1,10000), 'aaa');
```

```sql
INSERT 0 10000
```

This example shows the results from the `INSERT` command:

```sql
edb=# SELECT query, rows, total_time, shared_blks_dirtied FROM pg_stat_statements;
```

<table>
<thead>
<tr>
<th>query</th>
<th>rows</th>
<th>total_time</th>
<th>shared_blks_dirtied</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO t1 VALUES (generate_series (?,?), ?);</td>
<td>10000</td>
<td>13496.184</td>
<td>10003</td>
</tr>
</tbody>
</table>
The actual dirty rate is calculated as follows:

- The number of blocks dirtied per millisecond (ms) is \( \frac{10003}{13496.184} \) blocks per millisecond, which yields 0.74117247 blocks per millisecond.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 741.17247 blocks per second.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 6072 kilobytes per second.

The actual dirty rate of 6072 kilobytes per second is close to the dirty rate limit for the resource group, which is 6144 kilobytes per second.

By contrast, if you repeat the steps without the process belonging to any resource group, the dirty buffer rate is much higher:

```
edb=# CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SHOW edb_resource_group;
edb_resource_group
(1 row)
edb=# SELECT pg_stat_statements_reset();
pg_stat_statements_reset
--------------------------
(1 row)
edb=# INSERT INTO t1 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

This example shows the results from the `INSERT` command without the use of a resource group:

```
edb=# SELECT query, rows, total_time, shared_blks_dirtied
FROM pg_stat_statements;
-[ RECORD 1 ]---------------------
 query | INSERT INTO t1 VALUES (generate_series (?,?), ?,);
 rows  | 10000
 total_time | 2432.165
 shared_blks_dirtied | 10003
```

The total time was only 2432.165 milliseconds, compared to 13496.184 milliseconds when using a resource group with a dirty rate limit set to 6144 kilobytes per second.

The actual dirty rate without the use of a resource group is calculated as follows:

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 2432.165 ms, which yields 4.112797 blocks per millisecond.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 4112.797 blocks per second.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 33692 kilobytes per second.

The actual dirty rate of 33692 kilobytes per second is much higher than when the resource group with a dirty rate limit of 6144 kilobytes per second was used.

**Example: Multiple processes in a single group**

As stated previously, the dirty rate limit applies to the aggregate of all processes in the resource group. This concept is illustrated in the following example.

For this example, the inserts are performed simultaneously on two different tables in two separate `psql` sessions, each of which was added to resource group `resgrp_b` that has a `dirty_rate_limit` set to 6144 kilobytes per second.

**Session 1**

```
edb=# CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;
edb_resource_group
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
### INSERT commands in session 1 and session 2 started after the SELECT `pg_stat_statements_reset()` command in session 2 ran.

This example shows the results from the INSERT commands in the two sessions. **RECORD 3** shows the results from session 1. **RECORD 2** shows the results from session 2.

The INSERT commands in session 1 and session 2 started after the `pg_stat_statements_reset()` command in session 2 ran. This example shows the results from the INSERT commands in the two sessions. RECORD 3 shows the results from session 1. RECORD 2 shows the results from session 2.

```sql
> CREATE TABLE t2 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
> CREATE TABLE t1 (generate_series (1,10000), 'aaa');
> CREATE TABLE resgrp_b
```

<table>
<thead>
<tr>
<th>RECORD 1</th>
<th>query</th>
<th>rows</th>
<th>total_time</th>
<th>shared_blks_dirtied</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT <code>pg_stat_statements_reset()</code></td>
<td>1</td>
<td>8.43</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RECORD 2</th>
<th>query</th>
<th>rows</th>
<th>total_time</th>
<th>shared_blks_dirtied</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INSERT INTO t2 VALUES (generate_series (?,?), ?);</td>
<td>10000</td>
<td>30591.551</td>
<td>10003</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RECORD 3</th>
<th>query</th>
<th>rows</th>
<th>total_time</th>
<th>shared_blks_dirtied</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INSERT INTO t1 VALUES (generate_series (?,?), ?);</td>
<td>10000</td>
<td>33215.334</td>
<td>10003</td>
</tr>
</tbody>
</table>

The total time was 33215.334 milliseconds for session 1 and 30591.551 milliseconds for session 2. When only one session was active in the same resource group, the time was 13496.184 milliseconds. Thus, more active processes in the resource group result in a slower dirty rate for each active process in the group. The following calculations show this.

The actual dirty rate for session 1 is calculated as follows:

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 33215.334 ms, which yields 0.30115609 blocks per millisecond.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 301.15609 blocks per second.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 2467 kilobytes per second.

The actual dirty rate for session 2 is calculated as follows:

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 30591.551 ms, which yields 0.32698571 blocks per millisecond.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 326.98571 blocks per second.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 2679 kilobytes per second.

The combined dirty rate from session 1 (2467 kilobytes per second) and from session 2 (2679 kilobytes per second) yields 5146 kilobytes per second, which is below the set dirty rate limit of the resource group (6144 kilobytes per second).
Example: Multiple processes in multiple groups

In this example, two additional `psql` sessions are used along with the previous two sessions. The third and fourth sessions perform the same `INSERT` command in resource group `resgrp_c` with a `dirty_rate_limit` of 3072 kilobytes per second.

Repeat sessions 1 and 2 from the prior example using resource group `resgrp_b` with a `dirty_rate_limit` of 6144 kilobytes per second:

Session 3

```sql
edb=# CREATE TABLE t3 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SET edb_resource_group TO resgrp_c;
SET
edb=# SHOW edb_resource_group;

edb_resource_group
-------------------
resgrp_c
(1 row)

edb=# INSERT INTO t3 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

Session 4

```sql
edb=# CREATE TABLE t4 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SET edb_resource_group TO resgrp_c;
SET
edb=# SHOW edb_resource_group;

edb_resource_group
-------------------
resgrp_c
(1 row)

edb=# SELECT pg_stat_statements_reset();

pg_stat_statements_reset
-------------------------
(1 row)

edb=# INSERT INTO t4 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

Note

The `INSERT` commands in all four sessions started after the `SELECT pg_stat_statements_reset()` command in session 4 ran.

This example shows the results from the `INSERT` commands in the four sessions:

- **RECORD 3** shows the results from session 1.
- **RECORD 2** shows the results from session 2.
- **RECORD 4** shows the results from session 3.
- **RECORD 5** shows the results from session 4.

```sql
edb=# SELECT query, rows, total_time, shared_blks_dirtied FROM pg_stat_statements;

- [ RECORD 1 ]--------
  | query | SELECT pg_stat_statements_reset();
  | rows  | 1
  | total_time | 8.467
  | shared_blks_dirtied | 0
```
The times of session 1 (28407.435) and session 2 (31343.458) are close to each other, as they are both in the same resource group with `dirty_rate_limit` set to 6144. These times differ from the times of session 3 (52727.846) and session 4 (56063.697), which are in the resource group with `dirty_rate_limit` set to 3072. The latter group has a slower dirty rate limit, so the expected processing time is longer, as is the case for sessions 3 and 4.

The actual dirty rate for session 1 is calculated as follows:

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 28407.435 ms, which yields 0.35212612 blocks per millisecond.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 352.12612 blocks per second.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 2885 kilobytes per second.

The actual dirty rate for session 2 is calculated as follows:

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 31343.458 ms, which yields 0.31914156 blocks per millisecond.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 319.14156 blocks per second.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 2614 kilobytes per second.

The combined dirty rate from session 1 (2885 kilobytes per second) and from session 2 (2614 kilobytes per second) yields 5499 kilobytes per second, which is near the set dirty rate limit of the resource group (6144 kilobytes per seconds).

The actual dirty rate for session 3 is calculated as follows:

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 52727.846 ms, which yields 0.18971001 blocks per millisecond.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 189.71001 blocks per second.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 1554 kilobytes per second.

The actual dirty rate for session 4 is calculated as follows:

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 56063.697 ms, which yields 0.17842205 blocks per millisecond.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields 178.42205 blocks per second.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately 1462 kilobytes per second.

The combined dirty rate from session 3 (1554 kilobytes per second) and from session 4 (1462 kilobytes per second) yields 3016 kilobytes per second, which is near the set dirty rate limit of the resource group (3072 kilobytes per seconds).

This example shows how EDB Resource Manager keeps the aggregate dirty rate of the active processes in its groups close to the dirty rate limit set for each group.

### 8.3 Loading bulk data

EDB*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for EDB Postgres Advanced Server. The EDB*Loader command line utility loads data from an input source, typically a file, into one or more tables using a subset of the parameters offered by Oracle SQL*Loader.

#### 8.3.1 EDB*Loader key concepts and compatibility

**Key features**

EDB*Loader features include:

- Support for the Oracle SQL*Loader data loading methods (conventional path load, direct path load, and parallel direct path load)
Version compatibility restrictions

The important version compatibility restrictions between the EDB*Loader client and the database server are:

- When you invoke the EDB*Loader program (called `edbldr`), you pass in parameters and directive information to the database server. We strongly recommend that you use the version of the EDB*Loader client, the `edbldr` program supplied with the version of EDB Postgres Advanced Server you are using, to load data only into the database server. In general, use the same version for the EDB*Loader client and database server.

- Using EDB*Loader with connection poolers such as PgPool-II and PgBouncer isn't supported. EDB*Loader must connect directly to EDB Postgres Advanced Server version 14. Alternatively, you can use these commands for loading data through connection poolers:

  ```
  psql \copy
  jdbc copyIn
  psycopg2 copy_from
  ```

8.3.2 Overview of data loading methods

As with Oracle SQL*Loader, EDB*Loader supports three data loading methods:

- **Conventional path load** — Conventional path load is the default method used by EDB*Loader. Use basic insert processing to add rows to the table. The advantage of a conventional path load is that table constraints and database objects defined on the table are enforced during a conventional path load. Table constraints and database objects include primary keys, not null constraints, check constraints, unique indexes, foreign key constraints, triggers, and so on. One exception is that the EDB Postgres Advanced Server rules defined on the table aren't enforced. EDB*Loader can load tables on which rules are defined. However, the rules aren't executed. As a consequence, you can't load partitioned tables implemented using rules with EDB*Loader. See Conventional path load.

- **Direct path load** — A direct path load is faster than a conventional path load but requires removing most types of constraints and triggers from the table. See Direct path load.

- **Parallel direct path load** — A parallel direct path load provides even greater performance improvement by permitting multiple EDB*Loader sessions to run simultaneously to load a single table. See Parallel direct path load.

**Note**

Create EDB Postgres Advanced Server rules using the `CREATE RULE` command. EDB Postgres Advanced Server rules aren't the same database objects as rules and rule sets used in Oracle.

8.3.3 EDB*Loader error handling

EDB*Loader can load data files with either delimiter-separated or fixed-width fields in single-byte or multibyte character sets. The delimiter can be a string consisting of one or more single-byte or multibyte characters. Data file encoding and the database encoding can differ. Character set conversion of the data file to the database encoding is supported.

Each EDB*Loader session runs as a single, independent transaction. If an error occurs during the EDB*Loader session that aborts the transaction, all changes made during the session are rolled back. Generally, formatting errors in the data file don't result in an aborted transaction. Instead, the badly formatted records are written to a text file called the bad file. The reason for the error is recorded in the log file.

Records causing database integrity errors result in an aborted transaction and rollback. As with formatting errors, the record causing the error is written to the bad file and the reason is recorded in the log file.

**Note**

EDB*Loader differs from Oracle SQL*Loader in that a database integrity error results in a rollback in EDB*Loader. In Oracle SQL*Loader, only the record causing the error is rejected. Records that were previously inserted into the table are retained, and loading continues after the rejected record.

The following are examples of types of formatting errors that don't abort the transaction:

- Attempt to load non-numeric value into a numeric column
- Numeric value too large for a numeric column
- Character value too long for the maximum length of a character column
- Attempt to load improperly formatted date value into a date column
The following are examples of types of database errors that abort the transaction and result in the rollback of all changes made in the EDB*Loader session:

- Violating a unique constraint such as a primary key or unique index
- Violating a referential integrity constraint
- Violating a check constraint
- Error thrown by a trigger fired as a result of inserting rows

### 8.3.4 Building the EDB*Loader control file

When you invoke EDB*Loader, the list of arguments provided must include the name of a control file. The control file includes the instructions that EDB*Loader uses to load the tables from the input data file.

#### Contents of the control file

The control file includes information such as:

- The name of the input data file containing the data to load
- The name of the tables to load from the data file
- Names of the columns in the tables and their corresponding field placement in the data file
- Specification of whether the data file uses a delimiter string to separate the fields or if the fields occupy fixed column positions
- Optional selection criteria to choose the records from the data file to load into a given table
- The name of the file that collects illegally formatted records
- The name of the discard file that collects records that don’t meet the selection criteria of any table

#### Control file syntax

The syntax for the EDB*Loader control file is:

```plaintext
[ OPTIONS (<param>=<value> [, <param>=<value>] ...) ]

LOAD DATA
[ CHARACTERSET <charset> ]
[ INFILE '{ <data_file> | <stdin> }'
[ BADFILE '<bad_file>' ]
[ DISCARDFILE '<discard_file>' ]
[ ( DISCARDMAX | DISCARDS ) <max_discard_recs> ]
[ INSERT | APPEND | REPLACE | TRUNCATE ]
[ PRESERVE BLANKS ]
[ INTO TABLE <target_table> ]
[ WHEN <field_condition> [ AND <field_condition>] ]
[ FIELDS TERMINATED BY '<termstring>' ]
[ OPTIONALLY ENCLOSED BY '<enclstring>' ]
[ RECORDS DELIMITED BY '<delimstring>' ]
[ TRAILING NULLCOLS ]
[ <field_def> [, <field_def>] ]
...
```

Where `field_def` defines a field in the specified `data_file`. The field describes the location, data format, or value of the data to insert into `column_name` of `target_table`. The syntax of `field_def` is:

```plaintext
<column_name>
[
  CONSTANT <val>
  | FILLER [ POSITION (<start>:<end>) ] [ <fieldtype> ]
  | BOUNDFILLER [ POSITION (<start>:<end>) ] [ <fieldtype> ]
  | [ POSITION (<start>:<end>) ] [ <fieldtype> ]
]```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Specifying one of

```sql
[ NULLIF <field_condition> [ AND <field_condition>] ]
...
[ PRESCRIBE BLANKS ] [ "<expr>"
]
```

Where `fieldtype` is one of:

- `CHAR [<(length)>] | DATE [[<length>]] | TIMESTAMP [[<length>]] [ "<datemask>" ] | INTEGER EXTERNAL [[<length>]]
- FLOAT EXTERNAL [<(length)>] | DECIMAL EXTERNAL [<(length)>]
- ZONED EXTERNAL [<(length)>] | ZONED [<(precision> [,scale])]]

Setting the variables

The specification of `data_file`, `bad_file`, and `discard_file` can include the full directory path or a relative directory path to the filename. If the filename is specified alone or with a relative directory path, the file is then assumed to exist, in the case of `data_file`, relative to the current working directory from which you invoke `edbldr`. In the case of `bad_file` or `discard_file`, it's created.

You can include references to environment variables in the EDB*Loader control file when referring to a directory path or filename. Environment variable references are formatted differently on Windows systems than on Linux systems:

- On Linux, the format is `$ENV_VARIABLE` or `${ENV_VARIABLE}`.
- On Windows, the format is `%ENV_VARIABLE%`.

Where `ENV_VARIABLE` is the environment variable that's set to the directory path or filename.

Set the `EDBLDR_ENV_STYLE` environment variable to interpret environment variable references as Windows-styled references or Linux-styled references regardless of the operating system on which EDB*Loader resides. You can use this environment variable to create portable control files for EDB*Loader.

- On a Windows system, set `EDBLDR_ENV_STYLE` to `windows` to recognize Windows-style references in the control file.
- On a Linux system, set `EDBLDR_ENV_STYLE` to `linux` or `unix` to recognize Linux-style references in the control file.

The operating system account `enterprisedb` must have read permission on the directory and file specified by `data_file`. It must have write permission to the directories where `bad_file` and `discard_file` are written.

Note

The filenames for `data_file`, `bad_file`, and `discard_file` must have extensions `.dat`, `.bad`, and `.dsc`, respectively. If the provided filename doesn't have an extension, EDB*Loader assumes the actual filename includes the appropriate extension.

Example scenarios

Suppose an EDB*Loader session results in data format errors, the `BADFILE` clause isn't specified, and the `BAD` parameter isn't given on the command line when `edbldr` is invoked. In this case, a bad file is created with the name `control_file_base.bad` in the directory from which `edbldr` is invoked. `control_file_base` is the base name of the control file used in the `edbldr` session.

If all of the following conditions are true, the discard file isn't created even if the EDB*Loader session results in discarded records:

- The `DISCARDFILE` clause for specifying the discard file isn't included in the control file.
- The `DISCARD` parameter for specifying the discard file isn't included on the command line.
- The `DISCARDFILE` clause for specifying the maximum number of discarded records isn't included in the control file.
- The `DISCARDFILE` clause for specifying the maximum number of discarded records isn't included in the control file.

Suppose you don't specify the `DISCARDFILE` clause and the `DISCARD` parameter for explicitly specifying the discard file name, but you do specify `DISCARDFMAX` or `DISCARDS`. In this case, the EDB*Loader session creates a discard file using the data filename with an extension of `.dsc`.

Note

The keywords `DISCARD` and `DISCARDS` differ. `DISCARD` is an EDB*Loader command line parameter used to specify the discard file name. `DISCARDS` is a clause of the `LOAD DATA` directive that can appear only in the control file. Keywords `DISCARDS` and `DISCARDFMAX` provide the same functionality of specifying the maximum number of discarded records allowed before terminating the EDB*Loader session. Records loaded into the database before terminating the EDB*Loader session due to exceeding the `DISCARDFMAX` or `DISCARDS` settings are kept in the database. They aren't rolled back.

Specifying one of `INSERT`, `APPEND`, `REPLACE`, or `TRUNCATE` establishes the default action for adding rows to target tables. The default action is `INSERT`.
If you specify the `FIELDS TERMINATED BY` clause, then you can't specify the `POSITION (start:end)` clause for any `field_def`. Alternatively, if you don't specify the `FIELDS TERMINATED BY` clause, then every `field_def` must contain the `POSITION (start:end)` clause, the `fieldtype(length)` clause, or the `CONSTANT` clause.

For complete descriptions of the parameters available for each clause, see EDB*Loader control file parameters.

### 8.3.4.1 EDB*Loader control file examples

The following are some examples of control files and their corresponding data files.

#### Delimiter-separated field data file

This control file uses a delimiter-separated data file that appends rows to the `emp` table. The `APPEND` clause is used to allow inserting additional rows into the `emp` table.

```sql
LOAD DATA
INFIL 'emp.dat'
BADFILE 'emp.bad'

APPEND INTO TABLE emp
...
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '
TRAILING NULLCOLS
(
empno, ename, job, mgr, hiredate, sal, deptno, comm
)
```

The following is the corresponding delimiter-separated data file:

```
9101, ROGERS, CLERK, 7902, 17-DEC-10, 1980.00, 20
9102, PETERSON, SALESMAN, 7698, 20-DEC-10, 2600.00, 30, 2300.00
9103, WARREN, SALESMAN, 7698, 22-DEC-10, 5250.00, 30, 2500.00
9104, "JONES, JR.", MANAGER, 7839, 02-APR-09, 7975.00, 20
```

The use of the `TRAILING NULLCOLS` clause allows you to omit the last field supplying the `comm` column from the first and last records. The `comm` column is set to null for the rows inserted from these records.

Double quotation marks surround the value `"JONES, JR."` in the last record since the comma delimiter character is part of the field value.

This query displays the rows added to the table after the EDB*Loader session:

```sql
SELECT * FROM emp WHERE empno > 9100;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-10</td>
<td>1980.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>9102</td>
<td>PETERSON</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-DEC-10</td>
<td>2600.00</td>
<td>2300.00</td>
<td>30</td>
</tr>
<tr>
<td>9103</td>
<td>WARREN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-DEC-10</td>
<td>5250.00</td>
<td>2500.00</td>
<td>30</td>
</tr>
<tr>
<td>9104</td>
<td>&quot;JONES, JR.&quot;</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-09</td>
<td>7975.00</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

(4 rows)

#### Fixed-width field data file

This control file loads the same rows into the `emp` table. It uses a data file containing fixed-width fields. The `FIELDS TERMINATED BY` and `OPTIONALLY ENCLOSED BY` clauses are absent. Instead, each field includes the `POSITION` clause.
The following is the corresponding data file containing fixed-width fields:

```
LOAD DATA
INFILE 'emp_fixed.dat'
BADFILE 'emp_fixed.bad'
APPEND
INTO TABLE emp
TRAILING NULLCOLS
{
  empno  POSITION (1:4),
  ename  POSITION (5:14),
  job    POSITION (15:23),
  mgr    POSITION (24:27),
  hiredate POSITION (28:38),
  sal    POSITION (39:46),
  deptno POSITION (47:48),
  comm   POSITION (49:56)
}
```

Single physical record data file – RECORDS DELIMITED BY clause

This control file loads the same rows into the `emp` table but uses a data file with one physical record. Terminate each record loaded as a row in the table using a semicolon (`;`). The RECORDS DELIMITED BY clause specifies this value.

```
LOAD DATA
INFILE 'emp_recdelim.dat'
BADFILE 'emp_recdelim.bad'
APPEND
INTO TABLE emp
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
RECORDS DELIMITED BY ';'
TRAILING NULLCOLS
{
  empno,
  ename,
  job,
  mgr,
  hiredate,
  sal,
  deptno,
  comm
}
```

The following is the corresponding data file. The content is a single physical record in the data file. The record delimiter character is included following the last record, that is, at the end of the file.

```
9101,ROGERS,CLERK,7902,17-DEC-10,1980.00,20,;
9102,PETERSON,SALESMAN,7698,20-DEC-10,2600.00,30,2300.00,;
9103,WARREN,SALESMAN,7698,22-DEC-10,5250.00,30,2500.00,;
9104,JONES,JR.,MANAGER,7839,02-APR-09,7975.00,20,;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
This control file uses the `FILLER` clause in the data fields for the `sal` and `comm` columns. EDB*Loader ignores the values in these fields and sets the corresponding columns to null.

Using the same fixed-width data file as in the prior fixed-width field example, the resulting rows in the table appear as follows:

```
empno| ename     | job     | mgr|   hiredate        | sal   | comm   |deptno
------+-----------+---------+----+-------------------+-------+--------+------
9101 | ROGERS    | CLERK   |7902| 17-DEC-10 00:00:00|       |        |  20
9102 | PETERSON  | SALESMAN|7698| 20-DEC-10 00:00:00|       |        |  30
9103 | WARREN    | SALESMAN|7698| 22-DEC-10 00:00:00|       |        |  30
9104 | JONES, JR.| MANAGER |7839| 02-APR-09 00:00:00|       |        |  20
```

(4 rows)

This control file uses the `BOUNDFillER` clause in the data fields for the `job` and `mgr` columns. EDB*Loader ignores the values in these fields and sets the corresponding columns to null in the same manner as the `FILLER` clause. However, unlike columns with the `FILLER` clause, you can use columns with the `BOUNDFillER` clause in an expression, as shown for column `jobdesc`.

```
SELECT * FROM empjob WHERE empno > 9100;
```

Using the same fixed-width data file as in the prior fixed-width field example, the resulting rows in the table appear as follows:

```
empno| ename    | job | mgr | hiredate    | sal  | comm  |deptno
----|----------|-----|-----|-------------|------|-------|------
9101 | ROGERS   | CLERK |7902| 17-DEC-10 00:00:00| | |  20
9102 | PETERSON | SALESMAN|7698| 20-DEC-10 00:00:00| | |  30
9103 | WARREN   | SALESMAN|7698| 22-DEC-10 00:00:00| | |  30
9104 | JONES, JR.| MANAGER |7839| 02-APR-09 00:00:00| | |  20
```

(4 rows)

Using the same fixed-width data file as in the prior fixed-width field example, the resulting rows in the table appear as follows:

```
empno| ename    | job | mgr | hiredate    | sal  | comm  |deptno
----|----------|-----|-----|-------------|------|-------|------
9101 | ROGERS   | CLERK |7902| 17-DEC-10 00:00:00| | |  20
9102 | PETERSON | SALESMAN|7698| 20-DEC-10 00:00:00| | |  30
9103 | WARREN   | SALESMAN|7698| 22-DEC-10 00:00:00| | |  30
9104 | JONES, JR.| MANAGER |7839| 02-APR-09 00:00:00| | |  20
```

(4 rows)
The following is the delimiter-separated data file used in this example:

```
comm | FILLER,
jobdesc | "\|' for manager '||
:mgr"
)
```

The following is the corresponding data file containing fixed-width fields:

```
9101,ROGERS,CLERK,7982,17-DEC-10,1980.00,20
9102,PETERSON,SALESMAN,7698,20-DEC-10,2600.00,30
9103,WARREN,SALESMAN,7698,22-DEC-10,5250.00,30
9104,"JONES, JR.",MANAGER,7839,02-APR-09,7975.00,20
```

Field types with length specification

This control file contains the field-type clauses with the length specification:

```
LOAD DATA
INFILE 'emp_fixed.dat'
BADFILE 'emp_fixed.bad'
APPEND INTO TABLE emp
TRAILING NULLCOLS(47:48),
comm DECIMAL
)
```

Note

You can use the POSITION clause and the fieldtype(length) clause individually or in combination as long as each field definition contains at least one of the two clauses.
The resulting rows in the table appear as follows:

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-10</td>
<td>1980.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>9102</td>
<td>PETERSON</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-DEC-10</td>
<td>2600.00</td>
<td>2300.00</td>
<td>30</td>
</tr>
<tr>
<td>9103</td>
<td>WARREN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-DEC-10</td>
<td>5250.00</td>
<td>2500.00</td>
<td>30</td>
</tr>
<tr>
<td>9104</td>
<td>JONES, JR.</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-09</td>
<td>7975.00</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

The following is the control file:

```
LOAD DATA
INFILE 'emp_fixed_2.dat'
BADFILE 'emp_fixed_2.bad'
APPEND INTO TABLE emp
TRAILING NULLCOLS
{
  empno       POSITION (1:4),
  ename       POSITION (5:14),
  job         POSITION (15:23),
  mgr         POSITION (24:27),
  hiredate    POSITION (28:38),
  sal         POSITION (39:46),
  comm        POSITION (49:56)
}
```

The following is the corresponding data file:

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-10</td>
<td>1980.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>9102</td>
<td>PETERSON</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-DEC-10</td>
<td>2600.00</td>
<td>2300.00</td>
<td>30</td>
</tr>
<tr>
<td>9103</td>
<td>WARREN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-DEC-10</td>
<td>5250.00</td>
<td>2500.00</td>
<td>30</td>
</tr>
<tr>
<td>9104</td>
<td>JONES, JR.</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-09</td>
<td>7975.00</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

The resulting rows in the table appear as follows:

```
SELECT empno, ename, job, NVL(TO_CHAR(sal), '--null--') "sal",
NVL(TO_CHAR(comm), '--null--') "comm", deptno FROM emp WHERE empno > 9100;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9105</td>
<td>ARNOLDS</td>
<td>CLERK</td>
<td>3750</td>
<td>800</td>
<td>30</td>
</tr>
<tr>
<td>9106</td>
<td>JACKSON</td>
<td>ANALYST</td>
<td>4500</td>
<td>2000</td>
<td>40</td>
</tr>
</tbody>
</table>
The `sal` column for employee JONES, JR. is null since the job is MANAGER.

The `comm` values from the data file for employees PETERSON, WARREN, ARNOLDS, and MAXWELL are all loaded into the `comm` column of the `emp` table since these employees are either SALESMAN or members of department 30.

The `comm` value of 2000.00 in the data file for employee JACKSON is ignored, and the `comm` column of the `emp` table is set to null. This employee isn’t a SALESMAN or a member of department 30.

**SELECT statement in a field expression**

This example uses a `SELECT` statement in the expression of the field definition to return the value to load into the column:

```sql
LOAD DATA INFILE 'emp_fixed.dat' BADFILE 'emp_fixed.bad'
APPEND INTO TABLE emp TRAILING NULLCOLS
{
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23) "(SELECT dname FROM dept WHERE deptno = :
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      POSITION (47:48),
    comm        POSITION (49:56)
}
```

The following is the content of the `dept` table used in the `SELECT` statement:

```sql
SELECT * FROM dept;

```

<table>
<thead>
<tr>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>40</td>
</tr>
</tbody>
</table>

(4 rows)

The following is the corresponding data file:

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>hiredate</th>
<th>sal</th>
<th>deptno</th>
<th>comm</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>CLERK</td>
<td>799217-DEC-10</td>
<td>1980.00</td>
<td>2300.00</td>
<td>20</td>
</tr>
<tr>
<td>9102</td>
<td>PETERSON</td>
<td>SALESMAN</td>
<td>769820-DEC-10</td>
<td>2600.00</td>
<td>2500.00</td>
<td>30</td>
</tr>
<tr>
<td>9103</td>
<td>WARREN</td>
<td>SALESMAN</td>
<td>769822-DEC-10</td>
<td>5250.00</td>
<td>2500.00</td>
<td>30</td>
</tr>
<tr>
<td>9104</td>
<td>JONES, JR.</td>
<td>MANAGER</td>
<td>783902-APR-89</td>
<td>7975.00</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>9105</td>
<td>ARNOLDS</td>
<td>CLERK</td>
<td>769820-DEC-10</td>
<td>3750.00</td>
<td>800.00</td>
<td>30</td>
</tr>
<tr>
<td>9106</td>
<td>JACKSON</td>
<td>ANALYST</td>
<td>4580.00</td>
<td>4500.00</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>9107</td>
<td>MAXWELL</td>
<td>SALESMAN</td>
<td>2600.00</td>
<td>2600.00</td>
<td>1600.00</td>
<td>10</td>
</tr>
</tbody>
</table>

(7 rows)
The resulting rows in the table appear as follows:

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>RESEARCH</td>
<td>7902</td>
<td>17-DEC-10 00:00:00</td>
<td>1980.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>9102</td>
<td>PETERSON</td>
<td>SALES</td>
<td>7698</td>
<td>20-DEC-10 00:00:00</td>
<td>2600.00</td>
<td>2300.00</td>
<td>30</td>
</tr>
<tr>
<td>9103</td>
<td>WARREN</td>
<td>SALES</td>
<td>7698</td>
<td>22-DEC-10 00:00:00</td>
<td>5250.00</td>
<td>2500.00</td>
<td>30</td>
</tr>
<tr>
<td>9104</td>
<td>JONES, JR.</td>
<td>RESEARCH</td>
<td>7839</td>
<td>02-APR-09 00:00:00</td>
<td>7975.00</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

(4 rows)

Note

The `job` column contains the value from the `dname` column of the `dept` table returned by the `SELECT` statement instead of the job name from the data file.

Multiple INTO TABLE clauses

This example uses multiple INTO TABLE clauses. For this example, two empty tables are created with the same data definition as the `emp` table. The following CREATE TABLE commands create these two empty tables without inserting rows from the original `emp` table:

```sql
CREATE TABLE emp_research AS SELECT * FROM emp WHERE deptno = 99;
CREATE TABLE emp_sales AS SELECT * FROM emp WHERE deptno = 99;
```

This control file contains two INTO TABLE clauses. Without an APPEND clause, it uses the default operation of INSERT. For this operation, the tables `emp_research` and `emp_sales` must be empty.

```sql
LOAD DATA
INFILE 'emp_multitbl.dat'
BADFILE 'emp_multitbl.bad'
DISCARDFILE 'emp_multitbl.dsc'
INTO TABLE emp_research
WHEN (47:48) = '20'
TRAILING NULLCOLS

( empno      POSITION (1:4),
  ename      POSITION (5:14),
  job        POSITION (15:23),
  mgr        POSITION (24:27),
  hiredate   POSITION (28:38),
  sal        POSITION (39:46),
  deptno     CONSTANT '20',
  comm       POSITION (49:56)
)
INTO TABLE emp_sales
WHEN (47:48) = '30'
TRAILING NULLCOLS

( empno      POSITION (1:4),
  ename      POSITION (5:14),
  job        POSITION (15:23),
  mgr        POSITION (24:27),
  hiredate   POSITION (28:38),
  sal        POSITION (39:46),
  comm       POSITION (49:56)
)
The WHEN clauses specify that when the field designated by columns 47 through 48 contains 20, the record is inserted into the emp_research table. When that same field contains 30, the record is inserted into the emp_sales table. If neither condition is true, the record is written to the discard file emp_multitbl.dsc.

The CONSTANT clause is given for column deptno, so the specified constant value is inserted into deptno for each record. When the CONSTANT clause is used, it must be the only clause in the field definition other than the column name to which the constant value is assigned.

Column comm of the emp_sales table is assigned a SQL expression. Expressions can reference column names by prefixing the column name with a colon character (:

The following is the corresponding data file:

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-10 00:00:00</td>
<td>1980.00</td>
<td></td>
<td>20.00</td>
</tr>
<tr>
<td>9104</td>
<td>JONES, JR.</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-09 00:00:00</td>
<td>7975.00</td>
<td></td>
<td>20.00</td>
</tr>
<tr>
<td>9105</td>
<td>ARNOLDS</td>
<td>CLERK</td>
<td>7782</td>
<td>13-SEP-10 00:00:00</td>
<td>3750.00</td>
<td></td>
<td>10.00</td>
</tr>
<tr>
<td>9106</td>
<td>JACKSON</td>
<td>ANALYST</td>
<td>7566</td>
<td>03-JAN-11 00:00:00</td>
<td>4500.00</td>
<td></td>
<td>40.00</td>
</tr>
</tbody>
</table>

The records for employees ARNOLDS and JACKSON contain 10 and 40 in columns 47 through 48, which don't satisfy any of the WHEN clauses. EDB*Loader writes these two records to the discard file, emp_multitbl.dsc, with the following content:

The following are the rows loaded into the emp_research and emp_sales tables:

```sql
SELECT * FROM emp_research;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9101</td>
<td>ROGERS</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-10 00:00:00</td>
<td>1980.00</td>
<td></td>
<td>20.00</td>
</tr>
<tr>
<td>9104</td>
<td>JONES, JR.</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-09 00:00:00</td>
<td>7975.00</td>
<td></td>
<td>20.00</td>
</tr>
</tbody>
</table>

The following are the rows loaded into the emp_sales tables:

```sql
SELECT * FROM emp_sales;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9102</td>
<td>PETERSON</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-DEC-10 00:00:00</td>
<td>2500.00</td>
<td>2300.00</td>
<td>30.00</td>
</tr>
<tr>
<td>9103</td>
<td>WARREN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-DEC-10 00:00:00</td>
<td>5250.00</td>
<td>3813.00</td>
<td>30.00</td>
</tr>
</tbody>
</table>

8.3.5 Invoking EDB*Loader

You can run EDB*Loader as superuser or as a normal user.

8.3.5.1 Running EDB*Loader

Use the following command to invoke EDB*Loader from the command line:

```
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Description

You can specify parameters listed in the syntax diagram in a parameter file. Exceptions include the \_d option, \_p option, \_h option, and the PARFILE parameter. Specify the parameter file on the command line when you invoke edbldr using PARFILE=param_file. You can specify some parameters in the OPTIONS clause in the control file. For more information on the control file, see Building the EDB*Loader control file.

You can include the full directory path or a relative directory path to the file name when specifying control_file, data_file, bad_file, discard_file, log_file, and param_file. If you specify the file name alone or with a relative directory path, the file is assumed to exist in the case of control_file, data_file, or param_file relative to the current working directory from which edbldr is invoked. In the case of bad_file, discard_file, or log_file, the file is created. If you omit the \_d option, the \_p option, or the \_h option, the defaults for the database, port, and host are determined by the same rules as other EDB Postgres Advanced Server utility programs, such as edb-psql.

Requirements

- The control file must exist in the character set encoding of the client where edbldr is invoked. If the client is in an encoding different from the database encoding, then you must set the PGCLIENTENCODING environment variable on the client to the client's encoding prior to invoking edbldr. This technique ensures character set conversion between the client and the database server is done correctly.
- The file names must include these extensions:
  - control_file must use the .ctl extension.
  - data_file must use the .dat extension.
  - bad_file must use the .bad extension.
  - discard_file must use the .dsc extension.
  - log_file must include the .log extension.

If the provided file name doesn't have an extension, EDB*Loader assumes the actual file name includes the appropriate extension.

- The operating system account used to invoke edbldr must have read permission on the directories and files specified by control_file, data_file, and param_file.

- The operating system account enterprisedb must have write permission on the directories where bad_file, discard_file, and log_file are written.

Parameters

dbname

Name of the database containing the tables to load.

port

Port number on which the database server is accepting connections.

host
IP address of the host on which the database server is running.

USERID={ <username/password> | <username/> | <username> | / }

EDB*Loader connects to the database with <username>. <username> must be a superuser or a username with the required privileges. <password> is the password for <username>.

If you omit the USERID parameter, EDB*Loader prompts for username and password. If you specify USERID=username/, EDB*Loader either:

- Uses the password file specified by the environment variable PGPASSFILE if PGPASSFILE is set
- Uses the .pgpass password file (pgpass.conf on Windows systems) if PGPASSFILE isn’t set

If you specify USERID=username, EDB*Loader prompts for password. If you specify USERID=/, the connection is attempted using the operating system account as the user name.

Note

EDB*Loader ignores the EDB Postgres Advanced Server connection environment variables PGUSER and PGPASSWORD. See the PostgreSQL core documentation for information on the PGPASSFILE environment variable and the password file.

-c CONNECTION_STRING

connstr=CONNECTION_STRING

The -c or connstr option allows you to specify all the connection parameters supported by libpq. With this option, you can also specify SSL connection parameters or other connection parameters supported by libpq. If you provide connection options such as -d, -h, -p, or user=postgres/dbpass separately, they might override the values provided by the -c or connstr option.

CONTROL=control_file

control_file specifies the name of the control file containing EDB*Loader directives. If you don’t specify a file extension, an extension of .ctl is assumed.

For more information on the control file, see Building the EDB*Loader control file.

DATA=data_file

data_file specifies the name of the file containing the data to load into the target table. If you don’t specify a file extension, an extension of .dat is assumed. Specifying a data_file on the command line overrides the INFILE clause specified in the control file.

For more information about data_file, see Building the EDB*Loader control file.

BAD=bad_file

bad_file specifies the name of a file that receives input data records that can’t be loaded due to errors. Specifying bad_file on the command line overrides any BADFILE clause specified in the control file.

For more information about bad_file, see Building the EDB*Loader control file.

DISCARD=discard_file

discard_file is the name of the file that receives input data records that don’t meet any table’s selection criteria. Specifying discard_file on the command line overrides the DISCARDFILE clause in the control file.

For more information about discard_file, see Building the EDB*Loader control file.

DISCARDMAX=max_discard_recs

max_discard_recs is the maximum number of discarded records that can be encountered from the input data records before terminating the EDB*Loader session. Specifying max_discard_recs on the command line overrides the DISCARDMAX or DISCARDS clause in the control file.

For more information about max_discard_recs, see Building the EDB*Loader control file.

HANDLE_CONFLICTS={ FALSE | TRUE }

If any record insertion fails due to a unique constraint violation, EDB*Loader aborts the entire operation. You can instruct EDB*Loader to instead move the duplicate record to the BAD file and continue processing by setting HANDLE_CONFLICTS to TRUE. This behavior applies only if indexes are present. By default, HANDLE_CONFLICTS is set to FALSE.

Setting HANDLE_CONFLICTS to TRUE isn’t supported with direct path loading. If you set this parameter to TRUE when direct path loading, EDB*Loader throws an error.

LOG=log_file

log_file specifies the name of the file in which EDB*Loader records the results of the EDB*Loader session.
If you omit the `LOG` parameter, EDB*Loader creates a log file with the name `control_file_base.log` in the directory from which edbldr is invoked. `control_file_base` is the base name of the control file used in the EDB*Loader session. The operating system account `enterprisedb` must have write permission on the directory where the log file is written.

```plaintext
PARFILE=param_file
```

`param_file` specifies the name of the file that contains command line parameters for the EDB*Loader session. You can specify command line parameters listed in this section in `param_file` instead of on the command line. Exceptions are the `-d`, `-p`, and `-h` options, and the `PARFILE` parameter.

Any parameter given in `param_file` overrides the same parameter supplied on the command line before the `PARFILE` option. Any parameter given on the command line that appears after the `PARFILE` option overrides the same parameter given in `param_file`.

!!! Note Unlike other EDB*Loader files, there's no default file name or extension assumed for `param_file`. However, by Oracle SQL*Loader convention, .par is typically used as an extension. It isn't required.

```plaintext
DIRECT= { FALSE | TRUE }
```

If `DIRECT` is set to `TRUE`, EDB*Loader performs a direct path load instead of a conventional path load. The default value of `DIRECT` is `FALSE`.

Don’t set `DIRECT=true` when loading the data into a replicated table. If you’re using EDB*Loader to load data into a replicated table and set `DIRECT=true`, indexes might omit rows that are in a table or might contain references to rows that were deleted. EnterpiseDB doesn’t support direct inserts to load data into replicated tables.

For information about direct path loads, see Direct path load.

```plaintext
FREEZE= { FALSE | TRUE }
```

Set `FREEZE` to `TRUE` to copy the data with the rows frozen. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wraparound. For more information about frozen tuples, see the PostgreSQL core documentation.

You must specify a data-loading type of `TRUNCATE` in the control file when using the `FREEZE` option. `FREEZE` isn't supported for direct loading.

By default, `FREEZE` is `FALSE`.

```plaintext
ERRORS=error_count
```

`error_count` specifies the number of errors permitted before aborting the EDB*Loader session. The default is `50`.

```plaintext
PARALLEL= { FALSE | TRUE }
```

Set `PARALLEL` to `TRUE` to indicate that this EDB*Loader session is one of a number of concurrent EDB*Loader sessions participating in a parallel direct path load. The default value of `PARALLEL` is `FALSE`.

When `PARALLEL` is `TRUE`, the `DIRECT` parameter must also be set to `TRUE`.

For more information about parallel direct path loads, see Parallel direct path load.

```plaintext
ROWS=n
```

`n` specifies the number of rows that EDB*Loader commits before loading the next set of `n` rows.

```plaintext
SKIP=skip_count
```

Number of records at the beginning of the input data file to skip before loading begins. The default is `0`.

```plaintext
SKIP_INDEX_MAINTENANCE= { FALSE | TRUE }
```

If set to `TRUE`, index maintenance isn’t performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of `SKIP_INDEX_MAINTENANCE` is `FALSE`.

During a parallel direct path load, target table indexes aren’t updated. They’re marked as invalid after the load is complete.

You can use the `REINDEX` command to rebuild an index. For more information about the `REINDEX` command, see the PostgreSQL core documentation.

```plaintext
edb_resource_group=group_name
```

`group_name` specifies the name of an EDB Resource Manager resource group to which to assign the EDB*Loader session.

Any default resource group that was assigned to the session is overridden by the resource group given by the `edb_resource_group` parameter specified on the edbldr command line. An example of such a group is a database user running the EDB*Loader session who was assigned a default resource group with the `ALTER ROLE ... SET edb_resource_group` command.
Examples

This example invokes EDB*Loader using a control file named `emp.ctl` to load a table in database `edb`. The file is located in the current working directory.

```
$ /usr/edb/as14/bin/edblr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl
EDB*Loader: Copyright (c) 2007-2021, EnterpriseDB Corporation.
Successfully loaded (4) records
```

In this example, EDB*Loader prompts for the user name and password since they're omitted from the command line. In addition, the files for the bad file and log file are specified with the BAD and LOG command line parameters.

```
$ /usr/edb/as14/bin/edblr -d edb CONTROL=emp.ctl BAD=/tmp/emp.bad
LOG=/tmp/emp.log
Enter the user name : enterprisedb
Enter the password :
EDB*Loader: Copyright (c) 2007-2021, EnterpriseDB Corporation.
Successfully loaded (4) records
```

This example runs EDB*Loader using a parameter file located in the current working directory. The SKIP and ERRORS parameter default values are specified in the parameter file in addition to the CONTROL, BAD, and LOG files. The parameter file, `emp.par`, contains:

```
CONTROL=emp.ctl
BAD=/tmp/emp.bad
LOG=/tmp/emp.log
SKIP=1
ERRORS=10
```

Invoke EDB*Loader with the parameter file:

```
$ /usr/edb/as14/bin/edblr -d edb PARFILE=emp.par
Enter the user name : enterprisedb
Enter the password :
EDB*Loader: Copyright (c) 2007-2021, EnterpriseDB Corporation.
Successfully loaded (3) records
```

This example invokes EDB*Loader using a `connstr=` option that uses the `emp.ctl` control file located in the current working directory to load a table in a database named `edb`:

```
$ /usr/edb/as14/bin/edblr connstr="sslmode=verify-ca sslcompression=0
host=localhost dbname=edb port=5444 user=enterprisedb" CONTROL=emp.ctl
EDB*Loader: Copyright (c) 2007-2021, EnterpriseDB Corporation.
Successfully loaded (4) records
```

This example invokes EDB*Loader using a normal user. For this example, one empty table `bar` is created and a normal user `bob` is created. The `bob` user is granted all privileges on the table `bar`. The CREATE TABLE command creates the empty table. The CREATE USER command creates the user, and the GRANT command gives required privileges to the user `bob` on the `bar` table:

```
CREATE TABLE bar(i int);
CREATE USER bob identified by '123';
GRANT ALL on bar TO bob;
```

The control file and data file:

```
## Control file
EDBAS/ - (master) $ cat /tmp/edblr.ctl
LOAD DATA INFILE '/tmp/edblr.dat'
truncate into table bar
{
  i position(1:1)
}

## Data file
EDBAS/ - (master) $ cat /tmp/edblr.dat
1
2
3
5
```

Invoke EDB*Loader:
Exit codes

When EDB*Loader exits, it returns one of the following codes:

<table>
<thead>
<tr>
<th>Exit code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>All rows loaded successfully.</td>
</tr>
<tr>
<td>1</td>
<td>EDB*Loader encountered command line or syntax errors or aborted the load operation due to an unrecoverable error.</td>
</tr>
<tr>
<td>2</td>
<td>The load completed, but some or all rows were rejected or discarded.</td>
</tr>
<tr>
<td>3</td>
<td>EDB*Loader encountered fatal errors, such as OS errors. This class of errors is equivalent to the <code>FATAL</code> or <code>PANIC</code> severity levels of PostgreSQL errors.</td>
</tr>
</tbody>
</table>

8.3.5.2    Updating a table with a conventional path load

You can use EDB*Loader with a conventional path load to update the rows in a table, merging new data with the existing data. When you invoke EDB*Loader to perform an update, the server searches the table for an existing row with a matching primary key:

- If the server locates a row with a matching key, it replaces the existing row with the new row.
- If the server doesn’t locate a row with a matching key, it adds the new row to the table.

To use EDB*Loader to update a table, the table must have a primary key. You can’t use EDB*Loader to update a partitioned table.

Performing the update

To perform UPDATE, use the same steps as when performing a conventional path load:

1. Create a data file that contains the rows you want to update or insert.
2. Define a control file that uses the `INFILE` keyword to specify the name of the data file. For information about building the EDB*Loader control file, see Building the EDB*Loader control file.
3. Invoke EDB*Loader, specifying the database name, connection information, and the name of the control file. For information about invoking EDB*Loader, see Invoking EDB*Loader.

This example uses the `emp` table that’s distributed with the EDB Postgres Advanced Server sample data. By default, the table contains:

```sql
select * from emp;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7922</td>
<td>17-DEC-80 00:00:00</td>
<td>800.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81 00:00:00</td>
<td>1600.00</td>
<td>500.00</td>
<td>20</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
<td>1250.00</td>
<td>500.00</td>
<td>20</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
<td>2975.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7564</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
<td>1250.00</td>
<td>1400.00</td>
<td>20</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01-MAY-81 00:00:00</td>
<td>2850.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
<td>2450.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7780</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
<td>3000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>7839</td>
<td>17-NOV-81 00:00:00</td>
<td>5000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08-SEP-81 00:00:00</td>
<td>1500.00</td>
<td>0.00</td>
<td>20</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7788</td>
<td>23-MAY-87 00:00:00</td>
<td>1180.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
<td>950.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7962</td>
<td>FORD</td>
<td>ANALYST</td>
<td>7566</td>
<td>03-DEC-81 00:00:00</td>
<td>3000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23-JAN-82 00:00:00</td>
<td>1380.00</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>
(14 rows)

This control file (`emp_update.ctl`) specifies the fields in the table in a comma-delimited list. The control file performs an `UPDATE` on the `emp` table.

```sql
LOAD DATA
INFILE 'emp_update.dat'
BADFILE 'emp_update.bad'
DISCARDFILE 'emp_update.dsc'
UPDATE INTO TABLE
emp
```
The data that's being updated or inserted is saved in the `emp_update.dat` file. `emp_update.dat` contains:

```
Invoke EDB*Loader, specifying the name of the database (edb), the name of a database user and their associated password, and the name of the control file (emp_update.ctl):
```

```
edblr -d edb userid=user_name/password control=emp_update.ctl
```

Results of the update

After performing the update, the `emp` table contains:

```
7521,WARD,MANAGER,7839,22-FEB-81 00:00:00,3800.00,0.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,3500.00,0.00,20
7903,BAKER,SALESMAN,7521,10-JUN-13 00:00:00,1800.00,500.00,20
7904,MILLS,SALESMAN,7839,13-JUN-13 00:00:00,1500.00,500.00,20
7654,MARTIN,SALESMAN,7521,28-SEP-81 00:00:00,1500.00,400.00,20
```

The rows containing information for the three employees that are currently in the `emp` table are updated, while rows are added for the new employees (`BAKER` and `MILLS`).

### 8.3.5.3 Running a direct path load

During a direct path load, EDB*Loader writes the data directly to the database pages, which is then synchronized to disk. The insert processing associated with a conventional path load is bypassed, resulting in performance improvement. Bypassing insert processing reduces the types of constraints on the target table. The types of constraints permitted on the target table of a direct path load are:

- Primary key
- Not null constraints
- Indexes (unique or non-unique)

**Restrictions**

The restrictions on the target table of a direct path load are:

- Triggers aren't permitted.
- Check constraints aren't permitted.
- Foreign key constraints on the target table referencing another table aren't permitted.
- Foreign key constraints on other tables referencing the target table aren't permitted.
- You must not partition the table.
- Rules can exist on the target table, but they aren't executed.

**Note**

Currently, a direct path load in EDB*Loader is more restrictive than in Oracle SQL*Loader. The preceding restrictions don't apply to Oracle SQL*Loader in most cases. The following restrictions apply to a control file used in a direct path load:

```
FIELDS TERMINATED BY ",",

select * from emp;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Multiple table loads aren't supported. You can specify only one `INTO TABLE` clause in the control file.

You can't use SQL expressions in the data field definitions of the `INTO TABLE` clause.

The `FREEZE` option isn't supported for direct path loading.

Running the direct path load

To run a direct path load, add the `DIRECT=TRUE` option:

```
$ /usr/edb/as14/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl DIRECT=TRUE
EDB*Loader: Copyright (c) 2007-2021, EnterpriseDB Corporation.
Successfully loaded (4) records
```

8.3.5.4 Running a parallel direct path load

You can further improve the performance of a direct path load by distributing the loading process over two or more sessions running concurrently. Each session runs a direct path load into the same table.

Since the same table is loaded from multiple sessions, the input records to load into the table must be divided among several data files. This way, each EDB*Loader session uses its own data file, and the same record isn't loaded into the table more than once.

Restrictions

The target table of a parallel direct path load is under the same restrictions as a direct path load run in a single session.

The restrictions on the target table of a direct path load are:

- Triggers aren't permitted.
- Check constraints aren't permitted.
- Foreign key constraints on the target table referencing another table aren't permitted.
- Foreign key constraints on other tables referencing the target table aren't permitted.
- You must not partition the table.
- Rules can exist on the target table, but they aren't executed.

In addition, you must specify the `APPEND` clause in the control file used by each EDB*Loader session.

Running a parallel direct path load

To run a parallel direct path load, run EDB*Loader in a separate session for each participant of the parallel direct path load. You must include the `DIRECT=TRUE` and `PARALLEL=TRUE` parameters when invoking each such EDB*Loader session.

Each EDB*Loader session runs as an independent transaction. Aborting and rolling back changes of one of the parallel sessions doesn't affect the loading done by the other parallel sessions.

Note

In a parallel direct path load, each EDB*Loader session reserves a fixed number of blocks in the target table using turns. Some of the blocks in the last allocated chunk might not be used, and those blocks remain uninitialized. A later use of the `VACUUM` command on the target table might show warnings about these uninitialized blocks, such as the following:

```
WARNING: relation "emp" page 98264 is uninitialized --- fixing
WARNING: relation "emp" page 98265 is uninitialized --- fixing
WARNING: relation "emp" page 98266 is uninitialized --- fixing
```

This behavior is expected and doesn't indicate data corruption.

Indexes on the target table aren't updated during a parallel direct path load. They are therefore marked as invalid after the load is complete. You must use the `REINDEX` command to rebuild the indexes.

This example shows the use of a parallel direct path load on the `emp` table.

Note
If you attempt a parallel direct path load on the sample `emp` table provided with EDB Postgres Advanced Server, you must first remove the triggers and constraints referencing the `emp` table. In addition, the primary key column, `empno`, was expanded from `NUMBER(4)` to `NUMBER` in this example to allow for inserting more rows.

This is the control file used in the first session:

```sql
LOAD DATA
INFILE
'/home/user/loader/emp_parallel_1.dat'

APPEND
INTO TABLE
emp

FIELDS TERMINATED BY ',
"OPTIONALLY ENCLOSED BY
'',' TRAILING
NULLCOLS
{
empno,
ename,
job,
mgr,
hiredate,
sal,
depthno,
comm
}
```

You must specify the `APPEND` clause in the control file for a parallel direct path load.

This example invokes EDB*Loader in the first session. You must specify the `DIRECT=TRUE` and `PARALLEL=TRUE` parameters.

```
$ /usr/edb/as14/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp_parallel_1.ctl DIRECT=TRUE PARALLEL=TRUE
WARNING: index maintenance will be skipped with PARALLEL load
EDB*Loader: Copyright (c) 2007-2021, EnterpriseDB Corporation.
```

The control file used for the second session appears as follows. It’s the same as the one used in the first session, but it uses a different data file.

```sql
LOAD DATA
INFILE
'/home/user/loader/emp_parallel_2.dat'

APPEND
INTO TABLE
emp

FIELDS TERMINATED BY ',
"OPTIONALLY ENCLOSED BY
'',' TRAILING
NULLCOLS
{
empno,
ename,
job,
mgr,
hiredate,
sal,
depthno,
comm
}
```

This control file is used in a second session:

```
$ /usr/edb/as14/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp_parallel_2.ctl DIRECT=TRUE PARALLEL=TRUE
WARNING: index maintenance will be skipped with PARALLEL load
EDB*Loader: Copyright (c) 2007-2021, EnterpriseDB Corporation.
```

EDB*Loader displays a message in each session when the load operation completes:

```
Successfully loaded (10000) records
```

This query shows that the index on the `emp` table was marked `INVALID`:
Note

user_indexes is the view of indexes compatible with Oracle databases owned by the current user.

Queries on the emp table don’t use the index unless you rebuild it using the \texttt{REINDEX} command:

\begin{verbatim}
REINDEX INDEX emp_pk;
\end{verbatim}

A later query on user_indexes shows that the index is now marked as \texttt{VALID}:

\begin{verbatim}
SELECT index_name, status FROM user_indexes WHERE table_name = 'EMP';
\end{verbatim}

\begin{verbatim}
<table>
<thead>
<tr>
<th>index_name</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMP_PK</td>
<td>VALID</td>
</tr>
</tbody>
</table>
\end{verbatim}

(1 row)

8.3.5.5 Performing remote loading

EDB*Loader supports a feature called remote loading. In remote loading, the database containing the table to load is running on a database server on a host different from the one where EDB*Loader is invoked with the input data source.

This feature is useful if you have a large amount of data to load, and you don’t want to create a large data file on the host running the database server.

In addition, you can use the standard input feature to pipe the data from the data source, such as another program or script, directly to EDB*Loader. EDB*Loader then loads the table in the remote database. This feature bypasses having to create a data file on disk for EDB*Loader.

Requirements

Performing remote loading using standard input requires:

- The \texttt{edbldr} program must be installed on the client host on which to invoke it with the data source for the EDB*Loader session.
- The control file must contain the clause \texttt{INFILE 'stdin'}, so you can pipe the data directly into EDB*Loader’s standard input. For information on the \texttt{INFILE} clause and the EDB*Loader control file, see \texttt{Building the EDB*Loader control file}.
- All files used by EDB*Loader, such as the control file, bad file, discard file, and log file, must reside on or be created on the client host on which \texttt{edbldr} is invoked.
- When invoking EDB*Loader, use the \texttt{-h} option to specify the IP address of the remote database server. For more information, see \texttt{Invoking EDB*Loader}.
- Use the operating system pipe operator (\textbar) or input redirection operator (<) to supply the input data to EDB*Loader.

Loading a database

This example loads a database running on a database server at \texttt{192.168.1.14} using data piped from a source named \texttt{datasource}:

\begin{verbatim}
datasource | ./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password CONTROL=remote.ctl
\end{verbatim}

This example also shows how you can use standard input:

\begin{verbatim}
./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password CONTROL=remote.ctl < datasource
\end{verbatim}

8.4 Copying a database

EDB Clone Schema is an extension module for EDB Postgres Advanced Server that allows you to copy a schema and its database objects from a local or remote database (the source database) to a
receiving database (the target database).

The source and target databases can be either:

- The same physical database
- Different databases in the same database cluster
- Separate databases running under different database clusters on separate database server hosts

8.4.1 EDB Clone Schema key concepts and limitations

EDB Clone Schema functions

The EDB Clone Schema functions are created in the `edb_util` schema when the `parallel_clone` and `edb_cloneschema` extensions are installed.

Prerequisites

Verify the following conditions before using an EDB Clone Schema function:

- You’re connected to the target or local database as the database superuser defined in the `CREATE USER MAPPING` command for the foreign server of the target or local database.
- The `edb_util` schema is in the search path, or invoke the cloning function with the `edb_util` prefix.
- The target schema doesn’t exist in the target database.
- When using the remote copy functions, if the `on_tblspace` parameter is set to `true`, then the target database cluster contains all tablespaces that are referenced by objects in the source schema. Otherwise, creating the DDL statements for those database objects fails in the target schema, which causes a failure of the cloning process.
- When using the remote copy functions, if you set the `copy_acls` parameter to `true`, then all roles that have `GRANT` privileges on objects in the source schema exist in the target database cluster. Otherwise granting privileges to those roles fails in the target schema, which causes a failure of the cloning process.
- `pgAgent` is running against the target database if you’re using the non-blocking form of the function.

EDB Postgres Advanced Server includes `pgAgent` as a component. For information about `pgAgent`, see the `pgAdmin` documentation.

Overview of the functions

Use the following functions with EDB Clone Schema:

- `localcopyschema`. This function copies a schema and its database objects from a source database into the same database (the target) but with a different schema name from the original. Use this function when the source schema and the copy will reside within the same database. See `localcopyschema` for more information.
- `localcopyschema_nb`. This function performs the same purpose as `localcopyschema` but as a background job, which frees up the terminal from which the function was initiated. This function is referred to as a non-blocking function. See `localcopyschema_nb` for more information.
- `remotecopyschema`. This function copies a schema and its database objects from a source database to a different target database. Use this function when the source schema and the copy will reside in separate databases. The separate databases can reside in the same EDB Postgres Advanced Server database clusters or in different ones. See `remotecopyschema` for more information.
- `remotecopyschema_nb`. This function performs the same purpose as `remotecopyschema` but as a background job, which frees up the terminal from which the function was initiated. This function is a non-blocking function. See `remotecopyschema_nb` for more information.
- `process_status_from_log`. This function displays the status of the cloning functions. The information is obtained from a log file you specify when invoking a cloning function. See `process_status_from_log` for more information.
- `remove_log_file_and_job`. This function deletes the log file created by a cloning function. You can also use this function to delete a job created by the non-blocking form of the function. See `remove_log_file_and_job` for more information.

List of supported database objects

You can clone these database objects from one schema to another:

- Data types
- Tables including partitioned tables, excluding foreign tables
- Indexes
- Constraints
- Sequences
- View definitions
- Materialized views
- Private synonyms
- Table triggers, but excluding event triggers
- Rules
- Functions
- Procedures
- Packages
- Comments for all supported object types
- Access control lists (ACLs) for all supported object types
You can’t clone the following database objects:

- Large objects (Postgres LOBs and BFILEs)
- Logical replication attributes for a table
- Database links
- Foreign data wrappers
- Foreign tables
- Event triggers
- Extensions

For cloning objects that rely on extensions, see the limitations that follow.

- Row-level security
- Policies
- Operator class

Limitations

The following limitations apply:

- EDB Clone Schema is supported on EDB Postgres Advanced Server when you specify a dialect of Compatible with Oracle on the EDB Postgres Advanced Server Dialect dialog box during installation. It’s also supported when you include the --redwood-like keywords during a text-mode installation or cluster initialization.
- The source code in functions, procedures, triggers, packages, and so on, aren’t modified after being copied to the target schema. If such programs contain coded references to objects with schema names, the programs might fail when invoked in the target schema if such schema names are no longer consistent in the target schema.
- Cross-schema object dependencies aren’t resolved. If an object in the target schema depends on an object in another schema, this dependency isn’t resolved by the cloning functions.
- For remote cloning, if an object in the source schema depends on an extension, then you must create this extension in the public schema of the remote database before invoking the remote cloning function.
- At most, 16 copy jobs can run in parallel to clone schemas. Each job can have at most 16 worker processes to copy table data in parallel.
- You can’t cancel queries run by background workers.

8.4.2 Setting up EDB Clone Schema

To use EDB Clone Schema, you must first install several extensions along with the PL/Perl language on any database used as the source or target database by an EDB Clone Schema function.

In addition, it might help to modify some configuration parameters in the postgresql.conf file of the database servers.

Installing extensions

Perform this installation on any database to be used as the source or target database by an EDB Clone Schema function.

1. Install the following extensions on the database: postgres_fdw, dblink, adminpack and pgagent.

2. Ensure that pgAgent is installed before creating the pgagent extension. On Linux, you can use the edb-as<xx>-pgagent RPM package, where <xx> is the EDB Postgres Advanced Server version number to install pgAgent. On Windows, use StackBuilder Plus to download and install pgAgent.

Install the extensions:

```sql
CREATE EXTENSION postgres_fdw SCHEMA public;
CREATE EXTENSION dblink SCHEMA public;
CREATE EXTENSION adminpack;
CREATE EXTENSION pgagent;
```

For more information about using the CREATE EXTENSION command, see the PostgreSQL core documentation.

Modifying the configuration file

Modify the postgresql.conf file by adding $libdir/parallel_clone to the shared_preload_libraries configuration parameter:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/dbms_aq,$libdir/parallel_clone'
```

Installing PL/Perl
1. Install the Perl procedural language (PL/Perl) on the database, and run the `CREATE TRUSTED LANGUAGE plperl` command. For Linux, install PL/Perl using the `edb-as<xx>-server-plperl` RPM package, where `<xx>` is the EDB Postgres Advanced Server version number. For Windows, use the EDB Postgres Language Pack. For information on EDB Language Pack, see the EDB Postgres Language Pack.

2. Connect to the database as a superuser and run the following command:

   ```
   CREATE TRUSTED LANGUAGE plperl;
   ```

   For more information about using the `CREATE LANGUAGE` command, see the PostgreSQL core documentation.

---

**Setting configuration parameters**

You might need to modify configuration parameters in the `postgresql.conf` file.

**Performance configuration parameters**

You might need to tune the system for copying a large schema as part of one transaction. Tuning of configuration parameters is for the source database server referenced in a cloning function.

You might need to tune the following configuration parameters in the `postgresql.conf` file:

- `work_mem`. Specifies the amount of memory for internal sort operations and hash tables to use before writing to temporary disk files.
- `maintenance_work_mem`. Specifies the maximum amount of memory for maintenance operations such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY to use.
- `max_worker_processes`. Sets the maximum number of background processes that the system can support.
- `checkpoint_timeout`. Maximum time between automatic WAL checkpoints, in seconds.
- `checkpoint_completion_target`. Specifies the target of checkpoint completion as a fraction of total time between checkpoints.
- `checkpoint_flush_after`. Whenever more than `checkpoint_flush_after` bytes are written while performing a checkpoint, attempt to force the OS to issue these writes to the underlying storage.
- `max_wal_size`. Maximum size to let the WAL grow to between automatic WAL checkpoints.
- `max_locks_per_transaction`. Controls the average number of object locks allocated for each transaction. Individual transactions can lock more objects as long as the locks of all transactions fit in the lock table.

For information about the configuration parameters, see the PostgreSQL core documentation.

**Status logging**

Status logging by the cloning functions creates log files in the directory specified by the `log_directory` parameter in the `postgresql.conf` file for the database server to which you're connected when invoking the cloning function.

The default location is `PGDATA/log`:

```
log_directory = 'log'        # directory where log files are written, # can be absolute or relative to PGDATA
```

This directory must exist before running a cloning function.

The name of the log file is determined by what you specify in the parameter list when invoking the cloning function.

To display the status from a log file, use the `process_status_from_log` function.

To delete a log file, use the `remove_log_file_and_job` function, or delete it manually from the log directory.

---

**Installing EDB Clone Schema**

Install the EDB Clone Schema on any database to be used as the source or target database by an EDB Clone Schema function.

1. If you previously installed an older version of the `edb_cloneschema` extension, run the following command:

   ```
   DROP EXTENSION parallel_clone CASCADE;
   ```

   This command also drops the `edb_cloneschema` extension.

2. Install the extensions. Make sure that you create the `parallel_clone` extension before creating the `edb_cloneschema` extension.
Creating the foreign servers and user mappings

When using one of the local cloning functions `localcopyschema` or `localcopyschema_nb`, one of the required parameters includes a single, foreign server. This server is for identifying the database server and its database that's the source and receiver of the cloned schema.

When using one of the remote cloning functions `remotecopyschema` or `remotecopyschema_nb`, two of the required parameters include two foreign servers. The foreign server specified as the first parameter identifies the source database server and its database that's the provider of the cloned schema. The foreign server specified as the second parameter identifies the target database server and its database that's the receiver of the cloned schema.

For each foreign server, you must create a user mapping. When a selected database superuser invokes a cloning function, that superuser must be mapped to a database user name and password that has access to the foreign server that's specified as a parameter in the cloning function.

For general information about foreign data, foreign servers, and user mappings, see the PostgreSQL core documentation.

Foreign server and user mapping for local cloning functions

For the `localcopyschema` and `localcopyschema_nb` functions, the source and target schemas are both in the same database of the same database server. You must define and specify only one foreign server for these functions. This foreign server is also referred to as the local server because this server is the one to which you must be connected when invoking the `localcopyschema` or `localcopyschema_nb` function.

The user mapping defines the connection and authentication information for the foreign server. You must create this foreign server and user mapping in the database of the local server in which the cloning occurs.

This example creates the foreign server for the database containing the schema to clone and to receive the cloned schema:

```sql
CREATE SERVER local_server FOREIGN DATA WRAPPER postgres_fdw
OPTIONS(
    host 'localhost',
    port '5444',
    dbname 'edb'
);
```

For more information about using the `CREATE SERVER` command, see the PostgreSQL core documentation.

The user mapping for this server is:

```sql
CREATE USER MAPPING FOR enterprisedb SERVER local_server
OPTIONS
    (user 'enterprisedb',
     password 'password'
);
```

For more information about using the `CREATE USER MAPPING` command, see the PostgreSQL core documentation.

These psql commands show the foreign server and user mapping:

```
edb=# \des
```

<table>
<thead>
<tr>
<th>List of foreign servers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Owner</td>
</tr>
<tr>
<td>Foreign-data wrapper</td>
</tr>
<tr>
<td>Access privileges</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Version</td>
</tr>
<tr>
<td>FDW options</td>
</tr>
<tr>
<td>Description</td>
</tr>
</tbody>
</table>
List of user mappings

<table>
<thead>
<tr>
<th>Server</th>
<th>User name</th>
<th>FDW options</th>
</tr>
</thead>
<tbody>
<tr>
<td>local_server</td>
<td>enterprisedb</td>
<td>(&quot;user&quot; 'enterprisedb', password 'password')</td>
</tr>
</tbody>
</table>

When database superuser `enterprisedb` invokes a cloning function, the database user `enterprisedb` with its password is used to connect to `local_server` on the `localhost` with port `5444` to database `edb`.

In this case, the mapped database user, `enterprisedb`, and the database user, `enterprisedb`, used to connect to the local `edb` database are the same database user. However, that's not required.

For specific use of these foreign server and user mapping examples, see the example given in `localcopieschema`.

Foreign server and user mapping for remote cloning functions

For the `remotecopyschema` and `remotecopyschema_nb` functions, the source and target schemas are in different databases of either the same or different database servers. You must define and specify two foreign servers for these functions.

The foreign server defining the originating database server and its database containing the source schema to clone is referred to as the source server or the remote server.

The foreign server defining the database server and its database to receive the schema to clone is referred to as the target server or the local server. The target server is also referred to as the local server because this server is the one to which you must be connected when invoking the `remotecopyschema` or `remotecopyschema_nb` function.

The user mappings define the connection and authentication information for the foreign servers. You must create all of these foreign servers and user mappings in the target database of the target/local server. The database user for whom the user mappings are defined must be a superuser and the user connected to the local server when invoking an EDB Clone Schema function.

This example creates the foreign server for the local, target database that receives the cloned schema:

```sql
CREATE SERVER tgt_server FOREIGN DATA WRAPPER
          postgres_fdw
           OPTIONS( host 'localhost',
                 port '5444',
                 dbname 'tgtdb');
```

The user mapping for this server is:

```sql
CREATE USER MAPPING FOR enterprisedb SERVER
          tgt_server
     OPTIONS ( user 'tgtuser',
                 password 'tgtpassword');
```

This example creates the foreign server for the remote, source database that's the source for the cloned schema:

```sql
CREATE SERVER src_server FOREIGN DATA WRAPPER
          postgres_fdw
           OPTIONS( host '192.168.2.28',
                 port '5444',
                 dbname 'srcdb');
```

The user mapping for this server is:

```sql
CREATE USER MAPPING FOR enterprisedb SERVER
          src_server
     OPTIONS ( user 'srcuser',
                 password 'srcpassword');
```

Displaying foreign servers and user mappings

```sql
deu$ \deu
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
These psql commands show the foreign servers and user mappings:

```sql

tgtdb=# \des+

List of foreign servers
- [ RECORD 1 ]-----------------------------------------------------
  Name                 | src_server
  Owner                | tgtuser
  Foreign-data wrapper | postgres_fdw
  Access privileges    |
  Type                 |
  Version              |
  FDW options          | (host '192.168.2.28', port '5444', dbname 'srcdb')
  Description          ---------------------------------------------
- [ RECORD 2 ]-----------------------------------------------------
  Name                 | tgt_server
  Owner                | tgtuser
  Foreign-data wrapper | postgres_fdw
  Access privileges    |
  Type                 |
  Version              |
  FDW options          | (host 'localhost', port '5444', dbname 'tgtdb')
  Description          |
```

When database superuser `enterprisedb` invokes a cloning function, the database user `tgtuser` with password `tgtpassword` is used to connect to `tgt_server` on the `localhost` with port `5444` to database `tgtdb`.

In addition, database user `srcuser` with password `srcpassword` connects to `src_server` on host `192.168.2.28` with port `5444` to database `srcdb`.

```
tgtdb=# \deu+

List of user mappings
<table>
<thead>
<tr>
<th>Server</th>
<th>User name</th>
<th>FDW options</th>
</tr>
</thead>
<tbody>
<tr>
<td>src_server</td>
<td>enterprisedb</td>
<td>(&quot;user&quot; 'srcuser', password 'srcpassword')</td>
</tr>
<tr>
<td>tgt_server</td>
<td>enterprisedb</td>
<td>(&quot;user&quot; 'tgtuser', password 'tgtpassword')</td>
</tr>
</tbody>
</table>
```

Note

Be sure the `pg_hba.conf` file of the database server running the source database `srcdb` has an appropriate entry. This entry must permit connection from the target server location (address `192.168.2.27` in the following example) with the database user `srcuser` that was included in the user mapping for the foreign server `src_server` defining the source server and database.

```
# TYPE DATABASE USER ADDRESS METHOD
# "local" is for Unix domain socket connections only
local all all md5
# IPv4 local connections:
host srcdb srcuser 192.168.2.27/32 md5
```

For specific use of these foreign server and user mapping examples, see the example given in `remotecopyschema`.

### 8.4.3 Copying database objects from a local source to a target

There are two functions you can use with EDB Clone Schema to perform a local copy of a schema and its database objects:

- **localcopyschema** — This function copies a schema and its database objects from a source database into the same database (the target) but with a different schema name from the original. Use this function when the source schema and the copy will reside within the same database. See `localcopyschema` for more information.
- **localcopyschema_nb** — This function performs the same purpose as `localcopyschema` but as a background job, which frees up the terminal from which the function was initiated. This function is referred to as a non-blocking function. See `localcopyschema_nb` for more information.

#### Performing a local copy of a schema

The `localcopyschema` function copies a schema and its database objects in a local database specified in the `source_fdw` foreign server from the source schema to the specified target schema in the same database.

```
localcopyschema(
    <source_fdw> TEXT,
```
The function returns a Boolean value. If the function succeeds, then `true` is returned. If the function fails, then `false` is returned.

The `source_fdw`, `source_schema`, `target_schema`, and `log_filename` are required parameters while all other parameters are optional.

**Parameters**

- **source_fdw**
  
  Name of the foreign server managed by the `postgres_fdw` foreign data wrapper from which to clone database objects.

- **source_schema**
  
  Name of the schema from which to clone database objects.

- **target_schema**
  
  Name of the schema into which to clone database objects from the source schema.

- **log_filename**
  
  Name of the log file in which information from the function is recorded. The log file is created under the directory specified by the `log_directory` configuration parameter in the `postgresql.conf` file.

- **on_tblspace**
  
  Boolean value to specify whether to create database objects in their tablespaces. If `false`, then the `TABLESPACE` clause isn't included in the applicable `CREATE` DDL statement when added to the target schema. If `true`, then the `TABLESPACE` clause is included in the `CREATE` DDL statement when added to the target schema. The default value is `false`.

- **verbose_on**
  
  Boolean value to specify whether to print the DDLs in `log_filename` when creating objects in the target schema. If `false`, then DDLs aren't printed. If `true`, then DDLs are printed. The default value is `false`.

- **copy_acls**
  
  Boolean value to specify whether to include the access control list (ACL) while creating objects in the target schema. The access control list is the set of `GRANT` privilege statements. If `false`, then the access control list isn't included for the target schema. If `true`, then the access control list is included for the target schema. The default value is `false`.

- **worker_count**
  
  Number of background workers to perform the clone in parallel. The default value is `1`.

**Example**

This example shows the cloning of schema `edb` containing a set of database objects to target schema `edbcopy`. Both schemas are in database `edb` as defined by `local_server`.

The example is for the following environment:

- Host on which the database server is running: `localhost`
- Port of the database server: `5444`
- Database source/target of the clone: `edb`
- Foreign server (`local_server`) and user mapping with the information of the preceding bullet points
- Source schema: `edb`
- Target schema: `edbcopy`
- Database superuser to invoke `localcopyschema: enterprisedb`

Before invoking the function, database user `enterprisedb` connects to the database `edb`:

```
edb=# SET search_path TO "$user",public,edb_util;
```
The following displays the logging status using the `process_status_from_log` function:

```sql
edb# SELECT process_status_from_log('clone_edb_edbcopy');
```

```
process_status_from_log

```

Results

After the clone is complete, the following shows some of the database objects copied to the `edbcopy` schema:

```sql
edb# SET search_path TO edbcopy;
```

```
List of relations
Schema | Name   | Type | Owner       | Size | Description
-------+--------+------+-------------+------+----------------
edbcopy | dept   | table| enterprisedb| 8192 bytes |
edbcopy | emp    | table| enterprisedb| 8192 bytes |
edbcopy | jobhist| table| enterprisedb| 8192 bytes |
```

(3 rows)

```sql
edb# 
```

```
List of relations
Schema | Name    | Type | Owner
-------+---------+------+-------
edbcopy | salesemp| view | enterprisedb
```

(1 row)

```sql
edb# 
```

```
List of relations
Schema | Name        | Type | Owner | Table
-------+-------------+------+-------+-------
edbcopy | dept_dname_uq| index| enterprisedb | dept
edbcopy | dept_pk      | index| enterprisedb | dept
edbcopy | emp_pk       | index| enterprisedb | emp
edbcopy | jobhist_pk   | index| enterprisedb | jobhist
```

(4 rows)

```sql
edb# 
```

```
List of relations
Schema | Name    | Type | Owner
-------+---------+------+-------
edbcopy | next_empno | sequence | enterprisedb
```

(1 row)
Performing a local copy of a schema as a batch job

The `localcopy_schema_nb` function copies a schema and its database objects in a local database specified in the `source_fdw` foreign server from the source schema to the specified target schema in the same database. The copy occurs in a non-blocking manner as a job submitted to pgAgent.

```
localcopy_schema_nb(
  <source_fdw> TEXT,
  <source> TEXT,
  <target> TEXT,
  <log_filename> TEXT
  [, <on_tblspace> BOOLEAN
  [, <verbose_on> BOOLEAN
  [, <copy_acls> BOOLEAN
  [, <worker_count> INTEGER ]]])
```

The function returns an INTEGER value job ID for the job submitted to pgAgent. If the function fails, then null is returned.

The `source_fdw`, `source`, `target`, and `log_filename` parameters are required. All other parameters are optional.

After the pgAgent job completes, remove it with the `remove_log_file_and_job` function.

Parameters

- **source_fdw**
  Name of the foreign server managed by the `postgres_fdw` foreign data wrapper from which to clone database objects.

- **source**
  Name of the schema from which to clone database objects.

- **target**
  Name of the schema into which to clone database objects from the source schema.

- **log_filename**
  Name of the log file in which to record information from the function. The log file is created under the directory specified by the `log_directory` configuration parameter in the
### postgresql.conf file

**on_tblspace**

Boolean value to specify whether to create database objects in their tablespaces. If `false`, then the **TABLESPACE** clause isn't included in the applicable `CREATE` DDL statement when added to the target schema. If `true`, then the **TABLESPACE** clause is included in the `CREATE` DDL statement when added to the target schema. The default value is `false`.

**verbose_on**

Boolean value to specify whether to print the DDLs in `log_filename` when creating objects in the target schema. If `false`, then DDLs aren't printed. If `true`, then DDLs are printed. The default value is `false`.

**copy_acls**

Boolean value to specify whether to include the access control list (ACL) while creating objects in the target schema. The access control list is the set of **GRANT** privilege statements. If `false`, then the access control list isn't included for the target schema. If `true`, then the access control list is included for the target schema. The default value is `false`.

**worker_count**

Number of background workers to perform the clone in parallel. The default value is `1`.

### Example

The same cloning operation is performed as the example in `localcopyschema` but using the non-blocking function `localcopyschema_nb`.

You can use this command to see whether `pgAgent` is running on the appropriate local database:

```
[root@localhost -]# ps -ef | grep pgagent
root       4518      1  0 11:35 pts/1    00:00:00 pgagent -s /tmp/pgagent_edb_log
hostaddr=127.0.0.1 port=5444 dbname=edb user=enterprisedb password=password
root       4525   4399  0 11:35 pts/1    00:00:00 grep --color=auto pgagent
```

If `pgAgent` isn't running, start it by executing the `pgagent` option. The `pgagent` program file is located in the `bin` subdirectory of the EDB Postgres Advanced Server installation directory.

```
[root@localhost bin]# ./pgagent -l 2 -s /tmp/pgagent_edb_log hostaddr=127.0.0.1 port=5444
dbname=edb user=enterprisedb password=password
```

**Note**

The `pgagent -l 2` option starts `pgAgent` in **DEBUG** mode, which logs continuous debugging information into the log file specified with the `-s` option. Use a lower value for the `-l` option, or omit it entirely to record less information.

The `localcopyschema_nb` function returns the job ID shown as `4` in the example.

```
edb=# SELECT edb_util.localcopyschema_nb ('local_server','edb','edbcopy','clone_edb_edbcopy');
localcopyschema_nb
-------------------------
  4
(1 row)
```

The following displays the job status:

```
edb=# SELECT edb_util.process_status_from_log('clone_edb_edbcopy');
process_status_from_log
----------------------------------------------------------
(FINISH,"29-JUN-17 11:39:11.620093 -04:00",4618,INFO,"STAGE: FINAL","successfully cloned schema")
(1 row)
```

The following removes the `pgAgent` job:

```
edb=# SELECT edb_util.remove_log_file_and_job (4);
remove_log_file_and_job
------------------------
  t
(1 row)
```
8.4.4 Copying database objects from a remote source

There are two functions you can use with EDB Clone Schema to perform a remote copy of a schema and its database objects:

- `remotecopyschema` — This function copies a schema and its database objects from a source database to a different target database. Use this function when the source schema and the copy will reside in separate databases. The separate databases can reside in the same EDB Postgres Advanced Server database clusters or in different ones. See `remotecopyschema` for more information.
- `remotecopyschema_nb` — This function performs the same purpose as `remotecopyschema`, but as a background job, which frees up the terminal from which the function was initiated. This function is a non-blocking function. See `remotecopyschema_nb` for more information.

### Copying a remote schema

The `remotecopyschema` function copies a schema and its database objects from a source schema in the remote source database specified in the `source_fdw` foreign server to a target schema in the local target database specified in the `target_fdw` foreign server:

```plaintext
remotecopyschema(
  <source_fdw> TEXT,
  <target_fdw> TEXT,
  <source_schema> TEXT,
  <target_schema> TEXT,
  <log_filename> TEXT
[], <on_tblspace> BOOLEAN
[], <verbose_on> BOOLEAN
[], <copy_acls> BOOLEAN
[], <worker_count> INTEGER ))
```

The function returns a Boolean value. If the function succeeds, then `true` is returned. If the function fails, then `false` is returned.

The `source_fdw`, `target_fdw`, `source_schema`, `target_schema`, and `log_filename` are required parameters. All other parameters are optional.

### Parameters

**source_fdw**

Name of the foreign server managed by the `postgres_fdw` foreign data wrapper from which to clone database objects.

**target_fdw**

Name of the foreign server managed by the `postgres_fdw` foreign data wrapper to which to clone database objects.

**source_schema**

Name of the schema from which to clone database objects.

**target_schema**

Name of the schema into which to clone database objects from the source schema.

**log_filename**

Name of the log file in which information from the function is recorded. The log file is created under the directory specified by the `log_directory` configuration parameter in the `postgresql.conf` file.

**on_tblspace**

Boolean value to specify whether to create database objects in their tablespaces. If `false`, then the `TABLESPACE` clause isn’t included in the applicable `CREATE` DDL statement when added to the target schema. If `true`, then the `TABLESPACE` clause is included in the `CREATE` DDL statement when added to the target schema. The default value is `false`.

**verbose_on**

Boolean value to specify whether to print the DDLs in `log_filename` when creating objects in the target schema. If `false`, then DDLs aren’t printed. If `true`, then DDLs are printed. The default value is `false`.

**copy_acls**

Boolean value to specify whether to copy the acls from the source schema to the target schema. The default value is `false`.

**worker_count**

Integer value specifying the number of workers to use when performing the remote copy. The default value is `1`.

Note

If you specify `true` and a database object has a `TABLESPACE` clause, the tablespace must exist in the target database cluster. Otherwise, the cloning function fails.
**copy_acls**

Boolean value to specify whether to include the access control list (ACL) while creating objects in the target schema. The access control list is the set of \texttt{GRANT} privilege statements. If \texttt{false}, then the access control list isn't included for the target schema. If \texttt{true}, then the access control list is included for the target schema. The default value is \texttt{false}.

**Note**

If you specify \texttt{true}, a role with \texttt{GRANT} privilege must exist in the target database cluster. Otherwise, the cloning function fails.

**worker_count**

Number of background workers to perform the clone in parallel. The default value is 1.

**Example**

This example shows cloning schema \texttt{srcschema} in database \texttt{srcdb} (as defined by \texttt{src_server}) to target schema \texttt{tgtschema} in database \texttt{tgtdb} (as defined by \texttt{tgt_server}).

The source server environment:

- Host on which the source database server is running: 192.168.2.28
- Port of the source database server: 5444
- Database source of the clone: \texttt{srcdb}
- Foreign server (\texttt{src_server}) and user mapping with the information of the preceding bullet points
- Source schema: \texttt{srcschema}

The target server environment:

- Host on which the target database server is running: localhost
- Port of the target database server: 5444
- Database target of the clone: \texttt{tgtdb}
- Foreign server (\texttt{tgt_server}) and user mapping with the information of the preceding bullet points
- Target schema: \texttt{tgtschema}
- Database superuser to invoke \texttt{remotecopyschema}: enterprisedb

Before invoking the function, the connection database user \texttt{enterprisedb} connects to database \texttt{tgtdb}. A \texttt{worker_count} of 4 is specified for this function.

```sql
SELECT edb_util.remotecopyschema('src_server', 'tgt_server', 'srcschema', 'tgtschema', 'clone_rmt_src_tgt', worker_count => 4);
```

This example displays the status from the log file during various points in the cloning process:

```sql
SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
```

```sql
-----------------------------------------------------------------------------------------------------
| (RUNNING, "28-JUN-17 13:18:05.299953 -04:00", 4021, INFO, "STAGE: DATA-COPY", "[0][0] successfully copied data in [tgtschema.pgbench_tellers]") |
| (1 row) |
```

```sql
SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
```

```sql
-----------------------------------------------------------------------------------------------------
| (RUNNING, "28-JUN-17 13:18:06.634364 -04:00", 4022, INFO, "STAGE: DATA-COPY", "[0][1] successfully copied data in [tgtschema.pgbench_history]") |
| (1 row) |
```

```sql
SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
```

```sql
-----------------------------------------------------------------------------------------------------
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
**Results**

The following shows the cloned tables:

```sql
tgtdb=# \dt+
```

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgtschema</td>
<td>pgbench_accounts</td>
<td>table</td>
<td>enterprisedb</td>
<td>256 MB</td>
<td></td>
</tr>
<tr>
<td>tgtschema</td>
<td>pgbench_branches</td>
<td>table</td>
<td>enterprisedb</td>
<td>8192 bytes</td>
<td></td>
</tr>
<tr>
<td>tgtschema</td>
<td>pgbench_history</td>
<td>table</td>
<td>enterprisedb</td>
<td>25 MB</td>
<td></td>
</tr>
<tr>
<td>tgtschema</td>
<td>pgbench_tellers</td>
<td>table</td>
<td>enterprisedb</td>
<td>16 kB</td>
<td></td>
</tr>
</tbody>
</table>

(4 rows)

When the `remotecopyschema` function was invoked, four background workers were specified.

The following portion of the log file `clone_rmt_src_tgt` shows the status of the parallel data copying operation using four background workers:

<table>
<thead>
<tr>
<th>Log Message</th>
</tr>
</thead>
</table>

The `DATA-COPY` log message includes two square-bracket numbers, for example, `[0][3]`. The first number is the job index. The second number is the worker index. The worker index values range from 0 to 3 for the four background workers.

In case two clone schema jobs are running in parallel, the first log file has 0 as the job index, and the second has 1 as the job index.

---

**Copying a remote schema using a batch job**

The `remotecopyschema_nb` function copies a schema and its database objects from a source schema in the remote source database specified in the `source_fdw` foreign server to a target schema in the local target database specified in the `target_fdw` foreign server. Copying occurs in a non-blocking manner as a job submitted to pgAgent.
The function returns an INTEGER value job ID for the job submitted to pgAgent. If the function fails, then null is returned.

The source_fdw, target_fdw, source, target, and log_filename parameters are required. All other parameters are optional.

After the pgAgent job is complete, remove it with the remove_log_file_and_job function.

Parameters

source_fdw

Name of the foreign server managed by the postgres_fdw foreign data wrapper from which to clone database objects.

target_fdw

Name of the foreign server managed by the postgres_fdw foreign data wrapper to which to clone database objects.

source

Name of the schema from which to clone database objects.

target

Name of the schema into which to clone database objects from the source schema.

log_filename

Name of the log file in which to record information from the function. The log file is created under the directory specified by the log_directory configuration parameter in the postgresql.conf file.

on_tblspace

Boolean value to specify whether to create database objects in their tablespaces. If false, then the TABLESPACE clause isn't included in the applicable CREATE DDL statement when added to the target schema. If true, then the TABLESPACE clause is included in the CREATE DDL statement when added to the target schema. The default value is false.

Note

If you specify true is specified and a database object has a TABLESPACE clause, that tablespace must exist in the target database cluster. Otherwise, the cloning function fails.

verbose_on

Boolean value to specify whether to print the DDLs in log_filename when creating objects in the target schema. If false, then DDLs aren't printed. If true, then DDLs are printed. The default value is false.

copy_acls

Boolean value to specify whether to include the access control list (ACL) while creating objects in the target schema. The access control list is the set of GRANT privilege statements. If false, then the access control list isn't included for the target schema. If true, then the access control list is included for the target schema. The default value is false.

Note

If you specify true, a role with GRANT privilege must exist in the target database cluster. Otherwise the cloning function fails.

worker_count

Number of background workers to perform the clone in parallel. The default value is 1.

Example
The same cloning operation is performed as the example in `remotecopyschema` but using the non-blocking function `remotecopyschema_nb`.

This command starts pgAgent on the target database `tgtdb`. The `pgagent` program file is located in the `bin` subdirectory of the EDB Postgres Advanced Server installation directory.

```
[root@localhost bin]# ./pgagent -l 1 -s /tmp/pgagent_tgtdb_log hostaddr=127.0.0.1 port=5444 user=enterprisedb dbname=tgtdb password=password
```

**Results**

The `remotecopyschema_nb` function returns the job ID shown as `2` in the example:

```
tgtdb=# SELECT edb_util.remotecopyschema_nb ('src_server', 'tgt_server', 'srcschema', 'tgtschema', 'clone_rmt Src_tgt', worker_count => 4);
```

```
remotecopyschema_nb
---------------------
  2                   (1 row)
```

The following shows the completed status of the job:

```
tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt Src_tgt');
```

```
process_status_from_log
--------------------------------------------------------------------------------------------------------------------
(FINISH,"29-JUN-17 current:16:00.100284 -04:00",3849,INFO,"STAGE: FINAL","successfully clone schema into tgtschema")
(1 row)
```

The following command removes the log file and the pgAgent job:

```
tgtdb=# SELECT edb_util.remove_log_file_and_job ('clone_rmt Src_tgt', 2);
```

```
remove_log_file_and_job
-------------------------
  t                      (1 row)
```

### 8.4.5 Checking the status of the cloning process

The `process_status_from_log` function provides the status of a cloning function from its log file:

```
process_status_from_log
( <log_file> TEXT )
```

The function returns the following fields from the log file:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>Displays either STARTING, RUNNING, FINISH, or FAILED.</td>
</tr>
<tr>
<td>execution_time</td>
<td>When the command was executed. Displayed in timestamp format.</td>
</tr>
<tr>
<td>pid</td>
<td>Session process ID in which clone schema is getting called.</td>
</tr>
<tr>
<td>level</td>
<td>Displays either INFO, ERROR, or SUCCESSFUL.</td>
</tr>
<tr>
<td>stage</td>
<td>Displays either STARTUP, INITIAL, DDL-COLLECTION, PRE-DATA, DATA-COPY, POST-DATA, or FINAL.</td>
</tr>
<tr>
<td>message</td>
<td>Information respective to each command or failure.</td>
</tr>
</tbody>
</table>

**Parameters**

| log_file         | Name of the log file recording the cloning of a schema as specified when the cloning function was invoked. |
Example

The following shows the use of the `process_status_from_log` function:

```
edb# SELECT edb_util.process_status_from_log('clone_edb_edbcopy');
```

```
process_status_from_log
-----------------------------------------------
(FINISH,"26-JUN-17 11:57:03.214458 -04:00",3691,INFO,"STAGE: FINAL","successfully cloned schema")
(1 row)
```

8.4.6 Performing cleanup tasks

The `remove_log_file_and_job` function performs cleanup tasks by removing the log files created by the schema cloning functions and the jobs created by the non-blocking functions.

```
remove_log_file_and_job
(
  | <log_file> TEXT
  | <job_id> INTEGER
)
```

You can specify values for either or both of the two parameters when invoking the `remove_log_file_and_job` function:

- If you specify only `log_file`, then the function removes only the log file.
- If you specify only `job_id`, then the function removes only the job.
- If you specify both, then the function removes only the log file and the job.

Parameters

- **log_file**
  Name of the log file to remove.

- **job_id**
  Job ID of the job to remove.

Example

This example removes only the log file, given the log file name:

```
edb# SELECT edb_util.remove_log_file_and_job ('clone_edb_edbcopy');
```

```
remove_log_file_and_job
------------------------
t (1 row)
```

This example removes only the job, given the job ID:

```
edb# SELECT edb_util.remove_log_file_and_job (3);
```

```
remove_log_file_and_job
------------------------
t (1 row)
```

This example removes the log file and the job, given both values:
9 EDB Postgres Advanced Server security features

EDB Postgres Advanced Server extends Postgres security with features designed to limit unauthorized access.

9.1 Transparent Data Encryption overview

Transparent Data Encryption (TDE) is an optional encryption method that protects data in the database by encrypting the underlying files.

TDE is transparent to authorized users of the database, as no change is required in the applications or existing access policies. It’s supported by version 15 of EDB Postgres Advanced Server and EDB Postgres Extended Server with high availability.

TDE hardens your organization’s data security with minimum performance overhead and doesn’t require additional storage. TDE also enables developers to secure their data using secure encryption algorithms without changing their applications. The feature securely stores individual tablespaces and logs encrypted with their own encryption keys, ensuring security of user data while preserving ease of management for database administrators.

For information about how to enable transparent data encryption and work with the encrypted files, see the Transparent Data Encryption documentation.

9.2 Using EDB audit logging

EDB Postgres Advanced Server allows database and security administrators, auditors, and operators to track and analyze database activities using EDB audit logging. EDB audit logging generates audit log files, which can be configured to record information such as:

- When a role establishes a connection to an EDB Postgres Advanced Server database
- The database objects a role creates, modifies, or deletes when connected to EDB Postgres Advanced Server
- When any failed authentication attempts occur

The parameters specified in the configuration files `postgresql.conf` or `postgresql.auto.conf` control the information included in the audit logs.

9.2.1 Selecting SQL statements to audit

The `edb_audit_statement` permits inclusion of one or more comma-separated values to control the SQL statements to audit. The following is the general format:

```
edb_audit_statement = 'value_1[, value_2]...'
```

The comma-separated values can include or omit space characters following the comma. You can specify the values in any combination of lowercase or uppercase letters.

Overview of the parameters

The basic parameter values are the following:

- **all** – Audit and log every statement including any error messages on statements.
- **none** – Disable all auditing and logging. A value of **none** overrides any other value included in the list.
- **ddl** – Audit all data definition language (DDL) statements (CREATE, ALTER, and DROP) as well as GRANT and REVOKE data control language (DCL) statements.
- **dml** – Audit all data manipulation language (DML) statements (INSERT, UPDATE, DELETE, and TRUNCATE).
- **select** – Audit SELECT statements.
- **set** – Audit SET statements.
- **rollback** – Audit ROLLBACK statements.
- **error** – Log all error messages that occur. Otherwise, no messages about errors that occur on SQL statements related to any of the other preceding parameter values are logged except when **all** is used.
- `{ select | update | delete | insert }@groupname` – Selectively audit objects for specific DML statements (SELECT, UPDATE, DELETE, and INSERT) including (e) and excluding (E) groups on a given table. For more information, see `Object auditing`.

DDL and DCL statements describes additional parameter values for selecting particular DDL or DCL statements for auditing. DML statements describes additional parameter values for selecting...
particular DML statements for auditing.

If an unsupported value is included in the `edb_audit_statement` parameter, then an error occurs when applying the setting to the database server. See the database server log file for the error such as in the following example in which `create materialized vw` results in the error. The correct value is `create materialized view`.

```
2017-07-16 11:20:39 EDT LOG: invalid value for parameter
"edb_audit_statement": "create materialized vw, create sequence, grant"
2017-07-16 11:20:39 EDT FATAL: configuration file "/var/lib/edb/as14/data/postgresql.conf" contains errors
```

### DDL and DCL statements

When auditing DDL and DCL statements, the following general rules apply to the values that can be included in the `edb_audit_statement` parameter:

- If the `edb_audit_statement` parameter includes `ddl` or `all`, then all DDL statements are audited. In addition, the DCL statements `GRANT` and `REVOKE` are audited.
- If the `edb_audit_statement` is set to `none`, then no DDL nor DCL statements are audited.
- You can choose specific types of DDL and DCL statements for auditing by including a combination of values in the `edb_audit_statement` parameter.

Use the following syntax to specify an `edb_audit_statement` parameter value for DDL statements:

```
{ create | alter | drop } [ <object_type> ]
```

`object_type` is any of the following:

- `ACCESS METHOD`
- `AGGREGATE`
- `CAST`
- `COLLATION`
- `CONVERSION`
- `DATABASE`
- `EVENT TRIGGER`
- `EXTENSION`
- `FOREIGN TABLE`
- `FUNCTION`
- `INDEX`
- `LANGUAGE`
- `LARGE OBJECT`
- `MATERIALIZED VIEW`
- `OPERATOR`
- `OPERATOR CLASS`
- `OPERATOR FAMILY`
- `POLICY`
- `PUBLICATION`
- `ROLE`
- `RULE`
Descriptions of object types as used in SQL commands can be found in the PostgreSQL core documentation.

If `object_type` is omitted from the parameter value, then all of the specified command statements (create, alter, or drop) are audited.

Use the following syntax to specify an `edb_audit_statement` parameter value for DCL statements:

```
{ grant | revoke }
```

Examples

In this example, `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
logging_collector = 'on'
edb_audit_connect = 'all'
edb_audit_statement = 'create, alter, error'
```

Thus, only SQL statements invoked by the CREATE and ALTER commands are audited. Error messages are also included in the audit log.

The following is the database session that occurs:

```
$ psql edb
enterprisedb
Password
Password for user enterprisedb:
psql.bin (14.8.0)
Type "help" for help.
edb=# SHOW edb_audit_connect;
edb_audit_connect
-------------------
all
(1 row)
edb=# SHOW edb_audit_statement;
edb_audit_statement
--------------------
create, alter, error
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The resulting audit log file contains the following. (Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)

```


```

The resulting audit log file contains the following. (Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)

```


```
The CREATE and ALTER statements for the adminuser role and auditdb database are audited. Because error is included in the edb_audit_statement parameter, the error for the ALTER ROLE adminuser statement is also logged.

Similarly, the CREATE statements for schema edb and tables department and dept are audited.

There's no edb_audit_statement setting that results in auditing successfully processed DROP statements such as ddl, all, or drop. Thus the DROP TABLE department statement isn't in the audit log.

In this example, edb_audit_connect and edb_audit_statement are set with the following non-default values:

```
logging_collector = 'on'
ebd_audit_connect = 'all'
ebd_audit_statement = 'create view,create materialized view,create sequence,grant'
```

Thus, only SQL statements invoked by the CREATE VIEW, CREATE MATERIALIZED VIEW, CREATE SEQUENCE and GRANT commands are audited.

The following is the database session that occurs:

```
$ psql auditdb adminuser
Password for user adminuser:
psql.bin (14.8.8)
Type "help" for help.

auditdb=# SHOW edb_audit_connect;

edb_audit_connect
-------------------
all
 (1 row)

auditdb=# SHOW edb_audit_statement;

edb_audit_statement
---------------------
create view,create materialized view,create sequence,grant
 (1 row)

auditdb=# SET search_path TO edb;
SET
auditdb=# CREATE TABLE emp ( auditdb(#   empno NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY, auditdb(#   ename VARCHAR2(10),
```
The resulting audit log file contains the following. (Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)

```
[local],5ecb8351.9f6a,1,"authentication",2020-05-25 14:05:29 IST,4/28,0,AUDIT,00000,"connection authorized:user=adminuser database=auditdb",,"client backend","connect"

2020-05-25 14:12:06.318 IST,"adminuser","auditdb",40810,"
[local],5ecb8351.9f6a,2,"idle",2020-05-25 14:05:29 IST,4/34,0,AUDIT,00000,"statement: CREATE VIEW salesemp AS SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';",,"psql","client backend","CREATE VIEW","create"

[local],5ecb8351.9f6a,3,"idle",2020-05-25 14:05:29 IST,4/36,0,AUDIT,00000,"statement: CREATE MATERIALIZED VIEW managers AS SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'MANAGER';",,"psql","client backend","CREATE MATERIALIZED VIEW","create"

[local],5ecb8351.9f6a,4,"idle",2020-05-25 14:05:29 IST,4/38,0,AUDIT,00000,"statement: CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;",,"psql","client backend","CREATE SEQUENCE","create"

[local],5ecb8351.9f6a,6,"idle",2020-05-25 14:05:29 IST,4/39,0,AUDIT,00000,"statement: GRANT ALL ON dept TO PUBLIC;",,"psql","client backend","GRANT","grant"
```

The CREATE VIEW and CREATE MATERIALIZED VIEW statements are audited. The prior CREATE TABLE emp statement isn't audited because none of the values `create`, `create table`, `ddl`, nor all are included in the `edb_audit_statement` parameter.

The CREATE SEQUENCE and GRANT statements are audited because those values are included in the `edb_audit_statement` parameter.

In this example, `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
logging_collector = 'on'
edb_audit_connect = 'all'
edb_audit_statement = 'none'
```

The session for users who connect as ADMIN or SYSDBA can be fully audited. An admin user is connected to a database `auditdb` as `ADMINUSER`. The following ALTER USER command specifies to audit `ADMINUSER`.

The `CREATE VIEW` and `CREATE MATERIALIZED VIEW` statements are audited. The prior `CREATE TABLE emp` statement isn't audited because none of the values `create`, `create table`, `ddl`, nor all are included in the `edb_audit_statement` parameter.

The `CREATE SEQUENCE` and `GRANT` statements are audited because those values are included in the `edb_audit_statement` parameter.

In this example, `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
logging_collector = 'on'
edb_audit_connect = 'all'
edb_audit_statement = 'none'
```

The session for users who connect as ADMIN or SYSDBA can be fully audited. An admin user is connected to a database `auditdb` as `ADMINUSER`. The following ALTER USER command specifies to audit `ADMINUSER`.

The resulting audit log file contains the following. (Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)

```
[local],5ecb8351.9f6a,1,"authentication",2020-05-25 14:05:29 IST,4/28,0,AUDIT,00000,"connection authorized:user=adminuser database=auditdb",,"client backend","connect"

2020-05-25 14:12:06.318 IST,"adminuser","auditdb",40810,"
[local],5ecb8351.9f6a,2,"idle",2020-05-25 14:05:29 IST,4/34,0,AUDIT,00000,"statement: CREATE VIEW salesemp AS SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';",,"psql","client backend","CREATE VIEW","create"

[local],5ecb8351.9f6a,3,"idle",2020-05-25 14:05:29 IST,4/36,0,AUDIT,00000,"statement: CREATE MATERIALIZED VIEW managers AS SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'MANAGER';",,"psql","client backend","CREATE MATERIALIZED VIEW","create"

[local],5ecb8351.9f6a,4,"idle",2020-05-25 14:05:29 IST,4/38,0,AUDIT,00000,"statement: CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;",,"psql","client backend","CREATE SEQUENCE","create"

[local],5ecb8351.9f6a,6,"idle",2020-05-25 14:05:29 IST,4/39,0,AUDIT,00000,"statement: GRANT ALL ON dept TO PUBLIC;",,"psql","client backend","GRANT","grant"
```

The `CREATE VIEW` and `CREATE MATERIALIZED VIEW` statements are audited. The prior `CREATE TABLE emp` statement isn't audited because none of the values `create`, `create table`, `ddl`, nor all are included in the `edb_audit_statement` parameter.

The `CREATE SEQUENCE` and `GRANT` statements are audited because those values are included in the `edb_audit_statement` parameter.

In this example, `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
logging_collector = 'on'
edb_audit_connect = 'all'
edb_audit_statement = 'none'
```

The session for users who connect as ADMIN or SYSDBA can be fully audited. An admin user is connected to a database `auditdb` as `ADMINUSER`. The following ALTER USER command specifies to audit `ADMINUSER`.
Setting the `edb_audit_statement` parameter to `all` allows auditing of all of the SQL statements for an admin user.

The following is the database session that occurs:

```
$ psql auditdb adminuser
Password for user adminuser:
psql.bin (14.0.0)
Type "help" for help.

auditdb=# SHOW edb_audit_connect;

edb_audit_connect
-------------------
all
(1 row)

auditdb=# SHOW edb_audit_statement;

edb_audit_statement
---------------------
all
(1 row)

auditdb=# SET search_path TO edb;
SET

auditdb-> CREATE TABLE dept
auditdb-> (deptno NUMBER(2),
auditdb->    dname VARCHAR2(14),
auditdb->    loc VARCHAR2(13)));
CREATE TABLE

auditdb-> INSERT INTO dept VALUES (10, 'ACCOUNTING', 'NEW YORK');
INSERT 0 1

auditdb-> INSERT INTO dept VALUES (20, 'RESEARCH', 'DALLAS');
INSERT 0 1

auditdb-> INSERT INTO dept VALUES (30, 'SALES', 'CHICAGO');
INSERT 0 1

auditdb-> UPDATE dept SET loc = 'BEDFORD' WHERE deptno = 20;
UPDATE 1

auditdb-> SELECT * FROM dept;

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>BEDFORD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(3 rows)

auditdb-> DELETE FROM emp WHERE deptno = 40;
ERROR:  relation "emp" does not exist
LINE 1: DELETE FROM emp WHERE deptno = 40;

^ auditdb-> DELETE FROM dept WHERE deptno = 10;
DELETE 1

auditdb-> SELECT * FROM dept;

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>BEDFORD</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(2 rows)
```

The resulting audit log file contains the following. (Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)

```
2021-06-23 06:06:59.027 IST,"adminuser","auditdb","60218","[local]",60d3083b.
```
In this example, `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
EDB Postgres Advanced Server (EPAS)
```

The audit session for a user `carol` can be fully blocked by the database administrators using the `ALTER USER` command:

```
ALTER USER carol SET edb_audit_statement = "none";
```

The database administrator can allow a specific user to audit any SQL statements by specifying the `ALTER USER` command and setting the `edb_audit_statement` parameter to any value.

The following is the database session that occurs:

```
$ psql auditdb carol
Password for user carol:
```

---

**logging_collector** = 'on'

**edb_audit_connect** = "all"

**edb_audit_statement** = "all"
The resulting audit log file contains only the connection authentication information. Setting the `edb_audit_statement` parameter to `none` doesn’t allow the auditing of SQL statements for `carol`, so no audit logs are generated.

(Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)

```
```

DML statements

When auditing DML statements, the following general rules apply to the values that can be included in the `edb_audit_statement` parameter:

- If the `edb_audit_statement` parameter includes `dml` or `all`, then all DML statements are audited.
- If the `edb_audit_statement` is set to `none`, then no DML statements are audited.
- You can choose specific types of DML statements for auditing by including a combination of values in the `edb_audit_statement` parameter.

Use the following syntax to specify an `edb_audit_statement` parameter value for SQL INSERT, UPDATE, DELETE, or TRUNCATE statements:

```
{ insert | update | delete | truncate }
```

Example

In this example, `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
logging_collector = 'on'
edb_audit_connect = 'all'
edb_audit_statement = 'UPDATE, DELETE, error'
```

Thus, only SQL statements invoked by the UPDATE and DELETE commands are audited. All errors are also included in the audit log, including errors not related to the UPDATE and DELETE commands.

The following is the database session that occurs:

```
$ psql auditdb adminuser
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Password for user adminuser:
psql.bin
(14.0.0)
Type "help" for help.

auditdb=# SHOW edb_audit_connect;

edb_audit_connect
-------------------
all
(1 row)

auditdb=# SHOW edb_audit_statement;

edb_audit_statement
-----------------------
UPDATE, DELETE, error
(1 row)

deptno |   dname    |   loc
--------+------------+----------
10 | ACCOUNTING | NEW YORK
20 | RESEARCH   | DALLAS
30 | SALES      | CHICAGO
40 | OPERATIONS | BEDFORD
(4 rows)

The resulting audit log file contains the following. (Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)
9.2.2 Enabling audit logging

You can configure EDB Postgres Advanced Server to log all connections, disconnections, DDL statements, DCL statements, DML statements, and any statements resulting in an error.

- Enable auditing by setting the `edb_audit` parameter to `xml` or `csv`.
- Set the file rotation day when the new file is created by setting the parameter `edb_audit_rotation_day` to the desired value.
- To audit all connections, set the parameter `edb_audit_connect` to `all`.
- To audit all disconnections, set the parameter `edb_audit_disconnect` to `all`.
- To audit DDL, DCL, DML and other statements, set the parameter `edb_audit_statement` according to the instructions in Selecting SQL statements to audit.
- To specify the desired location of audit files, set the `edb_audit_directory` parameter.

Setting the `edb_audit_statement` parameter in the configuration file affects the entire database cluster.

You can further refine the type of statements that are audited. The type is controlled by the `edb_audit_statement` parameter according to the database in use as well as the database role running the session:

- You can set the `edb_audit_statement` parameter as an attribute of a specified database with the `ALTER DATABASE dbname SET edb_audit_statement` command. This setting overrides the `edb_audit_statement` parameter in the configuration file for statements executed when connected to database `dbname`.
- You can set the `edb_audit_statement` parameter as an attribute of a specified role with the `ALTER ROLE rolename SET edb_audit_statement` command. This setting overrides the `edb_audit_statement` parameter in the configuration file as well as any setting assigned to the database by the `ALTER DATABASE` command when the specified role is running the current session.
- You can set the `edb_audit_statement` parameter as an attribute of a specified role when using a specified database with the `ALTER ROLE rolename IN DATABASE dbname SET edb_audit_statement` command. This setting overrides the `edb_audit_statement` parameter in the configuration file. It also overrides any setting assigned to the database by the `ALTER DATABASE` command and as any setting assigned to the role with the `ALTER ROLE` command without the `IN DATABASE` clause.

Examples

The following are examples of this technique.

The database cluster is established with `edb_audit_statement` set to `all` as shown in its `postgresql.conf` file:

```sql
logging_collector = 'on'
edb_audit_statement = 'all'

# Statement type to be audited:
# none, dml, insert, update, delete, truncate,
# select, error, rollback, ddl, create,
# alter, grant, revoke, set,
# (select | update | delete | insert)@groupname
```

A database and role are established with the following settings for the `edb_audit_statement` parameter:

- `Database auditdb` with `ddl, insert, update, and delete`
- `Role admin` with `select, truncate, and set`
- `Role admin in database auditdb` with `create table, insert, and update`
The following shows creating and altering the database and role:

```bash
$ psql edb enterprisedb
Password for user enterprisedb:
psql.bin (14.0.8)
Type "help" for help.
edb=# SHOW edb_audit_statement;

edb_audit_statement
---------------------
all
(1 row)
```

The following shows the changes made and the resulting audit log file for three cases.

Case 1: Changes made in database `auditdb` by role `enterprisedb`

Only `ddl`, `insert`, `update`, and `delete` statements are audited:

```bash
$ psql auditdb enterprisedb
Password for user enterprisedb:
psql.bin (14.0.8)
Type "help" for help.
auditdb=# SHOW edb_audit_statement;

edb_audit_statement
---------------------
ddl, insert, update, delete
(1 row)
```

```sql
CREATE TABLE audit_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);

INSERT INTO audit_tbl VALUES (1, 'Row A');

UPDATE audit_tbl SET f2 = 'Row A' WHERE f1 = 1;
```

`f1 | f2`
```
1 | Row A
```

```sql
TRUNCATE audit_tbl;
```

The following audit log file shows entries only for the `CREATE TABLE`, `INSERT INTO audit_tbl`, and `UPDATE audit_tbl` statements. The `SELECT * FROM audit_tbl` and `TRUNCATE audit_tbl` statements weren't audited. (Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)

```
audit_tbl(f1 INTEGER PRIMARY KEY, f2 TEXT);",,,,,,,,,"psql","client backend","CREATE TABLE","","create"
Case 2: Changes made in database edb by role admin

Only select, truncate, and set statements are audited:

```sql
$ psql edb
admin
Password for user admin:
psql.bin
(14.0.0)
Type "help" for help.
edb=# SHOW edb_audit_statement;

edb audit_statement
----------
select, truncate, set
(1 row)

edb=# SET default_with_rowids TO TRUE;
SET
edb=# CREATE TABLE edb_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);
CREATE TABLE
edb=# INSERT INTO edb_tbl VALUES (1, 'Row 1');
INSERT INTO
edb=# SELECT * FROM edb_tbl;
GRADE | f2
-------+-------
1 | Row 1
(1 row)

edb=# TRUNCATE edb_tbl;
TRUNCATE TABLE
```

Continuation of the audit log file now appears as follows. The last two entries representing the second case show only the SET default_with_rowids TO TRUE, SELECT *, FROM edb_tbl, and TRUNCATE edb_tbl statements. The CREATE TABLE edb_tbl and INSERT INTO edb_tbl statements weren't audited.
Only create table, insert, and update statements are audited:

```
$ psql auditdb
Password for user admin:
psql.bin (14.0.8)
Type "help" for help.
auditdb=# SHOW edb_audit_statement;
```

```
-- edb_audit_statement
-----------------------------
create table, insert, update
(1 row)
```

```
auditdb=# CREATE TABLE audit_tbl_2 (f1 INTEGER PRIMARY KEY, f2 TEXT);
```

Continuation of the audit log file now appears as follows. The next-to-last two entries representing the third case show only CREATE TABLE audit_tbl_2 and INSERT INTO audit_tbl_2 statements. The SELECT * FROM audit_tbl_2 and TRUNCATE audit_tbl_2 statements weren't audited.

```
audit_tbl(f1 INTEGER PRIMARY KEY, f2 TEXT);","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","
```

```
audit_tbl VALUES (1, 'Row 1');","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","
```

```
audit_tbl_2 (f1 INTEGER PRIMARY KEY, f2 TEXT);","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","","
```

```
audit_tbl_2 VALUES (1, 'Row 1');","","","","","","","","","","","","
```

9.2.3 Using error codes to filter audit logs

EDB Postgres Advanced Server includes an extension that you can use to exclude from the EDB Postgres Advanced Server log files entries that include a user-specified error code. To filter audit log entries, you must first enable the extension by modifying the `postgresql.conf` file. Add the following value to those specified in the `shared_preload_libraries` parameter:
Then, use the `edb_filter_log.errcodes` parameter to specify any error codes you want to omit from the log files:

```
$libdir/edb_filter_log
```

Where `error_code` specifies one or more error codes that you want to omit from the log file. Provide multiple error codes in a comma-delimited list.

For example, if `edb_filter_log` is enabled, and `edb_filter_log.errcode` is set to `"23505,23502,22012"`, any log entries that return one of the following SQLSTATE errors are omitted from the log file:

- 23505 (for violating a unique constraint)
- 23502 (for violating a not-null constraint)
- 22012 (for dividing by zero)

For a complete list of the error codes supported by EDB Postgres Advanced Server audit log filtering, see the Postgres core documentation.

### 9.2.4 Using command tags to filter audit logs

Each entry in the log file contains a command tag, except for entries that display an error message. A command tag is the SQL command executed for that log entry. The command tag makes it possible to use tools later to scan the log file to find entries related to a SQL command.

This example is in XML form, formatted for easier review. The command tag is displayed as the `command_tag` attribute of the `event` element with values `CREATE ROLE`, `ALTER ROLE`, and `DROP ROLE`.

```xml
<event user="enterprisedb" database="edb" process_id="64234"
remote_host= "[\[local\]]"
   session_id="5ecbc7e6.faea" session_line_num="2"
   process_status="idle"
   virtual_transaction_id="4/30" type="create" command_tag="CREATE ROLE"
   application_name="psql" backend_type="client backend">
   <error_severity>AUDIT</error_severity>
   <message>statement: CREATE ROLE newuser WITH LOGIN;</message>
</event>

<event user="enterprisedb" database="edb" process_id="64234"
remote_host= "[\[local\]]"
   session_id="5ecbc7e6.faea" session_line_num="3"
   process_status="idle"
   virtual_transaction_id="4/31" type="error" sql_state_code="42601"
   application_name="psql" backend_type="client backend">
   <error_severity>ERROR</error_severity>
   <message>unrecognized role option &quot;super&quot;</message>
</event>

<query>ALTER ROLE newuser WITH SUPER USER;</query>
<query_pos>25</query_pos>
```

For a complete list of the error codes supported by EDB Postgres Advanced Server audit log filtering, see the Postgres core documentation.
You can use the GUC parameter `edb_filter_log.redact_password_commands` under the `postgresql.conf` file to redact stored passwords in the log file.

The syntax is:

```
[CREATE] [ALTER] { USER | ROLE | GROUP } identifier { [WITH] [ENCRYPTED] 
  'nonempty_string_literal' | IDENTIFIED BY { 'nonempty_string_literal' 
  | bareword } } [ REPLACE { 'nonempty_string_literal' | bareword } ]
```

To enable password redaction, you must first enable the parameter by modifying the `postgresql.conf` file. Add the following value to those specified in the `shared_preload_libraries` parameter:

```
$libdir/edb_filter_log
```

Then, set `edb_filter_log.redact_password_commands` to `true`:

```
edb_filter_log.redact_password_commands = true
```

After modifying the `postgresql.conf` file, you must restart the server for the changes to take effect.

Examples

When the following statement is logged by `log_statement`, the server redacts the password to `x`. For example, this command is added to the log file:

```
CREATE USER carol with login PASSWORD 'isafepwd';
```
It appears as:

When the following statement is logged by `log_statement`, the server identifies the new password, replace, and redact the password to `x`. For example, this command is added to the log file:

```
ALTER USER carol IDENTIFIED BY 'new_pass' REPLACE 'isafepwd';
```

It appears as:

The statement that includes a password is logged. The server redacts the password text to `x`. When the statement is logged as context for some other message, the server omits the statement from the context.

### 9.2.6 Archiving audit logs

EDB audit log archiving enables database administrators to control the space consumed by the audit log directory and helps manage the audit log files. The Audit Archiver is responsible for the compression, execution, and expiration of log files with `edb_audit_archiver_filename_prefix` present in the audit directory. The `edb_audit_archiver_timeout` parameter triggers the compression or expiration of audit log files at an appropriate time. For more information about audit archiving configuration parameters, see [Audit logging configuration parameters](#).

The audit archiver helps to:

- Prepare a list of log files from the audit log directory for compression.
- Determine the log files for compression based on the parameters `edb_audit_archiver_compress_time_limit` and `edb_audit_archiver_compress_size_limit`.
- Perform compression of the log files by specifying the compression command based on the `edb_audit_archiver_compress_command` parameter.
- Determine the log files to remove based on the `edb_audit_archiver_expire_time_limit` and `edb_audit_archiver_expire_size_limit` parameter.
- Execute the expiration command specified in the `edb_audit_archiver_expire_command` parameter to remove the log files.

### Rotating out older audit log files

To rotate out the older audit log files, you can set the log file rotation day when the new file is created. To do so, set the parameter `edb_audit_rotation_day` to the desired value. The audit log records are overwritten on a first-in, first-out basis if space isn’t available for more audit log records.

### Enabling compression and expiration of log files

To configure EDB Postgres Advanced Server to enable compression and expiration of the log files:

1. Enable audit log archiving by setting the `edb_audit_archiver` parameter to `on` in the `postgresql.conf` file.
2. To enable compression of log files, set the parameter `edb_audit_archiver_compress_size_limit` and `edb_audit_archiver_compress_time_limit` to the values you want.
3. To enable expiration of log files, set the parameter `edb_audit_archiver_expire_size_limit` and `edb_audit_archiver_expire_time_limit` to the values you want.
4. To enable both compression and expiration, set the parameters `edb_audit_archiver_compress_size_limit`, `edb_audit_archiver_compress_time_limit`, `edb_audit_archiver_expire_size_limit`, `edb_audit_archiver_expire_time_limit`, and `edb_audit_archiver_expire_command` to the values you want.

The following is an example of the configuration parameter settings in the `postgresql.conf` file.

```
logging_collector = 'on'
edb audits = 'csv'
edb audit archiver = 'on' # (Change requires restart)
```

### Examples

In this example, `edb_audit_archiver`, `edb_audit_archiver_compress_size_limit`, and `edb_audit_archiver_compress_time_limit` are set to enable compression of the audit log files greater than 10MB.

```
edb audit archiver = 'on'
edb audit archiver compress size limit = 10
```

---

*Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.*
Before compression, the audit log file appears as follows:

```
-rw-------. 1 enterprisedb edb 2097276 Jun 4 11:42 audit-20200811_114237.csv     <== Shows non-compressed audit log files
-rw-------. 1 enterprisedb edb 2097278 Jun 4 11:42 audit-20200811_114243.csv
-rw-------. 1 enterprisedb edb 2097289 Jun 4 11:42 audit-20200811_114245.csv
-rw-------. 1 enterprisedb edb 2097330 Jun 4 11:42 audit-20200811_114246.csv
-rw-------. 1 enterprisedb edb 2097343 Jun 4 11:42 audit-20200811_114248.csv
-rw-------. 1 enterprisedb edb 2097338 Jun 4 11:42 audit-20200811_114249.csv
```

The `edb_audit_archiver` parameter checks the log files, excluding the latest file. It retains at least 10MB of log files in the audit log directory and compresses the remaining log files. The `.gz` specifies the name of an already compressed log file. After compression, the audit log file appears as follows:

```
-rw-------. 1 enterprisedb edb 64119 Jun 4 11:42 audit-20200811_114237.csv.gz     <== Shows compressed audit log files
-rw-------. 1 enterprisedb edb 64323 Jun 4 11:42 audit-20200811_114243.csv.gz
-rw-------. 1 enterprisedb edb 64091 Jun 4 11:42 audit-20200811_114245.csv.gz
-rw-------. 1 enterprisedb edb 64152 Jun 4 11:42 audit-20200811_114246.csv.gz
-rw-------. 1 enterprisedb edb 2097343 Jun 4 11:42 audit-20200811_114248.csv     <== Shows non-compressed audit log files
-rw-------. 1 enterprisedb edb 2097338 Jun 4 11:42 audit-20200811_114249.csv
```

In this example, `edb_audit_archiver`, `edb_audit_archiver_expire_size_limit`, and `edb_audit_archiver_expire_time_limit` are set to enable expiration of the audit log files older than one hour.

```
edb_audit_archiver = 'on'
edb_audit_archiver_expire_size_limit = 0
edb_audit_archiver_expire_time_limit = 3600
```

Before compression, the audit log file appears as follows:

```
-rw-------. 1 enterprisedb edb 2097367 Jun 4 13:23 audit-20200811_132352.csv     <== Shows non-compressed audit log files scheduled for expiration/removal
-rw-------. 1 enterprisedb edb 2097395 Jun 4 13:24 audit-20200811_132353.csv
-rw-------. 1 enterprisedb edb 2097328 Jun 4 14:35 audit-20200811_143438.csv
-rw-------. 1 enterprisedb edb 2097276 Jun 4 14:35 audit-20200811_143502.csv
-rw-------. 1 enterprisedb edb 2097211 Jun 4 14:35 audit-20200811_143503.csv
-rw-------. 1 enterprisedb edb 2097152 Jun 4 14:35 audit-20200811_143504.csv
```

The `edb_audit_archiver` parameter checks the log files, excluding the latest file. It removes the log files older than one hour. After expiration, the audit log file appears as follows:

```
-rw-------. 1 enterprisedb edb 2097328 Jun 4 14:35 audit-20200811_143438.csv
-rw-------. 1 enterprisedb edb 2097276 Jun 4 14:35 audit-20200811_143502.csv
-rw-------. 1 enterprisedb edb 2097211 Jun 4 14:35 audit-20200811_143503.csv
-rw-------. 1 enterprisedb edb 2097152 Jun 4 14:35 audit-20200811_143504.csv
```

In this example, `edb_audit_archiver`, `edb_audit_archiver_compress_size_limit`, `edb_audit_archiver_compress_time_limit`, `edb_audit_archiver_expire_size_limit`, `edb_audit_archiver_expire_time_limit`, and `edb_audit_archiver_expire_command` are set to enable both compression and expiration of the audit log files.

```
edb_audit_archiver = 'on'
edb_audit_archiver_compress_size_limit = 4
edb_audit_archiver_compress_time_limit = 8
edb_audit_archiver_expire_size_limit = 5
edb_audit_archiver_expire_time_limit = 0
edb_audit_archiver_expire_command = 'cp %p /home/edb_audit/backup-log/'
```

Before compression, the audit log file appears as follows:

```
-rw-------. 1 enterprisedb edb 2097522 Jun 4 13:02 audit-20200811_125816.csv     <== Shows non-compressed audit log files ready to be removed/expired
-rw-------. 1 enterprisedb edb 2097510 Jun 4 13:02 audit-20200811_132352.csv
-rw-------. 1 enterprisedb edb 2097336 Jun 4 13:10 audit-20200811_130947.csv
-rw-------. 1 enterprisedb edb 2097276 Jun 4 13:10 audit-20200811_131034.csv
-rw-------. 1 enterprisedb edb 2097246 Jun 4 13:10 audit-20200811_131035.csv
-rw-------. 1 enterprisedb edb 2097289 Jun 4 13:10 audit-20200811_131036.csv
```

The `edb_audit_archiver` parameter checks the log files, excluding the latest file. It retains at least 4MB of log files in the audit log directory and compresses the remaining log files. While checking the log files for expiration, `edb_audit_archiver` retains at least 5MB of log files in the audit log directory and removes the remaining log files. After compression and expiration, the audit log file appears as follows:

```
-rw-------. 1 enterprisedb edb 63854 Jun 4 13:18 audit-20200811_131034.csv.gz
-rw-------. 1 enterprisedb edb 63513 Jun 4 13:18 audit-20200811_131035.csv.gz
-rw-------. 1 enterprisedb edb 63738 Jun 4 13:18 audit-20200811_131036.csv.gz
```
The expiration command is specified as `edb_audit_archiver_expire_command = 'cp %p /home/edb_audit/backup-log/'` in the `postgresql.conf` file. The `edb_audit_archiver` executes this command and copies the log files to a backup log directory before deleting them from the audit log directory. After compression and expiration, the backup log directory appears as follows:

```
-rw-------. 1 enterprisedb edb 2097522 Jun 4 13:02 audit-20200811_125816.csv.gz
-rw-------. 1 enterprisedb edb 2097510 Jun 4 13:02 audit-20200811_130215.csv.gz
-rw-------. 1 enterprisedb edb 2097336 Jun 4 13:18 audit-20200811_130947.csv.gz
```

### 9.2.7 Auditing objects

Object-level auditing allows selective auditing of objects for specific data manipulation language (DML) statements, such as `SELECT`, `UPDATE`, `DELETE`, and `INSERT`, on a given table. Object-level auditing also lets you include or exclude specific groups by specifying `(@)` or `(-)` with the `edb_audit_statement` parameter. For more information about DML statements, see Selecting SQL statements to audit.

Use the following syntax to specify an `edb_audit_statement` parameter value for `SELECT`, `UPDATE`, `DELETE`, or `INSERT` statements:

```
{ select | update | delete | insert }{@ | -}groupname
```

#### Example

In this example, `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
logging_collector = 'on'
edb_audit_connect = 'all'
edb_audit = 'csv'
edb_audit_statement = 'select, insert, update, delete'
```

The SQL statements invoked by the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands are audited.

The following is the database session that occurs:

```
$ psql edb
enterprisedb
Password
Password
for
for user enterprisedb:
psql.bin
(14.0.8)
Type "help" for help.
edb=# SHOW edb_audit_connect;

edb_audit_connect
---------------
all
(1 row)

edb=# SHOW edb_audit_statement;

edb_audit_statement
------------------------
select, insert, update, delete
(1 row)

edb=# CREATE TABLE emp
edb=# (empno NUMBER(4) NOT NULL,
edb=# ename VARCHAR2(10),
edb=# job VARCHAR2(9),
edb=# mgr NUMBER(4),
edb=# hiredate DATE,
edb=# sal NUMBER(7, 2),
edb=# comm NUMBER(7, 2),
edb=# deptno NUMBER(2));
CREATE TABLE
edb=# ALTER TABLE emp SET (edb_audit_group = 'low_security');
ALTER TABLE
edb=# CREATE TABLE dept
edb=# (deptno NUMBER(2),
edb=# dname VARCHAR2(14),
```
CREATE TABLE edb=# CREATE TABLE bonus
  (ename VARCHAR2(10),
   job VARCHAR2(9),
   sal NUMBER,
   comm NUMBER) with (edb_audit_group = 'high_security');

CREATE TABLE edb=# CREATE TABLE sal
  (grade NUMBER,
   losal NUMBER,
   hisal NUMBER) with (edb_audit_group = 'high_security');

SET edb_audit_statement = 'select@low_security@high_security, insert@high_security-low_security, update-low_security@high_security,
delete@low_security-high_security';

SET reloptions ---------------------------------
{edb_audit_group=low_security}
{edb_audit_group=low_security}
{edb_audit_group=high_security}
{edb_audit_group=high_security}
(4 rows)

setting--------------------------------------------------------------------------------------------------------------------------------------------
select@low_security@high_security, insert@high_security-low_security, update-low_security@high_security, delete@low_security-high_security
(1 row)

SELECT reloptions FROM pg_class WHERE relname IN('emp', 'dept', 'bonus', 'sal');

reloptions
-------------------
(edb_audit_group=low_security)
(edb_audit_group=low_security)
(edb_audit_group=high_security)
(edb_audit_group=high_security)

SELECT setting FROM pg_settings WHERE name = 'edb_audit_statement';

setting-------------------------------------------------------------------------------------------------------------------------------
select@low_security@high_security, insert@high_security-low_security, update-low_security@high_security, delete@low_security-high_security

SELECT * FROM emp;

empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+-------+--------
(0 rows)

SELECT * FROM dept;

deptno | dname | loc
--------+-------+-----
(0 rows)

SELECT * FROM bonus;

ename | job | sal | comm
-------+-----+-----+-------
(0 rows)

SELECT * FROM sal;

grade | losal | hisal
-------+-------+-------
(0 rows)

INSERT INTO emp VALUES (7369, 'SMITH', 'CLERK', 7902, TO_DATE('17-DEC-1980', 'DD-MON-YYYY'), 800, NULL, 10);
INSERT 0 1

INSERT INTO emp VALUES (7788, 'SCOTT', 'ANALYST', 7566, TO_DATE('09-DEC-1982', 'DD-MON-YYYY'), 3000, NULL, 20);
INSERT 0 1

INSERT INTO emp VALUES (7566, 'JONES', 'MANAGER', 7390, TO_DATE('2-APR-1981', 'DD-MON-YYYY'), 2975, NULL, 30);
INSERT 0 1

INSERT INTO dept VALUES (10, 'ACCOUNTING', 'NEW YORK');
For a table in the log file:

- `select@low_security@high_security` audits `SELECT` statements of the `low_security` and `high_security` audit groups.
- `insert@high_security-low_security` audits `INSERT` statements of `high_security` and excludes the insert statements of `low_security` audit group.
- `update-low_security-high_security` audits `UPDATE` statements of `high_security` and excludes the update statements of the `low_security` audit group.
- `delete@low_security-high_security` audits `DELETE` statements of `low_security` and excludes the delete statements of `high_security` audit group for a table in the log file.

The resulting audit log file contains the following. (Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)

```
155c2,1,"authentication",2021-07-20 01:45:27 IST,5/1,0,AUDIT,00000,"connection authorized: user=enterprisedb
database=edb",,,,,,,,,"client backend",","","connect"

2021-07-20 01:45:40.125 IST,"enterprisedb","edb",87490,"[local]",8f86f62f7.
155c2,1,"authentication",2021-07-20 01:45:27 IST,5/1,0,AUDIT,00000,"statement: SELECT * FROM emp;",,,,,,,,,
"psql","client backend","SELECT","","select"
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
9.3 Protecting against SQL injection attacks

EDB Postgres Advanced Server provides protection against SQL injection attacks. A SQL injection attack is an attempt to compromise a database by running SQL statements whose results provide clues to the attacker as to the content, structure, or security of that database.

Preventing a SQL injection attack is normally the responsibility of the application developer. The database administrator typically has little or no control over the potential threat. The difficulty for database administrators is that the application must have access to the data to function properly.

SQL/Protect:

- Allows a database administrator to protect a database from SQL injection attacks
- Provides a layer of security in addition to the normal database security policies by examining incoming queries for common SQL injection profiles
- Gives the control back to the database administrator by alerting the administrator to potentially dangerous queries and by blocking these queries.

9.3.1 SQL/Protect overview

SQL/Protect guards against different types of SQL injection attacks.

Types of SQL injection attacks

A number of different techniques are used to perpetrate SQL injection attacks. Each technique is characterized by a certain signature. SQL/Protect examines queries for the following signatures.
Unauthorized relations

While EDB Postgres Advanced Server allows administrators to restrict access to relations (such as tables and views), many administrators don’t perform this tedious task. SQL/Protect provides a learn mode that tracks the relations a user accesses.

This mode allows administrators to examine the workload of an application and for SQL/Protect to learn the relations an application is allowed to access for a given user or group of users in a role.

When SQL/Protect is switched to passive or active mode, the incoming queries are checked against the list of learned relations.

Utility commands

A common technique used in SQL injection attacks is to run utility commands, which are typically SQL data definition language (DDL) statements. An example is creating a user-defined function that can access other system resources.

SQL/Protect can block running all utility commands that aren’t normally needed during standard application processing.

SQL tautology

The most frequent technique used in SQL injection attacks is issuing a tautological WHERE clause condition (that is, using a condition that is always true).

The following is an example:

```
WHERE password = 'x' OR 'x'='x'
```

Attackers usually start identifying security weaknesses using this technique. SQL/Protect can block queries that use a tautological conditional clause.

Unbounded DML statements

A dangerous action taken during SQL injection attacks is running unbounded DML statements. These are UPDATE and DELETE statements with no WHERE clause. For example, an attacker might update all users’ passwords to a known value or initiate a denial of service attack by deleting all of the data in a key table.

Monitoring SQL injection attacks

SQL/Protect can monitor and report on SQL injection attacks.

Protected roles

Monitoring for SQL injection attacks involves analyzing SQL statements originating in database sessions where the current user of the session is a protected role. A protected role is an EDB Postgres Advanced Server user or group that the database administrator chooses to monitor using SQL/Protect. (In EDB Postgres Advanced Server, users and groups are collectively referred to as roles.)

You can customize each protected role for the types of SQL injection attacks it’s being monitored for. This approach provides different levels of protection by role and significantly reduces the user-maintenance load for DBAs.

You can’t make a role with the superuser privilege a protected role. If a protected non-superuser role later becomes a superuser, certain behaviors occur when that superuser tries to issue any command:

- SQL/Protect issues a warning message for every command issued by the protected superuser.
- The statistic in the column superusers of `edb_sql_protect_stats` is incremented with every command issued by the protected superuser. See Attack attempt statistics for information on this column.
- SQL/Protect in active mode prevents all commands issued by the protected superuser from running.

Either alter a protected role that has the superuser privilege so that it’s no longer a superuser, or revert it to an unprotected role.

Attack attempt statistics

SQL/Protect records each use of a command by a protected role that’s considered an attack. It collects statistics by type of SQL injection attack, as discussed in Types of SQL injection attacks.

You can access these statistics from the view `edb_sql_protect_stats`. You can easily monitor this view to identify the start of a potential attack.

The columns in `edb_sql_protect_stats` monitor the following:

- `username`. Name of the protected role.
- `superusers`. Number of SQL statements issued when the protected role is a superuser. In effect, any SQL statement issued by a protected superuser increases this statistic. See Protected roles for information about protected superusers.
- relations. Number of SQL statements issued referencing relations that weren't learned by a protected role. (These relations aren't in a role's protected relations list.)
- commands. Number of SQL statements issued by a protected role.
- tautology. Number of SQL statements issued by a protected role that contained a tautological condition.
- dml. Number of UPDATE and DELETE statements issued by a protected role that didn't contain a WHERE clause.

These statistics give database administrators the chance to react proactively in preventing theft of valuable data or other malicious actions.

If a role is protected in more than one database, the role's statistics for attacks in each database are maintained separately and are viewable only when connected to the respective database.

**Note**

SQL/Protect statistics are maintained in memory while the database server is running. When the database server is shut down, the statistics are saved to a binary file named `edb_sqlprotect.stat` in the `data/global` subdirectory of the EDB Postgres Advanced Server home directory.

**Attack attempt queries**

Each use of a command by a protected role that's considered an attack by SQL/Protect is recorded in the `edb_sql_protect_queries` view, which contains the following columns:

- **username.** Database user name of the attacker used to log into the database server.
- **ip_address.** IP address of the machine from which the attack was initiated.
- **port.** Port number from which the attack originated.
- **machine_name.** Name of the machine from which the attack originated, if known.
- **date_time.** Date and time when the database server received the query. The time is stored to the precision of a minute.
- **query.** The query string sent by the attacker.

The maximum number of offending queries that are saved in `edb_sql_protect_queries` is controlled by the `edb_sql_protect.max_queries_to_save` configuration parameter.

If a role is protected in more than one database, the role's queries for attacks in each database are maintained separately. They are viewable only when connected to the respective database.

### 9.3.2 Configuring SQL/Protect

You can configure how SQL/Protect operates.

**Prerequisites**

Meet the following prerequisites before configuring SQL/Protect:

- The library file (`sqlprotect.so` on Linux, `sqlprotect.dll` on Windows) needed to run SQL/Protect is installed in the `lib` subdirectory of your EDB Postgres Advanced Server home directory. For Windows, the EDB Postgres Advanced Server installer does this. For Linux, install the `edb-as<xx>-server-sqlprotect` RPM package, where `<xx>` is the EDB Postgres Advanced Server version number.
- You need the SQL script file `sqlprotect.sql` located in the `share/contrib` subdirectory of your EDB Postgres Advanced Server home directory.
- You must configure the database server to use SQL/Protect, and you must configure each database that you want SQL/Protect to monitor:
  - You must modify the database server configuration file `postgresql.conf` by adding and enabling configuration parameters used by SQL/Protect.
  - Install database objects used by SQL/Protect in each database that you want SQL/Protect to monitor.

**Configuring the module**

1. Edit the following configuration parameters in the `postgresql.conf` file located in the `data` subdirectory of your EDB Postgres Advanced Server home directory:
   - `shared_preload_libraries`. Add `@libdir/sqlprotect` to the list of libraries.
   - `edb_sqlprotect.enabled`. Controls whether SQL/Protect is actively monitoring protected roles by analyzing SQL statements issued by those roles and reacting according to the setting of `edb_sqlprotect.level`. When you're ready to begin monitoring with SQL/Protect, set this parameter to `on`. The default is `off`.
   - `edb_sqlprotect.level`. Sets the action taken by SQL/Protect when a SQL statement is issued by a protected role. The default behavior is `passive`. Initially, set this parameter to `learn`. See Setting the protection level for more information.
   - `edb_sqlprotect.max_protected_relations`. Sets the maximum number of relations to protect. The default is `64`.
   - `edb_sqlprotect.max_protected_relations`. Sets the maximum number of relations to protect per role. The default is `1024`.

---

EDB Postgres Advanced Server (EPAS)

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The total number of protected relations for the server is the number of protected relations times the number of protected roles. Every protected relation consumes space in shared memory. The space for the maximum possible protected relations is reserved during database server startup.

- `edb_sql_protect.max_queries_to_save`. Sets the maximum number of offending queries to save in the `edb_sql_protect_queries` view. The default is 5000. If the number of offending queries reaches the limit, additional queries aren't saved in the view but are accessible in the database server log file.

The minimum valid value for this parameter is 100. If you specify a value less than 100, the database server starts using the default setting of 5000. A warning message is recorded in the database server log file.

This example shows the settings of these parameters in the `postgresql.conf` file:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/sqlprotect'
# (change requires restart)

# Enable SQLProtect
edb_sql_protect.enabled = on
edb_sql_protect.level = learn
edb_sql_protect.max_protected_roles = 64
edb_sql_protect.max_protected_relations = 1024
edb_sql_protect.max_queries_to_save = 5000
```

2. After you modify the `postgresql.conf` file, restart the database server.

- **On Linux**: Invoke the EDB Postgres Advanced Server service script with the `restart` option.

```
systemctl restart edb-as-14
```

- **On Windows**: Use the Windows Services applet to restart the service named `edb-as-14`.

3. For each database that you want to protect from SQL injection attacks, connect to the database as a superuser (either `enterprisedb` or `postgres`, depending on your installation options). Then run the script `sqlprotect.sql`, located in the `share/contrib` subdirectory of your EDB Postgres Advanced Server home directory. The script creates the SQL/Protect database objects in a schema named `sqlprotect`.

This example shows the process to set up protection for a database named `edb`:

```
$ /usr/edb/as14/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (14.0.0, server 14.0.0)
Type "help" for help.

edb=# \'\i /usr/edb/as14/share/contrib/sqlprotect.sql
CREATE SCHEMA
GRANT
SET
CREATE TABLE
GRANT
CREATE TABLE
GRANT
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
DO
CREATE FUNCTION
CREATE FUNCTION
DO
CREATE VIEW
GRANT
DO
CREATE VIEW
GRANT
CREATE VIEW
GRANT
CREATE FUNCTION
CREATE FUNCTION
SET
```
Selecting roles to protect

After you create the SQL/Protect database objects in a database, you can select the roles for which to monitor SQL queries for protection and the level of protection to assign to each role.

Setting the protected roles list

For each database that you want to protect, you must determine the roles you want to monitor and then add those roles to the protected roles list of that database.

1. Connect as a superuser to a database that you want to protect with either psql or the Postgres Enterprise Manager client:

```
$ /usr/edb/as14/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (14.0.0, server 14.0.0)
Type "help" for help.
edb=#
```

2. Since the SQL/Protect tables, functions, and views are built under the sqlprotect schema, use the SET search_path command to include the sqlprotect schema in your search path. Doing so eliminates the need to schema-qualify any operation or query involving SQL/Protect database objects:

```
edb=# SET search_path TO sqlprotect;
SET
```

3. You must add each role that you want to protect to the protected roles list. This list is maintained in the table edb_sql_protect.

To add a role, use the function protect_role('rolename'). This example protects a role named appuser:

```
edb=# SELECT protect_role('appuser');
protect_role
-------------
(1 row)
```

You can list the roles that were added to the protected roles list with the following query:

```
edb=# SELECT * FROM edb_sql_protect;
               dbid | roleid | protect_relations | allow_utility_cmds | allow_tautology | allow_empty_dml
---------------+--------+-------------------+--------------------+-----------------+--
              13917 |  16671 | t                 | f                  | f               | f
(1 row)
```

A view is also provided that gives the same information using the object names instead of the object identification numbers (OIDs):

```
edb=# \x
Expanded display is on.
edb=# SELECT * FROM list_protected_users;
[- RECORD 1 ]------------
dbname | edb | username | protect_relations | allow_utility_cmds | allow_tautology | allow_empty_dml
--------+-----+----------+-------------------+--------------------+-----------------+--
dbname | edb | username | protect_relations | allow_utility_cmds | allow_tautology | allow_empty_dml
--------+-----+----------+-------------------+--------------------+-----------------+--
         |     | appuser  | t                 | f                  | f               | f
(1 row)
```

Setting the protection level

The edb_sql_protect.level configuration parameter sets the protection level, which defines the behavior of SQL/Protect when a protected role issues a SQL statement. The defined behavior applies to all roles in the protected roles lists of all databases configured with SQL/Protect in the database server.

You can set the edb_sql_protect.level configuration parameter in the postgresql.conf file to one of the following values to specify learn, passive, or active mode:

- **learn**. Tracks the activities of protected roles and records the relations used by the roles. Use this mode when first configuring SQL/Protect so the expected behaviors of the protected applications are learned.
monitoring protected roles

after you configure sql/protect in a database, add roles to the protected roles list, and set the desired protection level, you can activate sql/protect in learn, passive, or active mode. you can then start running your applications.

with a new sql/protect installation, the first step is to determine the relations that protected roles are allowed to access during normal operation. learn mode allows a role to run applications during which time sql/protect is recording the relations that are accessed. these are added to the role’s protected relations list stored in table edb_sql_protect_rel.

monitoring for protection against attack begins when you run sql/protect in passive or active mode. in passive and active modes, the role is permitted to access the relations in its protected relations list. these are the specified relations the role can access during typical usage.

however, if a role attempts to access a relation that isn’t in its protected relations list, sql/protect returns a warning or error severity-level message. the role’s attempted action on the relation might not be carried out, depending on whether the mode is passive or active.

learn mode

to activate sql/protect in learn mode:

1. set the parameters in the postgresql.conf file:

   ![code]

2. reload the postgresql.conf file. from the edb postgres advanced server application menu, select reload configuration > expert configuration.

   ![code]

   for an alternative method of reloading the configuration file, use the pg_reload_conf function. be sure you’re connected to a database as a superuser, and execute function pg_reload_conf:

   ![code]

3. allow the protected roles to run their applications.

   for example, the following queries are issued in the psql application by the protected role appuser:

   ![code]
SQL/Protect generates a `NOTICE` severity-level message, indicating the relation was added to the role’s protected relations list.

In SQL/Protect learn mode, SQL statements that are cause for suspicion aren’t prevented from executing. However, a message is issued to alert the user to potentially dangerous statements:

```sql
CREATE TABLE appuser_tab (f1 INTEGER);
NOTICE: SQLPROTECT: This command type is illegal for this user
CREATE TABLE
DELETE FROM appuser_tab;
NOTICE: SQLPROTECT: Learned relation: 16672
NOTICE: SQLPROTECT: Illegal Query: empty DML
DELETE 0
```

4. As a protected role runs applications, you can query the SQL/Protect tables to see that relations were added to the role’s protected relations list. Connect as a superuser to the database you’re monitoring, and set the search path to include the `sqlprotect` schema:

```sql
SELECT * FROM edb_sql_protect_rel;
```

<table>
<thead>
<tr>
<th>dbid</th>
<th>roleid</th>
<th>relid</th>
</tr>
</thead>
<tbody>
<tr>
<td>13917</td>
<td>16671</td>
<td>16384</td>
</tr>
<tr>
<td>13917</td>
<td>16671</td>
<td>16391</td>
</tr>
<tr>
<td>13917</td>
<td>16671</td>
<td>16672</td>
</tr>
</tbody>
</table>
(3 rows)

The `list_protected_rels` view provides more comprehensive information along with the object names instead of the OIDs:

```sql
SELECT * FROM list_protected_rels;
```

<table>
<thead>
<tr>
<th>Database</th>
<th>Protected User</th>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb</td>
<td>appuser</td>
<td>public</td>
<td>dept</td>
<td>Table</td>
<td>enterprisesdb</td>
</tr>
<tr>
<td>edb</td>
<td>appuser</td>
<td>public</td>
<td>emp</td>
<td>Table</td>
<td>enterprisesdb</td>
</tr>
<tr>
<td>edb</td>
<td>appuser</td>
<td>public</td>
<td>appuser_tab</td>
<td>Table</td>
<td>appuser</td>
</tr>
</tbody>
</table>
(3 rows)

**Passive mode**

After a role’s applications have accessed all relations they need, you can change the protection level so that SQL/Protect can actively monitor the incoming SQL queries and protect against SQL injection attacks.

Passive mode is a less restrictive protection mode than active.

1. To activate SQL/Protect in passive mode, set the following parameters in the `postgresql.conf` file:

   ```sql
   edb_sql_protect.enabled = on
   edb_sql_protect.level = passive
   ```

2. Reload the configuration file as shown in Step 2 of **Learn mode**.

   Now SQL/Protect is in passive mode. For relations that were learned, such as the `dept` and `emp` tables of the prior examples, SQL statements are permitted. No special notification to the client by SQL/Protect is required, as shown by the following queries run by user `appuser`:

```sql
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>
(4 rows)

```sql
SELECT empno, ename, job FROM emp WHERE deptno = 10;
```
SQL/Protect doesn't prevent any SQL statement from executing. However, it issues a message of WARNING severity level for SQL statements executed against relations that weren't learned. It also issues a warning for SQL statements that contain a prohibited signature:

```sql
CREATE TABLE appuser_tab_2 (f1 INTEGER);
WARNING: SQLPROTECT: This command type is illegal for this user
CREATE TABLE
```

```sql
INSERT INTO appuser_tab_2 VALUES (1);
WARNING: SQLPROTECT: Illegal Query: relations
INSERT 0 1
```

```sql
INSERT INTO appuser_tab_2 VALUES (2);
WARNING: SQLPROTECT: Illegal Query: relations
INSERT 0 1
```

```sql
SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
WARNING: SQLPROTECT: Illegal Query: relations
WARNING: SQLPROTECT: Illegal Query: tautology
```

3. Monitor the statistics for suspicious activity.

By querying the view `edb_sql_protect_stats`, you can see the number of times SQL statements executed that referenced relations that weren't in a role's protected relations list or contained SQL injection attack signatures.

The following is a query on `edb_sql_protect_stats`:

```sql
SET search_path TO sqlprotect;
SELECT * FROM edb_sql_protect_stats;
```

The following is a query on `edb_sql_protect_queries` view, you can see the SQL statements that were executed that referenced relations that weren't in a role's protected relations list or that contained SQL injection attack signatures.

The following code sample shows a query on `edb_sql_protect_queries`:

```sql
SELECT * FROM edb_sql_protect_queries;
```

- (1 row)
Active mode

In active mode, disallowed SQL statements are prevented from executing. Also, the message issued by SQL/Protect has a higher severity level of ERROR instead of WARNING.

1. To activate SQL/Protect in active mode, set the following parameters in the `postgresql.conf` file:

   - `edb_sql_protect.enabled = on`
   - `edb_sql_protect.level = active`

2. Reload the configuration file as shown in Step 2 of Learn mode.

This example shows SQL statements similar to those given in the examples of Step 2 in Passive mode. These statements are executed by the user `appuser` when `edb_sql_protect.level` is set to `active`:

```sql
CREATE TABLE appuser_tab_3 (f1 INTEGER);
ERROR: SQLPROTECT: This command type is illegal for this user
INSERT INTO appuser_tab_2 VALUES (1);
ERROR: SQLPROTECT: Illegal Query:
SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

The following shows the resulting statistics:

```sql
SELECT * FROM sqlprotect.edb_sql_protect_stats;

username | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
appuser   |      0     |     5     |     2    |     1     | 0
(1 row)
```

The following is a query on `edb_sql_protect_queries`:

```sql
SELECT * FROM sqlprotect.edb_sql_protect_queries;
```

Note

The `ip_address` and `port` columns don’t return any information if the attack originated on the same host as the database server using the Unix-domain socket (that is, `pg_hba.conf` connection type `local`).
### 9.3.3 Common maintenance operations

You must be connected as a superuser to perform these operations. Include the `sqlprotect` schema in your search path.

#### Adding a role to the protected roles list

Add a role to the protected roles list. Run `protect_role('rolename')`, as shown in this example:

```sql
SELECT protect_role('newuser');
```

<table>
<thead>
<tr>
<th>protect_role</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

#### Removing a role from the protected roles list

To remove a role from the protected roles list, use either of the following functions:

```
unprotect_role('rolename')
unprotect_role(roleoid)
```

The variation of the function using the OID is useful if you remove the role using the `DROP ROLE` or `DROP USER` SQL statement before removing the role from the protected roles list. If a query on a SQL/Protect relation returns a value such as `unknown (OID=16458)` for the user name, use the `unprotect_role(roleoid)` form of the function to remove the entry for the deleted role from the protected roles list.

Removing a role using these functions also removes the role’s protected relations list.

To delete the statistics for a role that was removed, use the `drop_stats` function.

To delete the offending queries for a role that was removed, use the `drop_queries` function.

This example shows the `unprotect_role` function:

```sql
SELECT unprotect_role('newuser');
```

<table>
<thead>
<tr>
<th>unprotect_role</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

Alternatively, you can remove the role by giving its OID of 16693:

```sql
SELECT unprotect_role(16693);
```

<table>
<thead>
<tr>
<th>unprotect_role</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Setting the types of protection for a role

You can change whether a role is protected from a certain type of SQL injection attack.

Change the Boolean value for the column in `edb_sql_protect` corresponding to the type of SQL injection attack for which you want to enable or disable protection of a role.

Be sure to qualify the following columns in your `WHERE` clause of the statement that updates `edb_sql_protect`:

- `dbid`. OID of the database for which you’re making the change.
- `roleid`. OID of the role for which you’re changing the Boolean settings.

For example, to allow a given role to issue utility commands, update the `allow_utility_cmds` column:

```
UPDATE edb_sql_protect SET allow_utility_cmds = TRUE WHERE dbid = 13917 AND roleid = 16671;
```

You can verify the change was made by querying `edb_sql_protect` or `list_protected_users`. In the following query, note that column `allow_utility_cmds` now contains `t`:

```
edb=# SELECT dbid, roleid, allow_utility_cmds FROM edb_sql_protect;
```

```
<table>
<thead>
<tr>
<th>dbid</th>
<th>roleid</th>
<th>allow_utility_cmds</th>
</tr>
</thead>
<tbody>
<tr>
<td>13917</td>
<td>16671</td>
<td>t</td>
</tr>
</tbody>
</table>
```

(1 row)

The updated rules take effect on new sessions started by the role since the change was made.

Removing a relation from the protected relations list

If SQL/Protect learns that a given relation is accessible for a given role, you can later remove that relation from the role’s protected relations list.

Delete the entry from the `edb_sql_protect_rel` table using any of the following functions:

```
unprotect_rel('rolename', 'relname')
unprotect_rel('rolename', 'schema', 'relname')
unprotect_rel(roleoid, reloid)
```

If the relation given by `relname` isn’t in your current search path, specify the relation’s schema using the second function format.

The third function format allows you to specify the OIDs of the role and relation, respectively, instead of their text names.

This example removes the `public.emp` relation from the protected relations list of the role `appuser`:

```
edb=# SELECT unprotect_rel('appuser', 'public', 'emp');
```

This query shows there’s no longer an entry for the `emp` relation:

```
edb=# SELECT * FROM list_protected_rels;
```

```
<table>
<thead>
<tr>
<th>Database</th>
<th>Protected User</th>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb</td>
<td>appuser</td>
<td>public</td>
<td>dept</td>
<td>Table</td>
<td>enterprisedb</td>
</tr>
<tr>
<td>edb</td>
<td>appuser</td>
<td>public</td>
<td>appuser_tab</td>
<td>Table</td>
<td>appuser</td>
</tr>
</tbody>
</table>
```

(2 rows)

SQL/Protect now issues a warning or completely blocks access (depending on the setting of `edb_sql_protect.level`) when the role attempts to use that relation.
Deleting statistics

You can delete statistics from the view `edb_sql_protect_stats` using either of the following functions:

```
drop_stats('rolename')
drop_stats(roleoid)
```

The form of the function using the OID is useful if you remove the role using the DROP ROLE or DROP USER SQL statement before deleting the role’s statistics using `drop_stats('rolename')`. If a query on `edb_sql_protect_stats` returns a value such as `unknown (OID=16458)` for the user name, use the `drop_stats(roleoid)` form of the function to remove the deleted role’s statistics from `edb_sql_protect_stats`.

This example shows the `drop_stats` function:

```
edb=# SELECT drop_stats('appuser');
  drop_stats
-----------
(1 row)
edb=# SELECT * FROM edb_sql_protect_stats;
  username  | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
   unknown  |          0 |         5 |        3 |         1 |   0
  appuser   |          0 |         5 |        2 |         1 |   0
(2 rows)
edb=# SELECT drop_stats(16693);
  drop_stats
-----------
(1 row)
edb=# SELECT * FROM edb_sql_protect_stats;
  username  | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
  appuser   |          0 |         5 |        2 |         1 |   0
(1 row)
```

Deleting offending queries

You can delete offending queries from the view `edb_sql_protect_queries` using either of the following functions:

```
drop_queries('rolename')
drop_queries(roleoid)
```

The variation of the function using the OID is useful if you remove the role using the DROP ROLE or DROP USER SQL statement before deleting the role’s offending queries using `drop_queries('rolename')`. If a query on `edb_sql_protect_queries` returns a value such as `unknown (OID=16454)` for the user name, use the `drop_queries(roleoid)` form of the function to remove the deleted role’s offending queries from `edb_sql_protect_queries`.

This example shows the `drop_queries` function:

```
edb=# SELECT drop_queries('appuser');
  drop_queries
-----------
(1 row)
```
This example uses the `drop_queries(roleoid)` form of the function when a role is dropped before deleting its queries:

```
username | query
---------+----------------------------------------------
unknown (OID=16454) | CREATE TABLE appuser_tab_2 (f1 INTEGER);
unknown (OID=16454) | INSERT INTO appuser_tab_2 VALUES (2);
unknown (OID=16454) | CREATE TABLE appuser_tab_3 (f1 INTEGER);
unknown (OID=16454) | INSERT INTO appuser_tab_2 VALUES (1);
unknown (OID=16454) | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

Disabling and enabling monitoring

If you want to turn off SQL/Protect monitoring, modify the `postgresql.conf` file, setting the `edb_sql_protect.enabled` parameter to `off`. After saving the file, reload the server configuration to apply the settings.

If you want to turn on SQL/Protect monitoring, modify the `postgresql.conf` file, setting the `edb_sql_protect.enabled` parameter to `on`. Save the file, and then reload the server configuration to apply the settings.

9.3.4 Backing up and restoring a SQL/Protect database

Backing up a database that's configured with SQL/Protect and then restoring the backup file to a new database requires considerations in addition to those normally associated with backup and restore procedures. These added considerations are mainly due to the use of object identification numbers (OIDs) in the SQL/Protect tables.

```
Note
This information applies if your backup and restore procedures result in re-creating database objects in the new database with new OIDs, such as when using the `pg_dump` backup program.
```

If you're backing up your EDB Postgres Advanced Server database server by using the operating system's copy utility to create a binary image of the EDB Postgres Advanced Server data files (file system backup method), then this information doesn't apply.

Object identification numbers in SQL/Protect tables

SQL/Protect uses two tables, `edb_sql_protect` and `edb_sql_protect_rel`, to store information on database objects such as databases, roles, and relations. References to these database objects in these tables are done using the objects' OIDs, not their text names. The OID is a numeric data type used by EDB Postgres Advanced Server to uniquely identify each database object.

When a database object is created, EDB Postgres Advanced Server assigns an OID to the object, which is then used when a reference to the object is needed in the database catalogs. If you create the same database object in two databases, such as a table with the same `CREATE TABLE` statement, each table is assigned a different OID in each database.
In a backup and restore operation that results in re-creating the backed-up database objects, the restored objects end up with different OIDs in the new database from what they were assigned in the original database. As a result, the OIDs referencing databases, roles, and relations stored in the `edb_sql_protect` and `edb_sql_protect_rel` tables are no longer valid when these tables are dumped to a backup file and then restored to a new database.

Two functions, `export_sqlprotect` and `import_sqlprotect`, are used specifically for backing up and restoring SQL/Protect tables to ensure the OIDs in the SQL/Protect tables reference the correct database objects after the tables are restored.

**Backing up the database**

Back up a database that was configured with SQL/Protect.

1. Create a backup file using `pg_dump`.

   This example shows a plain-text backup file named `/tmp/edb.dmp` created from database `edb` using the `pg_dump` utility program:

   ```
   $ cd /usr/edb/as14/bin
   $ ./pg_dump -U enterprisedb -Fp -f /tmp/edb.dmp edb
   Password:
   $ 
   ```

2. Connect to the database as a superuser, and export the SQL/Protect data using the `export_sqlprotect('sqlprotect_file')` function. `sqlprotect_file` is the fully qualified path to a file where the SQL/Protect data is saved.

   The `enterprisedb` operating system account (`postgres` if you installed EDB Postgres Advanced Server in PostgreSQL compatibility mode) must have read and write access to the directory specified in `sqlprotect_file`.

   ```
   edb# SELECT sqlprotect.export_sqlprotect('/tmp/sqlprotect.dmp');
   export_sqlprotect
   ------------
   (1 row)
   ```

   The files `/tmp/edb.dmp` and `/tmp/sqlprotect.dmp` comprise your total database backup.

**Restoring from the backup files**

1. Restore the backup file to the new database.

   This example uses the `psql` utility program to restore the plain-text backup file `/tmp/edb.dmp` to a newly created database named `newdb`:

   ```
   $ /usr/as14/bin/psql -d newdb -U enterprisedb -f /tmp/edb.dmp
   Password for user enterprisedb:
   SET
   SET
   SET
   SET
   SET
   COMMENT
   CREATE SCHEMA
   .
   .
   .
   ```

2. Connect to the new database as a superuser, and delete all rows from the `edb_sql_protect_rel` table.

   This deletion removes any existing rows in the `edb_sql_protect_rel` table that were backed up from the original database. These rows don't contain the correct OIDs relative to the database where the backup file was restored.

   ```
   newdb=# DELETE FROM sqlprotect.edb_sql_protect_rel;
   ```
3. Delete all rows from the `edb_sql_protect` table.

   This deletion removes any existing rows in the `edb_sql_protect` table that were backed up from the original database. These rows don't contain the correct OIDs relative to the database where the backup file was restored.

   ```sql
   newdb=# DELETE FROM sqlprotect.edb_sql_protect;
   ```

4. Delete any of the database’s statistics.

   This deletion removes any existing statistics for the database to which you’re restoring the backup. The following query displays any existing statistics:

   ```sql
   newdb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
   ```

   For each row that appears in the preceding query, use the `drop_stats` function, specifying the role name of the entry.

   For example, if a row appeared with `appuser` in the `username` column, issue the following command to remove it:

   ```sql
   newdb=# SELECT sqlprotect.drop_stats('appuser');
   ```

5. Delete any of the database’s offending queries.

   This deletion removes any existing queries for the database to which you’re restoring the backup. This query displays any existing queries:

   ```sql
   edb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
   ```

   For each row that appears in the preceding query, use the `drop_queries` function, specifying the role name of the entry. For example, if a row appeared with `appuser` in the `username` column, issue the following command to remove it:

   ```sql
   edb=# SELECT sqlprotect.drop_queries('appuser');
   ```

6. Make sure the role names that were protected by SQL/Protect in the original database are in the database server where the new database resides.

   If the original and new databases reside in the same database server, then you don’t need to do anything if you didn’t delete any of these roles from the database server.

7. Run the function `import_sqlprotect('sqlprotect_file')`, where `sqlprotect_file` is the fully qualified path to the file you created in Step 2 of Backing up the database.

   ```sql
   newdb=# SELECT sqlprotect.import_sqlprotect('/tmp/sqlprotect.dmp');
   ```

   Tables `edb_sql_protect` and `edb_sql_protect_rel` are populated with entries containing the OIDs of the database objects as assigned in the new database. The statistics view `edb_sql_protect_stats` also displays the statistics imported from the original database.

   The SQL/Protect tables and statistics are properly restored for this database. Use the following queries on the EDB Postgres Advanced Server system catalogs to verify:

   ```sql
   newdb=# SELECT datname, oid FROM pg_database;
   ```
| rolname    | oid  |
|------------+------|
| enterprisesdb | 10   |
| appuser    | 16671|
| newuser    | 16678|

(3 rows)

| relname | oid  |
|---------+------|
| appuser_tab | 16803 |
| dept     | 16809 |
| emp      | 16812 |

(3 rows)

| dbid  | roleid | protect_relations | allow_utility_cmds | allow_tautology | allow_empty_dml |
|-------+--------+-------------------+--------------------+-----------------+-----------------|
| 16679 | 16671  | t                 | t                  | f               | f               |

(1 row)

| dbid  | roleid | relid  |
|-------+--------+--------|
| 16679 | 16671  | 16809  |
| 16679 | 16671  | 16803  |

(2 rows)

| username | superusers | relations | commands | tautology | dml   |
|----------+------------+-----------+----------+-----------+------|
| appuser  | 0          | 5         | 2        | 1         | 0    |

(1 row)
Note the following about the columns in tables edb_sql_protect and edb_sql_protect_rel:

- **dbid**: Matches the value in the **oid** column from pg_database for newdb.
- **roleid**: Matches the value in the **oid** column from pg_roles for appuser.

Also, in table edb_sql_protect_rel, the values in the **relid** column match the values in the **oid** column of pg_class for relations dept and appuser_tab.

8. Verify that the SQLProtect configuration parameters are set as desired in the postgresql.conf file for the database server running the new database. Restart the database server or reload the configuration file as appropriate.

You can now monitor the database using SQL/Protect.

9.4 Generating SSL certificates

**sslutils** is a Postgres extension that provides SSL certificate generation functions to EDB Postgres Advanced Server for use by the EDB Postgres Enterprise Manager server.

Installing the extension

You install **sslutils** by using the **edb-as<xx>-server-sslutils** RPM package, where **<xx>** is the EDB Postgres Advanced Server version number.

Each parameter in the function's parameter list is described by parameter **n**, where **n** refers to the **n**th ordinal position (for example, first, second, or third) in the function's parameter list.

**openssl_rsa_generate_key**

The **openssl_rsa_generate_key** function generates an RSA private key. The function signature is:

```sql
openssl_rsa_generate_key(<integer>) RETURNS <text>
```

When invoking the function, pass the number of bits as an integer value. The function returns the generated key.

**openssl_rsa_key_to_csr**

The **openssl_rsa_key_to_csr** function generates a certificate signing request (CSR). The signature is:

```sql
openssl_rsa_key_to_csr(<text>, <text>, <text>, <text>, <text>, <text>, <text>, <text>) RETURNS <text>
```

The function generates and returns the certificate signing request.

**Parameters**

- **parameter 1**

The name of the RSA key file.
The common name (e.g., `agentN`) of the agent to use the signing request.

The name of the country where the server resides.

The name of the state where the server resides.

The location (city) in the state where the server resides.

The name of the organization unit requesting the certificate.

The email address of the user requesting the certificate.

```
openssl_csr_to_crt

The `openssl_csr_to_crt` function generates a self-signed certificate or a certificate authority certificate. The signature is:

```
openssl_csr_to_crt(<text>, <text>, <text>) RETURNS <text>
```

The function returns the self-signed certificate or certificate authority certificate.

Parameters

```
parameter 1
The name of the certificate signing the request.
```

```
parameter 2
The path to the certificate authority certificate, or `NULL` if generating a certificate authority certificate.
```

```
parameter 3
The path to the certificate authority's private key or, if argument 2 is `NULL`, the path to a private key.
```

```
openssl_rsa_generate_crl

The `openssl_rsa_generate_crl` function generates a default certificate revocation list. The signature is:

```
openssl_rsa_generate_crl(<text>, <text>) RETURNS <text>
```

The function returns the certificate revocation list.

Parameters

```
parameter 1
The path to the certificate authority certificate.
```

```
parameter 2
```
The path to the certificate authority private key.

9.5 Protecting proprietary source code

The EDB*Wrap utility protects proprietary source code and programs like functions, stored procedures, triggers, and packages from unauthorized scrutiny.

9.5.1 EDB*Wrap key concepts

The EDB*Wrap program translates a plaintext file that contains SPL or PL/pgSQL source code into a file that contains the same code in a form that's nearly impossible to read. Once you have the obfuscated form of the code, you can send that code to the PostgreSQL server, and the server stores those programs in obfuscated form. While EDB*Wrap does obscure code, table definitions are still exposed.

Everything you wrap is stored in obfuscated form. If you wrap an entire package, the package body source, as well as the prototypes contained in the package header and the functions and procedures contained in the package body, are stored in obfuscated form.

If you wrap a `CREATE PACKAGE` statement, you hide the package API from other developers. You might want to wrap the package body but not the package header so users can see the package prototypes and other public variables that are defined in the package body. To allow users to see the prototypes the package contains, use EDBWrap to obfuscate only the `CREATE PACKAGE BODY` statement in the `.edbwrap` input file, omitting the `CREATE PACKAGE` statement. The package header source is stored as plaintext, while the package body source and package functions and procedures are obfuscated.

You can't unwrap or debug wrapped source code and programs. Reverse engineering is possible but very difficult.

The entire source file is wrapped into one unit. Any `psql` meta-commands included in the wrapped file aren't recognized when the file is executed. Executing an obfuscated file that contains a `psql` meta-command causes a syntax error. `edbwrap` doesn't validate SQL source code. If the plaintext form contains a syntax error, `edbwrap` doesn't report it. Instead, the server reports an error and aborts the entire file when you try to execute the obfuscated form.

9.5.2 Obfuscating source code

EDB*Wrap is a command line utility that accepts a single input source file, obfuscates the contents, and returns a single output file. When you invoke the `edbwrap` utility, you must provide the name of the file that contains the source code to obfuscate. You can also specify the name of the file where `edbwrap` writes the obfuscated form of the code.

Overview of the command-line styles

```
edbwrap: offers three different command-line styles. The first style is compatible with Oracle's wrap utility:

edbwrap iname=<input_file> [oname=<output_file>]

The iname=<input_file> argument specifies the name of the input file. If input_file doesn't contain an extension, `edbwrap` searches for a file named `input_file.sql`. The optional oname=output_file argument specifies the name of the output file. If output_file doesn't contain an extension, `edbwrap` appends `.plb` to the name.

If you don't specify an output file name, `edbwrap` writes to a file whose name is derived from the input file name. `edbwrap` strips the suffix (typically `.sql`) from the input file name and adds `.plb`.
```

```
edbwrap offers two other command-line styles:

edbwrap --iname <input_file> [--oname <output_file>]
edbwrap -i <input_file> [-o <output_file>]

You can mix command-line styles. The rules for deriving input and output file names are the same regardless of the style you use.
```
Once edbwrap produces a file that contains obfuscated code, you typically feed that file into the PostgreSQL server using a client application such as edb-psql. The server executes the obfuscated code line by line and stores the source code for SPL and PL/pgSQL programs in wrapped form.

In summary, to obfuscate code with EDB*Wrap, you:

1. Create the source code file.
2. Invoke EDB*Wrap to obfuscate the code.
3. Import the file as if it were in plaintext form.

Creating the source code file

To use the EDB*Wrap utility, create the source code for the list_emp procedure in plaintext form:

```bash
cat listemp.sql
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno NUMBER(4);
    v_ename VARCHAR2(10);

    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;
/
```

Import the list_emp procedure with a client application such as edb-psql:

```bash
edb-psql
edbWelcome to edb-psql 8.4.3.2, the EnterpriseDB interactive
terminal.
Type: \copyright for distribution
terms
    \h for help with SQL commands
    \? for help with edb-psql
    \g or terminate with semicolon to execute
query
    \q to quit
ebd=# \i listemp.sql
CREATE PROCEDURE
View the plaintext source code stored in the server by examining the pg_proc system table:

ebd=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';
__EDBwrapped__ prosrc
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Invoking EDB\*Wrap

Ofuscate the plaintext file with EDB\*Wrap:

```
[bash]
edbwrap -i listemp.sql
```

EDB\*Wrap Utility: Release 8.4.3.2
Copyright (c) 2004-2021 EnterpriseDB Corporation. All Rights Reserved.

Using encoding UTF8 for input

Processing listemp.sql to listemp.plb

Examining the contents of the output file (listemp.plb) file reveals that the code is obfuscated:

```
[bash]
cat listemp.plb
$__EDBwrapped__$
UTF8
d+6DL30RVaGjYMIzkuoSzAQgtBw7MhYFuAFDzsyYfhd20rjwBv+bHr1FCyHFj595gh
movi+bv1+VRJ+HR2jb9qSovHkUyZPby3/GckBqgualRhlqGy KEFEdtUdpkYRLN
O/UXm60/P4H6EI88sAQmGvNOwl-srDjJq4ghZt5L5GtExvbDZTPb/s5sdx4660qQ
Qz3/gbVtkQ2bxcYMejwNRV6FAxFy6AM9J5LH7Fntxb5kjsxvFWR2XYZ23f
KFnrEnhDl10VtISEC5wELGwHvYvXOsf23m1R2IFns8MfF9fecnAT1vY9jR08j6-fc
or//faFbf=$
$__EDBwrapped__$
```

The second line of the wrapped file contains an encoding name. In this case, the encoding is UTF8. When you obfuscate a file, edbwrap infers the encoding of the input file by examining the locale. For example, if you're running edbwrap while your locale is set to en_UK.utf8, edbwrap assumes that the input file is encoded in UTF8. Be sure to examine the output file after running edbwrap, if the locale contained in the wrapped file doesn't match the encoding of the input file, change your locale and rewrap the input file.

Importing the obfuscated code to the PostgreSQL server

You can import the obfuscated code to the PostgreSQL server using the same tools that work with plaintext code:

```
[bash]
edb-psql
edb
Welcome to edb-psql 8.4.3.2, the EnterpriseDB interactive terminal.
Type: \copyright for distribution terms
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The `pg_proc` system table contains the obfuscated code:

```sql
edb=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';

prosrc
--------------------------------------------------------------------------------
$__EDBwrapped__$
UTF8
dw4B9Tz69J3WOsy0GgYJQa+G2sL3IOyxS8pPyuu0TFuYfE/EXtEatwwG3h3td3k
ea+Alp3s5d4/41dbNlwpegM3s94d0R9N0plfV7q0Zv4wQts/4C4v4lhfj

$__EDBwrapped__$
(1 row)
```

Invoke the obfuscated code in the same way that you invoke the plaintext form:

```sql
edb=# exec list_emp;

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
</tr>
</tbody>
</table>

EDB-SPL Procedure successfully completed
```

When you use `pg_dump` to back up a database, wrapped programs remain obfuscated in the archive file.

Be aware that audit logs produced by the Postgres server show wrapped programs in plaintext form. Source code is also displayed in plaintext in SQL error messages generated when the program executes.

**Note**

The bodies of the objects created by the following statements aren't stored in obfuscated form:

```sql
CREATE [OR REPLACE] TYPE type_name AS OBJECT
CREATE [OR REPLACE] TYPE type_name UNDER type_name
CREATE [OR REPLACE] TYPE BODY type_name
```

### 9.6 Managing user profiles

EDB Postgres Advanced Server allows a database superuser to create named profiles. The following sections describe how to manage profiles with EDB Postgres Advanced Server.
9.6.1 Profile management key concepts

A profile is a set of password attributes that allow you to easily manage a group of roles that share comparable authentication requirements. Each profile defines rules for password management that augment password and md5 authentication. The rules in a profile can:

- Count failed login attempts
- Lock an account due to excessive failed login attempts
- Mark a password for expiration
- Define a grace period after a password expiration
- Define rules for password complexity
- Define rules that limit password reuse

If the password requirements change, you can modify the profile to apply the new requirements to each user associated with that profile.

After creating the profile, you can associate the profile with one or more users. When a user connects to the server, the server enforces the profile that's associated with their login role. Profiles are shared by all databases in a cluster, but each cluster can have multiple profiles. A single user with access to multiple databases uses the same profile when connecting to each database in the cluster.

EDB Postgres Advanced Server creates a profile named default that's associated with a new role when the role is created unless you specify an alternative profile. If you upgrade to EDB Postgres Advanced Server from a previous server version, existing roles are automatically assigned to the default profile. You can't delete the default profile.

The default profile specifies the following attributes:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAILED_LOGIN_ATTEMPTS</td>
<td>UNLIMITED</td>
</tr>
<tr>
<td>PASSWORD_LOCK_TIME</td>
<td>UNLIMITED</td>
</tr>
<tr>
<td>PASSWORD_LIFE_TIME</td>
<td>UNLIMITED</td>
</tr>
<tr>
<td>PASSWORD_GRACE_TIME</td>
<td>UNLIMITED</td>
</tr>
<tr>
<td>PASSWORD_REUSE_TIME</td>
<td>UNLIMITED</td>
</tr>
<tr>
<td>PASSWORD_REUSE_MAX</td>
<td>UNLIMITED</td>
</tr>
<tr>
<td>PASSWORD_VERIFY_FUNCTION</td>
<td>NULL</td>
</tr>
<tr>
<td>PASSWORD_ALLOW_HASHED</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

A database superuser can use the ALTER PROFILE command to modify the values specified by the default profile. For more information about modifying a profile, see Altering a profile.

9.6.2 Creating a new profile

Use the CREATE PROFILE command to create a new profile. The syntax is:

```
CREATE PROFILE <profile_name> [LIMIT {<parameter> <value>} ...];
```

Include the LIMIT clause and one or more space-delimited parameter/value pairs to specify the rules enforced by EDB Postgres Advanced Server.

Parameters

- **profile_name** specifies the name of the profile.
- **parameter** specifies the attribute limited by the profile.
- **value** specifies the parameter limit.

EDB Postgres Advanced Server supports the following value for each parameter:

**FAILED_LOGIN_ATTEMPTS** specifies the number of failed login attempts that a user can make before the server locks them out of their account for the length of time specified by PASSWORD_LOCK_TIME. Supported values are:

- An INTEGER value greater than 0.
- DEFAULT — The value of FAILED_LOGIN_ATTEMPTS specified in the DEFAULT profile.
- UNLIMITED — The connecting user can make an unlimited number of failed login attempts.

**PASSWORD_LOCK_TIME** specifies the length of time that must pass before the server unlocks an account that was locked because of FAILED_LOGIN_ATTEMPTS. Supported values are:
The following command creates a profile named acctg. The profile specifies that if a user doesn't authenticate with the correct password in five attempts, the account is locked for one day:

```
CREATE PROFILE acctg LIMIT
  FAILED_LOGIN_ATTEMPTS 5
  PASSWORD_LOCK_TIME 1;
```

The following command creates a profile named sales. The profile specifies that a user must change their password every 90 days:

```
CREATE PROFILE sales LIMIT
  PASSWORD_LIFE_TIME 90
  PASSWORD_GRACE_TIME 3;
```

If the user doesn't change their password before the 90 days specified in the profile has passed, a warning appears at login. After a grace period of their days, their account can't invoke any commands until they change their password.
The following command creates a profile named `accts`. The profile specifies that a user can't reuse a password within 180 days of the last use of the password and must change their password at least five times before reusing the password:

```
CREATE PROFILE accts LIMIT
    PASSWORD_REUSE_TIME 180
    PASSWORD_REUSE_MAX 5;
```

The following command creates a profile named `resources`. The profile calls a user-defined function named `password_rules` that verifies that the password provided meets their standards for complexity:

```
CREATE PROFILE resources LIMIT
    PASSWORD_VERIFY_FUNCTION password_rules;
```

### 9.6.2.1 Creating a password function

When specifying `PASSWORD_VERIFY_FUNCTION`, you can provide a customized function that specifies the security rules to apply when your users change their password. For example, you can specify rules that stipulate that the new password must be at least \( n \) characters long and can't contain a specific value.

The password function has the following signature:

```
<function_name>(<user_name> VARCHAR2,
    <new_password> VARCHAR2,
    <old_password> VARCHAR2) RETURN boolean
```

Where:

- `<user_name>` is the name of the user.
- `<new_password>` is the new password.
- `<old_password>` is the user's previous password. If you reference this parameter in your function:
  - When a database superuser changes their password, the third parameter is always `NULL`.
  - When a user with the `CREATEROLE` attribute changes their password, the parameter passes the previous password if the statement includes the `REPLACE` clause. The `REPLACE` clause is optional syntax for a user with the `CREATEROLE` privilege.
  - When a user that isn't a database superuser and doesn't have the `CREATEROLE` attribute changes their password, the third parameter contains the previous password for the role.

The function returns a Boolean value. If the function returns `true` and doesn't raise an exception, the password is accepted. If the function returns `false` or raises an exception, the password is rejected. If the function raises an exception, the specified error message is displayed to the user. If the function doesn't raise an exception but returns `false`, the following error message is displayed:

```
ERROR: password verification for the specified password failed
```

The function must be owned by a database superuser and reside in the `sys` schema.

**Example**

This example creates a profile and a custom function. Then, the function is associated with the profile.

This `CREATE PROFILE` command creates a profile named `acctg_pwd_profile`:

```
CREATE PROFILE acctg_pwd_profile;
```

The following commands create a schema-qualified function named `verify_password`:

```
CREATE OR REPLACE FUNCTION sys.verify_password(user_name VARCHAR2,
    new_password VARCHAR2, old_password VARCHAR2)
RETURN boolean IMMUTABLE IS
BEGIN
    IF (length(new_password) < 5)
    THEN
        raise_application_error(-20001, 'too short');
    END IF;

    IF substring(new_password FROM old_password) IS NOT NULL
```
The function first ensures that the password is at least five characters long and then compares the new password to the old password. If the new password contains fewer than five characters or contains the old password, the function raises an error.

The following statement sets the ownership of the `verify_password` function to the `enterprisedb` database superuser:

```
ALTER FUNCTION verify_password(varchar2, varchar2, varchar2) OWNER TO enterprisedb;
```

Then, the `verify_password` function is associated with the profile:

```
ALTER PROFILE acctg_pwd_profile LIMIT PASSWORD_VERIFY_FUNCTION verify_password;
```

The following statements confirm that the function is working by first creating a test user (alice), and then attempting to associate invalid and valid passwords with her role:

```
CREATE ROLE alice WITH LOGIN PASSWORD 'temp_password' PROFILE acctg_pwd_profile;
```

Then, when alice connects to the database and attempts to change her password, she must adhere to the rules established by the profile function. A non-superuser without `CREATEROLE` must include the `REPLACE` clause when changing a password:

```
edb=> ALTER ROLE alice PASSWORD 'hey';
ERROR: missing REPLACE clause
```

The new password must be at least five characters long:

```
edb=> ALTER USER alice PASSWORD 'hey' REPLACE 'temp_password';
ERROR: EDB-20001: too short
CONTEXT:  edb-spl function verify_password(character varying,character varying,character varying) line 5 at procedure/function invocation statement
```

If the new password is acceptable, the command completes without error:

```
edb=> ALTER USER alice PASSWORD 'hello' REPLACE 'temp_password';
ALTER ROLE
```

If alice decides to change her password, the new password must not contain the old password:

```
edb=> ALTER USER alice PASSWORD 'helloworld' REPLACE 'hello';
ERROR: EDB-20002: includes old password
CONTEXT:  edb-spl function verify_password(character varying,character varying,character varying) line 10 at procedure/function invocation statement
```

To remove the verify function, set `password_verify_function` to `NULL`:

```
ALTER PROFILE acctg_pwd_profile LIMIT password_verify_function NULL;
```

Then, all password constraints are lifted:

```
edb=# ALTER ROLE alice PASSWORD 'hey';
ALTER ROLE
```

### 9.6.3 Altering a profile

Use the `ALTER PROFILE` command to modify a user-defined profile. EDB Postgres Advanced Server supports two forms of the command:

```
ALTER PROFILE <profile_name> RENAME TO <new_name>;
ALTER PROFILE <profile_name>
```
Include the `LIMIT` clause and one or more space-delimited parameter/value pairs to specify the rules enforced by EDB Postgres Advanced Server. Or use `ALTER PROFILE...RENAME TO` to change the name of a profile.

**Parameters**

- `profile_name` specifies the name of the profile.
- `new_name` specifies the new name of the profile.
- `parameter` specifies the attribute limited by the profile.
- `value` specifies the parameter limit.

See the table in Creating a new profile for a complete list of accepted parameter/value pairs.

**Examples**

The following example modifies a profile named `acctg_profile`:

```sql
ALTER PROFILE acctg_profile
LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

`acctg_profile` counts failed connection attempts when a login role attempts to connect to the server. The profile specifies that if a user doesn't authenticate with the correct password in three attempts, the account is locked for one day.

The following example changes the name of `acctg_profile` to `payables_profile`:

```sql
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

### 9.6.4 Dropping a profile

Use the `DROP PROFILE` command to drop a profile. The syntax is:

```sql
DROP PROFILE [IF EXISTS] <profile_name>
[CASCADE | RESTRICT];
```

Include the `IF EXISTS` clause to instruct the server not to throw an error if the specified profile doesn't exist. The server issues a notice if the profile doesn't exist.

Include the optional `CASCADE` clause to reassign any users that are currently associated with the profile to the `default` profile and then drop the profile. Include the optional `RESTRICT` clause to instruct the server not to drop any profile that's associated with a role. This is the default behavior.

**Parameters**

- `profile_name` The name of the profile being dropped.

**Examples**

This example drops a profile named `acctg_profile`:

```sql
DROP PROFILE acctg_profile CASCADE;
```

The command first reassociates any roles associated with the `acctg_profile` profile with the `default` profile and then drops the `acctg_profile` profile.

The following example drops a profile named `acctg_profile`:

```sql
DROP PROFILE acctg_profile RESTRICT;
```

The `RESTRICT` clause in the command instructs the server not to drop `acctg_profile` if any roles are associated with the profile.
9.6.5 Associating a profile with an existing role

After creating a profile, you can use the `ALTER USER... PROFILE` or `ALTER ROLE... PROFILE` command to associate the profile with a role. The command syntax related to profile management functionality is:

```
ALTER USER|ROLE <name> [WITH] option[...]
```

where `option` can be the following compatible clauses:

```
| PROFILE <profile_name>
| ACCOUNT
| {LOCK|UNLOCK}
| PASSWORD EXPIRE [AT '<timestamp>']
```

Or, `option` can be the following noncompatible clauses:

```
| PASSWORD SET AT '<timestamp>'
| LOCK TIME '<timestamp>'
| STORE PRIOR PASSWORD {'password' '<timestamp>}' [, ...]
```

For information about the administrative clauses of the `ALTER USER` or `ALTER ROLE` command that are supported by EDB Postgres Advanced Server, see the PostgreSQL core documentation.

Only a database superuser can use the `ALTER USER|ROLE` clauses that enforce profile management. The clauses enforce the following behaviors:

- Include the `PROFILE` clause and a `profile_name` to associate a predefined profile with a role or to change the predefined profile associated with a user.
- Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to place the user account in a locked or unlocked state.
- Include the `LOCK TIME '<timestamp>'` clause and a date/time value to lock the role at the specified time and unlock the role at the time indicated by the `PASSWORD_LOCK_TIME` parameter of the profile assigned to this role. If `LOCK TIME` is used with the `ACCOUNT LOCK` clause, only a database superuser can unlock the role with the `ACCOUNT UNLOCK` clause.
- Include the `PASSWORD EXPIRE` clause with the `AT '<timestamp>'` keywords to specify a date/time when the password associated with the role expires. If you omit the `AT '<timestamp>'` keywords, the password expires immediately.
- Include the `PASSWORD SET AT '<timestamp>'` keywords to set the password modification date to the input specified.
- Include the `STORE PRIOR PASSWORD` clause to modify the password history, adding the new password and the time the password was set.

Each login role can have only one profile. To discover the profile that’s currently associated with a login role, query the `profile` column of the `DBA_USERS` view.

**Parameters**

- `name`
  The name of the role with which to associate the specified profile.

- `password`
  The password associated with the role.

- `profile_name`
  The name of the profile to associate with the role.

- `timestamp`
  The date and time at which to enforce the clause. When specifying a value for `timestamp`, enclose the value in single quotes.

**Examples**

This command uses the `ALTER USER... PROFILE` command to associate a profile named `acctg` with a user named `john`:

```
ALTER USER john PROFILE acctg_profile;
```
The following command uses the `ALTER ROLE ... PROFILE` command to associate a profile named `acctg` with a user named `john`:

```
ALTER ROLE john PROFILE acctg_profile;
```

### 9.6.6 Unlocking a locked account

A database superuser can use clauses of the `ALTER USER|ROLE ...` command to lock or unlock a role. The syntax is:

```
ALTER USER|ROLE <name>
  ACCOUNT {
    LOCK | UNLOCK }
  LOCK TIME '<timestamp>'
```

Include the `ACCOUNT LOCK` clause to lock a role immediately. When locked, a role's `LOGIN` functionality is disabled. When you specify the `ACCOUNT LOCK` clause without the `LOCK TIME` clause, the state of the role doesn't change until a superuser uses the `ACCOUNT UNLOCK` clause to unlock the role.

Use the `ACCOUNT UNLOCK` clause to unlock a role.

Use the `LOCK TIME '<timestamp>'` clause to lock the account at the time specified by the given timestamp for the length of time specified by the `PASSWORD_LOCK_TIME` parameter of the profile associated with this role.

Combine the `LOCK TIME '<timestamp>'` clause and the `ACCOUNT LOCK` clause to lock an account at a specified time until the account is unlocked by a superuser invoking the `ACCOUNT UNLOCK` clause.

#### Parameters

- **name**
  
  The name of the role that's being locked or unlocked.

- **timestamp**
  
  The date and time when the role is locked. When specifying a value for `timestamp`, enclose the value in single quotes.

#### Note

This command (available only in EDB Postgres Advanced Server) is implemented to support Oracle-styled profile management.

#### Examples

This example uses the `ACCOUNT LOCK` clause to lock the role named `john`. The account remains locked until the account is unlocked with the `ACCOUNT UNLOCK` clause.

```
ALTER ROLE john ACCOUNT LOCK;
```

This example uses the `ACCOUNT UNLOCK` clause to unlock the role named `john`:

```
ALTER USER john ACCOUNT UNLOCK;
```

This example uses the `LOCK TIME '<timestamp>'` clause to lock the role named `john` on September 4, 2015:

```
ALTER ROLE john LOCK TIME 'September 4 12:00:00 2015';
```

The role remains locked for the length of time specified by the `PASSWORD_LOCK_TIME` parameter.

This example combines the `LOCK TIME '<timestamp>'` clause and the `ACCOUNT LOCK` clause to lock the role named `john` on September 4, 2015:

```
ALTER ROLE john LOCK TIME 'September 4 12:00:00 2015' ACCOUNT LOCK;
```

The role remains locked until a database superuser uses the `ACCOUNT UNLOCK` command to unlock the role.

### 9.6.7 Creating a new role associated with a profile
A database superuser can use clauses of the `CREATE USER|ROLE` command to assign a named profile to a role when creating the role or to specify profile management details for a role. The command syntax related to profile management functionality is:

```
CREATE USER|ROLE <name> [WITH] <option> [...]
```

where `option` can be the following compatible clauses:

```
| PROFILE <profile_name>
| ACCOUNT
|   {LOCK | UNLOCK}
|   [PASSWORD EXPIRE [AT '<timestamp>']]
```

Or, `option` can be the following noncompatible clauses:

```
| LOCK TIME '<timestamp>'
```

For information about the administrative clauses of the `CREATE USER` or `CREATE ROLE` command that are supported by EDB Postgres Advanced Server, see the PostgreSQL core documentation.

`CREATE ROLE|USER|... PROFILE` adds a role with an associated profile to an EDB Postgres Advanced Server database cluster.

Roles created with the `CREATE USER` command are by default login roles. Roles created with the `CREATE ROLE` command are by default not login roles. To create a login account with the `CREATE ROLE` command, you must include the `LOGIN` keyword.

Only a database superuser can use the `CREATE USER|ROLE` clauses that enforce profile management. These clauses enforce the following behaviors:

- Include the `PROFILE` clause and a `profile_name` to associate a predefined profile with a role or to change the predefined profile associated with a user.
- Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to place the user account in a locked or unlocked state.
- Include the `LOCK TIME '<timestamp>'` clause and a date/time value to lock the role at the specified time and unlock the role at the time indicated by the `PASSWORD_LOCK_TIME` parameter of the profile assigned to this role. If `LOCK TIME` is used with the `ACCOUNT LOCK` clause, only a database superuser can unlock the role with the `ACCOUNT UNLOCK` clause.
- Include the `PASSWORD EXPIRE` clause with the optional `AT '<timestamp>'` keywords to specify a date/time when the password associated with the role expires. If you omit the `AT '<timestamp>'` keywords, the password expires immediately.

Each login role can have only one profile. To discover the profile that's currently associated with a login role, query the `profile` column of the `DBA_USERS` view.

### Parameters

- **name**
  
The name of the role.

- **profile_name**
  
The name of the profile associated with the role.

- **timestamp**
  
The date and time when the clause is enforced. When specifying a value for `timestamp`, enclose the value in single quotes.

### Examples

This example uses `CREATE USER` to create a login role named `john` associated with the `acctg_profile` profile:

```
CREATE USER john PROFILE acctg_profile IDENTIFIED BY "lsafepwd";
```

`john` can log in to the server using the password `lsafepwd`.

This example uses `CREATE ROLE` to create a login role named `john` associated with the `acctg_profile` profile:

```
CREATE ROLE john PROFILE acctg_profile LOGIN PASSWORD "lsafepwd";
```

`john` can log in to the server using the password `lsafepwd`. 
9.6.8 Backing up profile management functions

A profile can include a `PASSWORD_VERIFY_FUNCTION` clause that refers to a user-defined function that specifies the behavior enforced by EDB Postgres Advanced Server. Profiles are global objects. They are shared by all of the databases in a cluster. While profiles are global objects, user-defined functions are database objects.

Invoking `pg_dumpall` with the `--g` or `--r` option creates a script that re-creates the definition of any existing profiles but that doesn't re-create the user-defined functions that are referred to by the `PASSWORD_VERIFY_FUNCTION` clause. Use the `pg_dump` utility to explicitly dump, and later restore, the database in which those functions reside.

The script created by `pg_dump` contains a command that includes the clause and function name to associate the restored function with the profile with which it was previously associated:

```
ALTER PROFILE ... LIMIT PASSWORD_VERIFY_FUNCTION
(function_name)
```

If the `PASSWORD_VERIFY_FUNCTION` clause is set to `DEFAULT` or `NULL`, the behavior is replicated by the script generated by the `pg_dumpall -g` or `pg_dumpall -r` command.

9.7 Redacting data

EDB Postgres Advanced Server includes features to help you to maintain, secure, and operate EDB Postgres Advanced Server databases. The DB Postgres Advanced Server `Data redaction` feature limits sensitive data exposure by dynamically changing data as it's displayed for certain users.

9.7.1 Data redaction key concepts

The DB Postgres Advanced Server `Data redaction` feature limits sensitive data exposure by dynamically changing data as it's displayed for certain users.

For example, a social security number (SSN) is stored as `021-23-9567`. Privileged users can see the full SSN, while other users see only the last four digits: `xxx-xx-9567`.

You implement data redaction by defining a function for each field to which to apply redaction. The function returns the value to display to the users subject to the data redaction.

For example, for the SSN field, the redaction function returns `xxx-xx-9567` for an input SSN of `021-23-9567`.

For a salary field, a redaction function always returns `$0.00`, regardless of the input salary value.

These functions are then incorporated into a redaction policy by using the `CREATE REDACTION POLICY` command. In addition to other options, this command specifies:

- The table on which the policy applies
- The table columns affected by the specified redaction functions
- Expressions to determine the affect session users
- The `edb_data_redaction` parameter in the `postgresql.conf` file then determines whether to apply data redaction.

By default, the parameter is enabled, so the redaction policy is in effect. The following occurs:

- Superusers and the table owner bypass data redaction and see the original data.
- All other users have the redaction policy applied and see the reformatted data.

If the parameter is disabled by having it set to `FALSE` during the session, then the following occurs:

- Superusers and the table owner bypass data redaction and see the original data.
- All other users get an error.

You can change a redaction policy using the `ALTER REDACTION POLICY` command. Or, you can eliminate it using the `DROP REDACTION POLICY` command.

9.7.2 Creating a data redaction policy

The `CREATE REDACTION POLICY` command defines a new data redaction policy for a table.

```
CREATE REDACTION POLICY <name> ON <table_name>
```

Synopsis
The `CREATE REDACTION POLICY` command defines a new column-level security policy for a table by redacting column data using a redaction function. A newly created data redaction policy is enabled by default. You can disable the policy using `ALTER REDACTION POLICY ... DISABLE`.

```
FOR ( expression )
```

This form adds a redaction policy expression.

```
ADD [ COLUMN ] <column_name> USING <funcname_clause>
```

This optional form adds a column of the table to the data redaction policy. The `USING` clause specifies a redaction function expression. You can use multiple `ADD [ COLUMN ]` forms if you want to add multiple columns of the table to the data redaction policy being created. The optional `WITH OPTIONS ( ... )` clause specifies a scope or an exception to the data redaction policy to apply. If you don't specify the scope or exception, the default value for scope is `query` and for exception is `none`.

### Parameters

- **name**
  - The name of the data redaction policy to create. This must be distinct from the name of any other existing data redaction policy for the table.

- **table_name**
  - The optionally schema-qualified name of the table the data redaction policy applies to.

- **expression**
  - The data redaction policy expression. No redaction is applied if this expression evaluates to false.

- **column_name**
  - Name of the existing column of the table on which the data redaction policy is being created.

- **funcname_clause**
  - The data redaction function that decides how to compute the redacted column value. Return type of the redaction function must be the same as the column type on which the data redaction policy is being added.

- **scope_value**
  - The scope identifies the query part to apply redaction for the column. Scope value can be `query`, `top_tlist`, or `top_tlist_or_error`. If the scope is `query`, then the redaction is applied on the column regardless of where it appears in the query. If the scope is `top_tlist`, then the redaction is applied on the column only when it appears in the query’s top target list. If the scope is `top_tlist_or_error`, the behavior is the same as the `top_tlist` but throws an error when the column appears anywhere else in the query.

- **exception_value**
  - The exception identifies the query part where redaction is exempted. Exception value can be `none`, `equal`, or `leakproof`. If exception is `none`, then there's no exemption. If exception is `equal`, then the column isn't redacted when used in an equality test. If exception is `leakproof`, the column isn't redacted when a leakproof function is applied to it.
Notes

You must be the owner of a table to create or change data redaction policies for it.

The superuser and the table owner are exempt from the data redaction policy.

Examples

This example shows how you can use this feature in production environments.

Create the components for a data redaction policy on the `employees` table:

```sql
CREATE TABLE employees (
    id integer GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    name varchar(40) NOT NULL,
    ssn varchar(11) NOT NULL,
    phone varchar(10),
    birthday date,
    salary money,
    email varchar(100)
);

-- Insert some data
INSERT INTO employees (name, ssn, phone, birthday, salary, email)
VALUES
    ('Sally Sample', '020-78-9345', '5081234567', '1961-02-02', 51234.34, 'sally.sample@enterprisedb.com'),
    ('Jane Doe', '123-33-9345', '6171234567', '1963-02-14', 62500.00, 'jane.doe@gmail.com'),
    ('Bill Foo', '123-89-9345', '9781234567', '1963-02-14', 45350, 'william.foe@hotmail.com');

-- Create a user hr who can see all the data in employees
CREATE USER hr;
-- Create a normal user
CREATE USER alice;
GRANT ALL ON employees TO hr, alice;

-- Create redaction function in which actual redaction logic
-- resides
CREATE OR REPLACE FUNCTION redact_ssn (ssn varchar(11)) RETURN varchar(11) IS
BEGIN
    /* replaces 020-12-9876 with xxx-xx-9876 */
    return overlay (ssn placing 'xxx-xx' from 1);
END;
CREATE OR REPLACE FUNCTION redact_salary () RETURN money IS
BEGIN
    return 0;
END;

CREATE REDACTION POLICY redact_policy_personal_info
    ON employees
    FOR (session_user != 'hr')
    ADD COLUMN ssn USING redact_ssn(ssn) WITH OPTIONS (SCOPE query, EXCEPTION equal),
    ADD COLUMN salary USING redact_salary();

The visible data for the hr user is:

```
-- hr can view all columns
data
```
The visible data for the normal user alice is:

```sql
-- Normal user cannot see salary and ssn number.
edb> \c edb
hr
edb> alice
edb> SELECT * FROM employees;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>ssn</th>
<th>phone</th>
<th>birthday</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sally Sample</td>
<td>028-78-9345</td>
<td>5081234567</td>
<td>02-FEB-61 00:00:00</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Jane Doe</td>
<td>123-33-9345</td>
<td>6171234567</td>
<td>14-FEB-63 00:00:00</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Bill Foo</td>
<td>123-89-9345</td>
<td>9781234567</td>
<td>14-FEB-63 00:00:00</td>
<td>$0.00</td>
</tr>
</tbody>
</table>
```

(3 rows)

But ssn data is accessible when used for equality check due to the exception_value setting:

```sql
-- Get ssn number starting from 123
edb> SELECT * FROM employees WHERE substring(ssn from 0 for 4) = '123';

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>ssn</th>
<th>phone</th>
<th>birthday</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sally Sample</td>
<td>xxx-xx-9345</td>
<td>5081234567</td>
<td>02-FEB-61 00:00:00</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Jane Doe</td>
<td>xxx-xx-9345</td>
<td>6171234567</td>
<td>14-FEB-63 00:00:00</td>
<td>$0.00</td>
</tr>
<tr>
<td></td>
<td>Bill Foo</td>
<td>xxx-xx-9345</td>
<td>9781234567</td>
<td>14-FEB-63 00:00:00</td>
<td>$0.00</td>
</tr>
</tbody>
</table>
```

(2 rows)

Caveats

- The data redaction policies created on inheritance hierarchies aren't cascaded. For example, if the data redaction policy is created for a parent, it isn't applied to the child table that inherits it, and vice versa. A user with access to these child tables can see the non-redacted data. For information about inheritance hierarchies, see the PostgreSQL core documentation.

- If the superuser or the table owner created any materialized view on the table and provided the access rights GRANT SELECT on the table and the materialized view to any non-superuser, then the non-superuser can access the non-redacted data through the materialized view.

- The objects accessed in the redaction function body must be schema qualified. Otherwise pg_dump might fail.

Compatibility

CREATE REDACTION POLICY is an EDB extension.

See also
9.7.3 Modifying a data redaction policy

The `ALTER REDACTION POLICY` command changes the definition of data redaction policy for a table.

### Synopsis

```
ALTER REDACTION POLICY <name> ON <table_name> RENAME TO <new_name>

ALTER REDACTION POLICY <name> ON <table_name> FOR ( <expression> )
}

ALTER REDACTION POLICY <name> ON <table_name> { ENABLE | DISABLE}

ALTER REDACTION POLICY <name> ON <table_name>
   ADD [ COLUMN ] <column_name> USING <funcname_clause>
   [ WITH OPTIONS ( [ <redaction_option> ] ) ]
   [ , <redaction_option> ]
}

ALTER REDACTION POLICY <name> ON <table_name>
   MODIFY [ COLUMN ] <column_name>
   { [ USING <funcname_clause> ]
   |
   [ WITH OPTIONS ( [ <redaction_option> ] ) ]
   [ , <redaction_option> ]
   }
}

ALTER REDACTION POLICY <name> ON <table_name>
   DROP [ COLUMN ] <column_name>
   { SCOPE <scope_value>| EXCEPTION <exception_value>
   }
```

Where `redaction_option` is:

```
[ SCOPE <scope_value>
| EXCEPTION <exception_value>
]
```

### Description

`ALTER REDACTION POLICY` changes the definition of an existing data redaction policy.

To use `ALTER REDACTION POLICY`, you must own the table that the data redaction policy applies to.

`FOR ( expression )`

This form adds or replaces the data redaction policy expression.

`ENABLE`

Enables the previously disabled data redaction policy for a table.
DISABLE

Disables the data redaction policy for a table.

ADD [ COLUMN ]

This form adds a column of the table to the existing redaction policy. See CREATE REDACTION POLICY for details.

MODIFY [ COLUMN ]

This form modifies the data redaction policy on the column of the table. You can update the redaction function clause or the redaction options for the column. The USING clause specifies the redaction function expression to update. The WITH OPTIONS ( ...) clause specifies the scope or the exception. For more details on the redaction function clause, the redaction scope, and the redaction exception, see CREATE REDACTION POLICY.

DROP [ COLUMN ]

This form removes the column of the table from the data redaction policy.

Parameters

name

The name of an existing data redaction policy to alter.

table_name

The optionally schema-qualified name of the table that the data redaction policy is on.

new_name

The new name for the data redaction policy. This must be distinct from the name of any other existing data redaction policy for the table.

expression

The data redaction policy expression.

column_name

Name of existing column of the table on which the data redaction policy is being altered or dropped.

funcname_clause

The data redaction function expression for the column. See CREATE REDACTION POLICY for details.

scope_value

The scope identifies the query part to apply redaction for the column. See CREATE REDACTION POLICY for the details.

exception_value

The exception identifies the query part where redaction are exempted. See CREATE REDACTION POLICY for the details.

Examples

Update the data redaction policy called redact_policy_personal_info on the table named employees:

```sql
ALTER REDACTION POLICY redact_policy_personal_info ON employees
FOR (session_user != 'hr' AND session_user != 'manager');
```

To update the data redaction function for the column ssn in the same policy:

```sql
ALTER REDACTION POLICY redact_policy_personal_info ON employees
MODIFY COLUMN ssn USING
redact_ssn_new(ssn);
```
## 9.7.4 Removing a data redaction policy

The `DROP REDACTION POLICY` command removes a data redaction policy from a table.

### Synopsis

```
DROP REDACTION POLICY [ IF EXISTS ] <name> ON <table_name>
[ CASCADE | RESTRICT ]
```

### Description

`DROP REDACTION POLICY` removes the specified data redaction policy from the table.

To use `DROP REDACTION POLICY`, you must own the table that the redaction policy applies to.

### Parameters

- **IF EXISTS**
  - Don’t throw an error if the data redaction policy doesn’t exist. A notice is issued in this case.
  
- **name**
  - The name of the data redaction policy to drop.
  
- **table_name**
  - The optionally schema-qualified name of the table that the data redaction policy is on.

- **CASCADE**
  - **RESTRICT**
    - These keywords don’t have any effect, as there are no dependencies on the data redaction policies.

### Examples

To drop the data redaction policy called `redact_policy_personal_info` on the table named `employees`:

```
DROP REDACTION POLICY redact_policy_personal_info ON employees;
```

### Compatibility

`ALTER REDACTION POLICY` is an EDB extension.
DROP REDACTION POLICY is an EDB extension.

See also

CREATE REDACTION POLICY, ALTER REDACTION POLICY

9.7.5 Data redaction system catalogs

System catalogs store the redaction policy information.

edb_redaction_column

The `edb_redaction_column` system catalog stores information about the data redaction policy attached to the columns of a table.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>oid</td>
<td></td>
<td>Row identifier (hidden attribute, must be explicitly selected)</td>
</tr>
<tr>
<td>rdpolicyid</td>
<td>oid</td>
<td><code>edb_redaction_policy.oid</code></td>
<td>The data redaction policy that applies to the described column</td>
</tr>
<tr>
<td>rdrelid</td>
<td>oid</td>
<td><code>pg_class.oid</code></td>
<td>The table that the described column belongs to</td>
</tr>
<tr>
<td>rdattnum</td>
<td>int2</td>
<td><code>pg_attribute.attnum</code></td>
<td>The number of the described column</td>
</tr>
<tr>
<td>rdscope</td>
<td>int2</td>
<td></td>
<td>The redaction scope: 1 = query, 2 = top_tlist, 4 = top_tlist_or_error</td>
</tr>
<tr>
<td>rdexception</td>
<td>int2</td>
<td></td>
<td>The redaction exception: 8 = none, 16 = equal, 32 = leakproof</td>
</tr>
<tr>
<td>rdfunexpr</td>
<td><code>pg_node_tree</code></td>
<td></td>
<td>Data redaction function expression</td>
</tr>
</tbody>
</table>

Note

The described column is redacted if the redaction policy `edb_redaction_column.rdpolicyid` on the table is enabled and the redaction policy expression `edb_redaction_policy.rdexpr` evaluates to `true`.

edb_redaction_policy

The catalog `edb_redaction_policy` stores information about the redaction policies for tables.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>oid</td>
<td></td>
<td>Row identifier (hidden attribute, must be explicitly selected)</td>
</tr>
<tr>
<td>rdname</td>
<td>name</td>
<td></td>
<td>The name of the data redaction policy</td>
</tr>
<tr>
<td>rdrelid</td>
<td>oid</td>
<td><code>pg_class.oid</code></td>
<td>The table to which the data redaction policy applies</td>
</tr>
<tr>
<td>rdenable</td>
<td>boolean</td>
<td></td>
<td>Is the data redaction policy enabled?</td>
</tr>
<tr>
<td>rdexpr</td>
<td><code>pg_node_tree</code></td>
<td></td>
<td>The data redaction policy expression</td>
</tr>
</tbody>
</table>

Note

The data redaction policy applies for the table if it’s enabled and the expression ever evaluated true.

9.8 Controlling data access (Virtual Private Database)

Virtual Private Database is a type of fine-grained access control using security policies. Fine-grained access control means that you can control access to data down to specific rows as defined by the security policy.

The rules that encode a security policy are defined in a policy function. A policy function is an SPL function with certain input parameters and return value. The security policy is the named association of the policy function to a particular database object, typically a table.
In EDB Postgres Advanced Server, you can write the policy function in any language it supports, such as SQL and PL/pgSQL, in addition to SPL.

Note

The database objects currently supported by EDB Postgres Advanced Server Virtual Private Database are tables. You can apply policies to views or synonyms.

The following are advantages of using Virtual Private Database:

- It provides a fine-grained level of security. Database-object-level privileges given by the `GRANT` command determine access privileges to the entire instance of a database object. Virtual Private Database provides access control for the individual rows of a database object instance.
- You can apply a different security policy depending on the type of SQL command (INSERT, UPDATE, DELETE, or SELECT).
- The security policy can vary dynamically for each applicable SQL command affecting the database object. Factors such as the session user of the application accessing the database object affect the security policy.
- Invoking the security policy is transparent to all applications that access the database object. You don't have to modify individual applications to apply the security policy.
- After you enable a security policy, no application (including new applications) can circumvent the security policy except by the system privilege described in the note that follows. Even superusers can’t circumvent the security policy except by the noted system privilege.

Note

The only way you can circumvent security policies is if the user is granted `EXEMPT ACCESS POLICY` system privilege. Use extreme care when granting the `EXEMPT ACCESS POLICY` privilege. A user with this privilege is exempted from all policies in the database.

The `DBMS_RLS` package provides procedures to create policies, remove policies, enable policies, and disable policies.

10 Managing performance

EDB Postgres Advanced Server extends Postgres with features designed to help database administrators manage performance.

10.1 Using the dynamic runtime instrumentation tools architecture (DRITA)

The Dynamic Runtime Instrumentation Tools Architecture (DRITA) enables a DBA to query catalog views to determine the wait events that affect the performance of individual sessions or the whole system. DRITA records the number of times each event occurs as well as the time spent waiting. You can use this information to diagnose performance problems.

DRITA compares snapshots to evaluate the performance of a system. A snapshot is a saved set of system performance data at a given point in time. A unique ID number identifies each snapshot. You can use snapshot ID numbers with DRITA reporting functions to return system performance statistics. DRITA consumes minimal system resources.

10.1.1 Taking a snapshot

EDB Postgres Advanced Server’s `postgresql.conf` file includes a configuration parameter named `timed_statistics` that controls collecting timing data. The valid parameter values are `TRUE` or `FALSE`. The default value is `FALSE`.

`timed_statistics` is a dynamic parameter that you can modify in the `postgresql.conf` file or while a session is in progress. To enable DRITA, you must either:

1. Modify the `postgresql.conf` file, setting the `timed_statistics` parameter to `TRUE`.

2. Connect to the server with the EDB-PSQL client and invoke the command:

```
SET timed_statistics = TRUE
```

3. After modifying the `timed_statistics` parameter, take a starting snapshot. A snapshot captures the current state of each timer and event counter. The server compares the starting snapshot to a later snapshot to gauge system performance. Use the `edbsnap()` function to take the beginning snapshot:

```
edb=# SELECT * FROM edbsnap();
edbsnap
```

```
Statement processed.
(1 row)
```

4. Run the workload that you want to evaluate. When the workload is complete or at a strategic point during the workload, take another snapshot:

```
edb=# SELECT * FROM edbsnap();
edbsnap
```

EDB Postgres Advanced Server (EPAS)

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
You can capture multiple snapshots during a session. Finally, you can use the DRITA functions and reports to manage and compare the snapshots to evaluate performance information.

### 10.1.2 Using DRITA functions

You can use DRITA functions to gather wait information and manage snapshots. DRITA functions are fully supported by EDB Postgres Advanced Server whether your installation is made compatible with Oracle databases or is in PostgreSQL-compatible mode.

#### Retrieving a list of current snapshots (get_snaps)

The `get_snaps()` function returns a list of the current snapshots. The signature is:

```sql
get_snaps()
```

This example uses the `get_snaps()` function to display a list of snapshots:

```sql
SELECT * FROM get_snaps();
```

```
1 25-JUL-18 09:49:04.224597
2 25-JUL-18 09:49:09.310395
3 25-JUL-18 09:49:14.378728
4 25-JUL-18 09:49:19.448875
5 25-JUL-18 09:49:24.52103
6 25-JUL-18 09:49:29.586889
7 25-JUL-18 09:49:34.65529
8 25-JUL-18 09:49:39.723095
9 25-JUL-18 09:49:44.788392
10 25-JUL-18 09:49:49.855821
11 25-JUL-18 09:49:54.919954
12 25-JUL-18 09:49:59.987707
```

The first column in the result list displays the snapshot identifier. The second column displays the date and time that the snapshot was captured.

#### Retrieving system wait information (sys_rpt)

The `sys_rpt()` function returns system wait information. The signature is:

```sql
sys_rpt(<beginning_id>, <ending_id>, <top_n>)
```

**Parameters**

`beginning_id`

An integer value that represents the beginning session identifier.

`ending_id`

An integer value that represents the ending session identifier.

`top_n`

The number of rows to return.

This example shows a call to the `sys_rpt()` function:

```sql
SELECT * FROM sys_rpt(9, 10, 10);
```
### Results

The information displayed in the result set includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT NAME</td>
<td>The name of the wait</td>
</tr>
<tr>
<td>COUNT</td>
<td>The number of times that the wait event occurred</td>
</tr>
<tr>
<td>WAIT TIME</td>
<td>The time of the wait event in seconds</td>
</tr>
<tr>
<td>% WAIT</td>
<td>The percentage of the total wait time used by this wait for this session</td>
</tr>
</tbody>
</table>

### Retrieving session wait information (sess_rpt)

The `sess_rpt()` function returns session wait information. The signature is:

```sql
sess_rpt(<beginning_id>, <ending_id>, <top_n>)
```

#### Parameters

- **<beginning_id>**
  - An integer value that represents the beginning session identifier.

- **<ending_id>**
  - An integer value that represents the ending session identifier.

- **<top_n>**
  - The number of rows to return.

This example shows a call to the `sess_rpt()` function:

```sql
SELECT * FROM sess_rpt(8, 9, 10);
```

### Results

<table>
<thead>
<tr>
<th>ID</th>
<th>USER</th>
<th>WAIT NAME</th>
<th>COUNT</th>
<th>TIME</th>
<th>% WAIT SES</th>
<th>% WAIT ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3581</td>
<td>enterprise</td>
<td>wal flush</td>
<td>8359</td>
<td>1.357593</td>
<td>30.62</td>
<td></td>
</tr>
<tr>
<td>3581</td>
<td>enterprise</td>
<td>wal write</td>
<td>8358</td>
<td>1.349153</td>
<td>30.43</td>
<td></td>
</tr>
<tr>
<td>3581</td>
<td>enterprise</td>
<td>wal file sync</td>
<td>8358</td>
<td>1.286437</td>
<td>29.02</td>
<td></td>
</tr>
<tr>
<td>3581</td>
<td>enterprise</td>
<td>query plan</td>
<td>33439</td>
<td>0.439324</td>
<td>9.91</td>
<td></td>
</tr>
<tr>
<td>3581</td>
<td>enterprise</td>
<td>db file extend</td>
<td>54</td>
<td>0.000585</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>3581</td>
<td>enterprise</td>
<td>db file read</td>
<td>54</td>
<td>0.000097</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>3581</td>
<td>enterprise</td>
<td>other lwlock acquire</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>3581</td>
<td>enterprise</td>
<td>ProcArrayLock</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>3581</td>
<td>enterprise</td>
<td>CLogControlLock</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

(18 rows)
The information displayed in the result set includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The process ID of the session</td>
</tr>
<tr>
<td>USER</td>
<td>The name of the user incurring the wait</td>
</tr>
<tr>
<td>WAIT NAME</td>
<td>The name of the wait event</td>
</tr>
<tr>
<td>COUNT</td>
<td>The number of times that the wait event occurred</td>
</tr>
<tr>
<td>TIME</td>
<td>The length of the wait event in seconds</td>
</tr>
<tr>
<td>% WAIT SES</td>
<td>The percentage of the total wait time used by this wait for this session</td>
</tr>
<tr>
<td>% WAIT ALL</td>
<td>The percentage of the total wait time used by this wait for all sessions</td>
</tr>
</tbody>
</table>

Retrieving session ID information for a specified backend (sessid_rpt)

The `sessid_rpt()` function returns session ID information for a specified backend. The signature is:

```
sessid_rpt(<beginning_id>, <ending_id>, <backend_id>)
```

Parameters

- **beginning_id**
  - An integer value that represents the beginning session identifier.

- **ending_id**
  - An integer value that represents the ending session identifier.

- **backend_id**
  - An integer value that represents the backend identifier.

This example shows a call to `sessid_rpt()`:

```
SELECT * FROM sessid_rpt(8, 9, 3501);
```

```
<table>
<thead>
<tr>
<th>ID</th>
<th>USER</th>
<th>WAIT NAME</th>
<th>COUNT</th>
<th>TIME</th>
<th>% WAIT SES</th>
<th>% WAIT ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3501</td>
<td>enterprise</td>
<td>CLogControlLock</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>3501</td>
<td>enterprise</td>
<td>ProcArrayLock</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>3501</td>
<td>enterprise</td>
<td>db file read</td>
<td>56</td>
<td>0.000541</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>3501</td>
<td>enterprise</td>
<td>db file extend</td>
<td>54</td>
<td>0.000578</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>3501</td>
<td>enterprise</td>
<td>query plan</td>
<td>33413</td>
<td>0.436901</td>
<td>9.87</td>
<td>9.87</td>
</tr>
<tr>
<td>3501</td>
<td>enterprise</td>
<td>wal file sync</td>
<td>8354</td>
<td>1.285607</td>
<td>29.04</td>
<td>29.04</td>
</tr>
<tr>
<td>3501</td>
<td>enterprise</td>
<td>wal write</td>
<td>8354</td>
<td>1.348192</td>
<td>30.46</td>
<td>30.46</td>
</tr>
<tr>
<td>3501</td>
<td>enterprise</td>
<td>wal flush</td>
<td>8354</td>
<td>1.354958</td>
<td>30.61</td>
<td>30.61</td>
</tr>
</tbody>
</table>
```

Results

The information displayed in the result set includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The process ID of the wait</td>
</tr>
<tr>
<td>USER</td>
<td>The name of the user that owns the session</td>
</tr>
<tr>
<td>WAIT NAME</td>
<td>The name of the wait event</td>
</tr>
<tr>
<td>COUNT</td>
<td>The number of times that the wait event occurred</td>
</tr>
<tr>
<td>TIME</td>
<td>The length of the wait in seconds</td>
</tr>
</tbody>
</table>
### Retrieving session wait information for a specified backend (sesshist_rpt)

The `sesshist_rpt()` function returns session wait information for a specified backend. The signature is:

```
sesshist_rpt(<snapshot_id>, <session_id>)
```

#### Parameters

- **snapshot_id**
  - An integer value that identifies the snapshot.

- **session_id**
  - An integer value that represents the session.

This example shows a call to the `sesshist_rpt()` function:

```
SELECT * FROM sesshist_rpt (9, 3501);
```

<table>
<thead>
<tr>
<th>ID</th>
<th>USER</th>
<th>SEQ</th>
<th>WAIT NAME</th>
<th>ELAPSED</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>N/A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_password_history</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_password_history</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_password_history</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_profile</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_profile_name_ind</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_profile_oid_ind</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_profile_password</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_resource_group</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_resource_group_n</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>edb_resource_group_o</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>pg_attribute</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>pg_attribute_relid_a</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>pg_attribute_relid_a</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>pg_auth_members</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3501</td>
<td>enterprise 1</td>
<td>0</td>
<td>query plan</td>
<td>13</td>
<td>pg_auth_members_memb</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Note**

The example was shortened. Over 1300 rows are actually generated.
Results

The information displayed in the result set includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>The system-assigned identifier of the wait</td>
</tr>
<tr>
<td>USER</td>
<td>The name of the user that incurred the wait</td>
</tr>
<tr>
<td>SEQ</td>
<td>The sequence number of the wait event</td>
</tr>
<tr>
<td>WAIT NAME</td>
<td>The name of the wait event</td>
</tr>
<tr>
<td>ELAPSED</td>
<td>The length of the wait event in microseconds</td>
</tr>
<tr>
<td>File</td>
<td>The relfilenode number of the file</td>
</tr>
<tr>
<td>Name</td>
<td>If available, the name of the file name related to the wait event</td>
</tr>
<tr>
<td># of Blks</td>
<td>The block number read or written for a specific instance of the event</td>
</tr>
<tr>
<td>Sum of Blks</td>
<td>The number of blocks read</td>
</tr>
</tbody>
</table>

Purging a range of snapshots from the snapshot tables (purgesnap)

The `purgesnap()` function purges a range of snapshots from the snapshot tables. The signature is:

`purgesnap(<beginning_id>, <ending_id>)`

Parameters

`beginning_id`

An integer value that represents the beginning session identifier.

`ending_id`

An integer value that represents the ending session identifier.

`purgesnap()` removes all snapshots between `beginning_id` and `ending_id` inclusive:

```sql
SELECT * FROM purgesnap(6, 9);
```

```
purgesnap
斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯斯ス
```

A call to the `get_snaps()` function after executing the example shows that snapshots 6 through 9 were purged from the snapshot tables:
Deleting records from the snapshot table (truncsnap)

Use the truncsnap() function to delete all records from the snapshot table. The signature is:

```sql
truncsnap()
```

For example:

```sql
SELECT * FROM truncsnap();
```

Snapshots truncated.

(1 row)

A call to the `get_snaps()` function after calling the `truncsnap()` function shows that all records were removed from the snapshot tables:

```sql
SELECT * FROM get_snaps();
```

(0 rows)

10.1.3 Performance tuning recommendations

Reviewing the reports

To use Dynamic Runtime Instrumentation Tools Architecture (DRITA) reports for performance tuning, review the top five events in a report. Look for any event that takes an especially large percentage of resources. In a streamlined system, user I/O generally makes up the largest number of waits. Evaluate waits in the context of CPU usage and total time. An event might not be significant if it takes two minutes out of a total measurement interval of two hours and the rest of the time is consumed by CPU time. Evaluate the component of response time (CPU "work" time or other "wait" time) that consumes the highest percentage of overall time.

When evaluating events, watch for:

<table>
<thead>
<tr>
<th>Event type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoint waits</td>
<td>Checkpoint waits might indicate that checkpoint parameters need to be adjusted (checkpoint_segments and checkpoint_timeout).</td>
</tr>
<tr>
<td>WAL-related waits</td>
<td>WAL-related waits might indicate wal_buffers are undersized.</td>
</tr>
<tr>
<td>SQL Parse waits</td>
<td>If the number of waits is high, try to use prepared statements.</td>
</tr>
<tr>
<td>db file random reads</td>
<td>If high, check for appropriate indexes and statistics.</td>
</tr>
<tr>
<td>db file random writes</td>
<td>If high, might need to decrease bgwriter_delay.</td>
</tr>
<tr>
<td>btree random lock acquires</td>
<td>Might indicate indexes are being rebuilt. Schedule index builds during less active time.</td>
</tr>
</tbody>
</table>

Also look at the hardware, the operating system, the network, and the application SQL statements in performance reviews.

Event descriptions
The following table lists the basic wait events that are displayed by DRITA.

<table>
<thead>
<tr>
<th>Event name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add in shmem lock acquire</td>
<td>Obsolete/unused.</td>
</tr>
<tr>
<td>bgwriter communication lock acquire</td>
<td>The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.</td>
</tr>
<tr>
<td>btree vacuum lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.</td>
</tr>
<tr>
<td>buffer free list lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory).</td>
</tr>
<tr>
<td>checkpoint lock acquire</td>
<td>A server process has waited for the short-term lock that prevents simultaneous checkpoints.</td>
</tr>
<tr>
<td>checkpoint start lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the bgwriter checkpoint schedule.</td>
</tr>
<tr>
<td>clog control lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the commit log.</td>
</tr>
<tr>
<td>control file lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes write access to the control file. This is usually a low number.</td>
</tr>
<tr>
<td>db file extend</td>
<td>A server process has waited for the operating system while adding a new page to the end of a file.</td>
</tr>
<tr>
<td>db file read</td>
<td>A server process has waited for a read from disk to complete.</td>
</tr>
<tr>
<td>db file write</td>
<td>A server process has waited for a write to disk to complete.</td>
</tr>
<tr>
<td>db file sync</td>
<td>A server process has waited for the operating system to flush all changes to disk.</td>
</tr>
<tr>
<td>first buf mapping lock acquire</td>
<td>The server has waited for a short-term lock that synchronizes access to the shared-buffer mapping table.</td>
</tr>
<tr>
<td>freespace lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the freespace map.</td>
</tr>
<tr>
<td>lwlock acquire</td>
<td>The server has waited for a short-term lock that isn't described elsewhere in this table.</td>
</tr>
<tr>
<td>multi xact gen lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes).</td>
</tr>
<tr>
<td>multi xact member lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes).</td>
</tr>
<tr>
<td>multi xact offset lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes).</td>
</tr>
<tr>
<td>oid gen lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available OID (object ID).</td>
</tr>
<tr>
<td>query plan</td>
<td>The server has computed the execution plan for a SQL statement.</td>
</tr>
<tr>
<td>rel cache init lock acquire</td>
<td>The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.</td>
</tr>
<tr>
<td>shmem index lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the shared-memory map.</td>
</tr>
<tr>
<td>signal lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the cache invalidation state.</td>
</tr>
<tr>
<td>sql parse</td>
<td>The server has parsed a SQL statement.</td>
</tr>
<tr>
<td>subtrans control lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the subtransaction log.</td>
</tr>
<tr>
<td>tablespace create lock acquire</td>
<td>The server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands.</td>
</tr>
<tr>
<td>two phase state lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the list of prepared transactions.</td>
</tr>
<tr>
<td>wal insert lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number can indicate that WAL buffers are sized too small.</td>
</tr>
<tr>
<td>wal write lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes write-ahead log flushes.</td>
</tr>
<tr>
<td>wal file sync</td>
<td>The server has waited for the write-ahead log to sync to disk. This is related to the wal_sync_method parameter which, by default, is 'sync'. You can gain better performance by changing this parameter to open_sync.</td>
</tr>
<tr>
<td>wal flush</td>
<td>The server has waited for the write-ahead log to flush to disk.</td>
</tr>
<tr>
<td>wal write</td>
<td>The server has waited for a write to the write-ahead log buffer. Expect this value to be high.</td>
</tr>
<tr>
<td>xid gen lock acquire</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available transaction ID.</td>
</tr>
</tbody>
</table>

Wait events related to lightweight locks

When wait events occur for lightweight locks, DRITA displays them as well. It uses a lightweight lock to protect a particular data structure in shared memory.

Certain wait events can be due to the server process waiting for one of a group of related lightweight locks, which is referred to as a lightweight lock tranche. DRITA doesn't display individual lightweight lock tranches, but it displays their summation with a single event named other lwlock acquire.
For a list and description of lightweight locks displayed by DRITA, see the PostgreSQL core documentation. Under Viewing Statistics, see the Wait Event Type table for more details.

This example displays lightweight locks ProcArrayLock, CLogControlLock, WALBufMappingLock, and XidGenLock.

```
postgres=# select * from sys_rpt(40,78,20);
```

```
<table>
<thead>
<tr>
<th>WAIT NAME</th>
<th>COUNT</th>
<th>WAIT TIME</th>
<th>% WAIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>wal flush</td>
<td>56107</td>
<td>44.456494</td>
<td>47.65</td>
</tr>
<tr>
<td>db file read</td>
<td>66123</td>
<td>19.543968</td>
<td>20.95</td>
</tr>
<tr>
<td>wal write</td>
<td>32886</td>
<td>12.780866</td>
<td>13.70</td>
</tr>
<tr>
<td>wal file sync</td>
<td>32833</td>
<td>11.792972</td>
<td>12.64</td>
</tr>
<tr>
<td>query plan</td>
<td>223576</td>
<td>4.539186</td>
<td>4.87</td>
</tr>
<tr>
<td>db file extend</td>
<td>2339</td>
<td>0.087038</td>
<td>0.09</td>
</tr>
<tr>
<td>other lwlock acquire</td>
<td>402</td>
<td>0.066591</td>
<td>0.07</td>
</tr>
<tr>
<td>ProcArrayLock</td>
<td>135</td>
<td>0.012942</td>
<td>0.01</td>
</tr>
<tr>
<td>CLogControlLock</td>
<td>212</td>
<td>0.010333</td>
<td>0.01</td>
</tr>
<tr>
<td>WALBufMappingLock</td>
<td>47</td>
<td>0.006068</td>
<td>0.01</td>
</tr>
<tr>
<td>XidGenLock</td>
<td>53</td>
<td>0.005296</td>
<td>0.01</td>
</tr>
</tbody>
</table>
(13 rows)
```

Wait events related to product features

DRITA also displays wait events that are related to certain EDB Postgres Advanced Server product features. These events and the other lwlock acquire event are listed in the following table.

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BulkLoadLock</td>
<td>The server has waited for access related to EDB*Loader.</td>
</tr>
<tr>
<td>EDBResourceManagerLock</td>
<td>The server has waited for access related to EDB Resource Manager.</td>
</tr>
<tr>
<td>other lwlock acquire</td>
<td>Summation of waits for lightweight lock tranches.</td>
</tr>
</tbody>
</table>

10.1.4 Simulating Statspack AWR reports

When taking a snapshot, performance data from system catalog tables is saved into history tables. The following functions return information comparable to the information contained in an Oracle Statspack/Automatic Workload Repository (AWR) report. These reporting functions report on the differences between two given snapshots:

- `stat_db_rpt()`
- `stat_tables_rpt()`
- `statio_tables_rpt()`
- `stat_indexes_rpt()`
- `statio_indexes_rpt()`

You can execute the reporting functions individually or you can execute all five functions by calling the `edbreport()` function.

```
edbreport()
```

The `edbreport()` function includes data from the other reporting functions, plus system information. The signature is:

```
edbreport(<beginning_id>, <ending_id>)
```

Parameters

- `beginning_id`  
  An integer value that represents the beginning session identifier.

- `ending_id`  
  An integer value that represents the ending session identifier.

EDB Postgres Advanced Server (EPAS)

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The call to the `edbreport()` function returns a composite report that contains system information and the reports returned by the other statspack functions:

```sql
SELECT * FROM edbreport(9, 10);
```

```
edbreport

EnterpriseDB Report for database acctg 25-JUL-18
Version: PostgreSQL 14.0 (EnterpriseDB EDB Postgres Advanced Server 14.0.0) on x86_64-pc-linux-gnu,
compiled by gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-18), 64-bit

Begin snapshot: 9 at 25-JUL-18 09:49:44.788392
End snapshot: 10 at 25-JUL-18 09:49:49.855821

Size of database acctg is 173 MB
Tablespace: pg_default Size: 231 MB Owner: enterprisedb
Tablespace: pg_global Size: 719 kB Owner: enterprisedb
Schema: pg_toast_temp_1 Size: 0 bytes Owner: enterprisedb
Schema: public Size: 158 MB Owner: enterprisedb

Report introduction

The information displayed in the report introduction includes the database name and version, the current date, the beginning and ending snapshot date and times, database and tablespace details, and schema information.

```

Top 10 Relations by pages

<table>
<thead>
<tr>
<th>TABLE</th>
<th>RELPAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgbench_accounts</td>
<td>16394</td>
</tr>
<tr>
<td>pgbench_history</td>
<td>391</td>
</tr>
<tr>
<td>pg_proc</td>
<td>145</td>
</tr>
<tr>
<td>pg_attribute</td>
<td>92</td>
</tr>
<tr>
<td>pg_depend</td>
<td>81</td>
</tr>
<tr>
<td>pg_collation</td>
<td>60</td>
</tr>
<tr>
<td>edb$stat_all_indexes</td>
<td>46</td>
</tr>
<tr>
<td>edb$statio_all_indexes</td>
<td>46</td>
</tr>
<tr>
<td>pg_description</td>
<td>44</td>
</tr>
<tr>
<td>edb$stat_all_tables</td>
<td>29</td>
</tr>
</tbody>
</table>

Top 10 Indexes by pages

<table>
<thead>
<tr>
<th>INDEX</th>
<th>RELPAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgbench_accounts_pkey</td>
<td>2745</td>
</tr>
<tr>
<td>pg_depend_reference_index</td>
<td>68</td>
</tr>
<tr>
<td>pg_depend_depender_index</td>
<td>63</td>
</tr>
<tr>
<td>pg_proc_prfname_args_nsp_index</td>
<td>53</td>
</tr>
<tr>
<td>pg_attribute_relid_attnam_index</td>
<td>25</td>
</tr>
<tr>
<td>pg_description_o_c_o_index</td>
<td>24</td>
</tr>
<tr>
<td>pg_attribute_relid_attnum_index</td>
<td>17</td>
</tr>
<tr>
<td>pg_proc_oid_index</td>
<td>14</td>
</tr>
<tr>
<td>pg_collation_name_enc_nsp_index</td>
<td>12</td>
</tr>
<tr>
<td>edb$stat_idx_pk</td>
<td>10</td>
</tr>
</tbody>
</table>

__OUTPUT__
Top 10 Indexes by pages

The information displayed in the **Top 18 Indexes by pages** section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX</td>
<td>The name of the index</td>
</tr>
<tr>
<td>RELPAGES</td>
<td>The number of pages in the Index</td>
</tr>
</tbody>
</table>

---

__OUTPUT__

Top 10 Relations by DML

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>UPDATES</th>
<th>DELETES</th>
<th>INSERTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>pgbench_accounts</td>
<td>117209</td>
<td>0</td>
<td>1000000</td>
</tr>
<tr>
<td>public</td>
<td>pgbench_tellers</td>
<td>117209</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>public</td>
<td>pgbench_branches</td>
<td>117209</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>public</td>
<td>pgbench_history</td>
<td>0</td>
<td>0</td>
<td>117209</td>
</tr>
</tbody>
</table>

---

Top 10 Relations by DML

The information displayed in the **Top 10 Relations by DML** section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>The name of the schema in which the table resides</td>
</tr>
<tr>
<td>RELATION</td>
<td>The name of the table</td>
</tr>
<tr>
<td>UPDATES</td>
<td>The number of UPDATES performed on the table</td>
</tr>
<tr>
<td>DELETES</td>
<td>The number of DELETES performed on the table</td>
</tr>
<tr>
<td>INSERTS</td>
<td>The number of INSERTS performed on the table</td>
</tr>
</tbody>
</table>

---

__OUTPUT__

DATA from pg_stat_database

<table>
<thead>
<tr>
<th>DATABASE</th>
<th>NUMBACKENDS</th>
<th>XACT COMMIT</th>
<th>XACT ROLLBACK</th>
<th>BLKS READ</th>
<th>BLKS HIT</th>
<th>HIT RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>acctg</td>
<td>0</td>
<td>8261</td>
<td>0</td>
<td>117</td>
<td>127985</td>
<td>99.91</td>
</tr>
</tbody>
</table>

---

DATA from pg_stat_database

The information displayed in the **DATA from pg_stat_database** section of the report includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE</td>
<td>The name of the database.</td>
</tr>
<tr>
<td>NUMBACKENDS</td>
<td>Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state. All other columns return the accumulated values since the last reset.</td>
</tr>
<tr>
<td>XACT COMMIT</td>
<td>Number of transactions in this database that were committed.</td>
</tr>
<tr>
<td>XACT ROLLBACK</td>
<td>Number of transactions in this database that were rolled back.</td>
</tr>
<tr>
<td>BLKS READ</td>
<td>Number of disk blocks read in this database.</td>
</tr>
<tr>
<td>BLKS HIT</td>
<td>Number of times disk blocks were found already in the buffer cache when a read wasn't necessary.</td>
</tr>
<tr>
<td>HIT RATIO</td>
<td>The percentage of times that a block was found in the shared buffer cache.</td>
</tr>
</tbody>
</table>

---

__OUTPUT__

DATA from pg_buffercache

<table>
<thead>
<tr>
<th>RELATION</th>
<th>BUFFERS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DATA from pg_buffercache

The information displayed in the DATA from pg_buffercache section of the report includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELATION</td>
<td>The name of the table</td>
</tr>
<tr>
<td>BUFFERS</td>
<td>The number of shared buffers used by the relation</td>
</tr>
</tbody>
</table>

**Note**

To obtain the report for DATA from pg_buffercache, the pg_buffercache module must be installed in the database. Perform the installation using the CREATE EXTENSION command.

For more information on the CREATE EXTENSION command, see the PostgreSQL core documentation.

```
__OUTPUT__

DATA from pg_stat_all_tables ordered by seq scan

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>SEQ</th>
<th>SCAN</th>
<th>REL TUP</th>
<th>READ</th>
<th>INS</th>
<th>UPD</th>
<th>DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>pgbench_branches</td>
<td>8258</td>
<td></td>
<td>82580</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>pgbench_tellers</td>
<td>8258</td>
<td></td>
<td>825800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pg_class</td>
<td>7</td>
<td></td>
<td>3969</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pg_index</td>
<td>5</td>
<td></td>
<td>950</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pg_namespace</td>
<td>4</td>
<td></td>
<td>144</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pg_database</td>
<td>2</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pg_am</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>pg_authid</td>
<td>1</td>
<td></td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>callback_queue_table</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>edb$session_wait_history</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>125</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DATA from pg_stat_all_tables ordered by seq scan

The information displayed in the DATA from pg_stat_all_tables ordered by seq scan section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>The name of the schema in which the table resides</td>
</tr>
<tr>
<td>RELATION</td>
<td>The name of the table</td>
</tr>
</tbody>
</table>
The information displayed in the **DATA from pg_stat_all_tables ordered by rel tup read** section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>The name of the schema in which the table resides</td>
</tr>
<tr>
<td>RELATION</td>
<td>The name of the table</td>
</tr>
<tr>
<td>SEQ SCAN</td>
<td>The number of sequential scans performed on the table</td>
</tr>
<tr>
<td>REL TUP READ</td>
<td>The number of tuples read from the table</td>
</tr>
<tr>
<td>IDX SCAN</td>
<td>The number of index scans performed on the table</td>
</tr>
<tr>
<td>IDX TUP READ</td>
<td>The number of index tuples read</td>
</tr>
<tr>
<td>INS</td>
<td>The number of rows inserted</td>
</tr>
<tr>
<td>UPD</td>
<td>The number of rows updated</td>
</tr>
<tr>
<td>DEL</td>
<td>The number of rows deleted</td>
</tr>
</tbody>
</table>

**DATA from pg_stat_all_tables ordered by rel tup read**

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>SEQ SCAN</th>
<th>REL TUP READ</th>
<th>IDX TUP READ</th>
<th>INS</th>
<th>UPD</th>
<th>DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>pgbench_tellers</td>
<td>8258</td>
<td>825800</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>public</td>
<td>pgbench_branches</td>
<td>8258</td>
<td>825800</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>7</td>
<td>3969</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_index</td>
<td>5</td>
<td>958</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_namespace</td>
<td>4</td>
<td>144</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_database</td>
<td>2</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_authid</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_am</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>callback_queue_table</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$session_wait_history</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>0</td>
<td>125</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The information displayed in the **DATA from pg_statio_all_tables** section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SCHEMA</strong></td>
<td>The name of the schema in which the table resides</td>
</tr>
<tr>
<td><strong>RELATION</strong></td>
<td>The name of the table</td>
</tr>
<tr>
<td><strong>HEAP READ</strong></td>
<td>The number of heap blocks read</td>
</tr>
<tr>
<td><strong>HEAP HIT</strong></td>
<td>The number of heap blocks hit</td>
</tr>
<tr>
<td><strong>IDX READ</strong></td>
<td>The number of index blocks read</td>
</tr>
<tr>
<td><strong>IDX HIT</strong></td>
<td>The number of index blocks hit</td>
</tr>
<tr>
<td><strong>TOAST READ</strong></td>
<td>The number of toast blocks read</td>
</tr>
<tr>
<td><strong>TOAST HIT</strong></td>
<td>The number of toast blocks hit</td>
</tr>
<tr>
<td><strong>TIDX READ</strong></td>
<td>The number of toast index blocks read</td>
</tr>
<tr>
<td><strong>TIDX HIT</strong></td>
<td>The number of toast index blocks hit</td>
</tr>
</tbody>
</table>

**DATA from pg_statio_all_tables**

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>HEAP</th>
<th>HEAP</th>
<th>IDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOAST</td>
<td>TOAST</td>
<td>TIDX</td>
<td>TIDX</td>
<td>READ</td>
</tr>
<tr>
<td>HIT</td>
<td>HIT</td>
<td>READ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>HEAP</th>
<th>HEAP</th>
<th>IDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>公开</td>
<td>pgbench_accounts</td>
<td>32</td>
<td>25816</td>
<td>0</td>
</tr>
<tr>
<td>公开</td>
<td>pgbench_tellers</td>
<td>0</td>
<td>24774</td>
<td>0</td>
</tr>
<tr>
<td>公开</td>
<td>pgbench_branches</td>
<td>0</td>
<td>16516</td>
<td>0</td>
</tr>
<tr>
<td>公开</td>
<td>pgbench_history</td>
<td>0</td>
<td>53</td>
<td>8364</td>
</tr>
<tr>
<td>公开</td>
<td>pg_class</td>
<td>0</td>
<td>199</td>
<td>0</td>
</tr>
<tr>
<td>公开</td>
<td>pg_attribute</td>
<td>0</td>
<td>198</td>
<td>0</td>
</tr>
<tr>
<td>公开</td>
<td>pg_proc</td>
<td>0</td>
<td>75</td>
<td>0</td>
</tr>
<tr>
<td>公开</td>
<td>pg_index</td>
<td>0</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>公开</td>
<td>pg_amop</td>
<td>0</td>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>公开</td>
<td>pg_namespace</td>
<td>0</td>
<td>28</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>INDEX</th>
<th>INDEX</th>
<th>IDX TUP</th>
<th>READ</th>
<th>IDX TUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOAST</td>
<td>TOAST</td>
<td>TIDX</td>
<td>TIDX</td>
<td>READ</td>
<td>HIT</td>
<td>READ</td>
</tr>
<tr>
<td>HIT</td>
<td>HIT</td>
<td>READ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DATA from pg_stat_all_indexes**

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>INDEX</th>
<th>INDEX</th>
<th>IDX TUP</th>
<th>READ</th>
<th>IDX TUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>公开</td>
<td>pgbench_accounts</td>
<td>196679</td>
<td>16516</td>
<td>196</td>
<td>402</td>
<td>402</td>
</tr>
<tr>
<td>公开</td>
<td>pg_attribute</td>
<td>16516</td>
<td>16516</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>公开</td>
<td>pg_proc</td>
<td>70</td>
<td>70</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>公开</td>
<td>pg_class</td>
<td>61</td>
<td>61</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>公开</td>
<td>pg_class_oid_index</td>
<td>31</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>公开</td>
<td>pg_type</td>
<td>19</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**__OUTPUT__**

**DATA from pg_statall_indexes**

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>INDEX</th>
<th>INDEX</th>
<th>IDX TUP</th>
<th>READ</th>
<th>IDX TUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>公开</td>
<td>pgbench_accounts</td>
<td>196679</td>
<td>16516</td>
<td>196</td>
<td>402</td>
<td>402</td>
</tr>
<tr>
<td>公开</td>
<td>pg_attribute</td>
<td>16516</td>
<td>16516</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>公开</td>
<td>pg_proc</td>
<td>70</td>
<td>70</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>公开</td>
<td>pg_class</td>
<td>61</td>
<td>61</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>公开</td>
<td>pg_class_oid_index</td>
<td>31</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>公开</td>
<td>pg_type</td>
<td>19</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The information displayed in the **DATA from pg_stat_all_indexes** section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>The name of the schema in which the index resides</td>
</tr>
<tr>
<td>RELATION</td>
<td>The name of the table on which the index is defined</td>
</tr>
<tr>
<td>INDEX</td>
<td>The name of the index</td>
</tr>
<tr>
<td>IDX SCAN</td>
<td>The number of indexes scans initiated on this index</td>
</tr>
<tr>
<td>IDX TUP READ</td>
<td>Number of index entries returned by scans on this index</td>
</tr>
<tr>
<td>IDX TUP FETCH</td>
<td>Number of live table rows fetched by simple index scans using this index</td>
</tr>
</tbody>
</table>

**DATA from pg_statio_all_indexes**

The information displayed in the **DATA from pg_statio_all_indexes** section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>The name of the schema in which the index resides</td>
</tr>
<tr>
<td>RELATION</td>
<td>The name of the table on which the index is defined</td>
</tr>
<tr>
<td>INDEX</td>
<td>The name of the index</td>
</tr>
<tr>
<td>IDX BLKS READ IDX BLKS HIT</td>
<td></td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>INDEX</th>
<th>IDX BLKS READ IDX BLKS HIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>pgbench_accounts</td>
<td>pgbench_accounts_pkey</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_attribute</td>
<td>pg_attribute_relid_attnum_index</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$stat_idx_pk</td>
<td>edb$stat_all_indexes</td>
<td>1</td>
</tr>
<tr>
<td>sys</td>
<td>edb$statio_idx_pk</td>
<td>edb$statio_all_indexes</td>
<td>1</td>
</tr>
<tr>
<td>sys</td>
<td>edb$statio_tab_pk</td>
<td>edb$statio_all_tables</td>
<td>2</td>
</tr>
<tr>
<td>sys</td>
<td>edb$statio_tab_pk</td>
<td>edb$statio_all_tables</td>
<td>0</td>
</tr>
<tr>
<td>sys</td>
<td>edb$session_wait_history</td>
<td>session_waits_hist_pk</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_proc</td>
<td>pg_proc_oid_index</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>pg_class_oid_index</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>pg_class_relnam_nsp_index</td>
<td>0</td>
</tr>
</tbody>
</table>
```
### Column name | Description
---|---
RELATION | The name of the table on which the index is defined
INDEX | The name of the index
IDX BLKS READ | The number of index blocks read
IDX BLKS HIT | The number of index blocks hit

### System Wait Information

The information displayed in the **System Wait Information** section includes:

<table>
<thead>
<tr>
<th>WAIT NAME</th>
<th>COUNT</th>
<th>WAIT TIME</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>wal flush</td>
<td>8359</td>
<td>1.357593</td>
<td>30.62</td>
</tr>
<tr>
<td>wal write</td>
<td>8358</td>
<td>1.349153</td>
<td>30.43</td>
</tr>
<tr>
<td>wal file sync</td>
<td>8358</td>
<td>1.286437</td>
<td>29.02</td>
</tr>
<tr>
<td>query plan</td>
<td>33439</td>
<td>0.439324</td>
<td>9.91</td>
</tr>
<tr>
<td>db file extend</td>
<td>54</td>
<td>0.000585</td>
<td>0.81</td>
</tr>
<tr>
<td>db file read</td>
<td>31</td>
<td>0.000307</td>
<td>0.01</td>
</tr>
<tr>
<td>other lWlock acquire</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
</tr>
<tr>
<td>CLogControlLock</td>
<td>0</td>
<td>0.000000</td>
<td>0.00</td>
</tr>
</tbody>
</table>

### Database Parameters from postgresql.conf

<table>
<thead>
<tr>
<th>CONTEXT</th>
<th>MINVAL</th>
<th>MAXVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_system_table_mods</td>
<td>off</td>
<td></td>
</tr>
<tr>
<td>application_name</td>
<td>psql.bin</td>
<td></td>
</tr>
<tr>
<td>array_nulls</td>
<td>on</td>
<td>on</td>
</tr>
<tr>
<td>authentication_timeout</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>autovacuum_analyze_scale_factor</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>autovacuum_analyze_threshold</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>autovacuum_freeze_max_age</td>
<td>200000000</td>
<td>200000000</td>
</tr>
<tr>
<td>autovacuum_max_workers</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>autovacuum_multixact_freeze_max_age</td>
<td>400000000</td>
<td>400000000</td>
</tr>
</tbody>
</table>

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Database Parameters from postgresql.conf

The information displayed in the Database Parameters from postgresql.conf section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAMETER</td>
<td>The name of the parameter</td>
</tr>
<tr>
<td>SETTING</td>
<td>The current value assigned to the parameter</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>The context required to set the parameter value</td>
</tr>
<tr>
<td>MINVAL</td>
<td>The minimum value allowed for the parameter</td>
</tr>
<tr>
<td>MAXVAL</td>
<td>The maximum value allowed for the parameter</td>
</tr>
</tbody>
</table>

stat_db_rpt()

The signature is:

```
stat_db_rpt(<beginning_id>, <ending_id>)
```

Parameters

beginning_id

An integer value that represents the beginning session identifier.

ending_id

An integer value that represents the ending session identifier.

This example shows the stat_db_rpt() function:

```
SELECT * FROM stat_db_rpt(9, 10);
```

DATA from pg_stat_database

The information displayed in the DATA from pg_stat_database section of the report includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE</td>
<td>The name of the database.</td>
</tr>
<tr>
<td>NUMBACKENDS</td>
<td>Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state. All other columns return the accumulated values since the last reset.</td>
</tr>
</tbody>
</table>
**XACT COMMIT**
The number of transactions in this database that were committed.

**XACT ROLLBACK**
The number of transactions in this database that were rolled back.

**BLKS READ**
The number of blocks read.

**BLKS HIT**
The number of blocks hit.

**HIT RATIO**
The percentage of times that a block was found in the shared buffer cache.

---

**stat_tables_rpt()**

The signature is:

```
function_name(<beginning_id>, <ending_id>, <top_n>, <scope>)
```

**Parameters**

**beginning_id**
An integer value that represents the beginning session identifier.

**ending_id**
An integer value that represents the ending session identifier.

**top_n**
The number of rows to return.

**scope**
Determines the tables the function returns statistics about. Specify **SYS**, **USER**, or **ALL**:

- Use **SYS** to return information about system-defined tables. A table is considered a system table if it's stored in the **pg_catalog**, **information_schema**, or **sys** schema.
- Use **USER** to return information about user-defined tables.
- Use **ALL** to return information about all tables.

The *stat_tables_rpt()* function returns a two-part report. The first portion of the report contains:

```
SELECT * FROM stat_tables_rpt(8, 9, 10, 'ALL');
```

---

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XACT COMMIT</td>
<td>The number of transactions in this database that were committed.</td>
</tr>
<tr>
<td>XACT ROLLBACK</td>
<td>The number of transactions in this database that were rolled back.</td>
</tr>
<tr>
<td>BLKS READ</td>
<td>The number of blocks read.</td>
</tr>
<tr>
<td>BLKS HIT</td>
<td>The number of blocks hit.</td>
</tr>
<tr>
<td>HIT RATIO</td>
<td>The percentage of times that a block was found in the shared buffer cache.</td>
</tr>
</tbody>
</table>
DATA from pg_stat_all_tables ordered by seq scan

The information displayed in the **DATA from pg_stat_all_tables ordered by seq scan** section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>The name of the schema in which the table resides</td>
</tr>
<tr>
<td>RELATION</td>
<td>The name of the table</td>
</tr>
<tr>
<td>SEQ SCAN</td>
<td>The number of sequential scans on the table</td>
</tr>
<tr>
<td>REL TUP READ</td>
<td>The number of tuples read from the table</td>
</tr>
<tr>
<td>IDX SCAN</td>
<td>The number of index scans performed on the table</td>
</tr>
<tr>
<td>IDX TUP READ</td>
<td>The number of index tuples read from the table</td>
</tr>
<tr>
<td>INS</td>
<td>The number of rows inserted</td>
</tr>
<tr>
<td>UPD</td>
<td>The number of rows updated</td>
</tr>
<tr>
<td>DEL</td>
<td>The number of rows deleted</td>
</tr>
</tbody>
</table>

The second portion of the report contains:

```sql
-- OUTPUT --
<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>SEQ SCAN</th>
<th>REL TUP</th>
<th>TUP READ</th>
<th>IDX</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>pgbench_tellers</td>
<td>8249</td>
<td>8</td>
<td>824908</td>
<td>0</td>
</tr>
<tr>
<td>public</td>
<td>pgbench_branches</td>
<td>8249</td>
<td>8</td>
<td>824900</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_class</td>
<td>7</td>
<td>3969</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_index</td>
<td>5</td>
<td>950</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

DATA from pg_stat_all_tables ordered by rel tup read

The information displayed in the **DATA from pg_stat_all_tables ordered by rel tup read** section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>The name of the schema in which the table resides</td>
</tr>
<tr>
<td>RELATION</td>
<td>The name of the table</td>
</tr>
<tr>
<td>SEQ SCAN</td>
<td>The number of sequential scans performed on the table</td>
</tr>
</tbody>
</table>

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
### statio_tables_rpt()

The signature is:

```sql
statio_tables_rpt(<beginning_id>, <ending_id>, <top_n>, <scope>)
```

**Parameters**

- **beginning_id**
  
  An integer value that represents the beginning session identifier.

- **ending_id**
  
  An integer value that represents the ending session identifier.

- **top_n**
  
  The number of rows to return.

- **scope**
  
  Determines the tables the function returns statistics about. Specify **SYS**, **USER** or **ALL**:
  
  - Use **SYS** to return information about system-defined tables. A table is considered a system table if it's stored in the `pg_catalog`, `information_schema`, or `sys` schema.
  - Use **USER** to return information about user-defined tables.
  - Use **ALL** to return information about all tables.

The `statio_tables_rpt()` function returns a report that contains:

```sql
SELECT * FROM statio_tables_rpt(9, 10, 10, 'SYS');
```

---

**Table:**

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REL TUP READ</td>
<td>The number of tuples read from the table</td>
</tr>
<tr>
<td>IDX SCAN</td>
<td>The number of index scans performed on the table</td>
</tr>
<tr>
<td>IDX TUP READ</td>
<td>The number of live rows fetched by index scans</td>
</tr>
<tr>
<td>INS</td>
<td>The number of rows inserted</td>
</tr>
<tr>
<td>UPD</td>
<td>The number of rows updated</td>
</tr>
<tr>
<td>DEL</td>
<td>The number of rows deleted</td>
</tr>
</tbody>
</table>
DATA from pg_statio_all_tables

The information displayed in the DATA from pg_statio_all_tables section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>The name of the schema in which the relation resides</td>
</tr>
<tr>
<td>RELATION</td>
<td>The name of the relation</td>
</tr>
<tr>
<td>HEAP READ</td>
<td>The number of heap blocks read</td>
</tr>
<tr>
<td>HEAP HIT</td>
<td>The number of heap blocks hit</td>
</tr>
<tr>
<td>IDX READ</td>
<td>The number of index blocks read</td>
</tr>
<tr>
<td>IDX HIT</td>
<td>The number of index blocks hit</td>
</tr>
<tr>
<td>TOAST READ</td>
<td>The number of toast blocks read</td>
</tr>
<tr>
<td>TOAST HIT</td>
<td>The number of toast blocks hit</td>
</tr>
<tr>
<td>TIDX READ</td>
<td>The number of toast index blocks read</td>
</tr>
<tr>
<td>TIDX HIT</td>
<td>The number of toast index blocks hit</td>
</tr>
</tbody>
</table>

stat_indexes_rpt() 

The signature is:

```
stat_indexes_rpt(<beginning_id>, <ending_id>, <top_n>, <scope>)
```

Parameters

beginning_id
An integer value that represents the beginning session identifier.

ending_id
An integer value that represents the ending session identifier.

top_n
The number of rows to return.

scope
Determines the tables the function returns statistics about. Specify SYS, USER or ALL:

- Use SYS to return information about system-defined tables. A table is considered a system table if it's stored in the pg_catalog, information_schema, or sys schema.
- Use USER to return information about user-defined tables.
- Use ALL to return information about all tables.

The stat_indexes_rpt() function returns a report that contains:

```
edb# SELECT * FROM stat_indexes_rpt(9, 10, 10, 'ALL');

stat_indexes_rpt
--------------------------------------------------------
DATA from pg_stat_all_indexes
```
### DATA from pg_stat_all_indexes

The information displayed in the **DATA from pg_stat_all_indexes** section includes:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCHEMA</td>
<td>The name of the schema in which the relation resides</td>
</tr>
<tr>
<td>RELATION</td>
<td>The name of the relation</td>
</tr>
<tr>
<td>INDEX</td>
<td>The name of the index</td>
</tr>
<tr>
<td>IDX SCAN</td>
<td>The number of indexes scanned</td>
</tr>
<tr>
<td>IDX TUP READ</td>
<td>The number of index tuples read</td>
</tr>
<tr>
<td>IDX TUP FETCH</td>
<td>The number of index tuples fetched</td>
</tr>
</tbody>
</table>

### statio_indexes_rpt()

The signature is:

```
statio_indexes_rpt(<beginning_id>, <ending_id>, <top_n>, <scope>)
```

**Parameters**

- **beginning_id**
  
  An integer value that represents the beginning session identifier.

- **ending_id**
  
  An integer value that represents the ending session identifier.

- **top_n**
  
  The number of rows to return.

- **scope**
  
  Determines the tables the function returns statistics about. Specify **SYS**, **USER** or **ALL**:

  - Use **SYS** to return information about system-defined tables. A table is considered a system table if it's stored in the **pg_catalog**, **information_schema**, or **sys** schema.
  - Use **USER** to return information about user-defined tables.
● Use `ALL` to return information about all tables.

The `statio_indexes_rpt()` function returns a report that contains:

```sql
edc=# SELECT * FROM statio_indexes_rpt(9, 18, 10, 'SYS');

statio_indexes_rpt
------------------------------------
DATA from pg_statio_all_indexes

<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>RELATION</th>
<th>INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>pg_attribute</td>
<td>db$stat_all_indexes</td>
</tr>
<tr>
<td>sys</td>
<td>pg_attribute_relid_attnum_index</td>
<td>db$stat_idx_pk</td>
</tr>
<tr>
<td>1</td>
<td>382</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>edb$statio_all_indexes</td>
<td>edb$statio_idx_pk</td>
</tr>
<tr>
<td>1</td>
<td>382</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>edb$statio_all_tables</td>
<td>edb$statio_tab_pk</td>
</tr>
<tr>
<td>2</td>
<td>262</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>edb$session_wait_history</td>
<td>session_waits_hist_pk</td>
</tr>
<tr>
<td>0</td>
<td>259</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>pg_proc</td>
<td>pg_proc_oid_index</td>
</tr>
<tr>
<td>0</td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>pg_class</td>
<td>pg_class_oid_index</td>
</tr>
<tr>
<td>0</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>pg_class</td>
<td>pg_class_rename_nsp_index</td>
</tr>
<tr>
<td>0</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>pg_catalog</td>
<td>pg_type</td>
<td>pg_type_oid_index</td>
</tr>
<tr>
<td>0</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

(14 rows)
```

**10.2 Using Index Advisor**

The Index Advisor utility helps determine the columns to index to improve performance in a given workload. Index Advisor considers B-tree (single-column or composite) index types. It doesn't identify other index types, that is, GIN, GiST, and Hash, that might improve performance. Index Advisor extension is installed with EDB Postgres Advanced Server, where you can configure and use it.

**10.2.1 Index Advisor overview**

Index Advisor works with EDB Postgres Advanced Server's query planner by creating hypothetical indexes that the query planner uses to calculate execution costs as if such indexes were available. Index Advisor identifies the indexes by analyzing SQL queries supplied in the workload.

You can use Index Advisor to analyze SQL queries in any of these ways:

- Invoke the Index Advisor utility program, supplying a text file containing the SQL queries that you want to analyze. Index Advisor generates a text file with `CREATE INDEX` statements for the recommended indexes.
- Provide queries at the EDB-PSQL command line that you want Index Advisor to analyze.
- Access Index Advisor through the Postgres Enterprise Manager (PEM) client. When accessed using the PEM client, Index Advisor works with SQL Profiler, providing indexing recommendations on code captured in SQL traces. For more information about using SQL Profiler and Index Advisor with PEM, see Using the Index Advisor in the PEM documentation.
Index Advisor attempts to make indexing recommendations on `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements. When invoking Index Advisor, you supply the workload in the form of either:

- If you're providing the command in an SQL file, a set of queries
- If you're specifying the SQL statement at the psql command line, an `EXPLAIN` statement

Index Advisor displays the query plan and estimated execution cost for the supplied query but doesn't execute the query.

During the analysis, Index Advisor compares the query execution costs with and without hypothetical indexes. If the execution cost using a hypothetical Index is less than the execution cost without it:

- Both plans are reported in the `EXPLAIN` statement output.
- Metrics that quantify the improvement are calculated.
- Index Advisor generates the `CREATE INDEX` statement needed to create the Index.

If no hypothetical index can be found that reduces the execution cost, Index Advisor displays only the original query plan output of the `EXPLAIN` statement.

**Note**

Index Advisor doesn’t create indexes on the tables. Use the `CREATE INDEX` statements supplied by Index Advisor to add any recommended indexes to your tables.

An extension supplied with EDB Postgres Advanced Server creates the table in which Index Advisor stores the indexing recommendations generated by the analysis. The extension also creates a function and a view of the table to simplify retrieving and interpreting the results.

If you choose to forgo running the script, Index Advisor logs recommendations in a temporary table that’s available only for the current Index Advisor session.

### 10.2.2 Index Advisor limitations

Prior to running the Index Advisor feature, review the following limitations:

- Index Advisor doesn’t consider index-only scans. It does consider index scans when making recommendations.
- Index Advisor ignores any computations found in the `WHERE` clause. Effectively, the index field in the recommendations isn’t any kind of expression. The field is a simple column name.
- Index Advisor doesn’t consider inheritance when recommending hypothetical indexes. If a query references a parent table, Index Advisor doesn’t make any index recommendations on child tables.
- Suppose you’re restoring a pg_dump backup file that includes the `index_advisor_log` table or any tables for which indexing recommendations were made and stored in the `index_advisor_log` table. Changes in object identifiers (OIDs) can result in broken links between the `index_advisor_log` table and the restored tables referenced by rows in the `index_advisor_log` table.
- If you need to display the recommendations made prior to the backup, you can replace the old OIDs in the `reloid` column of the `index_advisor_log` table with the new OIDs of the referenced tables using the SQL `UPDATE` statement:

  ```sql
  UPDATE index_advisor_log SET reloid = new_oid WHERE reloid = old_oid;
  ```

### 10.2.3 Configuring the Index Advisor

Index Advisor doesn’t require configuration to generate recommendations that are available only for the rest of the current session. To store the results of multiple sessions, you must create the `index_advisor_log` table, where EDB Postgres Advanced Server stores Index Advisor recommendations. To create the `index_advisor_log` table, create the extension `index_advisor`.

When selecting a storage schema for the Index Advisor table, function, and view, keep in mind that all users that invoke Index Advisor and query the result set must have USAGE privileges on the schema. The schema must be in the search path of all users that are interacting with Index Advisor.

**Storing the results of multiple sessions**

1. Place the selected schema at the start of your `search_path` parameter. For example, suppose your search path is currently:

   ```
   search_path=public, accounting
   ```

   If you want to create the Index Advisor objects in a schema named `advisor`, use the command:

   ```
   SET search_path = advisor, public, accounting;
   ```

2. Create the `index_advisor` extension, which creates the database objects. Connect to the database as the database superuser using psql, and enter the command:
5. **Grant privileges on the `index_advisor_log` table to all Index Advisor users.** This step isn't necessary if the Index Advisor user is a superuser or the owner of these database objects.

   - Grant `SELECT` and `INSERT` privileges on the `index_advisor_log` table to allow a user to invoke Index Advisor.
   - Grant `DELETE` privileges on the `index_advisor_log` table to allow the specified user to delete the table contents.
   - Grant `SELECT` privilege on the `index_recommendations` view.

### Example

This example shows creating Index Advisor database objects in a schema named `ia`. The schema is accessible to an Index Advisor user with user name `ia_user`.

```sql
# running as a entprisedb user
edb-psql -d edb -U enterprisedb
# running commands inside psql
CREATE SCHEMA ia;
SET search_path TO ia;
CREATE EXTENSION index_advisor;
GRANT USAGE ON SCHEMA ia TO ia_user;
GRANT SELECT, INSERT, DELETE ON index_advisor_log TO ia_user;
GRANT SELECT ON index_recommendations TO ia_user;
```

While using Index Advisor, the specified schema (ia) must be included in the `ia_user` `search_path` parameter.

### 10.2.4 Using Index Advisor

When you invoke Index Advisor, you must supply a workload. The workload is either a query specified at the command line or a file that contains a set of queries. These queries are executed by the `pg_advise_index()` function. After analyzing the workload, Index Advisor stores the result set in either a temporary table or in a permanent table. You can review the indexing recommendations generated by Index Advisor and use the `CREATE INDEX` statements generated by Index Advisor to create the recommended indexes.

**Note**

Don't run Index Advisor in read-only transactions.

The following examples assume that superuser `enterprisedb` is the Index Advisor user. The Index Advisor database objects were created in a schema in the `search_path` of superuser `enterprisedb`.

The examples use the table created with this statement:

```sql
CREATE TABLE t ( a INT, b INT );
INSERT INTO t SELECT s, 99999 - s FROM generate_series(0,99999) AS s;
ANALYZE t;
```

The resulting table contains the following rows:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99999</td>
</tr>
</tbody>
</table>
Using the `pg_advise_index` utility

When invoking the `pg_advise_index` utility, you must include the name of a file that contains the queries executed by `pg_advise_index`. The queries can be on the same line or on separate lines, but each query must be terminated by a semicolon. Queries in the file can't begin with the `EXPLAIN` keyword.

This example shows the contents of a sample `workload.sql` file:

```sql
SELECT * FROM t WHERE a = 500;
SELECT * FROM t WHERE b < 1000;
```

Run the `pg_advise_index` program:

```
$ pg_advise_index -d edb -h localhost -U enterprisedb -s 100M -o advisory.sql workload.sql
poolsize = 102400 KB
load workload from file 'workload.sql'
Analyzing queries .. done.
size = 2184 KB, benefit = 1684.720000
size = 2184 KB, benefit = 1655.520000
/* 1. t(a): size=2184 KB, benefit=1684.72 */
/* 2. t(b): size=2184 KB, benefit=1655.52 */
/* Total size = 4368KB */
```

In the code sample, the `-d`, `-h`, and `-U` options are psql connection options.

**-s**

An optional parameter that limits the maximum size of the indexes recommended by Index Advisor. If Index Advisor doesn't return a result set, `-s` might be set too low.

**-o**

The recommended indexes are written to the file specified after the `-o` option.

The information displayed by the `pg_advise_index` program is logged in the `index_advisor_log` table. In response to the command shown in the example, Index Advisor writes the following `CREATE INDEX` statements to the `advisory.sql` output file:

```sql
CREATE INDEX idx_t_1 on t (a);
CREATE INDEX idx_t_2 on t (b);
```

Creating the indexes

You can create the recommended indexes at the psql command line with the `CREATE_INDEX` statements in the file. Or you can create the indexes by executing the `advisory.sql` script.

```
$ edb-psql -d edb -h localhost -U enterprisedb -e -f advisory.sql
create index idx_t_1 on t (a);
CREATE INDEX
```
Using Index Advisor at the psql command line

Using Index Advisor at the psql command line

You can use Index Advisor to analyze SQL statements entered at the edb-psql or psql command line.

To load the Index Advisor plugin and use Index Advisor:

1. Connect to the server with the edb-psql command line utility, and load the Index Advisor plugin:

   ```
   $ edb-psql -d edb -U enterprise
   ...
   edb=# LOAD 'index_advisor';
   LOAD
   ```

2. Use the edb-psql command line to invoke each SQL command that you want Index Advisor to analyze. Index Advisor stores any recommendations for the queries in the index_advisor_log table. If the index_advisor_log table doesn’t exist in the user’s search_path, a temporary table is created with the same name. This temporary table exists only for the rest of the user’s current session.

After you load the Index Advisor plugin, Index Advisor analyzes all SQL statements and logs any indexing recommendations for the rest of the session.

If you want Index Advisor to analyze a query and make indexing recommendations without executing the query, preface the SQL statement with the EXPLAIN keyword. If you don’t preface the statement with the EXPLAIN keyword, Index Advisor analyzes the statement while the statement executes. It writes the indexing recommendations to the index_advisor_log table for later review.

In this example, the EXPLAIN statement displays the normal query plan. It’s followed by the query plan of the same query if the query were using the recommended hypothetical index:

```sql
create index idx_t_2 on t (b);
CREATE INDEX

Note
pg_advisor_index asks the backend process to load the index_advisor plugin first from $libdir/plugins. If not found, then it writes the error in the server log file and attempts to load from $libdir.
```

In this example, the EXPLAIN statement displays the normal query plan. It’s followed by the query plan of the same query if the query were using the recommended hypothetical index:

```sql
edbdb=1 EXPLAIN SELECT * FROM t WHERE a < 10000;
QUERY PLAN
--------------
Seq Scan on t (cost=0.00..1693.00 rows=10105 width=8)
  Filter: (a < 10000)
Result (cost=0.00..337.10 rows=10105 width=8)
  One-Time Filter: '===[ HYPOTHETICAL PLAN ]===':text
  -> Index Scan using "<hypothetical-index>:1" on t
      (cost=0.00..337.10 rows=10105 width=8)
      Index Cond: (a < 10000)
(6 rows)

edbdb=2 EXPLAIN SELECT * FROM t WHERE a = 100;
QUERY PLAN
--------------
Seq Scan on t (cost=8.00..1693.00 rows=1 width=8)
  Filter: (a = 100)
Result (cost=8.00..8.28 rows=1 width=8)
  One-Time Filter: '===[ HYPOTHETICAL PLAN ]===':text
  -> Index Scan using "<hypothetical-index>:3" on t
      (cost=8.00..8.28 rows=1 width=8)
      Index Cond: (a = 100)
(6 rows)
```

After loading the Index Advisor plugin, the default value of index_advisor.enabled is on. The Index Advisor plugin must be loaded to use a SET or SHOW command to display the current value of index_advisor.enabled.

You can use the index_advisor.enabled parameter to temporarily disable Index Advisor without interrupting the psql session:

```sql
edbdb=# SET index_advisor.enabled TO off;
SET
To enable Index Advisor, set the parameter to on:
```
10.2.5 Reviewing Index Advisor recommendations

You can review the index recommendations generated by Index Advisor in several ways. You can:

- Run the `show_index_recommendations` function.
- Query the `index_advisor_log` table.
- Query the `index_recommendations` view.

Using the `show_index_recommendations()` function

To review the recommendations of the Index Advisor utility using the `show_index_recommendations()` function, call the function, specifying the process ID of the session:

```
SELECT show_index_recommendations( pid );
```

Where `pid` is the process ID of the current session. If you don’t know the process ID of your current session, passing a value of `NULL` also returns a result set for the current session.

This code fragment shows an example of a row in a result set:

```
edb=# SELECT show_index_recommendations(null);

show_index_recommendations
----------------------------------------
create index idx_t_a on t(a);/* size: 2184 KB, benefit: 3040.62, gain: 1.3922666981456 */
(1 row)
```

In the example, `create index idx_t_a on t(a)` is the SQL statement needed to create the index suggested by Index Advisor. Each row in the result set shows:

- The command required to create the recommended index.
- The maximum estimated size of the index.
- The calculated benefit of using the index.
- The estimated gain that results from implementing the index.

You can display the results of all Index Advisor sessions from the following view:

```
SELECT * FROM index_recommendations;
```

**Querying the `index_advisor_log` table**

Index Advisor stores indexing recommendations in a table named `index_advisor_log`. Each row in the `index_advisor_log` table contains the result of a query where Index Advisor determines it can recommend a hypothetical index to reduce the execution cost of that query.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reloid</td>
<td>oid</td>
<td>OID of the base table for the index</td>
</tr>
<tr>
<td>relname</td>
<td>name</td>
<td>Name of the base table for the index</td>
</tr>
<tr>
<td>attrs</td>
<td>integer[]</td>
<td>Recommended index columns (identified by column number)</td>
</tr>
<tr>
<td>benefit</td>
<td>real</td>
<td>Calculated benefit of the index for this query</td>
</tr>
<tr>
<td>index_size</td>
<td>integer</td>
<td>Estimated index size in disk-pages</td>
</tr>
<tr>
<td>backend_pid</td>
<td>integer</td>
<td>Process ID of the process generating this recommendation</td>
</tr>
<tr>
<td>timestamp</td>
<td>timestamp</td>
<td>Date/time when the recommendation was generated</td>
</tr>
</tbody>
</table>

You can query the `index_advisor_log` table at the `psql` command line. This example shows the `index_advisor_log` table entries resulting from two Index Advisor sessions. Each session contains two queries and can be identified in the table by a different `backend_pid` value. For each session, Index Advisor generated two index recommendations.

```
edb=# SELECT * FROM index_advisor_log;
```
Index Advisor added the first two rows to the table after analyzing the following two queries executed by the `pg_advise_index` utility:

```sql
SELECT * FROM t WHERE a = 500;
SELECT * FROM t WHERE b < 1080;
```

- The value of 3442 in column `backend_pid` identifies these results as coming from the session with process ID 3442.
- The value of 1 in column `attrs` in the first row indicates that the hypothetical index is on the first column of the table (column `a` of table `t`).
- The value of 2 in column `attrs` in the second row indicates that the hypothetical index is on the second column of the table (column `b` of table `t`).

Index Advisor added the last two rows to the table after analyzing the following two queries executed at the psql command line:

```sql
EXPLAIN
SELECT * FROM t WHERE a < 10000;
EXPLAIN
SELECT * FROM t WHERE a = 100;
```

The values in the `benefit` column of the `index_advisor_log` table are calculated using the following formula:

```
benefit = (normal execution cost) - (execution cost with hypothetical index)
```

- The value of the `benefit` column for the last row of the `index_advisor_log` table shown in the example is calculated using the query plan for the following SQL statement:

```sql
EXPLAIN
SELECT * FROM t WHERE a = 100;
```

- The execution costs of the different execution plans are evaluated and compared:

```
benefit = (Seq Scan on t cost) - (Index Scan using <hypothetical-index>)
```

- The benefit is added to the table:

```
benefit = 1693.00 - 8.28
```
You can delete rows from the `index_advisor_log` table when you no longer need to review the results of the queries stored in the row.

**Querying the index_recommendations view**

The `index_recommendations` view contains the calculated metrics and the `CREATE INDEX` statements to create the recommended indexes for all sessions whose results are currently in the `index_advisor_log` table. You can display the results of all stored Index Advisor sessions by querying the `index_recommendations` view:

```sql
SELECT * FROM index_recommendations;
```

Using the example shown in Querying the `index_advisor_log` table, the `index_recommendations` view displays the following:

```
backend_pid | show_index_recommendations
-------------+---------------------------------------------------------------
3442    | create index idx_t_a on t(a);/* size: 2184 KB, benefit: 1684.72, gain: 0.771392654586624 */
3442    | create index idx_t_b on t(b);/* size: 2184 KB, benefit: 1655.52, gain: 0.758021539820856 */
3506    | create index idx_t_a on t(a);/* size: 2184 KB, benefit: 3040.62, gain: 1.39222666981456 */
3506    | create index idx_t_a on t(a);
3040.62, gain: 1.39222666981456 */
(3 rows)
```

In each session, the results of all queries that benefit from the same recommended index are combined to produce one set of metrics per recommended index, reflected in the fields named `benefit` and `gain`.

The formulas for the fields are:

- `size = MAX(index size of all queries)`
- `benefit = SUM(benefit of each query)`
- `gain = SUM(benefit of each query) / MAX(index size of all queries)`

For example, using the following query results from the process with a `backend_pid` of `3506`:

```
reloid  | relname | attrs | benefit | index_size | backend_pid | timestamp
--------+---------+-------+---------+------------+-------------+------------
16651   | t       | {1}   | 1355.9  | 2184       | 3506        | 22-MAR-11 16:48:28.317016 -04:00
16651   | t       | {1}   | 1684.72 | 2184       | 3506        | 22-MAR-11 16:51:45.927906 -04:00
```

The metrics displayed from the `index_recommendations` view for `backend_pid` `3506` are:

```
backend_pid | show_index_recommendations
----------------+----------------------------------
3506           | create index idx_t_a on t(a);/* size: 2184 KB,
benefit: 3040.62, gain: 1.39222666981456 */
```

The metrics from the view are calculated like this:

- `benefit = (benefit from 1st query) + (benefit from 2nd query)`
- `benefit = 1355.9 + 1684.72`
- `benefit = 3040.62`

As well as the following:

- `gain = ((benefit from 1st query) + (benefit from 2nd query)) / MAX(index size of all queries)`
- `gain = (1355.9 + 1684.72) / MAX(2184, 2184)`

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The gain metric is useful when comparing the relative advantage of the different recommended indexes derived during a given session. The larger the gain value, the better the cost effectiveness derived from the index weighed against the possible disk space consumption of the index.

10.3 Using dynamic resource tuning

EDB Postgres Advanced Server supports dynamic tuning of the database server to make the optimal use of the system resources available on the host machine where it’s installed. The two parameters that control this functionality are located in the `postgresql.conf` file. These parameters are:

- `edb_dynatune`
- `edb_dynatune_profile`

Implementing dynamic tuning

`edb_dynatune` determines how much of the host system’s resources for the database server to use. It bases the determination on the host machine’s total available resources and the intended use of the host machine.

When EDB Postgres Advanced Server is first installed, you set the `edb_dynatune` parameter according to the host machine’s use as a development machine, mixed-use machine, or dedicated server. For most purposes, the database administrator doesn’t need to adjust the configuration parameters in the `postgresql.conf` file to improve performance.

Here are some things to keep in mind when setting up dynamic tuning:

- You can change the value of the `edb_dynatune` parameter after the initial installation of EDB Postgres Advanced Server by editing the `postgresql.conf` file. You must restart the postmaster for the new configuration to take effect.
- You can set the `edb_dynatune` parameter to any integer value from 0 to 100. A value of 0 turns off the dynamic tuning feature, leaving the database server resource use under the control of the other configuration parameters in the `postgresql.conf` file.
- A low, non-zero value, that is, 1–33, dedicates the least amount of the host machine’s resources to the database server. Use this setting for a development machine where many other applications are being used.
- A value in the range of 34–66 dedicates a moderate amount of resources to the database server. You might use this setting for a dedicated application server that has a fixed number of other applications running on the same machine as EDB Postgres Advanced Server.
- The highest values, that is, 67–100, dedicate most of the server’s resources to the database server. Use this setting for a host machine that’s totally dedicated to running EDB Postgres Advanced Server.

Changing parameter values

After you select a value for `edb_dynatune`, you can further fine-tune database server performance by adjusting the other configuration parameters in the `postgresql.conf` file. Any adjusted setting overrides the corresponding value chosen by `edb_dynatune`.

To change the value of a parameter:

1. Uncomment the configuration parameter.
2. Specify the desired value.
3. Restart the database server.

Controlling tuning behavior

Use the `edb_dynatune_profile` parameter to control tuning aspects based on the expected workload profile on the database server. This parameter takes effect when you start the database server.

The table shows the possible values for `edb_dynatune_profile`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>oltp</code></td>
<td>Recommended when the database server is processing heavy online transaction processing workloads.</td>
</tr>
<tr>
<td><code>reporting</code></td>
<td>Recommended for database servers used for heavy data reporting.</td>
</tr>
<tr>
<td><code>mixed</code></td>
<td>Recommended for servers that provide a mix of transaction processing and data reporting.</td>
</tr>
</tbody>
</table>
10.4  Evaluating wait states

EDB Wait States is a tool for analyzing performance and tuning by allowing the collection and querying of wait event data. Wait events are recorded alongside other session activity and provide a snapshot of whether a session is waiting for I/O, CPU, IPC, locks, or timeouts. Snapshots of this information are gathered by the EDB Wait States background worker (BGW) at regular intervals.

The EDB Wait States interface allows you to control when and for how long the wait events are sampled and to extract the gathered samples in edb_wait_states_data for further analysis. By gathering this data over time, you can discover optimization opportunities and gain insight into what resources sessions are waiting on when performance is lower than expected.

See EDB Wait States for more information.

10.5  Using SQL Profiler

Inefficient SQL code is a leading cause of database performance problems. The challenge for database administrators and developers is locating and then optimizing this code in large, complex systems. SQL Profiler helps you locate and optimize poorly running SQL code.

Specific features and benefits of SQL Profiler include:

- **On-demand traces.** You can capture SQL traces at any time by manually setting up your parameters and starting the trace.
- **Scheduled traces.** If the current time isn’t convenient, you can also specify your trace parameters and schedule them to run later.
- **Save traces.** Execute your traces and save them for later review.
- **Trace filters.** Selectively filter SQL captures by database and by user, or capture every SQL statement sent by all users against all databases.
- **Trace output analyzer.** A graphical table lets you quickly sort and filter queries by duration or statement. A graphical or text-based EXPLAIN plan lays out your query paths and joins.
- **Index Advisor integration.** After you find your slow queries and optimize them, you can also let Index Advisor recommend the creation of underlying table indices to further improve performance.

See these topics in the Postgres Enterprise Manager documentation:

- Installing SQL Profiler
- Upgrading SQL Profiler
- Using SQL Profiler

11  Application programming

EDB Postgres Advanced Server includes features designed to increase application programmer productivity, such as user-defined objects, autonomous transactions, synonyms, and 200+ prepackaged utility functions.

11.1  Working with packages

A package is a named collection of functions, procedures, variables, cursors, user-defined record types, and records that are referenced using a common qualifier: the package identifier. Packages have the following characteristics:

- They provide a convenient means of organizing the functions and procedures that perform a related purpose. Permission to use the package functions and procedures depends on one privilege granted to the entire package. You must reference all of the package programs with a common name.
- You can declare certain functions, procedures, variables, types, and so on in the package as public. Public entities are visible and other programs that are given EXECUTE privilege on the package can reference them. For public functions and procedures, only their signatures are visible: the program names, parameters, if any, and return types of functions. The SPL code of these functions and procedures isn’t accessible to others. Therefore applications that use a package depend on only the information available in the signature and not in the procedural logic itself.
- You can declare other functions, procedures, variables, types, and so on in the package as private. Private entities can be referenced and used by function and procedures in the package but not by other external applications. Private entities are for use only by programs in the package.
- Function and procedure names can be overloaded in a package. You can define one or more functions/procedures with the same name but with different signatures. This capability enables you to create identically named programs that perform the same job but on different types of input.

11.1.1  Package components

Packages consist of two main components:

- The package specification, which is the public interface. You can reference these elements outside the package. Declare all database objects that are a part of a package in the specification.
- The package body, which contains the actual implementation of all the database objects declared in the package specification.

The package body implements the specifications in the package specification. It contains implementation details and private declarations that are invisible to the application. You can debug, enhance, or replace a package body without changing the specifications. Similarly, you can change the body without recompiling the calling programs because the implementation details are invisible to the application.
Package specification syntax

The package specification defines the user interface for a package (the API). The specification lists the functions, procedures, types, exceptions, and cursors that are visible to a user of the package.

The syntax used to define the interface for a package is:

```
CREATE [ OR REPLACE ] PACKAGE
    <package_name>
    [ <authorization_clause>
    ]
    [ { IS | AS }
    ]
    [ <declaration>; ]
    ...
    [ <procedure_or_function_declaration> ]
    ...
END [ <package_name> ]
;
```

Where `authorization_clause` :=

```
{ AUTHID DEFINER } | { AUTHID CURRENT_USER }
```

Where `procedure_or_function_declaration` :=

```
procedure_declaration | function_declaration
```

Where `procedure_declaration` :=

```
PROCEDURE proc_name [ argument_list ]
{ restriction_pragma; }
```

Where `function_declaration` :=

```
FUNCTION func_name [ argument_list ]
    RETURN rettype [ DETERMINISTIC ];
{ restriction_pragma; }
```

Where `argument_list` :=

```
( argument_declaration [ , ... ] )
```

Where `argument_declaration` :=

```
argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
```

Where `restriction_pragma` :=

```
PRAGMA RESTRICT_REFERENCES(name, restrictions)
```

Where `restrictions` :=

```
restriction [ , ... ]
```

Parameters

`package_name` is an identifier assigned to the package. Each package must have a unique name in the schema.

**AUTHID DEFINER**

If you omit the `AUTHID` clause or specify `AUTHID DEFINER`, the privileges of the package owner are used to determine access privileges to database objects.

**AUTHID CURRENT_USER**
If you specify AUTHID CURRENT_USER, the privileges of the current user executing a program in the package are used to determine access privileges.

**declaration** is an identifier of a public variable. You can access a public variable from outside the package using the syntax `package_name.variable`. There can be zero, one, or more public variables. Public variable definitions must come before procedure or function declarations.

**declaration** can be any of the following:

- Variable declaration
- Record declaration
- Collection declaration
- REF CURSOR and cursor variable declaration
- **TYPE** definitions for records, collections, and REF CURSOR
- Exception
- Object variable declaration

**proc_name**

The name of a public procedure.

**argname**

The name of an argument. The argument is referenced by this name in the function or procedure body.

**IN | IN OUT | OUT**

The argument mode. **IN** (the default) declares the argument for input only. **IN OUT** allows the argument to receive a value as well as return a value. **OUT** specifies the argument is for output only.

**argtype**

The data types of an argument. An argument type can be a base data type, a copy of the type of an existing column using **%TYPE**, or a user-defined type such as a nested table or an object type. Don't specify a length for any base type. For example, specify `VARCHAR2`, not `VARCHAR2(10)`.

Reference the type of a column by writing `tablename.columnname %TYPE`. Using this nomenclature can sometimes help make a procedure independent from changes to the definition of a table.

**DEFAULT value**

The **DEFAULT** clause supplies a default value for an input argument if you don’t supply one in the invocation. You can’t specify **DEFAULT** for arguments with modes **IN OUT** or **OUT**.

**func_name**

The name of a public function.

**rettype**

The return data type.

**DETERMINISTIC**

**DETERMINISTIC** is a synonym for **IMMUTABLE**. A **DETERMINISTIC** function can’t modify the database and always reaches the same result when given the same argument values. It doesn’t do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the function with all-constant arguments can be immediately replaced with the function value.

**restriction**

The following keywords are accepted for compatibility and ignored:

- **RNDS**
- **RNPS**
- **TRUST**
- **WINDS**
- **WNPS**

**Package body syntax**
Package implementation details reside in the package body. The package body can contain objects that aren't visible to the package user. EDB Postgres Advanced Server supports the following syntax for the package body:

```
CREATE [ OR REPLACE ] PACKAGE BODY
<package_name>
  ( IS | AS
  )
  [ <private_declaration>; ]
  ...
  [ <procedure_or_function_definition> ]
  ...
  [ <package_initializer> ]
END [ <package_name> ]
;
```

Where `procedure_or_function_definition` :=

```
procedure_definition | function_definition
```

Where `procedure_definition` :=

```
PROCEDURE proc_name[ argument_list ]
  [ options_list ]
  ( IS | AS
  )
procedure_body
  END [ proc_name ]
;
```

Where `procedure_body` :=

```
[ PRAGMA AUTONOMOUS_TRANSACTION;
 ] [ declaration; ] [, ...
BEGIN
  statement;
[...] [ EXCEPTION
    { WHEN exception [ OR exception] [...] } THEN statement;
  ]
[...] ]
```

Where `function_definition` :=

```
FUNCTION func_name [ argument_list ]
  RETURN rettype [ DETERMINISTIC
  ] [ options_list ]
  ( IS | AS
  )
function_body
  END [ func_name ]
;
```

Where `function_body` :=

```
[ PRAGMA AUTONOMOUS_TRANSACTION;
 ] [ declaration; ] [, ...
BEGIN
  statement;
[...] [ EXCEPTION
    { WHEN exception [ OR exception] [...] } THEN statement;
  ]
[...] ]
```

Where `argument_list` :=

```
( argument_declaration [, ...]
)
Where **argument_declaration** :=

argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]

Where **options_list** :=

option [ ... ]

Where **option** :=

- **STRICT**
- **LEAKPROOF**
- **PARALLEL** { UNSAFE | RESTRICTED | SAFE }
- **COST**
- **ROWS**
- **RESULT_ROWS**
- **SET** config_param { TO value | = value | FROM CURRENT }

Where **package_initializer** :=

BEGIN
statement;
[...]
END;

**Parameters**

**package_name**

**package_name** is the name of the package for which this is the package body. An package specification with this name must already exist.

**private_declaration**

**private_declaration** is an identifier of a private variable that any procedure or function can access in the package. There can be zero, one, or more private variables. **private_declaration** can be any of the following:

- Variable declaration
- Record declaration
- Collection declaration
- **REF CURSOR** and cursor variable declaration
- **TYPE** definitions for records, collections, and **REF CURSORS**
- Exception
- Object variable declaration

**proc_name**

The name of the procedure being created.

**PRAGMA AUTONOMOUS_TRANSACTION**

**PRAGMA AUTONOMOUS_TRANSACTION** is the directive that sets the procedure as an autonomous transaction.

**declaration**

A variable, type, **REF CURSOR**, or subprogram declaration. If you include subprogram declarations, declare them after all other variable, type, and **REF CURSOR** declarations.

**statement**

An SPL program statement. A **DECLARE - BEGIN - END** block is considered an SPL statement unto itself. Thus, the function body can contain nested blocks.

**exception**

An exception condition name such as **NO_DATA_FOUND**, **OTHERS**.

**func_name**

The name of the function being created.
The return data type, which can be any of the types listed for `argtype`. As for `argtype`, don't specify a length for `rettype`.

**DETERMINISTIC**

Include `DETERMINISTIC` to specify for the function to always return the same result when given the same argument values. A `DETERMINISTIC` function must not modify the database.

**Note** The `DETERMINISTIC` keyword is equivalent to the PostgreSQL `IMMUTABLE` option.

**Note** If `DETERMINISTIC` is specified for a public function in the package body, you must also specify it for the function declaration in the package specification. For private functions, there's no function declaration in the package specification.

**PRAGMA AUTONOMOUS_TRANSACTION**

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the function as an autonomous transaction.

**argname**

The name of a formal argument. The argument is referenced by this name in the procedure body.

**IN | IN OUT | OUT**

The argument mode. **IN** (the default) declares the argument for input only. **IN OUT** allows the argument to receive a value as well as return a value. **OUT** specifies the argument is for output only.

**argtype**

The data types of an argument. An argument type can be a base data type, a copy of the type of an existing column using `%TYPE`, or a user-defined type such as a nested table or an object type. Don't specify a length for any base type. For example, specify `VARCHAR2`, not `VARCHAR2(10)`.

Reference the type of a column by writing `tablename.columnname%TYPE`. Using this nomenclature can sometimes help make a procedure independent from changes to the definition of a table.

**DEFAULT value**

The **DEFAULT** clause supplies a default value for an input argument if you don't supply one in the procedure call. Don't specify **DEFAULT** for arguments with modes **IN OUT** or **OUT**.

The following options aren't compatible with Oracle databases. They're extensions to Oracle package syntax provided only by EDB Postgres Advanced Server.

**STRICT**

The **STRICT** keyword specifies for the function not to execute if called with a **NULL** argument. Instead the function returns **NULL**.

**LEAKPROOF**

The **LEAKPROOF** keyword specifies for the function not to reveal any information about arguments other than through a return value.

**PARALLEL { UNSAFE | RESTRICTED | SAFE }**

The **PARALLEL** clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to **UNSAFE**, the procedure or function can't be executed in parallel mode. The presence of such a procedure or function forces a serial execution plan. This is the default setting if you omit the **PARALLEL** clause.

When set to **RESTRICTED**, the procedure or function can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

When set to **SAFE**, the procedure or function can be executed in parallel mode without restriction.

**execution_cost**

**execution_cost** specifies a positive number giving the estimated execution cost for the function, in units of `cpu_operator_cost`. If the function returns a set, this is the cost per returned row. The default is `.0025`.

**result_rows**

**result_rows** is the estimated number of rows for the query planner to expect the function to return. The default is `1000`.
Use the **SET** clause to specify a parameter value for the duration of the function:

```sql
config_param specifies the parameter name.
value specifies the parameter value.
FROM CURRENT guarantees that the parameter value is restored when the function ends.
```

### Package_initializer

The statements in the **package_initializer** are executed once per user session when the package is first referenced.

**Note**

The **STRICT**, **LEAKPROOF**, **PARALLEL**, **COST**, **ROWS** and **SET** keywords provide extended functionality for EDB Postgres Advanced Server. Oracle doesn't support them.

### 11.1.1 Viewing packages and package body definition

You can view the package specification and package body definition using the `ps` meta-commands `\sps` and `\spb`, respectively.

#### Synopsis

```sql
\sps
[<schema_name>].<package_name>
\spb
[<schema_name>].<package_name>
```

#### Creating and viewing a package and a package body

Create a package and a package body `test_pkg` in the **public** schema:

```sql
edb=# CREATE OR REPLACE PACKAGE public.test_pkg IS
edb$# emp_name character varying(10);
edb$# PROCEDURE get_name(IN p_empno numeric);
edb$# FUNCTION display_counter() RETURN integer;
edb$# END;
CREATE PACKAGE
edb=#
edb=# CREATE OR REPLACE PACKAGE BODY public.test_pkg IS
edb$# v_counter integer;
edb$# PROCEDURE get_name(IN p_empno numeric) IS
edb$# BEGIN
edb$# SELECT ename INTO emp_name FROM emp WHERE empno = p_empno;
edb$# v_counter := v_counter + 1;
edb$# END;
edb$# FUNCTION display_counter() RETURN integer IS
edb$# BEGIN
edb$# RETURN v_counter;
edb$# END;
edb$# BEGIN
edb$# v_counter := 0;
edb$# DBMS_OUTPUT.PUT_LINE('Initialized counter');
edb$# END;
CREATE PACKAGE BODY
edb=#
```

Use `\sps` and `\spb` commands to view the definition of package and package body:

```sql
edb=# \sps
edb# \sps
test_pkg
CREATE OR REPLACE PACKAGE public.test_pkg IS
emp_name character varying(10);
```
PROCEDURE get_name(IN p_empno numeric);
FUNCTION display_counter() RETURN integer;
END
edb=#
edb=# \sp+  
CREATE OR REPLACE PACKAGE public.test_pkg IS
emp_name character varying(10);
PROCEDURE get_name(INOUT p_empno numeric);
FUNCTION display_counter(OUT p1 numeric, OUT p2 numeric) RETURN integer;
END

edb=# \sp+ CREATE OR REPLACE PACKAGE public.test_pkg IS
v_counter integer;
PROCEDURE get_name(IN p_empno numeric) IS
BEGIN
SELECT ename INTO emp_name FROM emp WHERE empno = p_empno;
v_counter := v_counter + 1;
END;
FUNCTION display_counter() RETURN integer IS
BEGIN
RETURN v_counter;
END;
BEGIN
v_counter := 0;
DBMS_OUTPUT.PUT_LINE('Initialized counter');
END
edb=

Viewing function and procedure definitions

You can also view the definition of individual functions and procedures using the \sf command.

Create the function and procedure:

edb=# CREATE OR REPLACE FUNCTION public.func1()
edb=# RETURNS integer
edb=# LANGUAGE edbspl
edb=# SECURITY DEFINER
edb=# AS $function$ begin return 10; end$function$;
CREATE FUNCTION
edb=# CREATE OR REPLACE PROCEDURE public.procl()
edb=# SECURITY DEFINER
edb=# AS $procedure$ begin null; end$procedure$;
edb=# LANGUAGE edbspl;
11.1.3 Creating packages

A package isn't an executable piece of code but a repository of code. When you use a package, you execute or make reference to an element within a package.

Creating the package specification

The package specification contains the definition of all the elements in the package that you can reference from outside of the package. These are called the public elements of the package, and they act as the package interface. The following code sample is a package specification:

```sql
-- Package specification for the 'emp_admin' package.
CREATE OR REPLACE PACKAGE emp_admin
IS
  FUNCTION get_dept_name
  (    p_deptno NUMBER DEFAULT 10
  )
  RETURN VARCHAR2;
  FUNCTION update_emp_sal
  (    p_empno NUMBER,
       p_raise NUMBER
  )
  RETURN NUMBER;
  PROCEDURE hire_emp
  (    p_empno NUMBER,
       p_ename VARCHAR2,
       p_job VARCHAR2,
       p_sal NUMBER,
       p_hiredate DATE DEFAULT sysdate,
       p_comm NUMBER DEFAULT 0,
       p_mgr NUMBER,
       p_deptno NUMBER DEFAULT 10
  );
  PROCEDURE fire_emp
  (    p_empno NUMBER
  );
END emp_admin;
```

This code sample creates the `emp_admin` package specification. This package specification consists of two functions and two stored procedures. You can also add the OR REPLACE clause to the CREATE PACKAGE statement for convenience.
Creating the package body

The body of the package contains the actual implementation behind the package specification. For the `emp_admin` package specification in the example, this code now create a package body that implements the specifications. The body contains the implementation of the functions and stored procedures in the specification.

```
-- -- Package body for the 'emp_admin'
-- package.
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
  -- Function that queries the 'dept' table based on the department number and returns the corresponding department name.
  FUNCTION get_dept_name
  ( p_deptno IN NUMBER DEFAULT 10 )
  RETURN VARCHAR2 IS
    v_dname VARCHAR2(14);
    BEGIN
      SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
      RETURN v_dname;
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno):
        RETURN '';
    END;
  -- Function that updates an employee's salary based on the employee number and salary increment/decrement passed as IN parameters. Upon successful completion the function returns the new updated salary.
  FUNCTION update_emp_sal
  ( p_empno IN NUMBER,
p_raise IN NUMBER )
  RETURN NUMBER IS
    v_sal NUMBER := 0;
    BEGIN
      SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
      v_sal := v_sal + p_raise;
      UPDATE emp SET sal = v_sal WHERE empno = p_empno;
      RETURN v_sal;
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found!');
        RETURN -1;
      WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
        RETURN -1;
    END;
  -- Procedure that inserts a new employee record into the 'emp'
table.
```
11.1.4 Referencing a package

To reference the types, items, and subprograms that are declared in a package specification, use the dot notation. For example:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
```

To invoke a function from the `emp_admin` package specification, execute the following SQL command:

```
SELECT emp_admin.get_dept_name(10) FROM DUAL;
```

This example invokes the `get_dept_name` function declared in the package `emp_admin`. It passes the department number as an argument to the function, which returns the name of the department. The value returned is `ACCOUNTING`, which corresponds to department number `10`.

11.1.5 Using packages with user-defined types

This example incorporates various user-defined types in the context of a package.

Package specification

The package specification of `emp_rpt` shows the declaration of a record type `emprec_typ` and a weakly typed `REF CURSOR`, `emp_refcur` as publicly accessible. It also shows two functions and two procedures. The function, `open_emp_by_dept`, returns the `REF CURSOR` type `EMP_REFCUR`. Procedures `fetch_emp` and `close_refcur` both declare a weakly typed `REF CURSOR` as a formal parameter.

```
CREATE OR REPLACE PACKAGE emp_rpt
```
The package body shows the declaration of several private variables: a static cursor `dept_cur`, a table type `depttab_typ`, a table variable `t_dept`, an integer variable `t_dept_max`, and a record variable `r_emp`.

```sql
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
  CURSOR dept_cur IS SELECT * FROM dept;
  TYPE depttab_typ IS TABLE OF dept%ROWTYPE
    INDEX BY BINARY_INTEGER;
  t_dept DEPTTAB_TYP;
  t_dept_max INTEGER := 1;
  r_emp EMPREC_TYP;

  FUNCTION get_dept_name
  ( p_deptno IN NUMBER ) RETURN VARCHAR2
  IS
    BEGIN
      FOR i IN 1..t_dept_max
      LOOP
        IF p_deptno = t_dept(i).deptno THEN
          RETURN t_dept(i).dname;
        END IF;
      END LOOP;
      RETURN 'Unknown';
    END;

  FUNCTION open_emp_by_dept
  ( p_deptno IN emp.deptno%TYPE ) RETURN EMP_REFCUR
  IS
    emp_by_dept EMP_REFCUR;
    BEGIN
      OPEN emp_by_dept FOR SELECT empno, ename FROM emp
    END;
  END;
END;
```

Package body

The package body shows the declaration of several private variables: a static cursor `dept_cur`, a table type `depttab_typ`, a table variable `t_dept`, an integer variable `t_dept_max`, and a record variable `r_emp`.

```sql
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
  CURSOR dept_cur IS SELECT * FROM dept;
  TYPE depttab_typ IS TABLE OF dept%ROWTYPE
    INDEX BY BINARY_INTEGER;
  t_dept DEPTTAB_TYP;
  t_dept_max INTEGER := 1;
  r_emp EMPREC_TYP;

  FUNCTION get_dept_name
  ( p_deptno IN NUMBER ) RETURN VARCHAR2
  IS
    BEGIN
      FOR i IN 1..t_dept_max
      LOOP
        IF p_deptno = t_dept(i).deptno THEN
          RETURN t_dept(i).dname;
        END IF;
      END LOOP;
      RETURN 'Unknown';
    END;

  FUNCTION open_emp_by_dept
  ( p_deptno IN emp.deptno%TYPE ) RETURN EMP_REFCUR
  IS
    emp_by_dept EMP_REFCUR;
    BEGIN
      OPEN emp_by_dept FOR SELECT empno, ename FROM emp
    END;
  END;
END;
```
This package contains an initialization section that loads the private table variable \texttt{t\_dept} using the private static cursor \texttt{dept\_cur.t\_dept}. \texttt{dept\_cur.t\_dept} serves as a department name lookup table in the function \texttt{get\_dept\_name}.

The function \texttt{open\_emp\_by\_dept} returns a \texttt{REF CURSOR} variable for a result set of employee numbers and names for a given department. This \texttt{REF CURSOR} variable can then be passed to the procedure \texttt{fetch\_emp} to retrieve and list the individual rows of the result set. Finally, the procedure \texttt{close\_refcur} can be used to close the \texttt{REF CURSOR} variable associated with this result set.

Using anonymous blocks

The following anonymous block runs the package function and procedures. In the anonymous block's declaration section, note the declaration of cursor variable \texttt{v\_emp\_cur} using the package's public \texttt{REF CURSOR} type, \texttt{EMP\_REFCUR}. \texttt{v\_emp\_cur} contains the pointer to the result set that's passed between the package function and procedures.
The following is the result of this anonymous block:

```sql
DECLARE
  v_deptno  dept.deptno%TYPE DEFAULT 30;
  v_emp_cur emp_rpt.EMP_REFCUR;
  r_emp    emp_rpt.EMPREC_TYP;
BEGIN
  v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
  DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #'|| v_deptno || ' | ' ||
                         emp_rpt.get_dept_name(v_deptno));
  DBMS_OUTPUT.PUT_LINE('EMPNO | ENAME');
  DBMS_OUTPUT.PUT_LINE('----- | -------');
  LOOP
    FETCH v_emp_cur INTO r_emp;
    EXIT WHEN v_emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(r_emp.empno || ' | ' || r_emp.ename);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('**********************');
  DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved!');
  CLOSE v_emp_cur;
END;
```

The following anonymous block shows another way to achieve the same result. Instead of using the package procedures `fetch_emp` and `close_refcur`, the logic of these programs is coded directly into the anonymous block. In the anonymous block’s declaration section, note the addition of record variable `r_emp`, declared using the package’s public record type, `EMPREC_TYP`.

The following is the result of this anonymous block.

```sql
<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7499</td>
<td>ALLEN</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
</tr>
</tbody>
</table>
```

6 rows were retrieved

### 11.1.6 Dropping a package

The syntax for deleting an entire package or the package body is:

```
DROP PACKAGE [ BODY ]
package_name;
```
If you omit the keyword **BODY**, both the package specification and the package body are deleted, that is, the entire package is dropped. If you specify the keyword **BODY**, then only the package body is dropped. The package specification remains intact. **package_name** is the identifier of the package to drop.

The following statement destroys only the package body of **emp_admin**:

```sql
DROP PACKAGE BODY emp_admin;
```

The following statement drops the entire **emp_admin** package:

```sql
DROP PACKAGE emp_admin;
```

### 11.2 Debugging programs

The debugger gives developers and DBAs the ability to test and debug server-side programs using a graphical, dynamic environment. The types of programs that you can debug are:

- SPL stored procedures
- functions
- triggers
- packages
- PL/pgSQL functions and triggers.

### 11.2.1 Configuring the debugger

The debugger is integrated with pgAdmin 4 and EDB Postgres Enterprise Manager. If you installed EDB Postgres Advanced Server on a Windows host, pgAdmin 4 is automatically installed. The pgAdmin 4 icon is in the Windows Start menu.

You can use the debugger in two basic ways to test programs:

- **Standalone debugging** — Use the debugger to start the program to test. Supply any input parameter values required by the program. You can immediately observe and step through the code of the program. Standalone debugging is the typical method used for new programs and for initial problem investigation.

- **In-context debugging** — In-context debugging is useful if it’s difficult to reproduce a problem using standalone debugging due to complex interaction with the calling application. Using this approach, the program to test is started by an application other than the debugger. You set a global breakpoint on the program to test. The application that makes the first call to the program encounters the global breakpoint. Then the application suspends execution. At that point, the debugger takes control of the called program. You can then observe and step through the code of the called program as it runs in the context of the calling application.

After you have completely stepped through the code of the called program in the debugger, the suspended application resumes executing.

The debugging tools and operations are the same whether using standalone or in-context debugging. The difference is in how to invoke the program being debugged.

If your EDB Postgres Advanced Server host is on a CentOS or Linux system, you can use `yum` to install pgAdmin4. Open a command line, assume superuser privileges, and enter:

```
yum install edb-pgadmin4-x86_64
```

On Linux, you must also install the `edb-as<xx>-server-pldebugger` RPM package, where `<xx>` is the EDB Postgres Advanced Server version number. Information about pgAdmin 4 is available at the pgAdmin website.

The RPM installation adds the pgAdmin4 icon to your Applications menu.

Before using the debugger, edit the `postgresql.conf` file (located in the `data` subdirectory of your EDB Postgres Advanced Server home directory). Add `$libdir/plugin_debugger` to the libraries listed in the `shared_preload_libraries` configuration parameter:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/plugin_debugger'
```

- On Linux, the `postgresql.conf` file is located in `/var/lib/edb/as<xx>/data`
- On Windows, the `postgresql.conf` file is located in `C:\Program Files\edb\as<xx>\data`

Where `<xx>` is the version of EDB Postgres Advanced Server.

After modifying the `shared_preload_libraries` parameter, restart the database server.

### 11.2.2 Starting the debugger

Use pgAdmin 4 to access the debugger for standalone debugging. To open the debugger:
1. Select the name of the stored procedure or function you want to debug in the pgAdmin Browser panel. Or, to debug a package, select the specific procedure or function under the package node of the package you want to debug.

2. Select Object > Debugging > Debug

You can use the Debugger window to pass parameter values when you are standalone debugging a program that expects parameters. When you start the debugger, the Debugger window opens to display any IN or IN OUT parameters the program expects. If the program declares no IN or IN OUT parameters, the Debugger window doesn’t open.

Use the fields on the Debugger window to provide a value for each parameter:

- The Name field contains the formal parameter name.
- The Type field contains the parameter data type.
- Select the Null? check box to indicate that the parameter is a NULL value.
- Select the Expression? check box if the Value field contains an expression.
- The Value field contains the parameter value that’s passed to the program.
- Select the Use Default? check box to indicate for the program to use the value in the Default Value field.
- The Default Value field contains the default value of the parameter.

If you’re debugging a procedure or function that’s a member of a package that has an initialization section, select the Debug Package Initializer check box to step into the package initialization section. This setting allows you to debug the initialization section code before debugging the procedure or function. If you don’t select the check box, the debugger executes the package initialization section without allowing you to see or step through the individual lines of code as they execute.

After entering the desired parameter values, select Debug to start the debugging process.

Note

The Debugger window doesn’t open during in-context debugging. Instead, the application calling the program to debug must supply any required input parameter values.

After you complete a full debugging cycle by stepping through the program code, the Debugger window reopens. You can enter new parameter values and repeat the debugging cycle or end the debugging session.

### 11.2.3 Debugger interface overview

The main debugger window contains two panels:

- The top Program Body panel displays the program source code.
- The bottom Tabs panel provides a set of tabs for different information.

Use the tool bar icons located at the top panel to access debugging functions.

**The Program Body panel**

The Program Body panel displays the source code of the program that’s being debugged. The figure shows that the debugger is about to execute the `SELECT` statement. The blue indicator in the program body highlights the next statement to execute.
The Tabs panel

You can use the bottom Tabs panel to view or modify parameter values or local variables or to view messages generated by \texttt{RAISE \texttt{INFO}} and function results.

The following is the information displayed by the tabs in the panel:

- The \texttt{Parameters} tab displays the current parameter values.
- The \texttt{Local variables} tab displays the value of any variables declared in the program.
- The \texttt{Messages} tab displays any results returned by the program as it executes.
- The \texttt{Results} tab displays any program results, such as the value from the \texttt{RETURN} statement of a function.
- The \texttt{Stack} tab displays the call stack.

The Stack tab

The \texttt{Stack} tab displays a list of programs that are currently on the call stack, that is, programs that were invoked but that haven't yet completed. When a program is called, the name of the program is added to the top of the list displayed in the \texttt{Stack} tab. When the program ends, its name is removed from the list.

The \texttt{Stack} tab also displays information about program calls. The information includes:

- The location of the call in the program
- The call arguments
- The name of the program being called

Reviewing the call stack can help you trace the course of execution through a series of nested programs. The figure shows that \texttt{emp_query_caller} is about to call a subprogram named \texttt{emp_query} and \texttt{emp_query_caller} is currently at the top of the call stack.
After the call to `emp_query` executes, `emp_query` is displayed at the top of the Stack tab, and its code is displayed in the Program Body panel.

After completing execution of the subprogram, control returns to the calling program (`emp_query_caller`), now displayed at the top of the Stack tab.
11.2.4 Running the debugger

You can perform the following operations to debug a program:

- Step through the program one line at a time.
- Execute the program until you reach a breakpoint.
- View and change local variable values within the program.

Considerations when using the program

- These instructions use the standalone debugging method. To start the debugger for in-context debugging, see Setting global breakpoint for in-context debugging.

Stepping through the code

Use the tool bar icons to step through a program with the debugger. The icons serve the following purposes:

- **Step into.** Execute the currently highlighted line of code.
- **Step over.** Execute a line of code, stepping over any subfunctions invoked by the code. The subfunction executes but is debugged only if it contains a breakpoint.
- **Continue/Start.** Execute the highlighted code and continue until the program encounters a breakpoint or completes.
- **Stop.** Halt a program.

Using breakpoints

As the debugger executes a program, it pauses when it reaches a breakpoint. When the debugger pauses, you can observe or change local variables or navigate to an entry in the call stack to observe variables or set other breakpoints. The next step into, step over, or continue operation forces the debugger to resume executing with the next line of code following the breakpoint.

These are the two types of breakpoints:

- **Local breakpoint** — You can set a local breakpoint at any executable line of code in a program. The debugger pauses execution when it reaches a line where a local breakpoint was set.
- **Global breakpoint** — A global breakpoint triggers when any session reaches that breakpoint. Set a global breakpoint if you want to perform in-context debugging of a program. When you set a global breakpoint on a program, the debugging session that set the global breakpoint waits until that program is invoked in another session. Only a superuser can set a global breakpoint.

Setting a local breakpoint

To create a local breakpoint, select the grey shaded margin to the left of the line of code where you want the local breakpoint set. The spot you select must be close to the right side of the margin as in the spot where the breakpoint dot is shown on source code line 12. When the breakpoint is created, the debugger displays a dark dot in the margin, indicating a breakpoint was set at the selected line of code.
You can set as many local breakpoints as you want. Local breakpoints remain in effect for the rest of a debugging session until you remove them.

Removing a local breakpoint

To remove a local breakpoint, select the breakpoint dot. The dot disappears.

To remove all of the breakpoints from the program that currently appears in the Program Body frame, select the **Clear all breakpoints** icon.

**Note**

When you perform any of these actions, only the breakpoints in the program that currently appears in the Program Body panel are removed. Breakpoints in called subprograms or breakpoints in programs that call the program currently appearing in the Program Body panel aren’t removed.

Setting a global breakpoint for in-context debugging

To set a global breakpoint for in-context debugging:

1. In the Browser panel, select the stored procedure, function, or trigger on which you want to set the breakpoint.

2. Select **Object > Debugging > Set Breakpoint**.

To set a global breakpoint on a trigger:

1. Expand the table node that contains the trigger.

2. Select the specific trigger you want to debug.

3. Select **Object > Debugging > Set Breakpoint**.

To set a global breakpoint in a package:

1. Select the specific procedure or function under the package node of the package you want to debug.

2. Select **Object > Debugging > Set Breakpoint**.

After you select **Set Breakpoint**, the Debugger window opens and waits for an application to call the program to debug.
The PSQL client invokes the `select_emp` function on which a global breakpoint was set.

```plaintext
$ psql edb
enterprisedb
psql.bin (14.0.0, server 14.0.0)
Type "help" for help.

edb=# SELECT
      select_emp(7900);

The `select_emp` function doesn't finish until you step through the program in the debugger.

You can now debug the program using the operations like step into, step over, and continue. Or you can set local breakpoints. After you step through executing the program, the calling application (PSQL) regains control, the `select_emp` function finishes executing, and its output is displayed.

```plaintext
INFO: Number    : 7900
INFO: Name      : JAMES
INFO: Hire Date : 12/03/1981
INFO: Salary    : 950.00
INFO: Commission: 0.00
INFO: Department: SALES
```

11.3 Using enhanced SQL and other miscellaneous features
### 11.3.1 Using the COMMENT command

In addition to allowing comments on objects supported by the PostgreSQL `COMMENT` command, EDB Postgres Advanced Server supports comments on other object types. The complete supported syntax is:

```sql
COMMENT ON
  | AGGREGATE <aggregate_name> ( <aggregate_signature> )
  | CAST (<source_type> AS <target_type>)
  | COLLATION <object_name>
  | COLUMN <relation_name>.<column_name>
  | CONSTRAINT <constraint_name> ON <table_name>
  | CONSTRAINT <constraint_name> ON DOMAIN <domain_name>
  | CONVERSION <object_name>
  | DATABASE <object_name>
  | DOMAIN <object_name>
  | EXTENSION <object_name>
  | EVENT TRIGGER <object_name>
  | FOREIGN DATA WRAPPER <object_name>
  | FOREIGN TABLE <object_name>
  | FUNCTION <func_name> (['__argmode__'] ['__argname__'] <argtype> [, ...])
  | INDEX <object_name>
  | LARGE OBJECT <large_object_oid>
  | MATERIALIZED VIEW <object_name>
  | OPERATOR <operator_name> (<left_type>, <right_type>)
  | OPERATOR CLASS <object_name> USING <index_method>
  | OPERATOR FAMILY <object_name> USING <index_method>
  | PACKAGE <object_name>
  | POLICY <policy_name> ON <table_name>
  | [ PROCEDURAL ] LANGUAGE <object_name>
  | PROCEDURE <proc_name> ([][]<argmode>['__argname__'] <argtype> [, ...])
  | PUBLIC SYNONYM <object_name>
  | ROLE <object_name>
  | RULE <rule_name> ON <table_name>
  | SCHEMA <object_name>
  | SEQUENCE <object_name>
  | SERVER <object_name>
  | TABLE <object_name>
  | TABLESPACE <object_name>
  | TEXT SEARCH CONFIGURATION <object_name>
  | TEXT SEARCH DICTIONARY <object_name>
  | TEXT SEARCH PARSER <object_name>
  | TEXT SEARCH TEMPLATE <object_name>
  | TRANSFORM FOR <type_name> LANGUAGE <lang_name>
  | TRIGGER <trigger_name> ON <table_name>
  | TYPE <object_name>
```
VIEW <object_name>
) IS '<text'>

Where aggregate_signature is:

* |
  [ [argmode] ] [ <argname> ] [ <argtype> ] [ , ... ]
  [ [ [argmode] ] [ <argname> ] [ <argtype> ] [ , ... ]
ORDER BY [ [argmode] ] [ <argname> ] [ <argtype> ] [ , ... ]

Parameters

object_name

The name of the object on which you’re commenting.

AGGREGATE aggregate_name (aggregate_signature)

Include the AGGREGATE clause to create a comment about an aggregate. aggregate_name specifies the name of an aggregate. aggregate_signature specifies the associated signature in one of the following forms:

* |
  [ [argmode] ] [ <argname> ] [ <argtype> ] [ , ... ]
  [ [ [argmode] ] [ <argname> ] [ <argtype> ] [ , ... ]
ORDER BY [ [argmode] ] [ <argname> ] [ <argtype> ] [ , ... ]

Where argmode is the mode of a function, procedure, or aggregate argument. argmode can be IN, OUT, INOUT, or VARIADIC. The default is IN.

argname is the name of an aggregate argument.

argtype is the data type of an aggregate argument.

CAST (source_type AS target_type)

Include the CAST clause to create a comment about a cast. When creating a comment about a cast, source_type specifies the source data type of the cast, and target_type specifies the target data type of the cast.

COLUMN relation_name.column_name

Include the COLUMN clause to create a comment about a column. column_name specifies the name of the column to which the comment applies. relation_name is the table, view, composite type, or foreign table in which a column resides.

CONSTRAINT constraint_name ON table_name

CONSTRAINT constraint_name ON DOMAIN domain_name

Include the CONSTRAINT clause to add a comment about a constraint. When you're creating a comment about a constraint, constraint_name specifies the name of the constraint. table_name or domain_name specifies the name of the table or domain on which the constraint is defined.

FUNCTION func_name ([argmode] [argname] argtype [ , ... ])]

Include the FUNCTION clause to add a comment about a function. func_name specifies the name of the function. argmode specifies the mode of the function. argmode can be IN, OUT, INOUT, or VARIADIC. The default is IN.

argname specifies the name of a function, procedure, or aggregate argument. argtype specifies the data type of a function, procedure, or aggregate argument.

large_object_oid

large_object_oid is the system-assigned OID of the large object about which you're commenting.

OPERATOR operator_name (left_type, right_type)

Include the OPERATOR clause to add a comment about an operator. operator_name specifies the optionally schema-qualified name of an operator on which you're commenting. left_type and
right_type are the optionally schema-qualified data types of the operator’s arguments.

OPERATOR CLASS object_name USING index_method

Include the OPERATOR CLASS clause to add a comment about an operator class. object_name specifies the optionally schema-qualified name of an operator on which you’re commenting. index_method specifies the associated index method of the operator class.

OPERATOR FAMILY object_name USING index_method

Include the OPERATOR FAMILY clause to add a comment about an operator family. object_name specifies the optionally schema-qualified name of an operator family on which you’re commenting. index_method specifies the associated index method of the operator family.

POLICY policy_name ON table_name

Include the POLICY clause to add a comment about a policy. policy_name specifies the name of the policy. table_name specifies the table that the policy is associated with.

PROCEDURE proc_name [{argmode} {argname} argtype [, ...]]

Include the PROCEDURE clause to add a comment about a procedure. proc_name specifies the name of the procedure. argmode specifies the mode of the procedure. argmode can be IN, OUT, INOUT, or VARIADIC. The default is IN.

argname specifies the name of a function, procedure, or aggregate argument. argtype specifies the data type of a function, procedure, or aggregate argument.

RULE rule_name ON table_name

Include the RULE clause to specify a comment on a rule. rule_name specifies the name of the rule. table_name specifies the name of the table on which the rule is defined.

TRANSFORM FOR type_name LANGUAGE lang_name

Include the TRANSFORM FOR clause to specify a comment on a TRANSFORM.

type_name specifies the name of the data type of the transform. lang_name specifies the name of the language of the transform.

TRIGGER trigger_name ON table_name

Include the TRIGGER clause to specify a comment on a trigger. trigger_name specifies the name of the trigger. table_name specifies the name of the table on which the trigger is defined.

text

The comment, written as a string literal, or NULL to drop the comment.

Note

Names of tables, aggregates, collations, conversions, domains, foreign tables, functions, indexes, operators, operator classes, operator families, packages, procedures, sequences, text search objects, types, and views can be schema qualified.

Example

This example adds a comment to a table named new_emp:

```
COMMENT ON TABLE new_emp IS 'This table contains information about new employees.';
```

For more information about using the COMMENT command, see the PostgreSQL core documentation.

11.3.2 Configuring logical decoding on standby

Logical decoding on a standby server allows you to create a logical replication slot on a standby server that can respond to API operations such as get, peek, and advance.

For more information about logical decoding, refer to the PostgreSQL core documentation.

For a logical slot on a standby server to work, you must set the hot_standby_feedback parameter to ON on the standby. The hot_standby_feedback parameter prevents VACUUM from removing recently dead rows that are required by an existing logical replication slot on the standby server. If a slot conflict occurs on the standby, the slots are dropped.
For logical decoding on a standby to work, you must set `wal_level` to `logical` on both the primary and standby servers. If you set `wal_level` to a value other than `logical`, then slots aren't created. If you set `wal_level` to a value other than `logical` on primary, and if existing logical slots are on standby, such slots are dropped. You can't create new slots.

When transactions are written to the primary server, the activity triggers the creation of a logical slot on the standby server. If a primary server is idle, creating a logical slot on a standby server might take noticeable time.

For more information about functions that support replication, see the PostgreSQL documentation. See also this logical decoding example.

### 11.3.3 Obtaining version information

The text string output of the `version()` function displays the name of the product, its version, and the host system on which it was installed.

For EDB Postgres Advanced Server, the `version()` output is in a format similar to the PostgreSQL community version. The first text word is PostgreSQL instead of EnterpriseDB as in EDB Postgres Advanced Server version 10 and earlier.

The general format of the `version()` output is:

```
PSTRLG $PG_VERSION_EXT (EnterpriseDB EDB Postgres Advanced Server $PG_VERSION) on $host
```

So for the current EDB Postgres Advanced Server, the version string appears as follows:

```
edb@45032=# select version();
```

![version output](image)

In contrast, for EDB Postgres Advanced Server 10, the version string was the following:

```
edb=# select version();
```

![version output](image)

### 11.4 Including embedded SQL commands

EDB enhanced ECPG (the PostgreSQL precompiler) to create ECPGPlus. ECPGPlus allows you to include Pro*C-compatible embedded SQL commands in C applications when connected to an EDB Postgres Advanced Server database. When you use ECPGPlus to compile an application, the SQL code syntax is checked and translated into C.

ECPGPlus supports:

- Oracle Dynamic SQL – Method 4 (ODS-M4)
- Pro*C-compatible anonymous blocks
- A `CALL` statement compatible with Oracle databases

As part of ECPGPlus’s Pro*C compatibility, you don’t need to include the `BEGIN DECLARE SECTION` and `END DECLARE SECTION` directives.

While most ECPGPlus statements work with community PostgreSQL, the `CALL` statement and the `EXECUTE...END EXEC` statement work only when the client application is connected to EDB Postgres Advanced Server.

#### 11.4.1 ECPGPlus overview

EDB enhanced ECPG (the PostgreSQL precompiler) to create ECPGPlus. ECPGPlus is a Pro*C-compatible version of the PostgreSQL C precompiler. ECPGPlus translates a program that combines C code and embedded SQL statements into an equivalent C program. As it performs the translation, ECPGPlus verifies that the syntax of each SQL construct is correct.
About ECPGPlus

The following diagram charts the path of a program containing embedded SQL statements as it's compiled into an executable:

Compilation of a program containing embedded SQL statements

To produce an executable from a C program that contains embedded SQL statements:

1. Pass the program (my_program.pgc in the diagram) to the ECPGPlus precompiler. ECPGPlus translates each SQL statement in my_program.pgc into C code that calls the ecpglib API and produces a C program (my_program.c).
2. Pass the C program to a C compiler. The C compiler generates an object file (my_program.o).
3. Pass the object file (my_program.o) as well as the ecpglib library file and any other required libraries to the linker, which in turn produces the executable (my_program).

While the ECPGPlus preprocessor validates the syntax of each SQL statement, it can’t validate the semantics. For example, the preprocessor confirms that an INSERT statement is syntactically correct, but it can’t confirm that the table mentioned in the INSERT statement exists.

Behind the scenes

A client application contains a mix of C code and SQL code made up of the following elements:

- C preprocessor directives
- C declarations (variables, types, functions, …)
- C definitions (variables, types, functions, …)
- SQL preprocessor directives
- SQL statements

For example:

```c
#include <stdio.h>
2 EXEC SQL INCLUDE sqlca;
3
4 extern void printInt(char *label, int val);
5 extern void printStr(char *label, char *val);
6 extern void printFloat(char *label, float val);
7
8 void displayCustomer(int custNumber)
9 {
10 EXEC SQL BEGIN DECLARE SECTION;
11 VARCHAR
12 custName[50];
```
In this code fragment:

- Line 1 specifies a directive to the C preprocessor.
  
  C preprocessor directives can be interpreted or ignored. The option is controlled by a command line option (\(-\text{C} \text{ PROC}\) entered when you invoke ECPGPlus. In either case, ECPGPlus copies each C preprocessor directive to the output file (4) without change. Any C preprocessor directive found in the source file appears in the output file.

- Line 2 specifies a directive to the SQL preprocessor.
  
  SQL preprocessor directives are interpreted by the ECPGPlus preprocessor and aren't copied to the output file.

- Lines 4 through 6 contain C declarations.
  
  C declarations are copied to the output file without change, except that each `VARCHAR` declaration is translated into an equivalent `struct` declaration.

- Lines 10 through 14 contain an embedded-SQL declaration section.
  
  C variables that you refer to in SQL code are known as host variables. If you invoke the ECPGPlus preprocessor in Pro*C mode (\(-\text{C} \text{ PROC}\)), you can refer to any C variable in a SQL statement. Otherwise you must declare each host variable in a `BEGIN/END DECLARATION SECTION` pair.

- Lines 16 through 19 contain a SQL statement.
  
  SQL statements are translated into calls to the ECPGPlus runtime library.

- Lines 21 through 23 contain C code.
  
  C code is copied to the output file without change.

Prefix any SQL statement with `EXEC SQL`. The SQL statement extends to the next (unquoted) semicolon. For example:

```c
printf("Updating employee salaries\n");
EXEC SQL UPDATE emp SET sal = sal * 1.25;
EXEC SQL COMMIT;
printf("Employee salaries updated\n");
```

When the preprocessor encounters this code fragment, it passes the C code (the first line and the last line) to the output file without translation and converts each `EXEC SQL` statement into a call to an `ecpglib` function. The result is similar to the following:

```c
printf("Updating employee salaries\n");

{ ECPGdo(__LINE__, 8, 1, NULL, 0,
  ECPGst_normal,
  "update emp set sal = sal * 1.25",
  ECPGt_EDIT, ECPGt_EORT);
}
{ ECPGtrans(__LINE__, NULL, "commit");
```
11.4.2 Installing and configuring ECPGPlus

On Windows, ECPGPlus is installed by the EDB Postgres Advanced Server installation wizard as part of the Database Server component. On Linux, you install ECPGPlus by running an executable.

Installing ECPGPlus

On Linux, install with the `edb-as<xx>-server-devel` RPM package, where `<xx>` is the EDB Postgres Advanced Server version number. On Linux, the executable is located in:

```
/usr/edb/as14/bin
```

On Windows, the executable is located in:

```
C:\Program Files\edb\as14\bin
```

When invoking the ECPGPlus compiler, the executable must be in your search path (`%PATH%` on Windows, `$PATH` on Linux). For example, the following commands set the search path to include the directory that holds the ECPGPlus executable file `ecpg`.

**On Windows:**

```
set EDB_PATH=C:\Program Files\edb\as14\bin
set PATH=%EDB_PATH%;%PATH%
```

**On Linux:**

```
export EDB_PATH=/usr/edb/as14/bin
export PATH=$EDB_PATH:$PATH
```

Constructing a makefile

A makefile contains a set of instructions that tell the make utility how to transform a program written in C that contains embedded SQL into a C program. To try the examples, you need:

- A C compiler and linker
- The make utility
- ECPGPlus preprocessor and library
- A makefile that contains instructions for ECPGPlus

The following code is an example of a makefile for the samples included in this documentation. To use the sample code, save it in a file named `makefile` in the directory that contains the source code file.

```makefile
INCLUDES = -I$(shell pg_config --includedir)
LIBPATH = -L $(shell pg_config -l)
CFLAGS += $(INCLUDES)
LDFLAGS += -g
LDDLBS += $(LIBPATH) -lecpg
-Lpq

.SUFFIXES: .pgc .pc

.pgc.c:
  ecpg -c $(INCLUDES)

.pc.c:
  ecpg -C PROC -c $(INCLUDES)
```

The first two lines use the `pg_config` program to locate the necessary header files and library directories:

```makefile
INCLUDES = -I$(shell pg_config --includedir)
```
The pg_config program is shipped with EDB Postgres Advanced Server. make knows to use the CFLAGS variable when running the C compiler and LDFLAGS and LDLIBS when invoking the linker. ECPG programs must be linked against the ECPG runtime library (-lecpg) and the libpq library (-lpq).

CFLAGS += $(INCLUDES)
-L $(LIBPATH) -lecpg -lpq

The pg_config program is shipped with EDB Postgres Advanced Server.

make knows to use the CFLAGS variable when running the C compiler and LDFLAGS and LDLIBS when invoking the linker. ECPG programs must be linked against the ECPG runtime library (-lecpg) and the libpq library (-lpq).

CFLAGS += $(INCLUDES)
-L $(LIBPATH) -lecpg -lpq

The sample makefile tells make how to translate a .pgc or a .pc file into a C program. Two lines in the makefile specify the mode in which the source file is compiled. The first compile option is:

```
pgc:  
    ecpg -c $(INCLUDES)
```

The first option tells make how to transform a file that ends in .pgc (presumably, an ECPG source file) into a file that ends in .c (a C program), using community ECPG, without the ECPGPlus enhancements. It invokes the ECPG precompiler with the -c flag, which instructs the compiler to convert SQL code into C, using the value of the INCLUDES variable and the name of the .pgc file.

```
pc:  
    ecpg -C PROC -c $(INCLUDES)
```

The second option tells make how to transform a file that ends in .pgc (an ECPG source file) into a file that ends in .c (a C program) using the ECPGPlus extensions. It invokes the ECPG precompiler with the -c flag, which instructs the compiler to convert SQL code to C. It also uses the -C PROC flag, which instructs the compiler to use ECPGPlus in Pro*C-compatibility mode, using the value of the INCLUDES variable and the name of the .pgc file.

When you run make, pass the name of the ECPG source code file you want to compile. For example, to compile an ECPG source code file named customer_list.pgc, use the command:

```
make customer_list
```

The make utility:

1. Consults the makefile located in the current directory.
2. Discovers that the makefile contains a rule that compiles customer_list.pgc into a C program (customer_list.c).
3. Uses the rules built into make to compile customer_list.c into an executable program.

ECPGPlus command line options

In the sample makefile, make includes the -C option when invoking ECPGPlus to invoke ECPGPlus in Pro*C-compatible mode.

If you include the -C PROC keywords at the command line, in addition to the ECPG syntax, you can use Pro*C command line syntax. For example:

```
$ ecpg -C PROC INCLUDE=/usr/edb/as14/include acct_update.c
```

To display a complete list of the other ECPGPlus options available, in the ECPGPlus installation directory, enter:

```
./ecpg --help
```

The command line options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>Generate C code from embedded SQL code.</td>
</tr>
<tr>
<td>-C &lt;mode&gt;</td>
<td>Specify a compatibility mode:</td>
</tr>
<tr>
<td></td>
<td>INFORMIX</td>
</tr>
<tr>
<td></td>
<td>INFORMIX_SE</td>
</tr>
<tr>
<td></td>
<td>PROC</td>
</tr>
<tr>
<td>-D &lt;symbol&gt;</td>
<td>Define a preprocessor symbol.</td>
</tr>
<tr>
<td></td>
<td>The -D keyword isn't supported when compiling in PROC mode. Instead, use the Oracle-style 'DEFINE=' clause.</td>
</tr>
<tr>
<td>-h</td>
<td>Parse a header file. This option includes option &quot;-c&quot;.</td>
</tr>
<tr>
<td>-I</td>
<td>Parse system. Include files as well.</td>
</tr>
<tr>
<td>-i</td>
<td>Search &lt;directory&gt; for include files.</td>
</tr>
<tr>
<td>-a</td>
<td>Write the result to &lt;outfile&gt;.</td>
</tr>
</tbody>
</table>
### 11.4.3 Using embedded SQL

These two examples show how to use embedded SQL with EDB Postgres Advanced Server.

#### Example: A simple query

The first code sample shows how to execute a `SELECT` statement that returns a single row, storing the results in a group of host variables. After declaring host variables, it connects to the `edb` sample database using a hard-coded role name and the associated password and queries the `emp` table. The query returns the values into the declared host variables. After checking the value of the `NULL` indicator variable, it prints a simple result set onscreen and closes the connection.

```c
/* print_emp.pgc */
#include <stdio.h>
int main(void) {
    EXEC SQL BEGIN DECLARE SECTION;
    int v_empno;
    char v_ename[40];
    double v_sal;
    double v_comm;
    short v_comm_ind;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL WHENEVER SQLERROR sqlprint;
    EXEC SQL CONNECT TO edb
        USER 'alice' IDENTIFIED BY '1safepw';
    EXEC SQL
        SELECT empno, ename, sal, comm
        INTO :v_empno, :v_ename, :v_sal, :v_comm INDICATOR:v_comm_ind
        FROM emp
        WHERE empno = 7369;
```
The code sample begins by including the prototypes and type definitions for the C `stdio` library and then declares the `main` function:

```c
#include <stdio.h>
int main(void)
{
...
```

Next, the application declares a set of host variables used to interact with the database server:

```c
EXEC SQL BEGIN DECLARE SECTION;
    int v_empno;
    char v_ename[40];
    double v_sal;
    double v_comm;
    short v_comm_ind;
EXEC SQL END DECLARE SECTION;
```

If you plan to precompile the code in PROC mode, you can omit the `BEGIN DECLARE...END DECLARE` section. For more information about declaring host variables, see Declaring host variables.

The data type associated with each variable in the declaration section is a C data type. Data passed between the server and the client application must share a compatible data type. For more information about data types, see the Supported C data types.

The next statement tells the server how to handle an error:

```c
EXEC SQL WHENEVER SQLERROR sqlprint;
```

If the client application encounters an error in the SQL code, the server prints an error message to `stderr` (standard error), using the `sqlprint()` function supplied with `ecpglib`. The next `EXEC SQL` statement establishes a connection with EDB Postgres Advanced Server:

```c
EXEC SQL CONNECT TO edb
    USER 'alice' IDENTIFIED BY '1safepwd';
```

In this example, the client application connects to the `edb` database using a role named alice with a password of `1safepwd`.

The code then performs a query against the `emp` table:

```c
EXEC SQL
    SELECT empno, ename, sal, comm INTO :v_empno, :v_ename, :v_sal, :v_comm INDICATOR :v_comm_ind
    FROM emp
WHERE empno = 7369;
```

The query returns information about employee number 7369.

The `SELECT` statement uses an `INTO` clause to assign the retrieved values (from the `empno`, `ename`, `sal`, and `comm` columns) into the `:v_empno`, `:v_ename`, `:v_sal`, and `:v_comm` host variables (and the `:v_comm_ind` null indicator). The first value retrieved is assigned to the first variable listed in the `INTO` clause, the second value is assigned to the second variable, and so on.

The `comm` column contains the commission values earned by an employee and can potentially contain a `NULL` value. The statement includes the `INDICATOR` keyword and a host variable to hold a null indicator.

The code checks the null indicator and displays the appropriate results:
If the null indicator is 0 (that is, false), the `comm` column contains a meaningful value, and the `printf` function displays the commission. If the null indicator contains a non-zero value, `comm` is `NULL`, and `printf` displays a value of `NULL`. A host variable (other than a null indicator) contains no meaningful value if you fetch a `NULL` into that host variable. You must use null indicators to identify any value that might be `NULL`.

The final statement in the code sample closes the connection to the server:

```c
EXEC SQL DISCONNECT;
```

Using indicator variables

The previous example included an indicator variable that identifies any row in which the value of the `comm` column (when returned by the server) was `NULL`. An indicator variable is an extra host variable that denotes if the content of the preceding variable is `NULL` or truncated. The indicator variable is populated when the contents of a row are stored. An indicator variable can contain the following values:

<table>
<thead>
<tr>
<th>Indicator value</th>
<th>Denotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>If an indicator is less than 0.</td>
<td>The value returned by the server was <code>NULL</code>.</td>
</tr>
<tr>
<td>If an indicator variable is equal to 0.</td>
<td>The value returned by the server was not <code>NULL</code>, and was not truncated.</td>
</tr>
<tr>
<td>If an indicator variable is greater than 0.</td>
<td>The value returned by the server was truncated when stored in the host variable.</td>
</tr>
</tbody>
</table>

When including an indicator variable in an `INTO` clause, you don't need to include the optional `INDICATOR` keyword.

You can omit an indicator variable if you're certain that a query never returns a `NULL` value into the corresponding host variable. If you omit an indicator variable and a query returns a `NULL` value, `ecpglib` raises a runtime error.

Declaring host variables

You can use a host variable in a SQL statement at any point that a value can appear in that statement. A host variable is a C variable that you can use to pass data values from the client application to the server and return data from the server to the client application. A host variable can be:

- An array
- A `typedef`
- A pointer
- A `struct`
- Any scalar C data type

The code fragments that follow show using host variables in code compiled in `PROC` mode and in non-`PROC` mode. The SQL statement adds a row to the `dept` table, inserting the values returned by the variables `v_deptno`, `v_dname`, and `v_loc` into the `deptno` column, the `dname` column, and the `loc` column, respectively.

If you're compiling in `PROC` mode, you can omit the `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION` directives. `PROC` mode permits you to use C function parameters as host variables:

```c
void addDept(int v_deptno, char v_dname, char v_loc)
{
    EXEC SQL INSERT INTO dept VALUES( :v_deptno, :v_dname, :v_loc);
}
```

If you aren't compiling in `PROC` mode, you must wrap embedded variable declarations with the `EXEC SQL BEGIN DECLARE SECTION` and the `EXEC SQL END DECLARE SECTION` directives:

```c
void addDept(int v_deptno, char v_dname, char v_loc)
{
    EXEC SQL BEGIN DECLARE SECTION;
    int v_deptno_copy = v_deptno;
    char v_dname_copy[14+1] = v_dname;
    char v_loc_copy[13+1] = v_loc;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL INSERT INTO dept VALUES( :v_deptno_copy, :v_dname_copy, :v_loc_copy);
}
```
You can also include the `INTO` clause in a `SELECT` statement to use the host variables to retrieve information:

```sql
EXEC SQL SELECT deptno, dname, loc
  INTO :v_deptno, :v_dname, v_loc FROM dept;
```

Each column returned by the `SELECT` statement must have a type-compatible target variable in the `INTO` clause. This is a simple example that retrieves a single row. To retrieve more than one row, you must define a cursor, as shown in the next example.

### Example: Using a cursor to process a result set

The code sample that follows shows using a cursor to process a result set. Four basic steps are involved in creating and using a cursor:

1. Use the `DECLARE CURSOR` statement to define a cursor.
2. Use the `OPEN CURSOR` statement to open the cursor.
3. Use the `FETCH` statement to retrieve data from a cursor.
4. Use the `CLOSE CURSOR` statement to close the cursor.

After declaring host variables, the example connects to the `edb` database using a user-supplied role name and password and queries the `emp` table. The query returns the values into a cursor named `employees`. The code sample then opens the cursor and loops through the result set a row at a time, printing the result set. When the sample detects the end of the result set, it closes the connection.
EXEC SQL BEGIN DECLARE SECTION;
  char *username = argv[1];
  char *password = argv[2];
  int v_empno;
  char v_ename[40];
  double v_sal;
  double v_comm;
  short v_comm_ind;
EXEC SQL END DECLARE SECTION;

arg[] is an array that contains the command line arguments entered when the user runs the client application. argv[1] contains the first command line argument (in this case, a username), and argv[2] contains the second command line argument (a password). The example omits the error-checking code you would normally include in a real-world application. The declaration initializes the values of username and password, setting them to the values entered when the user invoked the client application.

You might think that you can refer to argv[1] and argv[2] in a SQL statement instead of creating a separate copy of each variable. However, that doesn't work. All host variables must be declared in a BEGIN/END DECLARE SECTION, unless you're compiling in PROC mode. Since argv is a function parameter (not an automatic variable), you can't declare it in a BEGIN/END DECLARE SECTION. If you're compiling in PROC mode, you can refer to any C variable in a SQL statement.

The next statement tells the server to respond to an SQL error by printing the text of the error message returned by ECPGPlus or the database server:

EXEC SQL WHENEVER SQLERROR
  sqlprint;

Then, the client application establishes a connection with EDB Postgres Advanced Server:

EXEC SQL CONNECT TO edb USER :username IDENTIFIED BY :password;

The CONNECT statement creates a connection to the edb database, using the values found in the :username and :password host variables to authenticate the application to the server when connecting.

The next statement declares a cursor named employees:

EXEC SQL DECLARE employees CURSOR FOR
  SELECT empno, ename, sal, comm
  FROM emp;

employees contains the result set of a SELECT statement on the emp table. The query returns employee information from the following columns: empno, ename, sal, and comm. Notice that when you declare a cursor, you don't include an INTO clause. Instead, you specify the target variables (or descriptors) when you FETCH from the cursor.
Before fetching rows from the cursor, the client application must `OPEN` the cursor:

```sql
EXEC SQL OPEN
employees;
```

In the subsequent `FETCH` section, the client application loops through the contents of the cursor. The client application includes a `WHENEVER` statement that instructs the server to `break` (that is, terminate the loop) when it reaches the end of the cursor:

```sql
EXEC SQL WHENEVER NOT FOUND DO
break;
```

**FETCH**

The client application then uses a `FETCH` statement to retrieve each row from the cursor `INTO` the previously declared host variables:

```sql
for (;;) {
  EXEC SQL FETCH NEXT FROM
  employees
  INTO
    :v_empno, :v_ename, :v_sal, :v_comm INDICATOR :v_comm_ind;
}
```

The `FETCH` statement uses an `INTO` clause to assign the retrieved values into the `:v_empno`, `:v_ename`, `:v_sal`, and `:v_comm` host variables (and the `:v_comm_ind` null indicator). The first value in the cursor is assigned to the first variable listed in the `INTO` clause, the second value is assigned to the second variable, and so on.

The `FETCH` statement also includes the `INDICATOR` keyword and a host variable to hold a null indicator. If the `comm` column for the retrieved record contains a `NULL` value, `v_comm_ind` is set to a non-zero value, indicating that the column is `NULL`.

The code then checks the null indicator and displays the appropriate results:

```sql
if (v_comm_ind)
  printf("empno(%d), ename(%s), sal(%.2f)
    comm(NULL)\n",
    v_empno, v_ename, v_sal);
else
  printf("empno(%d), ename(%s), sal(%.2f)
    comm(%.2f)\n",
    v_empno, v_ename, v_sal,
    v_comm);
}
```

If the null indicator is `0` (that is, `false`), `v_comm` contains a meaningful value, and the `printf` function displays the commission. If the null indicator contains a non-zero value, `comm` is `NULL`, and `printf` displays the string 'NULL'. A host variable (other than a null indicator) contains no meaningful value if you fetch a `NULL` into that host variable. You must use null indicators for any value which may be `NULL`.

**CLOSE**

The final statements in the code sample close the cursor `employees` and the connection to the server:

```sql
EXEC SQL CLOSE
employees;
EXEC SQL DISCONNECT;
```

### 11.4.4 Using descriptors

Dynamic SQL allows a client application to execute SQL statements that are composed at runtime. This ability is useful when you don’t know the content or form for a statement when you’re writing a client application. ECFSPlus doesn’t allow you to use a host variable in place of an identifier (such as a table name, column name, or index name). Instead, use dynamic SQL statements to build a string that includes the information, and then execute that string. The string is passed between the client and the server in the form of a descriptor. A descriptor is a data structure that contains both the data and the information about the shape of the data.

**Overview of the client application flow**

A client application must use a `GET DESCRIPTOR` statement to retrieve information from a descriptor. The basic flow of a client application using dynamic SQL is:
1. Use an ALLOCATE DESCRIPTOR statement to allocate a descriptor for the result set (select list).
2. Use an ALLOCATE DESCRIPTOR statement to allocate a descriptor for the input parameters (bind variables).
3. Obtain, assemble, or compute the text of an SQL statement.
4. Use a PREPARE statement to parse and check the syntax of the SQL statement.
5. Use a DESCRIBE statement to describe the select list into the select-list descriptor.
6. Use a DESCRIBE statement to describe the input parameters into the bind-variables descriptor.
7. Prompt the user (if required) for a value for each input parameter. Use a SET DESCRIPTOR statement to assign the values into a descriptor.
8. Use a DECLARE CURSOR statement to define a cursor for the statement.
9. Use an OPEN CURSOR statement to open a cursor for the statement.
10. Use a FETCH statement to fetch each row from the cursor, storing each row in select-list descriptor.
11. Use a GET DESCRIPTOR command to interrogate the select-list descriptor to find the value of each column in the current row.
12. Use a CLOSE CURSOR statement to close the cursor and free any cursor resources.

### Descriptor attributes

A descriptor can contain these attributes.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Attribute description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARDINALITY</td>
<td>integer</td>
<td>The number of rows in the result set.</td>
</tr>
<tr>
<td>DATA</td>
<td>N/A</td>
<td>The data value.</td>
</tr>
</tbody>
</table>
| DATETIME_INTERVAL_CODE | integer      | If TYPE is 9:
|                    |              | 1 - DATE
|                    |              | 2 - TIME
|                    |              | 3 - TIMESTAMP
|                    |              | 4 - TIME WITH TIMEZONE
|                    |              | 5 - TIMESTAMP WITH TIMEZONE             |
| DATETIME_INTERVAL_PRECISION | integer   | Unused.                                |
| INDICATOR           | integer      | Indicates a NULL or truncated value.   |
| KEY_MEMBER          | integer      | Unused (returns FALSE).                |
| LENGTH              | integer      | The data length (as stored on server). |
| NAME                | string       | The name of the column in which the data resides. |
| NULLABLE            | integer      | Unused (returns TRUE).                 |
| OCTET_LENGTH        | integer      | The data length (in bytes) as stored on server. |
| PRECISION           | integer      | The data precision (if the data is of numeric type). |
| RETURNED_LENGTH     | integer      | Actual length of data item.            |
| RETURNED_OCTET_LENGTH | integer  | Actual length of data item.            |
| SCALE               | integer      | The data scale (if the data is of numeric type). |
Field | Type | Attribute description
---|---|---
| A numeric code that represents the data type of the column:
| 1 - SQL3_CHARACTER
| 2 - SQL3_NUMERIC
| 3 - SQL3_DECIMAL
| 4 - SQL3_INTEGER
| 5 - SQL3_SMALLINT
| 6 - SQL3_FLOAT
| 7 - SQL3_REAL
| 8 - SQL3_DOUBLE_PRECISION
| 9 - SQL3_DATE_TIME_TIMESTAMP
| 10 - SQL3_INTERVAL
| 12 - SQL3_CHARACTER_VARYING
| 13 - SQL3_ENUMERATED
| 14 - SQL3_BIT
| 15 - SQL3_BIT_VARYING
| 16 - SQL3_BOOLEAN

Example: Using a descriptor to return data

The following simple application executes an SQL statement entered by an end user. The code sample shows:

- How to use a SQL descriptor to execute a `SELECT` statement.
- How to find the data and metadata returned by the statement.

The application accepts an SQL statement from an end user, tests the statement to see if it includes the `SELECT` keyword, and executes the statement.

Using a SQL descriptor to execute a `SELECT` statement

When invoking the application, an end user must provide the name of the database on which to perform the SQL statement and a string that contains the text of the query.

For example, a user might invoke the sample with the following command:

```
./exec_stmt edb "SELECT * FROM emp"
```
EXEC SQL BEGIN DECLARE SECTION;
  char  *db  = argv[1];
  char  *stmt = argv[2];
  int  col_count;
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO :db;

EXEC SQL ALLOCATE DESCRIPTOR parse_desc;
  EXEC SQL PREPARE query FROM :stmt;
  EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;
  EXEC SQL GET DESCRIPTOR 'parse_desc' :col_count = COUNT;
if( col_count == 0 )
{
  EXEC SQL EXECUTE IMMEDIATE :stmt;
  if( sqlca.sqlcode >= 0 )
    EXEC SQL COMMIT;
}
else
{
  int row;

  EXEC SQL ALLOCATE DESCRIPTOR row_desc;
  EXEC SQL DECLARE my_cursor CURSOR FOR query;
  EXEC SQL OPEN my_cursor;

  for( row = 0; ; row++ )
  {
    EXEC SQL BEGIN DECLARE SECTION;
      int col;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL FETCH IN my_cursor INTO SQL DESCRIPTOR row_desc;
    if( sqlca.sqlcode != 0 )
      break;
    if( row == 0
      printf_meta_data( "row_desc" );
      printf("[RECORD %d]\n", row);
    for( col = 1; col <= col_count; col++ )
    {
      EXEC SQL BEGIN DECLARE SECTION;
        short ind;
      varchar val[40+1];
      varchar name[20+1];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL GET DESCRIPTOR 'row_desc'
      VALUE :col
        :val = DATA, :ind = INDICATOR, :name = NAME;
    if( :ind == -1

printf(" %s : <null>", name.arr);
    else if (ind > 0)
        printf(" %s : <truncated>", name.arr);
    else
        printf(" %s : %s", name.arr, val.arr);
}
printf("\n");
    printf("%d rows\n", row);
}
exit( 0 );
}

static void print_meta_data( char *desc_name )
{
    EXEC SQL BEGIN DECLARE SECTION;
        char *desc = desc_name;
        int col_count;
        int col;
    EXEC SQL END DECLARE SECTION;

    static char *types[] =
    {
        "unused",
        "CHARACTER",
        "NUMERIC",
        "DECIMAL",
        "INTEGER",
        "SMALLINT",
        "FLOAT",
        "REAL",
        "DOUBLE",
        "DATE_TIME",
        "INTERVAL",
        "unused",
        "CHARACTER_VARYING",
        "ENUMERATED",
        "BIT",
        "BIT_VARYING",
        "BOOLEAN",
        "abstract",
    };
    EXEC SQL GET DESCRIPTOR :desc :col_count = count;

    printf("%s\n", md1);
    printf("%s\n", md2);
    printf("%s\n", md3);

    for( col = 1; col <= col_count; col++ )
    {
        printf(" %s : %s", name.arr, val.arr);
    }
EXEC SQL BEGIN DECLARE SECTION;
    int type;
    int ret_len;
    varchar name[21];
EXEC SQL END DECLARE SECTION;

EXEC SQL GET DESCRIPTOR :desc
VALUE :col
    :name = NAME,
    :type = TYPE,
    :ret_len = RETURNED_OCTET_LENGTH;

    if( type > 0 && type < SQL3_abstract )
        type_name = types[type];
    else
        type_name = "unknown";

    printf( "%02d: %-20s %-17s %04d
    
", col, name.arr, type_name, ret_len);
}
}

printf( "\n" );

}/**
************************************************************

#include <stdio.h>
#include <stdlib.h>
#include <sql3types.h>
#include <sqlca.h>

The code sample begins by including the prototypes and type definitions for the C stdio and stdlib libraries, SQL data type symbols, and the SQLCA (SQL communications area) structure:

#include <stdio.h>
#include <stdlib.h>
#include <sql3types.h>
#include <sqlca.h>

The sample provides minimal error handling. When the application encounters a SQL error, it prints the error message to screen:

EXEC SQL WHENEVER SQLERROR
SQLPRINT;

Finding the data and metadata returned by the statement

The application includes a forward-declaration for a function named print_meta_data() that prints the metadata found in a descriptor:

static void print_meta_data( char * desc_name );

The following code specifies the column header information that the application uses when printing the metadata:

    char *md1 = "col field data ret";
    char *md2 = "num name type len";
    char *md3 = "--- -------------------- ----------------- ---
    
";

int main( int argc, char *argv[] )
{

The following declaration section identifies the host variables to contain the name of the database the application connects to, the content of the SQL statement, and a host variable for the number of columns in the result set (if any).

EXEC SQL BEGIN DECLARE SECTION;
    char *db = argv[1];
    char *stmt = argv[2];
    int col_count;
EXEC SQL END DECLARE SECTION;

The application connects to the database using the default credentials:

EXEC SQL CONNECT TO
    :db;

Next, the application allocates a SQL descriptor to hold the metadata for a statement:
The application uses a `PREPARE` statement to check the syntax of the string provided by the user:

```sql
EXEC SQL PREPARE query FROM :stmt;
```

It also uses a `DESCRIBE` statement to move the metadata for the query into the SQL descriptor:

```sql
EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;
```

Then, the application interrogates the descriptor to discover the number of columns in the result set and stores that in the host variable `col_count`.

```sql
EXEC SQL GET DESCRIPTOR parse_desc :col_count = COUNT;
```

If the column count is zero, the end user didn't enter a `SELECT` statement. The application uses an `EXECUTE IMMEDIATE` statement to process the contents of the statement:

```sql
if ( col_count == 0 )
{
    EXEC SQL EXECUTE IMMEDIATE :stmt;
}
```

If the statement executes successfully, the application performs a `COMMIT`:

```sql
if ( sqlca.sqlcode >= 0 )
    EXEC SQL COMMIT;
else
{
```

If the statement entered by the user is a `SELECT` statement (which we know because the column count is non-zero), the application declares a variable named `row`:

```sql
    int row;
```

Then, the application allocates another descriptor that holds the description and the values of a specific row in the result set:

```sql
EXEC SQL ALLOCATE DESCRIPTOR row_desc;
```

The application declares and opens a cursor for the prepared statement:

```sql
EXEC SQL DECLARE my_cursor CURSOR FOR query;
EXEC SQL OPEN my_cursor;
```

It loops through the rows in the result set:

```sql
for( row = 0; ; row++ )
{
    EXEC SQL BEGIN DECLARE SECTION;
    int col;
    EXEC SQL END DECLARE SECTION;
```

Then, it uses a `FETCH` to retrieve the next row from the cursor into the descriptor:

```sql
EXEC SQL FETCH IN my_cursor INTO SQL DESCRIPTOR row_desc;
```

The application confirms that the `FETCH` didn't fail. If the `FETCH` fails, the application reached the end of the result set and breaks the loop:

```sql
if ( sqlca.sqlcode != 0 )
    break;
```

The application checks to see if this is the first row of the cursor. If it is, the application prints the metadata for the row:

```sql
if ( row == 0 )
    print_meta_data( "row_desc" );
```

Next, it prints a record header containing the row number.
Then, it loops through each column in the row:

```c
for( col = 1; col <= col_count; col++ ) {
    EXEC SQL BEGIN DECLARE SECTION;
    short
    ind;
    varchar
    val[40+1];
    varchar name[20+1];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL GET DESCRIPTOR
    row_desc
    VALUE :val DATA, :ind INDICATOR, :name NAME;

    if( ind == -1 )
        printf( " %-20s : <null>
", name.arr );
    else if( ind > 0 )
        printf( " %-20s : <truncated>
", name.arr );
    else
        printf( " %-20s : %s
", name.arr, val.arr );
    printf( "\n" );
}
```

When the loop terminates, the application prints the number of rows fetched and exits:

```c
printf( "%d rows\n", row );
}
exit( 0 );
```

The `print_meta_data()` function extracts the metadata from a descriptor and prints the name, data type, and length of each column:

```c
static void print_meta_data( char *desc_name ) {
    EXEC SQL BEGIN DECLARE SECTION;
    char *desc = desc_name;
    int col_count;
    int col;
    EXEC SQL END DECLARE SECTION;

    The application declares host variables:

    EXEC SQL BEGIN DECLARE SECTION;
    char *desc = desc_name;
    int col_count;
    int col;
    EXEC SQL END DECLARE SECTION;

    The application then defines an array of character strings that map data type values (*numeric*) into data type names. We use the numeric value found in the descriptor to index into this array. For example, if we find that a given column is of type 2, we can find the name of that type (*NUMERIC*) by writing `types[2]`.

```c
static char *types[] =
{
    "unused",
    "CHARACTER",
    "NUMERIC",
    "DECIMAL",
    ""};
```
The application retrieves the column count from the descriptor. The program refers to the descriptor using a host variable (desc) that contains the name of the descriptor. In most scenarios, you use an identifier to refer to a descriptor. In this case, the caller provided the descriptor name, so we can use a host variable to refer to the descriptor.

The application prints the column headers defined at the beginning of this application:

```
EXEC SQL GET DESCRIPTOR :desc :col_count = count;
```

The application prints the column headers defined at the beginning of this application:

```
printf ( "%s\n", md1 );
printf ( "%s\n", md2 );
printf ( "%s\n", md3 );
```

Then, it loops through each column found in the descriptor and prints the name, type, and length of each column.

```
for( col = 1; col <= col_count; col++ )
{
  EXEC SQL BEGIN DECLARE SECTION;
    int type;
    int ret_len;
    varchar name[21];
  EXEC SQL END DECLARE SECTION;
  char *type_name;

  EXEC SQL GET DESCRIPTOR :desc
  VALUE :col :
  :name = NAME,
  :type = TYPE,
  :ret_len = RETURNED_OCTET_LENGTH;

  if( type > 0 && type < SQL3_abstract )
    type_name = types[type];
  else
    type_name = "unknown";

  printf( "%02d: %-20s %-17s %04d\n", col, name.arr, type_name, ret_len );
}
printf( "\n" );
```
Invoke the sample application with the following command:

```sql
./exec_stmt test "SELECT * FROM emp WHERE empno IN(7902, 7934)"
```

The application returns:

```
<table>
<thead>
<tr>
<th>col</th>
<th>field</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td></td>
<td></td>
</tr>
<tr>
<td>num</td>
<td>name</td>
<td>type</td>
</tr>
<tr>
<td>len</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>01</td>
<td>empno</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>0084</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>ename</td>
<td>CHARACTER_VARYING</td>
</tr>
<tr>
<td>0084</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>job</td>
<td>CHARACTER_VARYING</td>
</tr>
<tr>
<td>0087</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>mgr</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>0084</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>hiredate</td>
<td>DATE_TIME</td>
</tr>
<tr>
<td>0018</td>
<td></td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>sal</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>0087</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>comm</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>0080</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>deptno</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>0082</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[RECORD 1]

- empno          : 7902
- ename          : FORD
- job            : ANALYST
- mgr            : 7566
- hiredate       : 03-DEC-81
- sal            : 3000.00
- comm           : <null>
- deptno         : 20

[RECORD 2]

- empno          : 7934
- ename          : MILLER
- job            : CLERK
- mgr            : 7782
- hiredate       : 23-JAN-82
- sal            : 1300.00
- comm           : <null>
- deptno         : 10

2 rows
```

11.4.5 Building and executing dynamic SQL statements

The following examples show four techniques for building and executing dynamic SQL statements. Each example shows processing a different combination of statement and input types:

- The **first example** shows processing and executing a SQL statement that doesn't contain a `SELECT` statement and doesn't require input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 1.
- The **second example** shows processing and executing a SQL statement that doesn't contain a `SELECT` statement and contains a known number of input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 2.
- The **third example** shows processing and executing a SQL statement that might contain a `SELECT` statement and includes a known number of input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 3.
- The **fourth example** shows processing and executing a SQL statement that might contain a `SELECT` statement and includes an unknown number of input variables. This example corresponds to
Example: Executing a nonquery statement without parameters

This example shows how to use the `EXECUTE IMMEDIATE` command to execute a SQL statement, where the text of the statement isn't known until you run the application. You can't use `EXECUTE IMMEDIATE` to execute a statement that returns a result set. You can't use `EXECUTE IMMEDIATE` to execute a statement that contains parameter placeholders.

The `EXECUTE IMMEDIATE` statement parses and plans the SQL statement each time it executes, which can have a negative impact on the performance of your application. If you plan to execute the same statement repeatedly, consider using the `PREPARE/EXECUTE` technique described in Example: Executing a nonquery statement with a specified number of placeholders.

```c
/**********************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void handle_error(void);

int main(int argc, char *argv[])
{
    char *insertStmt;

    EXEC SQL WHENEVER SQLERROR DO handle_error();

    EXEC SQL CONNECT :argv[1];

    insertStmt = "INSERT INTO dept VALUES(50, 'ACCTG', 'SEATTLE');

    EXEC SQL EXECUTE IMMEDIATE :insertStmt;
    fprintf(stderr, "ok\n");
    EXEC SQL COMMIT RELEASE;
    exit(EXIT_SUCCESS);
}

static void handle_error(void)
{
    fprintf(stderr, "%s\n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    exit(EXIT_FAILURE);
}
/**********************************************************/
```

The code sample begins by including the prototypes and type definitions for the `stdio`, `string`, and `stdlib` libraries and providing basic infrastructure for the program:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void handle_error(void);

int main(int argc, char *argv[])
{
    char *insertStmt;

    EXEC SQL WHENEVER SQLERROR DO handle_error();

    The example then sets up an error handler. ECPGPlus calls the `handle_error()` function whenever a SQL error occurs:

    EXEC SQL CONNECT :argv[1];
```

Then, the example connects to the database using the credentials specified on the command line:
Next, the program uses an `EXECUTE IMMEDIATE` statement to execute a SQL statement, adding a row to the `dept` table:

```sql
insertStmt = "INSERT INTO dept VALUES(50, 'ACCTG', 'SEATTLE');
EXEC SQL EXECUTE IMMEDIATE :insertStmt;
```

If the `EXECUTE IMMEDIATE` command fails, ECPGPlus invokes the `handle_error()` function, which terminates the application after displaying an error message to the user. If the `EXECUTE IMMEDIATE` command succeeds, the application displays a message (`ok`) to the user, commits the changes, disconnects from the server, and terminates the application:

```c
fprintf(stderr, "ok\n");
EXEC SQL COMMIT RELEASE;
exit(EXIT_SUCCESS);
}
```

ECPGPlus calls the `handle_error()` function whenever it encounters a SQL error. The `handle_error()` function prints the content of the error message, resets the error handler, rolls back any changes, disconnects from the database, and terminates the application:

```c
static void handle_error(void)
{
fprintf(stderr, "%s\n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL WHENEVER SQLERROR DO handle_error();
EXEC SQL CONNECT :argv[1];
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK RELEASE;
exit(EXIT_FAILURE);
}
```

Example: Executing a nonquery statement with a specified number of placeholders

To execute a nonquery command that includes a known number of parameter placeholders, you must first `PREPARE` the statement (providing a statement handle) and then `EXECUTE` the statement using the statement handle. When the application executes the statement, it must provide a value for each placeholder found in the statement.

When an application uses the `PREPARE/EXECUTE` mechanism, each SQL statement is parsed and planned once but might execute many times, providing different values each time.

ECPGPlus converts each parameter value to the type required by the SQL statement, if possible. Otherwise, ECPGPlus reports an error.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
static void handle_error(void)
{
    char *stmtText;
    EXEC SQL WHENEVER SQLERROR DO handle_error();
    EXEC SQL CONNECT :argv[1];
    stmtText = "INSERT INTO dept VALUES(?, ?, ?)";
    EXEC SQL PREPARE stmtHandle FROM :stmtText;
    EXEC SQL EXECUTE stmtHandle USING :argv[2], :argv[3], :argv[4];
    fprintf(stderr, "ok\n");
    EXEC SQL COMMIT RELEASE;
    exit(EXIT_SUCCESS);
}
```
The code sample begins by including the prototypes and type definitions for the C <stdio.h>, <string.h>, <stdlib.h>, and <sqlca.h> libraries and providing basic infrastructure for the program:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>

static void handle_error(void);

int main(int argc, char **argv)
{
    char *stmtText;

    stmtText = "INSERT INTO dept VALUES(?, ?, ?);"

    EXEC SQL PREPARE stmtHandle FROM :stmtText;

    EXEC SQL COMMIT;
    EXEC SQL ROLLBACK;
    exit(EXIT_SUCCESS);
}

/*********************************************************************/
```

The example then sets up an error handler. ECPGPlus calls the handle_error() function whenever a SQL error occurs.

```c
EXEC SQL WHENEVER SQLERROR DO
    handle_error();

Then, the example connects to the database using the credentials specified on the command line:

```c
EXEC SQL CONNECT :argv[1];
```

After parsing and planning the statement, the application uses the EXECUTE statement to execute the statement associated with the statement handle, substituting user-provided values for the parameter markers:

```c
EXEC SQL EXECUTE stmtHandle USING :argv[2], :argv[3], :argv[4];
```

The example then sets up an error handler. ECPGPlus calls the handle_error() function whenever it encounters a SQL error. The handle_error() function prints the content of the error message, resets the error handler, rolls back any changes, disconnects from the database, and terminates the application:

```c
static void handle_error(void)
{
    printf("%s\n", sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK;
    exit(EXIT_FAILURE);
}
```
Example: Executing a query with a known number of placeholders

This example shows how to execute a query with a known number of input parameters and with a known number of columns in the result set. This method uses the `PREPARE` statement to parse and plan a query and then opens a cursor and iterates through the result set.

```c
/***********************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[]) {
    VARCHAR empno[10];
    VARCHAR ename[20];

    EXEC SQL WHENEVER SQLERROR DO handle_error();

    EXEC SQL CONNECT :argv[1];

    EXEC SQL PREPARE queryHandle
        FROM "SELECT empno, ename FROM emp WHERE deptno = ?";

    EXEC SQL DECLARE empCursor CURSOR FOR queryHandle;

    EXEC SQL OPEN empCursor USING :argv[2];

    EXEC SQL WHENEVER NOT FOUND DO break;

    while(true) {
        EXEC SQL FETCH empCursor INTO :empno, :ename;
        printf("%-10s %s\n", empno.arr, ename.arr);
    }

    EXEC SQL CLOSE empCursor;

    EXEC SQL COMMIT RELEASE;
    exit(EXIT_SUCCESS);
}

static void handle_error(void) {
    printf("%s\n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    exit(EXIT_FAILURE);
}
/***********************************************************/
```

The code sample begins by including the prototypes and type definitions for the C `stdio`, `string`, `stdlib`, and `sqlca` libraries and providing basic infrastructure for the program.
The example then sets up an error handler. ECPGPlus calls the `handle_error()` function whenever a SQL error occurs:

```c
EXEC SQL WHENEVER SQLERROR DO
    handle_error();
```

Then, the example connects to the database using the credentials specified on the command line:

```c
EXEC SQL CONNECT
    :argv[1];
```

Next, the program uses a `PREPARE` statement to parse and plan a query that includes a single parameter marker. If the `PREPARE` statement succeeds, it creates a statement handle that you can use to execute the statement. (In this example, the statement handle is named `stmtHandle`.) You can execute a given statement multiple times using the same statement handle.

```c
EXEC SQL PREPARE
    stmtHandle
    FROM "SELECT empno, ename FROM emp WHERE deptno = ?";
```

The program then declares and opens the cursor `empCursor`, substituting a user-provided value for the parameter marker in the prepared `SELECT` statement. The `OPEN` statement includes a `USING` clause, which must provide a value for each placeholder found in the query:

```c
EXEC SQL DECLARE
    empCursor CURSOR FOR
    stmtHandle;
EXEC SQL OPEN
    empCursor USING
    :argv[2];
EXEC SQL WHENEVER NOT FOUND DO
    break;
while(true) {
```

The program iterates through the cursor and prints the employee number and name of each employee in the selected department:

```c
EXEC SQL FETCH
    empCursor INTO
    :empno, :ename;
    printf("%-10s %s\n", empno.arr, ename.arr);
}
```

The program then closes the cursor, commits any changes, disconnects from the server, and terminates the application:

```c
EXEC SQL CLOSE
    empCursor;
EXEC SQL COMMIT
    RELEASE;
exit(EXIT_SUCCESS);
}
```

The application calls the `handle_error()` function whenever it encounters a SQL error. The `handle_error()` function prints the content of the error message, resets the error handler, rolls back any changes, disconnects from the database, and terminates the application:

```c
static void handle_error(void)
{
    printf("%s\n", sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR
        CONTINUE;
    EXEC SQL ROLLBACK
        RELEASE;
    exit(EXIT_FAILURE);
}
```

Example: Executing a query with an unknown number of variables

```c
static void handle_error(void)
{
    int main(int argc, char argv)[]
    {
        ...}
```
This example shows executing a query with an unknown number of input parameters or columns in the result set. This type of query might occur when you prompt the user for the text of the query or when a query is assembled from a form on which the user chooses from a number of conditions (i.e., a filter).

```c
#include <stdio.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>

SQLDA *params;
SQLDA *results;

static void allocateDescriptors(int count,
                                int varNameLength,
                                int indNameLenth);

static void bindParams(void);
static void displayResultSet(void);

int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;
    char *username = argv[1];
    char *password = argv[2];
    char *stmtText = argv[3];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL WHENEVER SQLERROR sqlprint;
    EXEC SQL CONNECT TO test USER :username IDENTIFIED BY :password;
    params = sqlald(20, 64, 64);
    results = sqlald(20, 64, 64);
    EXEC SQL PREPARE stmt FROM :stmtText;
    EXEC SQL DECLARE dynCursor CURSOR FOR stmt;
    bindParams();
    EXEC SQL OPEN dynCursor USING DESCRIPTOR params;
    displayResultSet(20);
}

static void bindParams(void)
{
    EXEC SQL DESCRIBE BIND VARIABLES FOR stmt INTO params;
    if (params->F < 0)
        fprintf(stderr, "Too many parameters required!
    else
        { int i;
            params->N = params->F;
            for (i = 1; i < params->F; i++)
            { char *paramName = params->S[i];
                int nameLen = params->C[i];
                char paramValue[255];
```
printf("Enter value for parameter %.s:", namelen, paramName);
fgets(paramValue, sizeof(paramValue), stdin);
params->T[i] = 1; /* Data type = Character (1) */
params->L[i] = strlen(paramValue) - 1;
params->V[i] = strdup(paramValue);
}
}
}
}

static void displayResultSet(void)
{
  EXEC SQL DESCRIBE SELECT LIST FOR stmt INTO results;

  if (results->F < 0)
    fprintf(stderr, "Too many columns returned by query\n");
  else if (results->F == 0)
    return;
  else
  {
    int col;

    results->N = results->F;
    for (col = 0; col < results->F; col++)
    {
      int null_permitted,
      length;
      sqlnull(results->T[col],&results->T[col],&null_permitted);
      switch (results->T[col])
      {
        case 2: /* NUMERIC */
          {
            int precision, scale;
            sqlprc(results->L[col], &precision, &scale);
            if (precision == 0)
              precision = 38;
            length = precision + 3;
            break;
          }
        case 12: /* DATE */
          {
            length = 30;
            break;
          }
        default: /* Others */
          {
            length = results->L[col] + 1;
            break;
          }
      }
      results->V[col] = realloc(results->V[col], length);
      results->L[col] = length;
      results->T[col] = 1;
    }
  }
The code sample begins by including the prototypes and type definitions for the `stdio` and `stdlib` libraries. In addition, the program includes the `sqlda.h` and `sqlcpr.h` header files. `sqlda.h` defines the SQLDA structure used throughout this example. `sqlcpr.h` defines a small set of functions used to interrogate the metadata found in an SQLDA structure.

Next, the program declares pointers to two SQLDA structures. The first SQLDA structure (`params`) is used to describe the metadata for any parameter markers found in the dynamic query text. The second SQLDA structure (`results`) contains both the metadata and the result set obtained by executing the dynamic query.

The program then declares two helper functions, which are defined near the end of the code sample:

```
static void bindParams(void);
static void displayResultSet(void);
```

Next, the program declares three host variables. The first two (`username` and `password`) are used to connect to the database server. The third host variable (`stmtText`) is a NULL-terminated C string containing the text of the query to execute. The values for these three host variables are derived from the command-line arguments. When the program begins to execute, it sets up an error handler and then connects to the database server:

```
EXEC SQL WHENEVER NOT FOUND DO
  break;
while (1)
  {
    const char *delimiter = "";
    EXEC SQL FETCH dynCursor USING DESCRIPTOR results;
    for (col = 0; col < results->F;
        col++)
      {
        if (*results->I[col] == -1)
          printf("%s", delimiter, "
        else
          printf("%s", delimiter, results->V[col]);
        delimiter = ",
      }
printf("\n");
  }
} /********************************************************************/
```

The code sample begins by including the prototypes and type definitions for the `stdio` and `stdlib` libraries. In addition, the program includes the `sqlda.h` and `sqlcpr.h` header files. `sqlda.h` defines the SQLDA structure used throughout this example. `sqlcpr.h` defines a small set of functions used to interrogate the metadata found in an SQLDA structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
```

Next, the program declares pointers to two SQLDA structures. The first SQLDA structure (`params`) is used to describe the metadata for any parameter markers found in the dynamic query text. The second SQLDA structure (`results`) contains both the metadata and the result set obtained by executing the dynamic query.

```
SQLDA *params;
SQLDA *results;
```

The program then declares two helper functions, which are defined near the end of the code sample:

```
static void bindParams(void);
static void displayResultSet(void);
```

Next, the program declares three host variables. The first two (`username` and `password`) are used to connect to the database server. The third host variable (`stmtText`) is a NULL-terminated C string containing the text of the query to execute. The values for these three host variables are derived from the command-line arguments. When the program begins to execute, it sets up an error handler and then connects to the database server:

```
int main(int argc, char *argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;
    char *username = argv[1];
    char *password = argv[2];
    char *stmtText = argv[3];
  EXEC SQL END DECLARE SECTION;

  EXEC SQL WHENEVER SQLERROR
    sqlprint;
  EXEC SQL CONNECT TO
    test
    USER :username
    IDENTIFIED BY :password;
```

Next, the program calls the `sqlald()` function to allocate the memory required for each descriptor. Each descriptor contains pointers to arrays of:

- Column names
- Indicator names
- Data types
- Lengths
- Data values
When you allocate an SQLDA descriptor, you specify the maximum number of columns you expect to find in the result set (for SELECT-list descriptors) or the maximum number of parameters you expect to find the dynamic query text (for bind-variable descriptors). In this case, we specify that we expect no more than 20 columns and 20 parameters. You must also specify a maximum length for each column or parameter name and each indicator variable name. In this case, we expect names to be no more than 64 bytes long.

See SQLDA structure for a complete description of the SQLDA structure.

After allocating the SELECT-list and bind descriptors, the program prepares the dynamic statement and declares a cursor over the result set.

Next, the program calls the bindParams() function. The bindParams() function examines the bind descriptor (params) and prompts the user for a value to substitute in place of each parameter marker found in the dynamic query.

bindParams();

Finally, the program opens the cursor (using any parameter values supplied by the user) and calls the displayResultSet() function to print the result set produced by the query:

displayResultSet();

The bindParams() function determines whether the dynamic query contains any parameter markers. If so, it prompts the user for a value for each parameter and then binds that value to the corresponding marker. The DESCRIBE BIND VARIABLE statement populates the params SQLDA structure with information describing each parameter marker:

static void bindParams(void)
{
  EXEC SQL DESCRIBE BIND VARIABLES FOR stmt INTO params;

  if (params->F < 0) {
    fprintf(stderr, "Too many parameters required\n");
  } else {
    int i;
    params->N = params->F;
  }

  for (i = 0; i < params->F; ++i) {
    char *paramName = params->S[i];
    int nameLen = params->C[i];
    char paramValue[255];

    printf("Enter value for parameter %.*s:
", nameLen, paramName);
    fgets(paramValue, sizeof(paramValue), stdin);
    params->T[i] = 1; /* Data type = Character (1)*/
  }
}

After prompting the user for a value for a given parameter, the program binds that value to the parameter by setting:

- params->T[i] to indicate the data type of the value
- params->L[i] to the length of the value (we subtract one to trim off the trailing new-line character added by fgets())
- params->V[i] to point to a copy of the NULL-terminated string provided by the user
The `displayResultSet()` function loops through each row in the result set and prints the value found in each column. `displayResultSet()` starts by executing a `DESCRIBE SELECT LIST` statement. This statement populates an SQLDA descriptor (results) with a description of each column in the result set.

```c
static void
displayResultSet(void)
{
    EXEC SQL DESCRIBE SELECT LIST FOR stmt INTO results;

    if (results->F < 0)
        fprintf(stderr, "Too many columns returned by query \n\n");
    else if (results->F == 0)
        return;
    else
    {
        int col;
        results->N = results->F;

        for (col = 0; col < results->F; col++)
        {
            int null_permitted,
                length;

            sqlnul(&results->T[col], &results->T[col],
                   &null_permitted);

            if (null_permitted)
                results->T[col] = 2;
            else
                sqlprc(&results->L[col], &precision,
                       &scale);

            if (precision == 0)
                precision = 38;
            length = precision + 3;
            break;
        }
    }
```

Next, the program enters a loop, iterating once for each column in the result set:

```c
for (col = 0; col < results->F; col++)
{
    int null_permitted,
        length;

    sqlnul(&results->T[col], &results->T[col],
           &null_permitted);

    if (null_permitted)
        results->T[col] = 2;
    else
        sqlprc(&results->L[col], &precision,
               &scale);

    if (precision == 0)
        precision = 38;
    length = precision + 3;
    break;
}
```

To decode the type code found in `results->T`, the program invokes the `sqlnul()` function (see the description of the `T` member of the SQLDA structure in the EDB Postgres Advanced Server (EPAS) SQLDA structure). This call to `sqlnul()` modifies `results->T[col]` to contain only the type code (the nullability flag is copied to `null_permitted`). This step is needed because the `DESCRIBE SELECT LIST` statement encodes the type of each column and the nullability of each column into the `T` array.

```c
sqlnul(&results->T[col], &results->T[col],
       &null_permitted);
```

After decoding the actual data type of the column, the program modifies the results descriptor to tell ECPGPlus to return each value in the form of a NULL-terminated string. Before modifying the descriptor, the program must compute the amount of space required to hold each value. To make this computation, the program examines the maximum length of each column (results->V[col]) and the data type of each column (results->T[col]).

For numeric values (where `results->T[col]` = 2), the program calls the `sqlprc()` function to extract the precision and scale from the column length. To compute the number of bytes required to hold a numeric value in string form, `displayResultSet()` starts with the precision (that is, the maximum number of digits) and adds three bytes for a sign character, a decimal point, and a NULL terminator.

```c
switch (results->T[col])
{
    case 2: /* NUMERIC */
        {
            int precision, scale;

            sqlprc(&results->L[col], &precision,
                   &scale);

            if (precision == 0)
                precision = 38;
            length = precision + 3;
            break;
        }
```

For date values, the program uses a hard-coded length of 30. In a real-world application, you might want to more carefully compute the amount of space required:

```c
case 12: /* DATE */
    {
        length = 30;
        break;
    }
```

For date values, the program uses a hard-coded length of 30. In a real-world application, you might want to more carefully compute the amount of space required.
For a value of any type other than date or numeric, `displayResultSet()` starts with the maximum column width reported by `DESCRIBE SELECT LIST` and adds one extra byte for the NULL terminator. Again, in a real-world application you might want to include more careful calculations for other data types:

```c
default: /* Others */
{
    length = results->L[col] + 1;
    break;
}
}
```

After computing the amount of space required to hold a given column, the program:

- Allocates enough memory to hold the value
- Sets `results->L[col]` to indicate the number of bytes found at `results->V[col]`
- Sets the type code for the column (`results->T[col]`) to 1 to instruct the upcoming `FETCH` statement to return the value in the form of a NULL-terminated string

```c
results->V[col] = malloc(length);
results->L[col] = length;
results->T[col] = 1;
}
```

At this point, the results descriptor is configured such that a `FETCH` statement can copy each value into an appropriately sized buffer in the form of a NULL-terminated string.

Next, the program defines a new error handler to break out of the upcoming loop when the cursor is exhausted.

```c
EXEC SQL WHENEVER NOT FOUND DO
break;
while (1)
{
    const char *delimiter = "";

    The program executes a `FETCH` statement to fetch the next row in the cursor into the `results` descriptor. If the `FETCH` statement fails (because the cursor is exhausted), control transfers to the end of the loop because of the `EXEC SQL WHENEVER` directive found before the top of the loop.

```c
EXEC SQL FETCH dynCursor USING DESCRIPTOR results;
```

The `FETCH` statement populates the following members of the results descriptor:

- `results->I[col]` indicates whether the column contains a NULL value (-1) or a non-NULL value (0). If the value is non-NULL but too large to fit into the space provided, the value is truncated and `results->T[col]` contains a positive value.
- `results->V[col]` contains the value fetched for the given column (unless `results->I[col]` indicates that the column value is NULL).
- `results->L[col]` contains the length of the value fetched for the given column.

Finally, `displayResultSet()` iterates through each column in the result set, examines the corresponding NULL indicator, and prints the value. The result set isn’t aligned. Instead, each value is separated from the previous value by a comma.

```c
for (col = 0; col < results->F; col++)
{
    if (*results->I[col] == -1)
        printf("%.*s", delimiter, "<null>");
    else
        printf("%.*s", delimiter, results->V[col]);
    delimiter = "",
}
printf("\n");`
```

```
*/}

11.4.6    Error handling

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
ECPGPlus provides two methods to detect and handle errors in embedded SQL code. A client application can:

- Examine the sqlca data structure for error messages and supply customized error handling for your client application.
- Include `EXEC SQL WHENEVER` directives to instruct the ECPGPlus compiler to add error-handling code.

### Error handling with sqlca

The SQL communications area (sqlca) is a global variable used by `ecpglib` to communicate information from the server to the client application. After executing a SQL statement such as an `INSERT` or `SELECT` statement, you can inspect the contents of sqlca to determine if the statement completed successfully or if the statement failed.

sqlca has the following structure:

```c
struct {
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[8];
} sqlca;
```

Use the following directive to implement sqlca functionality:

```sql
EXEC SQL INCLUDE sqlca;
```

If you include the `ecpg` directive, you don't need to `#include` the `sqlca.h` file in the client application's header declaration.

The EDB Postgres Advanced Server sqlca structure contains the following members:

- **sqlcaid** — Contains the string "SQLCA".
- **sqlabc** — `sqlabc` contains the size of the sqlca structure.
- **sqlcode** — The sqlcode member was deprecated with SQL 92. EDB Postgres Advanced Server supports sqlcode for backward compatibility. Use the sqlstate member when writing new code. sqlcode is an integer value. A positive sqlcode value indicates that the client application encountered a harmless processing condition. A negative value indicates a warning or error.

If a statement processes without error, sqlcode contains a value of 0. If the client application encounters an error or warning during a statement's execution, sqlcode contains the last code returned.

The SQL standard defines only a positive value of 100, which indicates that the most recent SQL statement processed returned or affected no rows. Since the SQL standard doesn't define other sqlcode values, be aware that the values assigned to each condition can vary from database to database.

- **sqlerrm** is a structure embedded in sqlca, composed of two members:
  - **sqlerrml** — Contains the length of the error message currently stored in sqlerrmc.
  - **sqlerrmc** — Contains the null-terminated message text associated with the code stored in sqlstate. If a message exceeds 149 characters, `ecpglib` truncates the error message.
  - **sqlerrp** — Contains the string "NOT SET".
- **sqlerrd** is an array that contains six elements:
  - **sqlerrd[0]** — Contains the OID of the processed row (if applicable).
  - **sqlerrd[1]** — Contains the number of processed or returned rows.
  - **sqlerrd[2]**, **sqlerrd[3]**, **sqlerrd[4]** and **sqlerrd[5]** are unused.
sqlwarn is an array that contains 8 characters:

- sqlwarn[6] — Contains a value of 'W' if any other element in sqlwarn is set to 'W'.
- sqlwarn[1] — Contains a value of 'W' if a data value was truncated when it was stored in a host variable.
- sqlwarn[2] — Contains a value of 'W' if the client application encounters a nonfatal warning.
- sqlwarn[3], sqlwarn[4], sqlwarn[5], sqlwarn[6], and sqlwarn[7] are unused.

sqlstate is a five-character array that contains a SQL-compliant status code after the execution of a statement from the client application. If a statement processes without error, sqlstate contains a value of 08000. sqlstate isn’t a null-terminated string.

sqlstate codes are assigned in a hierarchical scheme:

- The first two characters of sqlstate indicate the general class of the condition.
- The last three characters of sqlstate indicate a specific status within the class.

If the client application encounters multiple errors (or warnings) during an SQL statement’s execution, sqlstate contains the last code returned.

List of sqlstate and sqlcode values

The following table lists the sqlstate and sqlcode values, as well as the symbolic name and error description for the related condition.

<table>
<thead>
<tr>
<th>sqlstate</th>
<th>sqlcode</th>
<th>Symbolic name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YE001</td>
<td>-12</td>
<td>ECPG_OUT_OF_MEMORY</td>
<td>Virtual memory is exhausted.</td>
</tr>
<tr>
<td>YE002</td>
<td>-280</td>
<td>ECPG_UNSUPPORTED</td>
<td>The preprocessor generated an unrecognized item. Might indicate incompatibility between the preprocessor and the library.</td>
</tr>
<tr>
<td>97001, or 87002</td>
<td>-281</td>
<td>ECPG_TOO_MANY_ARGUMENTS</td>
<td>The program specifies more variables than the command expects.</td>
</tr>
<tr>
<td>97001, or 87002</td>
<td>-282</td>
<td>ECPG_TOO FEW ARGUMENTS</td>
<td>The program specified fewer variables than the command expects.</td>
</tr>
<tr>
<td>21000</td>
<td>-283</td>
<td>ECPG_TOO_MANY MATCHES</td>
<td>The SQL command returned multiple rows, but the statement was prepared to receive a single row.</td>
</tr>
<tr>
<td>42804</td>
<td>-284</td>
<td>ECPG_INT_FORMAT</td>
<td>The host variable (defined in the C code) is of type INT, and the selected data is of a type that can't be converted into an INT. ecpglib uses the strtol() function to convert string values into numeric form.</td>
</tr>
<tr>
<td>42804</td>
<td>-285</td>
<td>ECPG_UINT_FORMAT</td>
<td>The host variable (defined in the C code) is an unsigned INT, and the selected data is of a type that can't be converted into an unsigned INT. ecpglib uses the strtoul() function to convert string values into numeric form.</td>
</tr>
<tr>
<td>42804</td>
<td>-286</td>
<td>ECPG_FLOAT_FORMAT</td>
<td>The host variable (defined in the C code) is of type FLOAT, and the selected data is of a type that can't be converted into a FLOAT. ecpglib uses the strtod() function to convert string values into numeric form.</td>
</tr>
<tr>
<td>42804</td>
<td>-211</td>
<td>ECPG_CONVERT BOOL</td>
<td>The host variable (defined in the C code) is of type BOOL, and the selected data can't be stored in a BOOL.</td>
</tr>
<tr>
<td>YE002</td>
<td>-2-1</td>
<td>ECPG_EMPTY</td>
<td>The statement sent to the server was empty.</td>
</tr>
<tr>
<td>22002</td>
<td>-213</td>
<td>ECPG_MISSING INDICATOR</td>
<td>A NULL indicator variable wasn't supplied for the NULL value returned by the server. (The client application received an unexpected NULL value.).</td>
</tr>
<tr>
<td>42804</td>
<td>-214</td>
<td>ECPG_NO ARRAY</td>
<td>The server returned an array, and the corresponding host variable can't store an array.</td>
</tr>
<tr>
<td>42804</td>
<td>-215</td>
<td>ECPG_DATA NOT ARRAY</td>
<td>The server returned a value that isn’t an array into a host variable that expects an array value.</td>
</tr>
<tr>
<td>88003</td>
<td>-220</td>
<td>ECPG_NO_CONN</td>
<td>The client application attempted to use a nonexistent connection.</td>
</tr>
<tr>
<td>YE002</td>
<td>-221</td>
<td>ECPG_NOT_CONN</td>
<td>The client application attempted to use an allocated but closed connection.</td>
</tr>
<tr>
<td>26000</td>
<td>-230</td>
<td>ECPG_INVALID_STMT</td>
<td>The statement wasn’t prepared.</td>
</tr>
<tr>
<td>33800</td>
<td>-240</td>
<td>ECPG_UNKNOWN_DESCRIPTOR</td>
<td>The specified descriptor isn’t found.</td>
</tr>
<tr>
<td>97009</td>
<td>-241</td>
<td>ECPG_INVALID_DESCRIPTOR_INDEX</td>
<td>The descriptor index is out of range.</td>
</tr>
<tr>
<td>YE002</td>
<td>-242</td>
<td>ECPG_UNKNOWN_DESCRIPTOR_ITEM</td>
<td>The client application requested an invalid descriptor item (internal error).</td>
</tr>
<tr>
<td>97006</td>
<td>-243</td>
<td>ECPG_VAR NOT NUMERIC</td>
<td>A dynamic statement returned a numeric value for a non-numeric host variable.</td>
</tr>
<tr>
<td>97006</td>
<td>-244</td>
<td>ECPG_VAR NOT CHAR</td>
<td>A dynamic SQL statement returned a CHAR value, and the host variable isn’t a CHAR.</td>
</tr>
<tr>
<td>-480</td>
<td>ECPG_PGSQL</td>
<td>The server returned an error message. The resulting message contains the error text.</td>
<td></td>
</tr>
<tr>
<td>88007</td>
<td>-481</td>
<td>ECPG_TRANS</td>
<td>The server can't start, commit, or roll back the specified transaction.</td>
</tr>
</tbody>
</table>
Implementing simple error handling for client applications

Use the EXEC SQL WHENEVER directive to implement simple error handling for client applications compiled with ECPGPlus. The syntax of the directive is:

```
EXEC SQL WHENEVER <condition> <action>;
```

This directive instructs the ECPG compiler to insert error-handling code into your program.

The code instructs the client application to perform a specified action if the client application detects a given condition. The condition can be one of the following:

- **SQLERROR**: A SQLERROR condition exists when `sqlca.sqlcode` is less than zero.
- **SQLWARNING**: A SQLWARNING condition exists when `sqlca.sqlwarn[0]` contains a 'W'.
- **NOT FOUND**: A NOT FOUND condition exists when `sqlca.sqlcode` is `ECPG_NOT_FOUND` (when a query returns no data).

You can specify that the client application perform one of the following actions if it encounters one of the previous conditions:

- **CONTINUE**: Specify CONTINUE to instruct the client application to continue processing, ignoring the current condition. CONTINUE is the default action.
- **DO CONTINUE**: An action of DO CONTINUE generates a CONTINUE statement in the emitted C code. If it encounters the condition, it skips the rest of the code in the loop and continues with the next iteration. You can use it only in a loop.
- **GOTO label** or **GO TO label**: Use a C goto statement to jump to the specified label.
- **SQLPRINT**: Print an error message to stderr (standard error), using the sqlprint() function. The sqlprint() function prints sql error followed by the contents of sqlca.sqlerrm.sqlerrmc.
- **STOP**: Call exit(1) to signal an error and terminate the program.
- **DO BREAK**: Execute the C break statement. Use this action in loops or switch statements.
- **CALL name(args)** or **DO name(args)**: Invoke the C function specified by the name parameter, using the parameters specified in the args parameter.

**Example**

The following code fragment prints a message if the client application encounters a warning and aborts the application if it encounters an error:

```
EXEC SQL WHENEVER SQLWARNING
SQLPRINT;
```
The ECPGPlus compiler processes your program from top to bottom, even though the client application might not execute from top to bottom. The compiler directive is applied to each line in order and remains in effect until the compiler encounters another directive. If the control of the flow in your program isn’t top to bottom, consider adding error handling directives to any parts of the program that might be missed during compiling.

11.5 Using the stored procedural language

EDB Postgres Advanced Server’s stored procedural language (SPL) is a highly productive, procedural programming language for writing custom procedures, functions, triggers, and packages for EDB Postgres Advanced Server. It provides:

- Full procedural programming functionality to complement the SQL language
- A single, common language to create stored procedures, functions, triggers, and packages for the EDB Postgres Advanced Server database
- A seamless development and testing environment
- The use of reusable code
- Ease of use

For reference information about the SPL program types, programming statements, control structures, collection types, and collection methods, see Stored procedural language (SPL) reference.

11.5.1 Types of SPL programs

SPL is a procedural, block-structured language. You can create four different types of programs using SPL: procedures, functions, triggers, and packages.

In addition, you can use SPL to create subprograms. A subprogram refers to a subprocedure or a subfunction. These are nearly identical in appearance to procedures and functions. They differ in that procedures and functions are standalone programs. They are stored individually in the database, and you can invoke them from other SPL programs or from PSQL. You can invoke subprograms only from the standalone program where they were created.

11.5.1.1 SPL block structure overview

Regardless of whether the program is a procedure, function, subprogram, or trigger, an SPL program has the same block structure. A block consists of up to three sections: an optional declaration section, a mandatory executable section, and an optional exception section. Minimally, a block has an executable section that consists of one or more SPL statements between the keywords BEGIN and END.

Use the optional declaration section to declare variables, cursors, types, and subprograms that are used by the statements in the executable and exception sections. Declarations appear just before the BEGIN keyword of the executable section. Depending on the context in which the block appears, the declaration section can begin with the keyword DECLARE.

You can include an exception section in the BEGIN - END block. The exception section begins with the keyword EXCEPTION and continues until the end of the block in which it appears. If an exception is thrown by a statement in the block, program control might go to the exception section where the thrown exception is handled, depending on the exception and the contents of the exception section.

The following is the general structure of a block:

```
[ [ DECLARE
  ]
  <pragmas>
  <declarations>
 ]
BEGIN
  <statements>
  [ EXCEPTION
    WHEN <exception_condition> THEN
      <statements> [ , ...]
  ]
END;
```

- **pragmas** are the directives (AUTONOMOUS_TRANSACTION is the currently supported pragma).
- **declarations** are one or more variable, cursor, type, or subprogram declarations that are local to the block. If subprogram declarations are included, you must declare them after all other variable, cursor, and type declarations. Terminate each declaration with a semicolon. The use of the keyword DECLARE depends on the context in which the block appears.
- **statements** are one or more SPL statements. Terminate each statement with a semicolon. You must also terminate the end of the block denoted by the keyword END with a semicolon.
- **if present, the keyword EXCEPTION marks the beginning of the exception section. exception_condition is a conditional expression testing for one or more types of exceptions. If an exception matches one of the exceptions in exception_condition, the statements following the WHEN exception_condition clause are executed. There can be one or more WHEN exception_condition clauses, each followed by statements.**
A BEGIN/END block is considered a statement, thus you can nest blocks. The exception section can also contain nested blocks.

The following is the simplest possible block, consisting of the NULL statement in the executable section. The NULL statement is an executable statement that does nothing.

```
BEGIN
    NULL;
END;
```

The following block contains a declaration section as well as the executable section:

```
DECLARE
    v_numerator NUMBER(2);
    v_denominator NUMBER(2);
    v_result NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 14;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator || ' is ' || v_result);
END;
```

In this example, three numeric variables are declared of data type NUMBER. Values are assigned to two of the variables, and one number is divided by the other. Results are stored in a third variable and then displayed. The output is:

```
75 divided by 14 is 5.36
```

The following block consists of a declaration, an executable, and an exception:

```
DECLARE
    v_numerator NUMBER(2);
    v_denominator NUMBER(2);
    v_result NUMBER(5,2);
BEGIN
    v_numerator := 75;
    v_denominator := 0;
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE(v_numerator || ' divided by ' || v_denominator || ' is ' || v_result);
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.PUT_LINE('An exception occurred');
END;
```

The following output shows that the statement in the exception section is executed as a result of the division by zero:

```
An exception occurred
```

### 11.5.1.2 Anonymous blocks

You typically write blocks as part of a procedure, function, subprogram, or trigger. You name procedure, function, and trigger programs and store them in the database if you want to reuse them.

For quick, one-time execution such as testing, you can enter the block without providing a name or storing it in the database. A block without a name and that isn't stored in the database is called an anonymous block. Once the block is executed and erased from the application buffer, you can't execute it again unless the block code you enter into the application again.

Typically, the same block of code executes many times. To run a block of code repeatedly without reentering the code each time, with some simple modifications, you can turn an anonymous block into a procedure or function. See Procedures overview and Functions overview.
### 11.5.1.3 Procedures overview

Procedures are standalone SPL programs that you invoke or call as an individual SPL program statement. When called, procedures can optionally receive values from the caller in the form of input parameters. They can optionally return values to the caller in the form of output parameters.

#### 11.5.1.3.1 Creating a procedure

The `CREATE PROCEDURE` command defines and names a standalone procedure that's stored in the database.

If you include a schema name, then the procedure is created in the specified schema. Otherwise it's created in the current schema. The name of the new procedure must not match any existing procedure with the same input argument types in the same schema. However, procedures of different input argument types can share a name. This is called overloading.

**Note**

Overloading of procedures is an EDB Postgres Advanced Server feature. Overloading of stored, standalone procedures isn't compatible with Oracle databases.

#### Updating the definition of an existing procedure

To update the definition of an existing procedure, use `CREATE OR REPLACE PROCEDURE`. You can't change the name or argument types of a procedure this way. Attempting to do so creates a new, distinct procedure. When using `OUT` parameters, you can't change the types of any `OUT` parameters except by dropping the procedure.

```
CREATE [OR REPLACE] PROCEDURE <name> [ ( <parameters> ) ]

[ IMMUABLE
  | STABLE
  | VOLATILE
  | DETERMINISTIC
  | [ NOT ]
  | CALLED ON NULL
  | INPUT
  | RETURNS NULL ON NULL
  | INPUT
  | STRICT
  | [ EXTERNAL ] SECURITY
  | INVOKER
  | [ EXTERNAL ] SECURITY
  | DEFINER
  | AUTHID
  | DEFINER
  | AUTHID
  | CURRENT_USER
  | PARALLEL { UNSAFE | RESTRICTED | SAFE
  ]

  | COST
  | <execution_cost>
  | ROWS
  | <result_rows>
  | SET
  | <configuration_parameter>
  | { TO <value> | = <value> | FROM CURRENT
  } ...

  | IS | AS

  | [ PRAGMA AUTONOMOUS_TRANSACTION;
  ]

  | [ <declarations>
  ]

BEGIN
  <statements>
END [ <name> ]
```

Where:

- `<name>` is the identifier of the procedure.
parameters

is a list of formal parameters.

declarations

definitions are variable, cursor, type, or subprogram declarations. If you include subprogram declarations, you must declare them after all other variable, cursor, and type declarations.

statements

statements are SPL program statements. The BEGIN - END block can contain an EXCEPTION section.

IMMUTABLE

STABLE

VOLATILE

These attributes inform the query optimizer about the behavior of the procedure. You can specify only one choice. VOLATILE is the default behavior.

- IMMUTABLE indicates that the procedure can't modify the database and always reaches the same result when given the same argument values. It doesn't perform database lookups or otherwise use information not directly present in its argument list. If you include this clause, you can immediately replace any call of the procedure with all-constant arguments with the procedure value.

- STABLE indicates that the procedure can't modify the database. It also indicates that, in a single table scan, it consistently returns the same result for the same argument values but that its result might change across SQL statements. Use this selection for procedures that depend on database lookups, parameter variables (such as the current time zone), and so on.

- VOLATILE indicates that the procedure value can change even in a single table scan, so you can't make optimizations. You must classify any function that has side effects as volatile, even if its result is predictable, to prevent calls from being optimized away.

DETERMINISTIC

DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC procedure can't modify the database and always reaches the same result when given the same argument values. It doesn't do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the procedure with all-constant arguments can be immediately replaced with the procedure value.

[ NOT ] LEAKPROOF

A LEAKPROOF procedure has no side effects and reveals no information about the values used to call the procedure.

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

- CALLED ON NULL INPUT (the default) indicates that the procedure is called normally when some of its arguments are NULL. It's the author's responsibility to check for NULL values if necessary and respond appropriately.

- RETURNS NULL ON NULL INPUT or STRICT indicates that the procedure always returns NULL when any of its arguments are NULL. If these clauses are specified, the procedure isn't executed when there are NULL arguments. Instead a NULL result is assumed automatically.

[ EXTERNAL ] SECURITY DEFINER

SECURITY DEFINER specifies for the procedure to execute with the privileges of the user that created it. This is the default. The key word EXTERNAL is allowed for SQL conformance but is optional.

[ EXTERNAL ] SECURITY INVOKER

The SECURITY INVOKER clause indicates for the procedure to execute with the privileges of the user that calls it. The key word EXTERNAL is allowed for SQL conformance but is optional.

AUTHID DEFINER

AUTHID CURRENT_USER

- The AUTHID DEFINER clause is a synonym for [EXTERNAL] SECURITY DEFINER. If you omit the AUTHID clause or specify AUTHID DEFINER, the rights of the procedure owner determine access privileges to database objects.

- The AUTHID CURRENT_USER clause is a synonym for [EXTERNAL] SECURITY INVOKER. If you specify AUTHID CURRENT_USER, the rights of the current user executing the procedure determine access privileges.

PARALLEL { UNSAFE | RESTRICTED | SAFE }
The `PARALLEL` clause enables the use of parallel sequential scans, that is, parallel mode. A parallel sequential scan uses multiple workers to scan a relation in parallel during a query, in contrast to a serial sequential scan.

- When the value is set to `UNSAFE`, you can't execute the procedure in parallel mode. The presence of such a procedure forces a serial execution plan. This is the default setting.
- When the value is set to `RESTRICTED`, you can execute the procedure in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that's parallel restricted, that relation isn't chosen for parallelism.
- When the value is set to `SAFE`, you can execute the procedure in parallel mode without restriction.

\[
\text{COST execution\_cost}
\]

`execution\_cost` is a positive number giving the estimated execution cost for the procedure, in units of `cpu\_operator\_cost`. If the procedure returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

\[
\text{ROWS result\_rows}
\]

`result\_rows` is a positive number giving the estimated number of rows for the planner to expect the procedure to return. This setting is allowed only when the procedure is declared to return a set. The default is 1000 rows.

\[
\text{SET configuration\_parameter \{ TO value | = value | FROM CURRENT \}}
\]

The `SET` clause causes the specified configuration parameter to be set to the specified value when the procedure is entered and then restored to its prior value when the procedure exits. `SET FROM CURRENT` saves the session’s current value of the parameter as the value to apply when the procedure is entered.

If a `SET` clause is attached to a procedure, then the effects of a `SET LOCAL` command executed inside the procedure for the same variable are restricted to the procedure. The configuration parameter’s prior value is restored at procedure exit. An ordinary `SET` command (without `LOCAL`) overrides the `SET` clause, much as it does for a previous `SET LOCAL` command. The effects of this command persist after procedure exit, unless the current transaction is rolled back.

\[
\text{PRAGMA AUTONOMOUS\_TRANSACTION}
\]

`PRAGMA AUTONOMOUS\_TRANSACTION` is the directive that sets the procedure as an autonomous transaction.

**Note**

- The `STRICT`, `LEAKPROOF`, `PARALLEL`, `COST`, `ROWS` and `SET` keywords provide extended functionality for EDB Postgres Advanced Server and aren't supported by Oracle.
- By default, stored procedures are created as `SECURITY DEFINERS`, but when written in plpgsql, the stored procedures are created as `SECURITY INVOKERS`.

**Example**

This example shows a simple procedure that takes no parameters:

\[
\text{CREATE OR REPLACE PROCEDURE simple\_procedure IS BEGIN DBMS\_OUTPUT\_PUT\_LINE('That's all folks!'); END simple\_procedure;}
\]

Store the procedure in the database by entering the procedure code in EDB Postgres Advanced Server.

This example shows using the `AUTHID DEFINER` and `SET` clauses in a procedure declaration. The `update\_salary` procedure conveys the privileges of the role that defined the procedure to the role that's calling the procedure while the procedure executes.

\[
\text{CREATE OR REPLACE PROCEDURE update\_salary(id INT, new\_salary NUMBER) \{ SET SEARCH\_PATH = 'public' SET WORK\_MEM = '1MB' \} AUTHID DEFINER IS BEGIN UPDATE emp SET salary = new\_salary WHERE emp\_id = id; END;}
\]

Include the `SET` clause to set the procedure’s search path to `public` and the work memory to `1MB`. These settings don't affect other procedures, functions, and objects.

In the example, the `AUTHID DEFINER` clause temporarily grants privileges to a role that otherwise might not be allowed to execute the statements in the procedure. To instruct the server to use the privileges associated with the role invoking the procedure, replace the `AUTHID DEFINER` clause with the `AUTHID CURRENT\_USER` clause.
11.5.1.3.2 Calling a procedure

You can invoke a procedure from another SPL program by specifying the procedure name followed by any parameters and a semicolon:

\[
\text{name} \ [ (\ [\text{parameters}\ ]\ )\ ];
\]

Where:

- **name** is the identifier of the procedure.
- **parameters** is a list of parameters.

**Note**

- If there are no parameters to pass, you can call the procedure with an empty parameter list, or you can omit the parentheses.
- The syntax for calling a procedure is the same as in the preceding syntax diagram when executing it with the **EXEC** command in PSQL or EDB*Plus. See SQL reference for information about the **EXEC** command.

This example calls the procedure from an anonymous block:

```
BEGIN
  simple_procedure;
END;
```

That's all folks!

**Note**

Each application has its own way of calling a procedure. For example, a Java application uses the application programming interface JDBC.

11.5.1.3.3 Deleting a procedure

You can delete a procedure from the database using the **DROP PROCEDURE** command.

```
DROP \[ IF \ EXISTS \] \name\ [ (\ [\text{parameters}\ ]\ )\ ];
```

Where **name** is the name of the procedure to drop.

**Note**

- The specification of the parameter list is required in EDB Postgres Advanced Server under certain circumstances such as in an overloaded procedure. Oracle requires that you always omit the parameter list.
- Using **IF EXISTS**, **CASCADE**, or **RESTRICT** isn't compatible with Oracle databases. See SQL reference for information on these options.

This example drops the procedure **simple_procedure**:

```
DROP \text{PROCEDURE} simple\_procedure;
```

11.5.1.4 Functions overview

Functions are standalone SPL programs that are invoked as expressions. When evaluated, a function returns a value that's substituted in the expression in which the function is embedded. Functions can optionally take values from the calling program in the form of input parameters.

In addition to the fact that the function returns a value, a function can also optionally return values to the caller in the form of output parameters. However, we don't encourage the use of output parameters in functions.

11.5.1.4.1 Creating a function
The `CREATE FUNCTION` command defines and names a standalone function to store in the database. If a schema name is included, then the function is created in the specified schema. Otherwise it’s created in the current schema. The name of the new function must not match any existing function with the same input argument types in the same schema. However, functions of different input argument types can share a name. Sharing a name is called overloading.

Note

Overloading functions is an EDB Postgres Advanced Server feature. Overloading stored, standalone functions isn’t compatible with Oracle databases.

Updating the definition of an existing function

To update the definition of an existing function, use `CREATE OR REPLACE FUNCTION`. You can’t change the name or argument types of a function this way. If you try to, you instead create a new, distinct function. Also, `CREATE OR REPLACE FUNCTION` doesn’t let you change the return type of an existing function. To do that, you must drop and recreate the function. When using `OUT` parameters, you can’t change the types of any `OUT` parameters except by dropping the function.

```sql
CREATE [ OR REPLACE ] FUNCTION <name> [ ( <parameters> ) ]
  RETURN <data_type>
  [ 
    IMMUTABLE
    | STABLE
    | VOLATILE
    | DETERMINISTIC
    | [ NOT ]
    | LEAKPROOF
    | CALLED ON NULL
    | INPUT
    | RETURNS NULL ON NULL
    | INPUT
    | STRICT
    | [ [ external ] SECURITY
    | INVOKER
    | [ [ external ] SECURITY
    | DEFINER
    | AUTHID
    | DEFINER
    | AUTHID
    | CURRENT_USER
    | PARALLEL ( UNSAFE | RESTRICTED | SAFE )
    | COST
    <execution_cost>
    | ROWS
    <result_rows>
    | SET
    <configuration_parameter>
    [ TO <value> | = <value> | FROM CURRENT
    ]
    { IS | AS }
  ]
  [ PRAGMA AUTONOMOUS_TRANSACTION;
  ]
  [ <declarations> ]
  BEGIN
  <statements>
  END [ ( <name> ) ];
```

Where:

- `<name>` is the identifier of the function.
- `<parameters>` is a list of formal parameters.
- `<data_type>` is the data type of the value returned by the function’s `RETURN` statement.
- `<declarations>` are variable, cursor, type, or subprogram declarations. If you include subprogram declarations, you must declare them after all other variable, cursor, and type declarations.
- `<statements>` are SPL program statements. The `BEGIN - END` block can contain an `EXCEPTION` section.
These attributes inform the query optimizer about the behavior of the function. You can specify only one. VOLATILE is the default behavior.

- IMMUTABLE: indicates that the function can’t modify the database and always reaches the same result when given the same argument values. It doesn’t do database lookups or otherwise use information not directly present in its argument list. If you include this clause, any call of the function with all-constant arguments can be immediately replaced with the function value.

- STABLE: indicates that the function can’t modify the database. It also indicates that, in a single table scan, it consistently returns the same result for the same argument values but that its result might change across SQL statements. Select this attribute for functions that depend on database lookups, parameter variables (such as the current time zone), and so on.

- VOLATILE: indicates that the function value can change even in a single table scan, so no optimizations can be made. Classify any function that has side effects as volatile, even if its result is predictable, to prevent calls from being optimized away.

DETERMINISTIC

DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC function can’t modify the database and always reaches the same result when given the same argument values. It doesn’t do database lookups or otherwise use information not directly present in its argument list. If you include this clause, you can replace any call of the function with all-constant arguments with the function value.

[ NOT ] LEAKPROOF

A LEAKPROOF function has no side effects and reveals no information about the values used to call the function.

CALLED ON NULL INPUT

RETURNS NULL ON NULL INPUT

STRICT

- CALLED ON NULL INPUT (the default) indicates for the procedure to be called normally when some of its arguments are NULL. It is the author’s responsibility to check for NULL values if necessary and respond appropriately.

- RETURNS NULL ON NULL INPUT or STRICT indicates that the procedure always returns NULL when any of its arguments are NULL. If you specify these clauses, the procedure isn’t executed when there are NULL arguments. Instead, a NULL result is assumed automatically.

[ EXTERNAL ] SECURITY DEFINER

SECURITY DEFINER (the default) specifies for the function to execute with the privileges of the user that created it. The key word EXTERNAL is allowed for SQL conformance but is optional.

[ EXTERNAL ] SECURITY INVOKER

The SECURITY INVOKER clause indicates for the function to execute with the privileges of the user that calls it. The key word EXTERNAL is allowed for SQL conformance but is optional.

AUTHID DEFINER

AUTHID CURRENT_USER

- The AUTHID DEFINER clause is a synonym for [EXTERNAL] SECURITY DEFINER. If the AUTHID clause is omitted or if AUTHID DEFINER is specified, the rights of the function owner determine access privileges to database objects.

- The AUTHID CURRENT_USER clause is a synonym for [EXTERNAL] SECURITY INVOKER. If AUTHID CURRENT_USER is specified, the rights of the current user executing the function determine access privileges.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

The PARALLEL clause enables the use of parallel sequential scans, that is, parallel mode. A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

- When this value is set to UNSAFE, you can’t execute the function in parallel mode. The presence of such a function in a SQL statement forces a serial execution plan. This is the default setting.

- When this value is set to RESTRICTED, you can execute the function in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that’s parallel restricted, that relation isn’t chosen for parallelism.

- When this value is set to SAFE, you can execute the function in parallel mode with no restriction.

COST execution_cost

execution_cost is a positive number giving the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary.
result_rows is a positive number giving the estimated number of rows for the planner to expect the function to return. This is allowed only when the function is declared to return a set. The default assumption is 1000 rows.

The `SET configuration_parameter { TO value | = value | FROM CURRENT }` causes the specified configuration parameter to be set to the specified value when the function is entered and then restored to its prior value when the function exits. `SET FROM CURRENT` saves the session's current value of the parameter as the value to apply when the function is entered.

If a `SET` clause is attached to a function, then the effects of a `SET LOCAL` command executed inside the function for the same variable are restricted to the function. The configuration parameter's prior value is restored at function exit. A `SET` command without `LOCAL` overrides the `SET` clause, much as it does for a previous `SET LOCAL` command. The effects of such a command persist after procedure exit, unless the current transaction is rolled back.

```
PRAGMA AUTONOMOUS_TRANSACTION
```

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the function as an autonomous transaction.

**Note**

The `STRICT`, `LEAKPROOF`, `PARALLEL`, `COST`, `ROWS` and `SET` keywords provide extended functionality for EDB Postgres Advanced Server and aren't supported by Oracle.

### Examples

This example shows a simple function that takes no parameters:

```
CREATE OR REPLACE FUNCTION simple_function
IS
BEGIN
  RETURN 'That''s All Folks!';
END;
```

This function takes two input parameters:

```
CREATE OR REPLACE FUNCTION emp_comp
  (p_sal NUMBER,
p_comm NUMBER)
RETURN NUMBER
IS
BEGIN
  RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
```

This example uses the `AUTHID CURRENT_USER` clause and `STRICT` keyword in a function declaration:

```
CREATE OR REPLACE FUNCTION dept_salaries(dept_id int)
RETURN NUMBER
STRICT
  AUTHID CURRENT_USER
BEGIN
  RETURN QUERY (SELECT sum(salary) FROM emp WHERE deptno = dept_id);
END;
```

Include the `STRICT` keyword to instruct the server to return `NULL` if any input parameter passed is `NULL`. If a `NULL` value is passed, the function doesn't execute.

The `dept_salaries` function executes with the privileges of the role that's calling the function. If the current user doesn't have the privileges to perform the `SELECT` statement querying the `emp` table (to display employee salaries), the function reports an error. To instruct the server to use the privileges associated with the role that defined the function, replace the `AUTHID CURRENT_USER` clause with the `AUTHID DEFINER` clause.

### 11.5.1.4.2 Calling a function

You can use a function anywhere an expression can appear in an SPL statement. Invoke a function by specifying its name followed by any parameters enclosed in parentheses.
**name** is the name of the function.

**parameters** is a list of parameters.

Note

If there are no parameters to pass, you can call the function with an empty parameter list, or you can omit the parentheses.

This example shows how to call the function from another SPL program:

```sql
BEGIN
  DBMS_OUTPUT.PUT_LINE(simple_function);
END;
```

That's All Folks!

You typically use a function in a SQL statement, as this example shows:

```sql
SELECT empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM",
       emp_comp(sal, comm) "YEARLY COMPENSATION"
FROM emp;
```

### 11.5.1.4.3 Deleting a function

You can delete a function from the database using **DROP FUNCTION**.

```sql
DROP FUNCTION [ IF EXISTS ] <name> [ ( <parameters> ) ]
  [ CASCADE | RESTRICT ];
```

Where **name** is the name of the function to drop.

Note

- Specifying the parameter list is required in EDB Postgres Advanced Server under certain circumstances such as in an overloaded function. Oracle requires that you always omit the parameter list.

- Use of **IF EXISTS**, **CASCADE**, or **RESTRICT** isn't compatible with Oracle databases. See the SQL reference for information on these options.

This example drops the function **simple_function**:

```sql
DROP FUNCTION
  simple_function;
```

### 11.5.1.5 Procedure and function parameters

An important aspect of using procedures and functions is the capability to pass data from the calling program to the procedure or function and to receive data back from the procedure or function.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>SAL</th>
<th>COMM</th>
<th>YEARLY COMPENSATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>800.00</td>
<td></td>
<td>19200.00</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>1600.00</td>
<td>300.00</td>
<td>45600.00</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>1250.00</td>
<td>500.00</td>
<td>42000.00</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>2975.00</td>
<td></td>
<td>71400.00</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>1250.00</td>
<td>1400.00</td>
<td>63600.00</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>2850.00</td>
<td></td>
<td>68400.00</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>2450.00</td>
<td></td>
<td>58800.00</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>3800.00</td>
<td></td>
<td>72000.00</td>
</tr>
<tr>
<td>7839</td>
<td>JONES</td>
<td>1250.00</td>
<td>500.00</td>
<td>42000.00</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>1500.00</td>
<td>1200.00</td>
<td>36000.00</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>1100.00</td>
<td></td>
<td>22800.00</td>
</tr>
<tr>
<td>7900</td>
<td>KING</td>
<td>5000.00</td>
<td></td>
<td>120000.00</td>
</tr>
<tr>
<td>7902</td>
<td>TURNER</td>
<td>1500.00</td>
<td>1200.00</td>
<td>36000.00</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>1300.00</td>
<td></td>
<td>31200.00</td>
</tr>
</tbody>
</table>
```

(14 rows)
11.5.1.5.1 Declaring parameters

Declare parameters in the procedure or function definition, and enclose them in parentheses following the procedure or function name. Parameters declared in the procedure or function definition are known as formal parameters. When you invoke the procedure or function, the calling program supplies the actual data to use in the called program's processing as well as the variables that receive the results of the called program's processing. The data and variables supplied by the calling program when the procedure or function is called are referred to as the actual parameters.

The following is the general format of a formal parameter declaration:

\[
(\text{name} \ [\text{IN | OUT | IN OUT}] \ \text{\textbf{data_type}} \ [\text{DEFAULT <value> }])
\]

- name is an identifier assigned to the formal parameter.
- Whether a parameter is IN, OUT, or IN OUT is referred to as the parameter's mode. If specified, IN defines the parameter for receiving input data into the procedure or function. An IN parameter can also be initialized to a default value. If specified, OUT defines the parameter for returning data from the procedure or function. If specified, IN OUT allows the parameter to be used for both input and output. If all of IN, OUT, and IN OUT are omitted, then the parameter acts as if it were defined as IN by default.
- data_type defines the data type of the parameter.
- value is a default value assigned to an IN parameter in the called program if you don't specify an actual parameter in the call.

This example shows a procedure that takes parameters:

```sql
CREATE OR REPLACE PROCEDURE emp_query
(
    p_deptno IN NUMBER,
    p_empno IN OUT NUMBER,
    p_ename IN OUT VARCHAR2,
    p_job OUT VARCHAR2,
    p_hiredate OUT DATE,
    p_sal OUT NUMBER
) IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
    INTO p_empno, p_ename, p_job, p_hiredate, p_sal
    FROM emp
    WHERE deptno = p_deptno
    AND (empno = p_empno
    OR ename = UPPER(p_ename));
END;
```

In this example, p_deptno is an IN formal parameter, p_empno and p_ename are IN OUT formal parameters, and p_job, p_hiredate and p_sal are OUT formal parameters.

Note: In the example, no maximum length was specified on the VARCHAR2 parameters, and no precision and scale were specified on the NUMBER parameters. It's illegal to specify a length, precision, scale, or other constraints on parameter declarations. These constraints are inherited from the actual parameters that are used when the procedure or function is called.

The emp_query procedure can be called by another program, passing it the actual parameters. This example is another SPL program that calls emp_query:

```sql
DECLARE
    v_deptno NUMBER(2);
    v_empno NUMBER(4);
    v_ename VARCHAR2(10);
    v_job VARCHAR2(9);
    v_hiredate DATE;
    v_sal NUMBER;
BEGIN
    v_deptno := 30;
    v_empno := 7900;
    v_ename := ' '; 
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name : ' || v_ename);
END;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
In this example, `v_deptno`, `v_empno`, `v_ename`, `v_job`, `v_hiredate`, and `v_sal` are the actual parameters.

The output from the example is:

```
Department : 30
Employee No: 7900
Name : JAMES
Job : CLERK
Hire Date : 03-DEC-81
Salary : 950
```

### 11.5.1.5.2 Positional versus named parameter notation

You can use either *positional* or *named* parameter notation when passing parameters to a function or procedure.

- If you specify parameters using *positional notation*, you must list the parameters in the order that they are declared. If you specify parameters with named notation, the order of the parameters doesn’t matter.

- If you specify parameters using named notation, list the name of each parameter followed by an arrow (`=>`) and the parameter value. Named notation is more verbose but makes your code easier to read and maintain.

This example uses positional and named parameter notation:

```sql
CREATE OR REPLACE PROCEDURE emp_info
(
    p_deptno IN NUMBER,
    p_empno OUT NUMBER,
    p_ename OUT VARCHAR2,
)
IS
BEGIN
    dbms_output.put_line('Department Number =' || p_deptno);
    dbms_output.put_line('Employee Number = ' || p_empno);
    dbms_output.put_line('Employee Name = ' || p_ename);
END;

emp_info(38, 7455, 'Clark');
```

To call the procedure using positional notation, pass the following:

```
emp_info(38, 7455, 'Clark');
```

To call the procedure using named notation, pass the following:

```
emp_info(p_ename => 'Clark', p_empno => 7455, p_deptno => 38);
```

If you used named notation, you don’t need to rearrange a procedure’s parameter list if the parameter list changes, the parameters are reordered, or an optional parameter is added.

When an argument has a default value and the argument isn’t a trailing argument, you must use named notation to call the procedure or function. This example shows a procedure with two leading default arguments:

```sql
CREATE OR REPLACE PROCEDURE check_balance
(
    p_customerID IN NUMBER DEFAULT NULL,
    p_balance IN NUMBER DEFAULT NULL,
    p_amount IN
)
IS
DECLARE
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
You can omit nontrailing argument values when you call this procedure only by using named notation. When using positional notation, only trailing arguments are allowed to default. You can call this procedure with the following arguments:

You can use a combination of positional and named notation, referred to as mixed notation, to specify parameters. This example shows using mixed parameter notation:

```
create or replace procedure emp_info
(
    p_deptno  in  number,
    p_empno   in out number,
    p_ename   in out varchar2,
)
is
begin
    dbms_output.put_line('Department Number = ' || p_deptno);
    dbms_output.put_line('Employee Number = ' || p_empno);
    dbms_output.put_line('Employee Name = ' || p_ename);
end;
```

You can call the procedure using mixed notation:

```
emp_info(38, p_ename => 'Clark', p_empno => 7455);
```

When using mixed notation, named arguments can't precede positional arguments.

### 11.5.1.5.3 Parameter modes

A parameter has one of three possible modes: **IN**, **OUT**, or **IN OUT**. The following characteristics of a formal parameter depend on its mode:

- Its initial value when the procedure or function is called
- Whether the called procedure or function can modify the formal parameter
- How the actual parameter value is passed from the calling program to the called program
- What happens to the formal parameter value when an unhandled exception occurs in the called program

The following table summarizes the behavior of parameters according to their mode.

<table>
<thead>
<tr>
<th>Mode property</th>
<th>IN</th>
<th>IN OUT</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formal parameter initialized to:</td>
<td>Actual parameter value</td>
<td>Actual parameter value</td>
<td>Actual parameter value</td>
</tr>
<tr>
<td>Formal parameter modifiable by the called program?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Actual parameter contains: (after normal called program termination)</td>
<td>Original actual parameter value prior to the call</td>
<td>Last value of the formal parameter</td>
<td>Last value of the formal parameter</td>
</tr>
</tbody>
</table>
Actual parameter contains: (after a handled exception in the called program)

<table>
<thead>
<tr>
<th>Mode property</th>
<th>IN</th>
<th>IN OUT</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual parameter contains: (after an unhandled exception in the called program)</td>
<td>Original actual parameter value prior to the call</td>
<td>Original actual parameter value prior to the call</td>
<td>Original actual parameter value prior to the call</td>
</tr>
</tbody>
</table>

As shown by the table:

- The **IN** formal parameter is initialized to the actual parameter with which it's called unless it was explicitly initialized with a default value. You can reference the **IN** parameter in the called program. However, the called program can't assign a new value to the **IN** parameter. After control returns to the calling program, the actual parameter always contains the same value that it had prior to the call.

- Like an **IN** parameter, an **IN OUT** formal parameter is initialized to the actual parameter with which it's called. Like an **OUT** parameter, an **IN OUT** formal parameter can be modified by the called program. The last value in the formal parameter is passed to the calling program's actual parameter if the called program ends without an exception. If a handled exception occurs, the value of the actual parameter takes on the last value assigned to the formal parameter. If an unhandled exception occurs, the value of the actual parameter remains as it was prior to the call.

- The **OUT** formal parameter is initialized to the actual parameter with which it's called. The called program can reference and assign new values to the formal parameter. If the called program ends without an exception, the actual parameter takes on the value last set in the formal parameter. If a handled exception occurs, the value of the actual parameter takes on the last value assigned to the formal parameter. If an unhandled exception occurs, the value of the actual parameter remains as it was prior to the call.

### 11.5.1.5.4 Using default values in parameters

You can set a default value of a formal parameter by including the **DEFAULT** clause or using the assignment operator (:::) in the **CREATE PROCEDURE** or **CREATE FUNCTION** statement.

**Syntax**

The general form of a formal parameter declaration is:

```sql
(<name> [ IN|OUT|IN OUT ] <data_type> [[DEFAULT | := ] <expr> ])
```

- **name** is an identifier assigned to the parameter.
- **IN|OUT|IN OUT** specifies the parameter mode.
- **data_type** is the data type assigned to the variable.
- **expr** is the default value assigned to the parameter. If you don't include a **DEFAULT** clause, the caller must provide a value for the parameter.

The default value is evaluated every time you invoke the function or procedure. For example, assigning **SYSDATE** to a parameter of type **DATE** causes the parameter to have the time of the current invocation, not the time when the procedure or function was created.

**Example**

This example uses the assignment operator to set a default value of **SYSDATE** into the parameter **hiredate**:

```sql
CREATE OR REPLACE PROCEDURE hire_emp (
    p_empno NUMBER,
    p_ename VARCHAR2,
    p_hiredate DATE := SYSDATE
) IS
    BEGIN
        INSERT INTO emp(empno, ename, hiredate)
        VALUES(p_empno, p_ename, p_hiredate);
        DBMS_OUTPUT.PUT_LINE('Hired!');
    END hire_emp;
```

If the parameter declaration includes a default value, you can omit the parameter from the actual parameter list when you call the procedure. Calls to the sample procedure (hire_emp) must include two arguments: the employee number (**p_empno**) and employee name (**p_ename**). The third parameter (**p_hiredate**) defaults to the value of **SYSDATE**.
If you do include a value for the actual parameter when you call the procedure, that value takes precedence over the default value. This command adds an employee with a hiredate of February 15, 2010, regardless of the current value of SYSDATE.

You can write the same procedure by substituting the `DEFAULT` keyword for the assignment operator:

```sql
CREATE OR REPLACE PROCEDURE hire_emp
(
  p_empno NUMBER,
  p_ename VARCHAR2,
  p_hiredate DATE DEFAULT SYSDATE
)
IS
BEGIN
  INSERT INTO emp(empno, ename, hiredate)
  VALUES(p_empno, p_ename, p_hiredate);
  DBMS_OUTPUT.PUT_LINE('Hired!');
END hire_emp;
```

### 11.5.1.6 Subprograms: subprocedures and subfunctions

You can use the capability and functionality of SPL procedure and function programs to your advantage to build well-structured and maintainable programs by organizing the SPL code into subprocedures and subfunctions.

Subprocedures and subfunctions have the following characteristics:

- The syntax, structure, and functionality of subprocedures and subfunctions are almost identical to standalone procedures and functions. The major difference is the use of the keyword `PROCEDURE` or `FUNCTION` instead of `CREATE PROCEDURE` or `CREATE FUNCTION` to declare the subprogram.
- Subprocedures and subfunctions provide isolation for the identifiers (that is, variables, cursors, types, and other subprograms) declared within itself. That is, you can't access or alter these identifiers from the upper, parent-level SPL programs or subprograms outside of the subprocedure or subfunction. This ensures that the subprocedure and subfunction results are reliable and predictable.
- The declaration section of subprocedures and subfunctions can include its own subprocedures and subfunctions. Thus, a multi-level hierarchy of subprograms can exist in the standalone program. In the hierarchy, a subprogram can access the identifiers of upper-level parent subprograms and also invoke upper-level parent subprograms. However, the same access to identifiers and invocation can't be done for lower-level child subprograms in the hierarchy.

You can declare and invoke subprocedures and subfunctions from any of the following types of SPL programs:

- Standalone procedures and functions
- Anonymous blocks
- Triggers
- Packages
- Procedure and function methods of an object type body
- Subprocedures and subfunctions declared in any of the preceding programs

#### 11.5.1.6.1 Creating a subprocedure

The `PROCEDURE` clause specified in the declaration section defines and names a subprocedure local to that block.

- The term block refers to the SPL block structure consisting of an optional declaration section, a mandatory executable section, and an optional exception section. Blocks are the structures for standalone procedures and functions, anonymous blocks, subprograms, triggers, packages, and object type methods.
- The phrase the identifier is local to the block means that the identifier (that is, a variable, cursor, type, or subprogram) is declared in the declaration section of that block. Therefore, the SPL code can access it in the executable section and optional exception section of that block.

**Declaring subprocedures**

You can declare subprocedures only after all the other variable, cursor, and type declarations included in the declaration section. Subprograms must be the last set of declarations.
PROCEDURE <name> [ ( <parameters> ) ]

| IS | AS |
| --- |

[ PRAGMA AUTONOMOUS_TRANSACTION; ]

[ <declarations> ]

BEGIN
  <statements>
END [ <name> ];

Where:

- **name** is the identifier of the subprocedure.
- **parameters** is a list of formal parameters.
- **PRAGMA AUTONOMOUS_TRANSACTION** is the directive that sets the subprocedure as an autonomous transaction.
- **declarations** are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, you must declare them after all other variable, cursor, and type declarations.
- **statements** are SPL program statements. The **BEGIN - END** block can contain an **EXCEPTION** section.

Example: Subprocedure in an anonymous block

This example is a subprocedure in an anonymous block:

```sql
DECLARE
  PROCEDURE list_emp IS
    v_empno NUMBER(4);
    v_ename VARCHAR2(10);
    CURSOR emp_cur IS
      SELECT empno, ename FROM emp ORDER BY empno;
  BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('Subprocedure list_emp:');
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
      FETCH emp_cur INTO v_empno, v_ename;
      EXIT WHEN emp_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
  END;
END;
BEGIN
  list_emp;
END;
```

Invoking this anonymous block produces the following output:

```
Subprocedure list_emp:
EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
```
Example: Subprocedure in a trigger

This example is a subprocedure in a trigger:

```sql
CREATE OR REPLACE TRIGGER dept_audit_trig
AFTER INSERT OR UPDATE OR DELETE ON dept
DECLARE
    v_action VARCHAR2(24);
    PROCEDURE display_action
        (p_action IN VARCHAR2)
        IS
            BEGIN
                DBMS_OUTPUT.PUT_LINE('User ' || USER || ' ' || p_action || ' dept on ' || TO_CHAR(SYSDATE, 'YYYY-MM-DD'));
            END;
            display_action(v_action);
    END;
BEGIN
    IF INSERTING THEN
        v_action := 'added';
    ELSIF UPDATING THEN
        v_action := 'updated';
    ELSIF DELETING THEN
        v_action := 'deleted';
    END IF;
    display_action(v_action);
END;
```

Invoking this trigger produces the following output:

```
INSERT INTO dept VALUES (50,'HR','DENVER');
```

```
User enteredb added dept on 2016-07-26
```

11.5.1.6.2 Creating a subfunction

The `FUNCTION` clause specified in the declaration section defines and names a subfunction local to that block.

- The term `block` refers to the SPL block structure consisting of an optional declaration section, a mandatory executable section, and an optional exception section. Blocks are the structures for standalone procedures and functions, anonymous blocks, subprograms, triggers, packages, and object type methods.

- The phrase `the identifier is local to the block` means that the identifier (that is, a variable, cursor, type, or subprogram) is declared in the declaration section of that block and is therefore accessible by the SPL code in the executable section and optional exception section of that block.

Declaring a subfunction

```sql
FUNCTION <name> [ (<parameters>) ]
RETURN <data_type>
    [ IS | AS
    ]
    [ PRAGMA AUTONOMOUS_TRANSACTION;
]```
[<declarations>]
BEGIN
    <statements>
END [<name>];

Where:

- **name** is the identifier of the subfunction.

- **parameters** is a list of formal parameters.

- **data_type** is the data type of the value returned by the function's **RETURN** statement.

- **PRAGMA AUTONOMOUS_TRANSACTION** is the directive that sets the subfunction as an autonomous transaction.

- **declarations** are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations.

- **statements** are SPL program statements. The **BEGIN - END** block can contain an **EXCEPTION** section.

**Example: Recursive subfunction**

This example shows the use of a recursive subfunction:

```sql
DECLARE
FUNCTION factorial
( n BINARY_INTEGER )
RETURN BINARY_INTEGER
IS BEGIN
IF n = 1 THEN
    RETURN n;
ELSE
    RETURN n * factorial(n-1);
END IF;
END factorial;
BEGIN
    FOR i IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE(i || '! = ' || factorial(i));
    END LOOP;
END;
```

The following is the output:

```
__OUTPUT__
i! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

### 11.5.1.6.3 Declaring block relationships

You can declare the relationship between blocks in an SPL program. The ability to invoke subprograms and access identifiers declared in a block depends on this relationship.

**About block relationships**

The following are the basic terms:

- A **block** is the basic SPL structure consisting of an optional declaration section, a mandatory executable section, and an optional exception section. Blocks implement standalone procedure and function programs, anonymous blocks, triggers, packages, and subprocedures and subfunctions.
An identifier (variable, cursor, type, or subprogram) local to a block means that it’s declared in the declaration section of the given block. You can access such local identifiers from the executable section and optional exception section of the block.

- The parent block contains the declaration of another block, that is, the child block.
- Descendent blocks are the set of blocks forming the child relationship starting from a given parent block.
- Ancestor blocks are the set of blocks forming the parental relationship starting from a given child block.
- The set of descendent (ancestor) blocks form a hierarchy.
- The level is an ordinal number of a given block from the highest ancestor block. For example, given a standalone procedure, the subprograms declared in the declaration section of this procedure are all at the same level, such as level 1. Additional subprograms in the declaration section of the subprograms declared in the standalone procedure are at the next level, that is, level 2.
- The sibling blocks are the set of blocks that have the same parent block, that is, they are all locally declared in the same block. Sibling blocks are at the same level relative to each other.

**Example**

The following schematic of a set of procedure declaration sections provides an example of a set of blocks and their relationships to their surrounding blocks.

The two vertical lines on the left-hand side of the blocks indicate there are two pairs of sibling blocks. block_1a and block_1b is one pair, and block_2a and block_2b is the second pair.

The relationship of each block with its ancestors is shown on the right-hand side of the blocks. Three hierarchical paths are formed when progressing up the hierarchy from the lowest-level child blocks.

The first consists of block_0, block_1a, block_2a, and block_3. The second is block_0, block_1a, and block_2b. The third is block_0, block_1b, and block_2b.

```sql
CREATE PROCEDURE block_0
IS
  +----- PROCEDURE block_1a  ------- Local to block_0
  |     |
  | IS |
  | |   |
  |   |   |
  |   |   |
  | +--- PROCEDURE block_2a  ---- Local to block_1a and descendant
  |   |   |
  | IS |
  | |   |
  |   |   |
  |   |   |
  | +--- PROCEDURE block_3   -- Local to block_1a and descendant
  |   |   |
  | IS |
  | |   |
  |   | Siblings |
  | |   |
  |   |
  | |   |
  | +--- END block_3;
  |   |   |
  | IS |
  | |   |
  |   |
  | |   |
  | +--- END block_2a;
  |   |   |
  | IS |
  | |   |
  |   |
  | |   |
  | +--- END block_2b;
  |
  |   |   |
  | IS |
  | |   |
  |   |
  | |   |
  | +--- END block_1a;  ---------
  +----- PROCEDURE block_1b;  ------- Local to block_0
  |     |
  IS |
  | |   |
  |   |   |
  |   |   |
  |   |   |
  +-----
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The rules for invoking subprograms based on block location are described starting with Invoking subprograms. The rules for accessing variables based on block location are described in Accessing subprogram variables.

11.5.1.6.4 Invoking subprograms

Invoke a subprogram in the same manner as a standalone procedure or function by specifying its name and any actual parameters.

You can invoke the subprogram with zero, one, or more qualifiers. Qualifiers are the names of the parent subprograms or labeled anonymous blocks forming the ancestor hierarchy from which the subprogram was declared.

Overview of subprograms

Invoke the subprogram using a dot-separated list of qualifiers ending with the subprogram name and any of its arguments:

```
[[<qualifier_1>][...]<qualifier_n>]<subprog> [[<arguments>]]
```

Specifying qualifiers

If specified, qualifier_n is the subprogram in which subprog was declared in its declaration section. The preceding list of qualifiers must reside in a continuous path up the hierarchy from qualifier_n to qualifier_l. qualifier_l can be any ancestor subprogram in the path as well as any of the following:

- Standalone procedure name containing the subprogram
- Standalone function name containing the subprogram
- Package name containing the subprogram
- Object type name containing the subprogram in an object type method
- An anonymous block label included prior to the DECLARE keyword if a declaration section exists, or prior to the BEGIN keyword if there is no declaration section

Note

qualifier_l can't be a schema name. If it is, an error is thrown when invoking the subprogram. This EDB Postgres Advanced Server restriction isn't compatible with Oracle databases, which allow the use of the schema name as a qualifier.

arguments is the list of actual parameters to pass to the subprocedure or subfunction.

Searching for subprograms

When you invoke the subprogram, the search for the subprogram occurs as follows:

- The invoked subprogram name of its type (that is, subprocedure or subfunction) along with any qualifiers in the specified order (referred to as the invocation list) is used to find a matching set of blocks residing in the same hierarchical order. The search begins in the block hierarchy where the lowest level is the block from where the subprogram is invoked. The declaration of the subprogram must be in the SPL code prior to the code line where it's invoked when the code is observed from top to bottom. (You can achieve an exception to this requirement using a forward declaration. See Using forward declarations.)
- If the invocation list doesn't match the hierarchy of blocks starting from the block where the subprogram is invoked, a comparison is made by matching the invocation list starting with the parent of the previous starting block. In other words, the comparison progresses up the hierarchy.
- If there are sibling blocks of the ancestors, the invocation list comparison also includes the hierarchy of the sibling blocks but always comparing in an upward level. It doesn't compare the
This comparison process continues up the hierarchies until the first complete match is found, in which case the located subprogram is invoked. The formal parameter list of the matched subprogram must comply with the actual parameter list specified for the invoked subprogram. Otherwise an error occurs when invoking the subprogram.

If no match is found after searching up to the standalone program, then an error is thrown when invoking the subprogram.

Note

The EDB Postgres Advanced Server search algorithm for subprogram invocation isn't completely compatible with Oracle databases. For Oracle, the search looks for the first match of the first qualifier (that is, qualifier_1). When such a match is found, all remaining qualifiers, the subprogram name, subprogram type, and arguments of the invocation must match the hierarchy content where the matching first qualifier is found. Otherwise an error is thrown. For EDB Postgres Advanced Server, a match isn't found unless all qualifiers, the subprogram name, and the subprogram type of the invocation match the hierarchy content. If such an exact match isn't initially found, EDB Postgres Advanced Server continues the search progressing up the hierarchy.

You can access the location of subprograms relative to the block from where the invocation is made as follows:

- You can invoke subprograms declared in the local block from the executable section or the exception section of the same block.
- You can invoke subprograms declared in the parent or other ancestor blocks from the child block of the parent or other ancestors.
- You can call subprograms declared in sibling blocks from a sibling block or from any descendent block of the sibling.

However, you can't access the following locations of subprograms relative to the block from where the invocation is made:

- Subprograms declared in blocks that are descendants of the block from where the invocation is attempted
- Subprograms declared in blocks that are descendants of a sibling block from where the invocation is attempted

Invoking locally declared subprograms

This example contains a single hierarchy of blocks contained in the standalone procedure `level_0`. In the executable section of the procedure `level_1a`, the means of invoking the local procedure `level_2a` are shown, with and without qualifiers.

Access to the descendant of the local procedure `level_2a`, which is the procedure `level_3a`, isn't permitted, with or without qualifiers. These calls are commented out in the example.

```
CREATE OR REPLACE PROCEDURE level_0
IS
    PROCEDURE level_1a
    IS
        PROCEDURE level_2a
        IS
            PROCEDURE level_3a
            IS
                BEGIN
                    DBMS_OUTPUT.PUT_LINE('........ BLOCK level_3a');
                    DBMS_OUTPUT.PUT_LINE('........ END BLOCK level_3a');
                END level_3a;
                BEGIN
                    DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
                    DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
                END level_2a;
                BEGIN
                    DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
                    called level_2a; -- Local block
                    called level_1a.level_2a; -- Qualified local block
                    called level_0.level_1a.level_2a; -- Double qualified local block
                    -- Error - Descendant of local block
                    level_2a.level_3a; -- Error - Descendant of local block
                    -- Error - Descendant of local block
                    DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
                END level_1a;
                BEGIN
                    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
                    level_1a;
                    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
                END level_0;
```

When the standalone procedure is invoked, the output is the following. This output indicates that the procedure `level_2a` is successfully invoked from the calls in the executable section of the
procedure level_1a:

BEGIN
level_0;
END;

BLOCK level_0
.. BLOCK level_1a
........ BLOCK level_2a
........ END BLOCK level_2a
........ BLOCK level_2a
........ END BLOCK level_2a
........ BLOCK level_2a
........ END BLOCK level_2a
.. END BLOCK level_1a
END BLOCK level_0

If you try to run the procedure level_0 with any of the calls to the descendent block uncommented, then an error occurs.

Invoking subprograms declared in ancestor blocks

This example shows how to invoke subprograms that are declared in parent and other ancestor blocks relative to the block where the invocation is made.

In this example, the executable section of procedure level_3a invokes the procedure level_2a, which is its parent block. v_cnt is used to avoid an infinite loop.

```
CREATE OR REPLACE PROCEDURE level_0 IS
  v_cnt NUMBER(2) := 8;
PROCEDURE level_1a IS
  PROCEDURE level_2a IS
    PROCEDURE level_3a IS
      BEGIN
        DBMS_OUTPUT.PUT_LINE('........ BLOCK level_3a');
        v_cnt := v_cnt + 1;
        IF v_cnt < 2 THEN
          level_2a; -- Parent block called
        END IF;
        DBMS_OUTPUT.PUT_LINE('........ END BLOCK level_3a');
      END level_3a;
      BEGIN
        DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
        level_3a;
        -- Local block called
        DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
      END level_2a;
      BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
        level_2a;
        -- Local block called
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
      END level_1a;
      BEGIN
        DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
        level_1a;
        DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
      END level_0;
END level_0;
```

The following is the output:

```
BEGIN
  v_cnt := 8;
  level_1a;
  level_2a;
  level_3a;
END
```

The following is the output:
In a similar example, the executable section of the procedure level_3a invokes the procedure level_1a, which is further up the ancestor hierarchy. v_cnt is again used to avoid an infinite loop.

```
CREATE OR REPLACE PROCEDURE level_0
IS
  v_cnt NUMBER(2) := 0;
  PROCEDURE level_1a
  IS
    PROCEDURE level_3a
    IS
      BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_0');
        level_1a;
      END;
      BEGIN
        DBMS_OUTPUT.PUT_LINE('... BLOCK level_0');
        level_1a;
      END;
      BEGIN
        DBMS_OUTPUT.PUT_LINE('.... BLOCK level_0');
        level_1a;
      END;
      BEGIN
        DBMS_OUTPUT.PUT_LINE('...... BLOCK level_0');
        level_1a;
      END;
    END level_3a;
    BEGIN
      DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
      level_2a;
    END;
    BEGIN
      DBMS_OUTPUT.PUT_LINE('... BLOCK level_1a');
      level_2a;
    END;
    BEGIN
      DBMS_OUTPUT.PUT_LINE('.... BLOCK level_1a');
      level_2a;
    END;
  END level_1a;
BEGIN
  DBMS_OUTPUT.PUT_LINE('BLOCK level_0);
  level_0;
END;
```

The following is the output:

```
BEGIN
  level_0;
END;
```

```
BLOCK level_0
.. BLOCK level_1a
... BLOCK level_2a
.... BLOCK level_3a
.. BLOCK level_1a
... BLOCK level_2a
.... BLOCK level_3a
.. BLOCK level_1a
... BLOCK level_2a
.... BLOCK level_3a
.. END BLOCK level_3a
.. END BLOCK level_1a
END BLOCK level_0
```
Invoking subprograms declared in sibling blocks

These examples show how you can invoke subprograms that are declared in a sibling block relative to the local, parent, or other ancestor blocks from where the invocation of the subprogram is made.

In this example, the executable section of the procedure `level_1b` invokes the procedure `level_1a`, which is its sibling block. Both are local to the standalone procedure `level_0`.

Invoking `level_2a` or, equivalently, `level_1a.level_2a` from the procedure `level_1b` is commented out as this call results in an error. Invoking a descendant subprogram (`level_2a`) of a sibling block (`level_1a`) isn't permitted.

```sql
CREATE OR REPLACE PROCEDURE level_0
IS
  v_cnt NUMBER(2) := 0;
PROCEDURE level_1a
IS
PROCEDURE level_2a
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
  DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
END level_2a;
BEGIN
  DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
  DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
BEGIN
  -- Sibling block called level_2a;
  -- Error - Descendant of sibling block level_1a.
  DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
END level_1b;
BEGIN
  DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
  level_1b;
BEGIN
  DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
  level_1b;
BEGIN
  DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
END;
```

The following is the output:

```
BEGIN
  level_0;
END;
```

In this example, the procedure `level_1a` is successfully invoked. It's the sibling of the procedure `level_1b`, which is an ancestor of the procedure `level_3b`. 

```
BEGIN
  level_0;
END;
```

```
BLOCK level_0
  .. BLOCK level_1b
  .. BLOCK level_1a
  .. END BLOCK level_1a
  .. END BLOCK level_1b
END BLOCK level_0
```
11.5.1.6.5 Using forward declarations

When you want to invoke a subprogram, you must declare it in the hierarchy of blocks in the standalone program prior to where you invoke it. In other words, when scanning the SPL code from beginning to end, the subprogram declaration must appear before its invocation.

However, you can construct the SPL code so that the full declaration of the subprogram appears in the SPL code after the point in the code where it’s invoked. (The full declaration includes its optional...
declaration section, its mandatory executable section, and optional exception section.)

You can do this by inserting a forward declaration in the SPL code prior to its invocation. The forward declaration is the specification of a subprocedure or subfunction name, formal parameters, and return type if it's a subfunction.

You must specify the full subprogram consisting of the optional declaration section, the executable section, and the optional exception section in the same declaration section as the forward declaration. However, it can appear following other subprogram declarations that invoke this subprogram with the forward declaration.

Typical use of a forward declaration is when two subprograms invoke each other:

```sql
DECLARE
FUNCTION add_one
(    p_add IN NUMBER ) RETURN NUMBER;
FUNCTION test_max
(    p_test IN NUMBER ) RETURN NUMBER IS
BEGIN
IF p_test < 5 THEN
    RETURN add_one(p_test);
END IF;
DBMS_OUTPUT.PUT('Final value is ');
RETURN p_test;
END;
FUNCTION add_one
(    p_add IN NUMBER ) RETURN NUMBER IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Increase by 1');
    RETURN test_max(p_add + 1);
END;
BEGIN
    DBMS_OUTPUT.PUT_LINE(test_max(3));
END;
```

Subfunction `test_max` invokes subfunction `add_one`, which also invokes subfunction `test_max`. A forward declaration is required for one of the subprograms, which is implemented for `add_one` at the beginning of the anonymous block declaration section.

The resulting output from the anonymous block is as follows:

```
_OUTPUT_
Increase by 1
Increase by 1
Final value is 5
```

### 11.5.1.6.6 Overloading subprograms

Generally, subprograms of the same type (subprocedure or subfunction) with the same name and same formal parameter specification can appear multiple times in the same standalone program as long as they aren't sibling blocks (that is, the subprograms aren't declared in the same local block).

You can invoke each subprogram individually depending on the use of qualifiers and the location where the subprogram invocation is made.

However, it's possible to declare subprograms, even as siblings, that are of the same subprogram type and name as long as certain aspects of the formal parameters differ. These characteristics (subprogram type, name, and formal parameter specification) are generally known as a program's signature.

The declaration of multiple subprograms where the signatures are identical except for certain aspects of the formal parameter specification is referred to as subprogram overloading.

#### Requirements

The particular overloaded subprogram to invoke is determined by a match of the actual parameters specified by the subprogram invocation and the formal parameter lists of the overloaded subprograms.
Any of the following differences permit overloaded subprograms:

- The number of formal parameters are different.
- At least one pair of data types of the corresponding formal parameters (that is, compared according to the same order of appearance in the formal parameter list) are different but aren't aliases.

The following differences alone don't permit overloaded subprograms:

- Different formal parameter names
- Different parameter modes (IN, IN OUT, OUT) for the corresponding formal parameters
- For subfunctions, different data types in the RETURN clause

One of the differences allowing overloaded subprograms is different data types.

Using aliases

Certain data types have alternative names referred to as aliases, which can be used for the table definition.

For example, you can specify fixed-length character data types as CHAR or CHARACTER. You can specify variable-length character data types as CHAR VARYING, CHARACTER VARYING, VARCHAR, or VARCHAR2. For integers, there are BINARY_INTEGER, PLS_INTEGER, and INTEGER data types. For numbers, there are NUMBER, NUMERIC, DEC, and DECIMAL data types.

For detailed information about the data types supported by EDB Postgres Advanced Server, see Data types.

Thus, when attempting to create overloaded subprograms, the formal parameter data types aren't considered different if the specified data types are aliases of each other.

You can determine if certain data types are aliases of other types by displaying the table definition containing the data types.

Example: Data types and aliases

The following table definition contains some data types and their aliases:

```sql
CREATE TABLE data_type_aliases
(
  dt_BLOB        BLOB,
  dt_LONG_RAW    LONG,
  dt_RAW         RAW,
  dt_BYTEA       BYTEA,
  dt_INTEGER     INTEGER,
  dt_BINARY_INTEGER   BINARY_INTEGER,
  dt_PLS_INTEGER  PLS_INTEGER,
  dt_DOUBLE_PRECISION DOUBLE PRECISION,
  dt_FLOAT       FLOAT,
  dt_NUMBER      NUMBER,
  dt_DECIMAL     DECIMAL,
  dt_NUMERIC     NUMERIC,
  dt_CHAR        CHAR,
  dt_CHARACTER   CHARACTER,
  dt_VARCHAR2    VARCHAR2,
  dt_CHAR_VARYING CHAR VARYING(4),
  dt_VARCHAR     VARCHAR(4)
);
```

Using the PSQL `\d` command to display the table definition, the Type column displays the data type internally assigned to each column based on its data type in the table definition.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt_blob</td>
<td>bytea</td>
</tr>
<tr>
<td>dt_long_raw</td>
<td>bytea</td>
</tr>
<tr>
<td>dt_raw</td>
<td>bytea(4)</td>
</tr>
<tr>
<td>dt_bytea</td>
<td>bytea</td>
</tr>
</tbody>
</table>
In the example, the base set of data types are bytea, integer, real, double precision, numeric, character, and character varying.

When attempting to declare overloaded subprograms, a pair of formal parameter data types that are aliases aren't enough to allow subprogram overloading. Thus, parameters with data types INTEGER and PLS_INTEGER can't overload a pair of subprograms. However, data types INTEGER and REAL, INTEGER and FLOAT, or INTEGER and NUMBER can overload the subprograms.

Note
The overloading rules based on formal parameter data types aren't compatible with Oracle databases. Generally, the EDB Postgres Advanced Server rules are more flexible, and certain combinations are allowed in EDB Postgres Advanced Server that result in an error when attempting to create the procedure or function in Oracle databases.

For certain pairs of data types used for overloading, you might need to cast the arguments specified by the subprogram invocation to avoid an error encountered during runtime of the subprogram. Invoking a subprogram must include the actual parameter list that can specifically identify the data types. Certain pairs of overloaded data types might require the CAST function to explicitly identify data types. For example, pairs of overloaded data types that might require casting during the invocation are CHAR and VARCHAR2, or NUMBER and REAL.

Example: Overloaded subfunctions

This example shows a group of overloaded subfunctions invoked from an anonymous block. The executable section of the anonymous block contains the use of the CAST function to invoke overloaded functions with certain data types.

```sql
DECLARE
    FUNCTION add_it
    (p_add_1 IN BINARY_INTEGER,
    p_add_2 IN BINARY_INTEGER
    ) RETURN VARCHAR2 IS BEGIN
        RETURN 'add_it BINARY_INTEGER: ' || TO_CHAR(p_add_1 + p_add_2, '9999.9999');
    END add_it;
END;
```
```sql
DECLARE
    FUNCTION add_it
    (p_add_1 IN NUMBER,
    p_add_2 IN NUMBER
    ) RETURN VARCHAR2 IS BEGIN
        RETURN 'add_it NUMBER: ' || TO_CHAR(p_add_1 + p_add_2, '9999.9999');
    END add_it;
END;
```
```sql
DECLARE
    FUNCTION add_it
    (p_add_1 IN REAL,
    p_add_2 IN REAL
    ) RETURN VARCHAR2 IS BEGIN
        RETURN 'add_it REAL: ' || TO_CHAR(p_add_1 + p_add_2, '9999.9999');
    END add_it;
END;
```
11.5.1.6.7 Accessing subprogram variables

You can access variables declared in blocks, such as subprograms or anonymous blocks, from the executable section or the exception section of other blocks depending on their relative location.

Accessing a variable means being able to reference it in a SQL statement or an SPL statement as you can with any local variable.

**Note**

If the subprogram signature contains formal parameters, you can access these in the same way as local variables of the subprogram. All discussion related to variables of a subprogram also applies to formal parameters of the subprogram.

Accessing variables includes not only those defined as a data type but also includes others such as record types, collection types, and cursors.

At most one qualifier can access the variable. The qualifier is the name of the subprogram or labeled anonymous block in which the variable was locally declared.

**Syntax**

The syntax to reference a variable is:

```
[<qualifier>.]<variable>
```

If specified, `<qualifier>` is the subprogram or labeled anonymous block in which `<variable>` was declared in its declaration section (that is, it's a local variable).

**Note**

In EDB Postgres Advanced Server, in only one circumstance are two qualifiers are permitted. This scenario is for accessing public variables of packages where you can specify the reference in the following format:

```
schema_name.package_name.public_variable_name
```

For more information about supported package syntax, see [Built-in packages](#).

**Requirements**
You can access variables in the following ways:

- Variables can be accessed as long as the block in which the variable was locally declared is in the ancestor hierarchical path starting from the block containing the reference to the variable. Such variables declared in ancestor blocks are referred to as global variables.
- If a reference to an unqualified variable is made, the first attempt is to locate a local variable of that name. If such a local variable doesn’t exist, then the search for the variable is made in the parent of the current block, and so forth, proceeding up the ancestor hierarchy. If such a variable isn’t found, then an error occurs when the subprogram is invoked.
- If a reference to a qualified variable is made, the same search process is performed but searching for the first match of the subprogram or labeled anonymous block that contains the local variable. The search proceeds up the ancestor hierarchy until a match is found. If such a match isn’t found, then an error occurs when the subprogram is invoked.

You can’t access the following location of variables relative to the block from where the reference to the variable is made:

- Variables declared in a descendent block
- Variables declared in a sibling block, a sibling block of an ancestor block, or any descendants within the sibling block

**Note**

The EDB Postgres Advanced Server process for accessing variables isn’t compatible with Oracle databases. For Oracle, you can specify any number of qualifiers, and the search is based on the first match of the first qualifier in a similar manner to the Oracle matching algorithm for invoking subprograms.

Accessing variables with the same name

This example shows similar access attempts when all variables in all blocks have the same name:

```sql
CREATE OR REPLACE PROCEDURE level_0
IS
  v_common VARCHAR2(20) := 'Value from level_0';
PROCEDURE level_1a
IS
  v_common VARCHAR2(20) := 'Value from level_1a';
PROCEDURE level_2a
IS
  v_common VARCHAR2(20) := 'Value from level_2a';
BEGIN
  DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
  DBMS_OUTPUT.PUT_LINE('............ v_common: ' || v_common);
  DBMS_OUTPUT.PUT_LINE('............ level_2a.v_common: ' || level_2a.v_common);
  DBMS_OUTPUT.PUT_LINE('............ level_1a.v_common: ' || level_1a.v_common);
  DBMS_OUTPUT.PUT_LINE('............ level_0.v_common: ' || level_0.v_common);
  DBMS_OUTPUT.PUT_LINE('........ END BLOCK level_2a');
END level_2a;
BEGIN
  DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
  DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
  DBMS_OUTPUT.PUT_LINE('.... level_0.v_common: ' || level_0.v_common);
  level_2a;
  DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
END level_1a;
PROCEDURE level_1b
IS
  v_common VARCHAR2(20) := 'Value from level_1b';
BEGIN
  DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
  DBMS_OUTPUT.PUT_LINE('.... v_common: ' || v_common);
  DBMS_OUTPUT.PUT_LINE('.... level_0.v_common: ' || level_0.v_common);
  level_2a;
  DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
END level_1b;
BEGIN
  DBMS_OUTPUT.PUT_LINE(' BLOCK level_0');
END level_0;
```
The following is the output showing the content of each variable when the procedure is invoked:

```
BEGIN
level_0;
END;

BLOCK level_8
  .. v_common: Value from level_0
  .. BLOCK level_1a
    .... v_common: Value from level_1a
    .... level_0.v_common: Value from level_8
    ...... BLOCK level_2a
    ....... v_common: Value from level_2a
    ....... level_2a.v_common: Value from level_2a
    ....... level_1a.v_common: Value from level_1a
    ....... level_0.v_common: Value from level_0
    ...... END BLOCK level_2a
  ... END BLOCK level_1a
  .. BLOCK level_1b
    .... v_common: Value from level_1b
    .... level_0.v_common: Value from level_0
    .. END BLOCK level_1b
END BLOCK level_8
```

Using labels to qualify access to variables

You can also use the labels on anonymous blocks to qualify access to variables. This example shows variable access in a set of nested anonymous blocks:

```
DECLARE
  v_common VARCHAR2(20) := 'Value from level_0';
BEGIN
  DBMS_OUTPUT.PUT_LINE('.. BLOCK level_0');
  DBMS_OUTPUT.PUT_LINE('.. v_common: ' ||
    v_common);
  <<level_1a>>
  DECLARE
    v_common VARCHAR2(20) := 'Value from level_1a';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
    DBMS_OUTPUT.PUT_LINE('.. v_common: ' ||
      v_common);
    <<level_2a>>
    DECLARE
      v_common VARCHAR2(20) := 'Value from level_2a';
    BEGIN
      DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2a');
      DBMS_OUTPUT.PUT_LINE('...... v_common: ' ||
        v_common);
      DBMS_OUTPUT.PUT_LINE('...... level_1a.v_common: ' ||
        level_1a.v_common);
      DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2a');
    END;
  END;
  <<level_1b>>
  DECLARE
    v_common VARCHAR2(20) := 'Value from level_1b';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
```
The following is the output showing the content of each variable when the anonymous block is invoked:

```
__OUTPUT__
BLOCK level_0
.. v_common: Value from level_0
.. BLOCK
level_1a
.... v_common: Value from level_1a
....... BLOCK
level_2a
........ v_common: Value from level_2a
......... level_1a.v_common: Value from level_1a
........ END BLOCK
level_2a
.. END BLOCK
level_1a
.. BLOCK
level_1b
.... v_common: Value from level_1b
....... level_1b.v_common: Value from level_1b
.... END BLOCK
level_1b
.. END BLOCK
level_1b
END BLOCK level_0
END BLOCK level_0
END BLOCK level_0;
```

Examples

Example: Accessing record types in parent blocks

This example is an object type whose object type method, `display_emp`, contains the record type `emp_typ` and the subprocedure `emp_sal_query`. The record variable `r_emp` declared locally to `emp_sal_query` can access the record type `emp_typ` declared in the parent block `display_emp`.

```sql
CREATE OR REPLACE TYPE emp_pay_obj_typ AS OBJECT
(
  empno NUMBER(4),
  MEMBER PROCEDURE display_emp(SELF IN OUT emp_pay_obj_typ)
);
CREATE OR REPLACE TYPE BODY emp_pay_obj_typ AS
  MEMBER PROCEDURE display_emp (SELF IN OUT emp_pay_obj_typ)
  IS
    TYPE emp_typ IS RECORD
    (
      ename emp.ename%TYPE,
      job emp.job%TYPE,
      hiredate emp.hiredate%TYPE,
      sal emp.sal%TYPE,
      deptno emp.deptno%TYPE
    );
    PROCEDURE emp_sal_query
    (p_empno IN emp.empno%TYPE
    )
    IS
      r_emp emp_typ;
      v_avgsal emp.sal%TYPE;
    ```
The following is the output displayed when an instance of the object type is created and the procedure \texttt{display_emp} is invoked:

```
DECLARE
  v_emp EMP_PAY_OBJ_TYP;
BEGIN
  v_emp := emp_pay_obj_typ(7900);
  v_emp.display_emp;
END;
```

Employee #: 7900
Name : JAMES
Job : CLERK
Hire Date : 03-DEC-81 00:00:00
Salary : 950.00
Dept #: 38
Employee's salary does not exceed the department average of 1566.67

Example: Accessing an upper-level procedure

This example is a package with three levels of subprocedures. A record type, collection type, and cursor type declared in the upper-level procedure can be accessed by the descendent subprocedure.

```
CREATE OR REPLACE PACKAGE emp_dept_pkg
IS
  PROCEDURE display_emp
  (p_deptno NUMBER);
END;
```

```
CREATE OR REPLACE PACKAGE BODY emp_dept_pkg
IS
  PROCEDURE display_emp
  (p_deptno NUMBER);
END;
```
The following is the output displayed when the top-level package procedure is invoked:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
</tr>
</tbody>
</table>

Example: Accessing variables in blocks

This example shows how variables in various blocks are accessed, with and without qualifiers. The lines that are commented out show attempts to access variables that result in an error.

```plsql
CREATE OR REPLACE PROCEDURE level_0
IS
  v_level_0 VARCHAR2(20) := 'Value from level_0';
PROCEDURE level_1a
IS
  v_level_1a VARCHAR2(20) := 'Value from level_1a';
```
PROCEDURE
level_2a
IS
  v_level_2a VARCHAR2(20) := 'Value from level_2a';
BEGIN
  DBMS_OUTPUT.PUT_LINE('.... BLOCK level_2a');
  DBMS_OUTPUT.PUT_LINE('..... v_level_2a : ' || v_level_2a);
  DBMS_OUTPUT.PUT_LINE('...... v_level_1a : ' || v_level_1a);
  DBMS_OUTPUT.PUT_LINE('........ level_1a.v_level_1a : ' || level_1a.v_level_1a);
  DBMS_OUTPUT.PUT_LINE('......... v_level_0 : ' || v_level_0);
  DBMS_OUTPUT.PUT_LINE('........ level_0.v_level_0 : ' || level_0.v_level_0);
  DBMS_OUTPUT.PUT_LINE('.... END BLOCK level_2a');
END level_2a;
BEGIN
  level_1a;
  level_1b;
  DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
BEGIN
  level_0;
END;

The following is the output showing the content of each variable when the procedure is invoked:

```
BEGIN
  level_0;
END;

BLOCK level_0
  .. v_level_0: Value from level_0
  .. BLOCK level_1a
    .... BLOCK level_2a
    ....... v_level_2a: Value from level_2a
    ......... v_level_1a: Value from level_1a
    ........ v_level_0: Value from level_0
    ........ level_0.v_level_0: Value from level_0
    .... END BLOCK level_2a
```
11.5.1.7 Compilation errors in procedures and functions

When the EDB Postgres Advanced Server parsers compile a procedure or function, they confirm that both the `CREATE` statement and the program body (the portion of the program that follows the `AS` keyword) conform to the grammar rules for SPL and SQL constructs. By default, the server stops compiling if a parser detects an error. The parsers detect syntax errors in expressions, but they don't detect semantic errors. Semantic errors include an expression referencing a nonexistent column, table, or function, or a value of incorrect type.

Setting an error count compilation limit

Setting `spl.max_error_count` instructs the server to stop parsing if it encounters the specified number of errors in SPL code or when it encounters an error in SQL code. The default value of `spl.max_error_count` is 10. The maximum value is 1000. Setting `spl.max_error_count` to a value of 1 instructs the server to stop parsing when it encounters the first error in either SPL or SQL code.

You can use the `SET` command to specify a value for `spl.max_error_count` for your current session. The syntax is:

```
SET spl.max_error_count = <number_of_errors>
```

Where `number_of_errors` specifies the number of SPL errors that can occur before the server stops compiling. For example:

```
SET spl.max_error_count = 6
```

Example

The example instructs the server to continue past the first five SPL errors it encounters. When the server encounters the sixth error, it stops validating and prints six detailed error messages and one error summary.

To save time when developing new code or when importing code from another source, you might want to set the `spl.max_error_count` configuration parameter to a relatively high number of errors.

If you instruct the server to continue parsing in spite of errors in the SPL code in a program body, and the parser encounters an error in a segment of SQL code, there can be more errors in any SPL or SQL code that follows the incorrect SQL code. For example, the following function results in two errors:

```
CREATE FUNCTION computeBonus(baseSalary number) RETURN number AS
BEGIN
  bonus := baseSalary * 1.10;
  total := bonus + 100;
  RETURN bonus;
END;
```

ERROR: "bonus" is not a known variable
ERROR: "total" is not a known variable
ERROR: compilation of SPL function/procedure "computebonus" failed due to 2 errors

This example adds a `SELECT` statement to the example. The error in the `SELECT` statement masks the other errors that follow.

```
CREATE FUNCTION computeBonus(employeeName number) RETURN number AS
BEGIN
  SELECT salary INTO baseSalary FROM emp
```

11.5.1.8 Program security

You can control the security over whether a user can execute an SPL program. You can also control the database objects an SPL program can access for any given user executing the program. These are controlled by the following:

- Privilege to execute a program
- Privileges granted on the database objects (including other SPL programs) that a program attempts to access
- Whether the program is defined with definer’s rights or invoker’s rights

11.5.1.8.1 EXECUTE privilege

An SPL program (function, procedure, or package) can begin execution only if any of the following are true:

- The current user is a superuser.
- The current user was granted EXECUTE privilege on the SPL program.
- The current user inherits EXECUTE privilege on the SPL program by virtue of being a member of a group that has this privilege.
- EXECUTE privilege was granted to the PUBLIC group.

Whenever an SPL program is created in EDB Postgres Advanced Server, EXECUTE privilege is granted to the PUBLIC group by default. Therefore, any user can immediately execute the program.

You can remove this default privilege by using the REVOKE EXECUTE command. For example:

```
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
```

You can then grant explicit EXECUTE privilege on the program to individual users or groups.

```
GRANT EXECUTE ON PROCEDURE list_emp TO john;
```

Now, user john can execute the list_emp program. Other users who don't meet any of the required conditions can't.

Once a program begins to execute, the next aspect of security is the privilege checks that occur if the program attempts to perform an action on any database object including:

- Reading or modifying table or view data
- Creating, modifying, or deleting a database object such as a table, view, index, or sequence
- Obtaining the current or next value from a sequence
- Calling another program (function, procedure, or package)

Each such action can be protected by privileges on the database object either allowed or disallowed for the user.

It's possible for a database to have more than one object of the same type with the same name, but each such object belongs to a different schema in the database. If this is the case, which object is being referenced by an SPL program? For more information, see Database object name resolution.

11.5.1.8.2 Database object name resolution

You can reference a database object inside an SPL program either by its qualified name or by an unqualified name. A qualified name is in the form of `schema_name`, where `schema` is the name of the schema under which the database object with identifier `name` exists. An unqualified name doesn't have the `schema` portion. When a reference is made to a qualified name, there is no ambiguity as to which database object is intended. It either does or doesn't exist in the specified schema.

Locating an object with an unqualified name, however, requires the use of the current user's search path. When a user becomes the current user of a session, a default search path is always associated with that user. The search path consists of a list of schemas that are searched in left-to-right order for locating an unqualified database object reference. The object is considered nonexistent if it can't be found in any of the schemas in the search path. You can display the default search path in PSQL using the `SHOW search_path` command.
In this example, a new database is created along with two users:

- **hr_mgr**, who owns a copy of the entire sample application in schema **hr_mgr**
- **sales_mgr**, who owns a schema named **sales_mgr** that has a copy of only the **emp** table containing only the employees who work in sales

The procedure **list_emp**, function **hire_clerk**, and package **emp_admin** are used in this example. All of the default privileges that are granted upon installation of the sample application are removed and then explicitly regranted so as to present a more secure environment.

Programs **list_emp** and **hire_clerk** are changed from the default of definer's rights to invoker's rights. Then, when **sales_mgr** runs these programs, they act on the **emp** table in the **sales_mgr** schema since the **sales_mgr** search path and privileges are used for name resolution and authorization checking.

Programs **get_dept_name** and **hire_emp** in the **emp_admin** package are then executed by **sales_mgr**. In this case, the **dept** table and **emp** table in the **hr_mgr** schema are accessed as **hr_mgr** is the owner of the **emp_admin** package which is using definer's rights. Since the default search path is in effect with the **$user** placeholder, the schema matching the user (in this case, **hr_mgr**) is used to find the tables.
Step 1: Create database and users

As user `enterprisedb`, create the hr database:

```sql
CREATE DATABASE hr;
```

Switch to the hr database and create the users:

```sql
\c hr enterprisedb
CREATE USER hr_mgr IDENTIFIED BY password;
CREATE USER sales_mgr IDENTIFIED BY password;
```

Step 2: Create the sample application

Create the entire sample application, owned by hr mgr, in the hr_mgr schema.

```sql
\c hr_mgr
\i /usr/edb/as14/share/edb-sample.sql
```

BEGIN
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE VIEW
CREATE SEQUENCE
.
.
.
CREATE PACKAGE
CREATE PACKAGE BODY
COMMIT

Step 3: Create the emp table in schema sales_mgr

Create a subset of the emp table owned by sales mgr in the sales mgr schema.

```sql
\c -
hr_mgr
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
\c -
sales_mgr
CREATE TABLE emp AS SELECT * FROM hr_mgr.emp WHERE job = 'SALESMAN';
```

The GRANT USAGE ON SCHEMA command allows sales mgr access into the hr_mgr’s schema to make a copy of the hr_mgr emp table. This step is required in EDB Postgres Advanced Server and isn’t compatible with Oracle databases. Oracle doesn’t have the concept of a schema that’s distinct from its user.

Step 4: Remove default privileges

Remove all privileges to later illustrate the minimum required privileges needed.

```sql
\c -
hr_mgr
REVOKE USAGE ON SCHEMA hr_mgr FROM sales_mgr;
REVOKE ALL ON dept FROM PUBLIC;
REVOKE ALL ON emp FROM PUBLIC;
REVOKE ALL ON next_empno FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION new_empno() FROM PUBLIC;
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
```
Step 5: Change list_emp to invoker’s rights

While connected as user hr_mgr, add the AUTHID CURRENT_USER clause to the list_emp program and resave it in EDB Postgres Advanced Server. When performing this step, be sure you’re logged in as hr_mgr. Otherwise the modified program might end up in the public schema instead of in the hr_mgr schema.

```sql
CREATE OR REPLACE PROCEDURE list_emp
AUTHID CURRENT_USER
IS
  v_empno NUMBER(4);
  v_ename VARCHAR2(10);
  CURSOR emp_cur IS
    SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
  OPEN emp_cur;
  DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
  DBMS_OUTPUT.PUT_LINE('----- -------');
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
  END LOOP;
  CLOSE emp_cur;
END;
END;
```

Step 6: Change hire_clerk to invoker’s rights and qualify call to new_empno

While connected as user hr_mgr, add the AUTHID CURRENT_USER clause to the hire_clerk program.

Also, after the BEGIN statement, fully qualify the reference new_empno to hr_mgr.new_empno to ensure the hire_clerk function call to the new_empno function resolves to the hr_mgr schema.

When resaving the program, be sure you’re logged in as hr_mgr. Otherwise the modified program might end up in the public schema instead of in the hr_mgr schema.

```sql
CREATE OR REPLACE FUNCTION hire_clerk
  (p_ename VARCHAR2,
   p_deptno NUMBER)
RETURN NUMBER
AUTHID CURRENT_USER
IS
  v_empno NUMBER(4);
  v_ename VARCHAR2(10);
  v_job VARCHAR2(9);
  v_mgr NUMBER(4);
  v_hiredate DATE;
  v_sal NUMBER(7,2);
  v_comm NUMBER(7,2);
  v_deptno NUMBER(2);
BEGIN
  v_empno := hr_mgr.new_empno;
  INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK',
    7782,
    TRUNC(SYSDATE), 950.00, NULL, p_deptno);
  SELECT ename, job, mgr, hiredate, sal, comm, deptno
  into v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm,
  v_deptno
  FROM emp WHERE empno = v_empno;
```
Step 7: Grant required privileges

While connected as user hr_mgr, grant the privileges needed so sales_mgr can execute the list_emp procedure, hire_clerk function, and emp_admin package. The only data object sales_mgr has access to is the emp table in the sales_mgr schema. sales_mgr has no privileges on any table in the hr_mgr schema.

```
GRANT USAGE ON SCHEMA hr_mgr TO sales_mgr;
GRANT EXECUTE ON PROCEDURE list_emp TO sales_mgr;
GRANT EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) TO sales_mgr;
GRANT EXECUTE ON FUNCTION new_empno() TO sales_mgr;
GRANT EXECUTE ON PACKAGE emp_admin TO sales_mgr;
```

Step 8: Run programs list_emp and hire_clerk

Connect as user sales_mgr, and run the following anonymous block:

```
\c sales_mgr
DECLARE
  v_empno NUMBER(4);
BEGIN
  hr_mgr.list_emp;
  DBMS_OUTPUT.PUT_LINE('*** Adding new employee ***');
  v_empno := hr_mgr.hire_clerk('JONES',40);
  DBMS_OUTPUT.PUT_LINE('*** After new employee added ***');
  hr_mgr.list_emp;
END;
```

EMPNO   ENAME
------   ------
  7499    ALLEN
  7521    WARD
  7654    MARTIN
  7844    TURNER

*** Adding new employee
 *** Department : 40
The table and sequence accessed by the programs of the anonymous block are shown in the following diagram. The gray ovals represent the schemas of sales_mgr and hr_mgr. The current user during each program execution is shown in parenthesis in bold red font.

![Diagram showing access to sales_mgr and hr_mgr schemas]

**Fig. 1: Invokers Rights Programs**

Selecting from the sales_mgr emp table shows that the update was made in this table.

```sql
SELECT empno, ename, hiredate, sal, deptno,
hr_mgr.emp_admin.get_dept_name(deptno) FROM
sales_mgr.emp;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>hiredate</th>
<th>sal</th>
<th>deptno</th>
<th>get_dept_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>20-FEB-81 00:00:00</td>
<td>1600.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>22-FEB-81 00:00:00</td>
<td>1250.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>28-SEP-81 00:00:00</td>
<td>1250.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>08-SEP-81 00:00:00</td>
<td>1500.00</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>8000</td>
<td>JONES</td>
<td>08-NOV-07 00:00:00</td>
<td>950.00</td>
<td>40</td>
<td>OPERATIONS</td>
</tr>
</tbody>
</table>
```

The following diagram shows that the SELECT command references the emp table in the sales_mgr schema. However, the dept table referenced by the get_dept_name function in the emp_admin package is from the hr_mgr schema since the emp_admin package has definer's rights and is owned by hr_mgr. The default search path setting with the $user placeholder resolves the access by user hr_mgr to the dept table in the hr_mgr schema.

**Fig. 2: Access to dept Table**
Step 9: Run program hire_emp in the emp_admin package

While connected as user sales_mgr, run the hire_emp procedure in the emp_admin package.

EXEC hr_mgr.emp_admin.hire_emp(9001, 'ALICE', 'SALESMAN', 8000, TRUNC(SYSDATE), 1000, 7369, 40);

This diagram shows that the hire_emp procedure in the emp_admin definer's rights package updates the emp table belonging to hr_mgr. The object privileges of hr_mgr are used and the default search path setting with the $user placeholder resolves to the schema of hr_mgr.

Fig. 3: Definer's Rights Package

Now connect as user hr_mgr. The following SELECT command verifies that the new employee was added to the hr_mgr emp table since the emp_admin package has definer's rights and hr_mgr is the emp_admin owner.

\c hr_mgr

SELECT empno, ename, hiredate, sal, deptno, hr_mgr.emp_admin.get_dept_name(deptno) FROM hr_mgr.emp;

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>hiredate</th>
<th>sal</th>
<th>deptno</th>
<th>get_dept_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>17-DEC-80 00:00:00</td>
<td>808.00</td>
<td>28</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>20-FEB-81 00:00:00</td>
<td>1608.00</td>
<td>38</td>
<td>SALES</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>22-FEB-81 00:00:00</td>
<td>1258.00</td>
<td>38</td>
<td>SALES</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>02-APR-81 00:00:00</td>
<td>2975.00</td>
<td>28</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>28-SEP-81 00:00:00</td>
<td>1258.00</td>
<td>38</td>
<td>SALES</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>01-MAY-81 00:00:00</td>
<td>2858.00</td>
<td>38</td>
<td>SALES</td>
</tr>
<tr>
<td>7872</td>
<td>CLARK</td>
<td>09-JUN-81 00:00:00</td>
<td>2456.00</td>
<td>18</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>19-APR-81 00:00:00</td>
<td>3808.00</td>
<td>28</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>17-NOV-81 00:00:00</td>
<td>5908.00</td>
<td>18</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>08-SEP-81 00:00:00</td>
<td>1508.00</td>
<td>38</td>
<td>SALES</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>23-MAY-87 00:00:00</td>
<td>1108.00</td>
<td>28</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7908</td>
<td>JAMES</td>
<td>03-DEC-81 00:00:00</td>
<td>958.00</td>
<td>38</td>
<td>SALES</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>03-DEC-81 00:00:00</td>
<td>3808.00</td>
<td>28</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>23-JAN-82 00:00:00</td>
<td>1308.00</td>
<td>18</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>9001</td>
<td>ALICE</td>
<td>08-NOV-07 00:00:00</td>
<td>8800.00</td>
<td>48</td>
<td>OPERATIONS</td>
</tr>
</tbody>
</table>
(15 rows)

11.5.2 Using variable declarations

SPL is a block-structured language. The first section that can appear in a block is the declaration. The declaration contains the definition of variables, cursors, and other types that you can use in SPL statements contained in the block.

11.5.2.1 Declaring a variable

Generally, you must declare all variables used in a block in the declaration section of the block. A variable declaration consists of a name that's assigned to the variable and its data type. Optionally, you can initialize the variable to a default value in the variable declaration.
Syntax

The general syntax of a variable declaration is:

```sql
<nname> <ntype> [ { := | DEFAULT } { <expression> | NULL } ]
```

- `<nname>` is an identifier assigned to the variable.
- `<ntype>` is the data type assigned to the variable.
- `[ := expression ]`, if given, specifies the initial value assigned to the variable when the block is entered. If the clause isn't given then the variable is initialized to the SQL `NULL` value.

The default value is evaluated every time the block is entered. So, for example, assigning `SYSDATE` to a variable of type `DATE` causes the variable to have the time of the current invocation, not the time when the procedure or function was precompiled.

Example: Variable declarations that use defaults

This procedure shows some variable declarations that use defaults consisting of string and numeric expressions:

```sql
CREATE OR REPLACE PROCEDURE dept_salary_rpt
(p_deptno NUMBER)
IS
todays_date DATE := SYSDATE;
rpt_title VARCHAR2(60) := 'Report For Department # ' || p_deptno || ' on ' || todays_date;
base_sal NUMBER := 35525;
base_comm_rate NUMBER := 1.33333;
base_annual NUMBER := ROUND(base_sal * base_comm_rate, 2);
BEGIN
DBMS_OUTPUT.PUT_LINE(rpt_title);
DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

The following output of the procedure shows that default values in the variable declarations are assigned to the variables:

```
EXEC dept_salary_rpt(20);
```

Report For Department # 20 on 10-JUL-07 16:44:45
Base Annual Salary: 47366.55

11.5.2.2 Using %TYPE in variable declarations

Often, variables are declared in SPL programs that are used to hold values from tables in the database. To ensure compatibility between the table columns and the SPL variables, make sure their data types are the same.

However, often a change is made to the table definition. If the data type of the column is changed, you might need to make the corresponding change to the variable in the SPL program.

Instead of coding the specific column data type into the variable declaration, you can use the column attribute `%TYPE`. Specify a qualified column name in dot notation or the name of a previously declared variable as a prefix to `%TYPE`. The data type of the column or variable prefixed to `%TYPE` is assigned to the variable being declared. If the data type of the given column or variable changes, the new data type is associated with the variable, and you don't need to modify the declaration code.

Note

You can use the `%TYPE` attribute with formal parameter declarations as well.
name is the identifier assigned to the variable or formal parameter that's being declared.

column is the name of a column in table or view.

variable is the name of a variable that was declared prior to the variable identified by name.

Note

The variable doesn't inherit any of the column's other attributes that you specify on the column with the NOT NULL clause or the DEFAULT clause.

Example: Defining parameters using %TYPE

In this example, a procedure:

- Queries the emp table using an employee number
- Displays the employee's data
- Finds the average salary of all employees in the department to which the employee belongs
- Compares the chosen employee's salary with the department average

```sql
CREATE OR REPLACE PROCEDURE emp_sal_query
(
    p_empno   IN   NUMBER
) IS
    v_ename   VARCHAR2(18);
    v_job     VARCHAR2(9);
    v_hiredate DATE;
    v_sal     NUMBER(7,2);
    v_deptno  NUMBER(2);
    v_avgsal  NUMBER(7,2); BEGIN
    SELECT ename, job, hiredate, sal,
           deptno INTO v_ename, v_job, v_hiredate, v_sal,
           v_deptno FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);
    v_avgsal := AVG(sal) FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee’s salary is more than the department average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee’s salary does not exceed the department average of ' || v_avgsal);
    END IF;
END;
```

Alternatively, you can write the procedure without explicitly coding the emp table data types into the declaration section of the procedure:

```sql
CREATE OR REPLACE PROCEDURE emp_sal_query
(
    p_empno   IN   emp.empno%TYPE
) IS
    v_ename   emp.ename%TYPE;
    v_job     emp.job%TYPE;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
p_empno shows an example of a formal parameter defined using %TYPE.

v_avgsal shows the use of %TYPE referring to another variable instead of a table column.

The following is sample output from executing this procedure:

```
EXEC emp_sal_query(7698);
```

Employee #: 7698
Name : BLAKE
Job : MANAGER
Hire Date : 01-MAY-81 00:00:00
Salary : 2850.00
Dept #: 30

Employee’s salary is more than the department average of 1566.67

11.5.2.3 Using %ROWTYPE in record declarations

The %TYPE attribute provides an easy way to create a variable that depends on a column’s data type. Using the %ROWTYPE attribute, you can define a record that contains fields that correspond to all columns of a given table. Each field takes on the data type of its corresponding column. The fields in the record don’t inherit any of the columns’ other attributes like those specified with the NOT NULL clause or the DEFAULT clause.

A record is a named, ordered collection of fields. A field is similar to a variable. It has an identifier and data type but has the additional property of belonging to a record. You must reference it using dot notation with the record name as its qualifier.

Syntax

You can use the %ROWTYPE attribute to declare a record. The %ROWTYPE attribute is prefixed by a table name. Each column in the named table defines an identically named field in the record with the same data type as the column.

```xml
<record> <table>%ROWTYPE;
```
record is an identifier assigned to the record.

Example

This example shows how you can modify the emp_sal_query procedure from Using %TYPE in variable declarations to use emp%ROWTYPE to create a record named r_emp instead of declaring individual variables for the columns in emp:

```sql
CREATE OR REPLACE PROCEDURE emp_sal_query
(
    p_empno IN emp.empno%TYPE
) IS
    r_emp emp%ROWTYPE;
    v_avgsal emp.sal%TYPE;
    BEGIN
        SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
        DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
        DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
        DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);
        SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
        IF r_emp.sal > v_avgsal THEN
            DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department average of ' || v_avgsal);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department average of ' || v_avgsal);
        END IF;
    END;
```

11.5.2.4 User-defined record types and record variables

You can declare records based on a table definition using the %ROWTYPE attribute, as shown in Using %ROWTYPE in record declarations. You can also define a new record structure that isn't tied to a particular table definition.

You use the TYPE IS RECORD statement to create the definition of a record type. A record type is a definition of a record made up of one or more identifiers and their corresponding data types. You can't use a record type by itself to manipulate data.

Syntax

The syntax for a TYPE IS RECORD statement is:

```sql
TYPE <rec_type> IS RECORD ( <fields> )
```

Where <fields> is a comma-separated list of one or more field definitions of the following form:

```sql
<field_name> <data_type> [NOT NULL] [= DEFAULT] [default_value]
```
Where:

- `rec_type` is an identifier assigned to the record type.
- `field_name` is the identifier assigned to the field of the record type.
- `data_type` specifies the data type of `field_name`.
- The `DEFAULT` clause assigns a default data value for the corresponding field. The data type of the default expression must match the data type of the column. If you don't specify a default, then the default is `NULL`.

A record variable or record is an instance of a record type. A record is declared from a record type. The properties of the record such as its field names and types are inherited from the record type.

The following is the syntax for a record declaration:

```sql
<record> <rectype>
```

`record` is an identifier assigned to the record variable. `rectype` is the identifier of a previously defined record type. Once declared, you can't then use a record to hold data.

Use dot notation to reference the fields in the record:

```sql
<record>.<field>
```

`record` is a previously declared record variable and `field` is the identifier of a field belonging to the record type from which `record` is defined.

Example

This `emp_sal_query` procedure uses a user-defined record type and record variable:

```sql
CREATE OR REPLACE PROCEDURE emp_sal_query
(
    p_empno IN emp.empno%TYPE)
IS
    TYPE emp_typ IS RECORD
    (       ename emp.ename%TYPE,
        job emp.job%TYPE,
        hiredate emp.hiredate%TYPE,
        sal emp.sal%TYPE,
        deptno emp.deptno%TYPE
    );
    r_emp emp_typ;
    v_avgsal emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
    INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
    FROM emp
    WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept # : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
    FROM emp
    WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal
        THEN
            DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department average of ' || v_avgsal);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Employee''s salary is less than the department average of ' || v_avgsal);
    END IF;
END;
```

This procedure uses a user-defined record type and record variable. It selects employee information from the `emp` table based on the `empno` parameter and calculates the average salary for the department. It then compares the employee's salary with the department's average salary and outputs the result.
Instead of specifying data type names, you can use the %TYPE attribute for the field data types in the record type definition.

The following is the output from executing this stored procedure:

```sql
EXEC emp_sal_query(7698);
Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

11.5.3 Working with transactions

There might be times when you want all updates to a database to occur successfully or for none to occur in case of any error. A set of database updates that occur successfully as a single unit or not at all is called a transaction.

A common example in banking is a funds transfer between two accounts. The two parts of the transaction are the withdrawal of funds from one account and the deposit of the funds in another account. Both parts of this transaction must occur for the bank’s books to balance. The deposit and withdrawal are one transaction.

You can create an SPL application that uses a style of transaction control compatible with Oracle databases if the following conditions are met:

- The edb_stmt_level_tx parameter is set to TRUE. This prevents the action of unconditionally rolling back all database updates in the BEGIN/END block if any exception occurs.
- The application isn’t running in autocommit mode. If autocommit mode is on, each successful database update is immediately committed and can’t be undone. The manner in which autocommit mode is turned on or off depends on the application.

The three main transaction commands are COMMIT, ROLLBACK, and PRAGM_AUTONOMOUS_TRANSACTION.

11.5.3.1 About transactions

A transaction begins when the first SQL command is encountered in the SPL program. All subsequent SQL commands are included as part of that transaction.

The transaction ends when one of the following occurs:

- An unhandled exception occurs. In this case, the effects of all database updates made during the transaction are rolled back, and the transaction is aborted.
- A COMMIT command is encountered. In this case, the effect of all database updates made during the transaction become permanent.
- A ROLLBACK command is encountered. In this case, the effects of all database updates made during the transaction are rolled back, and the transaction is aborted. If a new SQL command is encountered, a new transaction begins.
- Control returns to the calling application, such as Java or PSQI. In this case, the action of the application determines whether the transaction is committed or rolled back. The exception is when the transaction is in a block in which PRAGMA AUTONOMOUS_TRANSACTION was declared. In this case, the commitment or rollback of the transaction occurs independently of the calling program.

Note

Unlike Oracle, DDL commands such as CREATE TABLE don’t implicitly occur in their own transaction. Therefore, DDL commands don’t cause an immediate database commit as in Oracle, and you can roll back DDL commands just like DML commands.

A transaction can span one or more BEGIN/END blocks, or a single BEGIN/END block can contain one or more transactions.

11.5.3.2 COMMIT

The COMMIT command makes all database updates from the current transaction permanent and ends the current transaction.

```sql
COMMIT [ WORK ];
```

You can use the COMMIT command in anonymous blocks, stored procedures, or functions. In an SPL program, it can appear in the executable section and the exception section.
In this example, the third `INSERT` command in the anonymous block results in an error. The effect of the first two `INSERT` commands is retained as shown by the first `SELECT` command. Even after issuing a `ROLLBACK` command, the two rows remain in the table, as shown by the second `SELECT` command verifying that they were indeed committed.

```sql
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

BEGIN
    INSERT INTO dept VALUES (50, 'FINANCE', 'DALLAS');
    INSERT INTO dept VALUES (60, 'MARKETING', 'CHICAGO');
    COMMIT;
    INSERT INTO dept VALUES (78, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>50</td>
<td>FINANCE</td>
<td>DALLAS</td>
</tr>
<tr>
<td>60</td>
<td>MARKETING</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>

(6 rows)

```sql
ROLLBACK;
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>50</td>
<td>FINANCE</td>
<td>DALLAS</td>
</tr>
<tr>
<td>60</td>
<td>MARKETING</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>

(6 rows)

### 11.5.3 ROLLBACK

The `ROLLBACK` command undoes all database updates made during the current transaction and ends the current transaction.

```sql
ROLLBACK [ WORK ];
```

You can use the `ROLLBACK` command in anonymous blocks, stored procedures, or functions. In an SPL program, it can appear in the executable section and the exception section.

In this example, the exception section contains a `ROLLBACK` command. Even though the first two `INSERT` commands execute successfully, the third causes an exception that results in the rollback of all the `INSERT` commands in the anonymous block.

```sql
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;
BEGIN
```
<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>

This example uses both COMMIT and ROLLBACK. First, the following stored procedure is created. It inserts a new employee.

```
INSERT INTO dept VALUES (58, 'FINANCE', 'DALLAS');
INSERT INTO dept VALUES (68, 'MARKETING', 'CHICAGO');
INSERT INTO dept VALUES (78, 'HUMAN RESOURCES', 'CHICAGO');

END;

SELECT * FROM dept;

deptno | dname    | loc   
--------|----------|-------|
        |          |       |
--------|----------|-------|
10 | ACCOUNTING | NEW YORK |
20 | RESEARCH   | DALLAS  |
30 | SALES      | CHICAGO |
40 | OPERATIONS | BOSTON  |
```

This example uses both COMMIT and ROLLBACK. First, the following stored procedure is created. It inserts a new employee.

```
\set AUTOCOMMIT off
SET edb_stmt_level Tx TO ON;
CREATE OR REPLACE PROCEDURE emp_insert
(
  p_empno   IN emp.empno%TYPE,
p_ename    IN emp.ename%TYPE,
p_job      IN emp.job%TYPE,
p_mgr      IN emp.mgr%TYPE,
p_hiredate IN emp.hiredate%TYPE,
p_sal      IN emp.sal%TYPE,
p_comm     IN emp.comm%TYPE,
p_deptno   IN emp.deptno%TYPE
)
IS BEGIN
  INSERT INTO emp VALUES
  (
    p_empno,
p_ename,
p_job,
p_mgr,
p_hiredate,
p_sal,
p_comm,
p_deptno);
  DBMS_OUTPUT.PUT_LINE('Added employee...');
  DBMS_OUTPUT.PUT_LINE('Employee #: ' || p_empno);
  DBMS_OUTPUT.PUT_LINE('Name   : ' || p_ename);
  DBMS_OUTPUT.PUT_LINE('Job    : ' || p_job);
  DBMS_OUTPUT.PUT_LINE('Manager: ' || p_mgr);
  DBMS_OUTPUT.PUT_LINE('Hire Date: ' || p_hiredate);
  DBMS_OUTPUT.PUT_LINE('Salary : ' || p_sal);
  DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);
  DBMS_OUTPUT.PUT_LINE('Dept #: ' || p_deptno);
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
DBMS_OUTPUT.PUT_LINE('----------------------
');
END;

This procedure has no exception section. Any errors are propagated up to the calling program.

Then the following anonymous block runs. The COMMIT command is used after all calls to the emp_insert procedure and the ROLLBACK command in the exception section.

```
BEGIN
  emp_insert(9601, 'FARRELL', 'ANALYST', 7902, '03-MAR-08', 5000, NULL, 40);
  emp_insert(9602, 'TYLER', 'ANALYST', 7900, '25-JAN-08', 4800, NULL, 40);
  COMMIT;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
    ROLLBACK;
END;
```

Added employee...
Employee #: 9601
Name: FARRELL
Job: ANALYST
Manager: 7902
Hire Date: 03-MAR-08 00:00:00
Salary: 5000
Commission: 
Dept #: 40
----------------------

Added employee...
Employee #: 9602
Name: TYLER
Job: ANALYST
Manager: 7900
Hire Date: 25-JAN-08 00:00:00
Salary: 4800
Commission: 
Dept #: 40
----------------------

The following SELECT command shows that employees Farrell and Tyler were successfully added:

```
SELECT *
FROM emp
WHERE empno > 9600;
```

```
empno | ename | job   | mgr | hiredate       | sal   | comm | deptno
------|-------|-------|-----|----------------+-------+-------+-------
9601| FARRELL| ANALYST|7902| 03-MAR-08 00:00:00 | 5000.00 |       | 40
9602| TYLER | ANALYST|7900| 25-JAN-08 00:00:00 | 4800.00 |       | 40
(2 rows)
```

Next, execute the following anonymous block:

```
BEGIN
  emp_insert(9603, 'HARRISON', 'SALESMAN', 7902, '13-DEC-07', 5000, 3000, 20);
  emp_insert(9604, 'JARVIS', 'SALESMAN', 7902, '05-MAY-08', 4800, 4100, 11);
  COMMIT;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('An error occurred - roll back inserts');
    ROLLBACK;
END;
```

Added employee...
Employee #: 9603
Name: HARRISON
Job: SALESMAN
Manager: 7902
Hire Date: 13-DEC-07 00:00:00

The following SELECT command shows that employees Farrell and Tyler were successfully added:
Salary : 5000
Commission : 3000
Dept # : 20

SQLERRM: insert or update on table "emp" violates foreign key constraint "emp_ref_dept_fk"
An error occurred - roll back inserts

A SELECT command run against the table produces the following:

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9601</td>
<td>FARRELL</td>
<td>ANALYST</td>
<td>7902</td>
<td>03-MAR-08 00:00:00</td>
<td>5000.00</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>9602</td>
<td>TYLER</td>
<td>ANALYST</td>
<td>7900</td>
<td>25-JAN-08 00:00:00</td>
<td>4800.00</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

The ROLLBACK command in the exception section successfully undoes the insert of employee Harrison. Employees Farrell and Tyler are still in the table as their inserts were made permanent by the COMMIT command in the first anonymous block.

Note

Executing a COMMIT or ROLLBACK in a plpgsql procedure throws an error if an Oracle-style SPL procedure is on the runtime stack.

11.5.3.4 PRAGMA AUTONOMOUS_TRANSACTION

A stored procedural language (SPL) program can be declared as an autonomous transaction by specifying the following directive in the declaration section of the SPL block. An autonomous transaction is an independent transaction started by a calling program.

PRAGMA AUTONOMOUS_TRANSACTION;

A commit or rollback of SQL commands in the autonomous transaction has no effect on the commit or rollback in any transaction of the calling program. A commit or rollback in the calling program has no effect on the commit or rollback of SQL commands in the autonomous transaction.

Requirements and restrictions

The following SPL programs can include PRAGMA AUTONOMOUS_TRANSACTION:

- Standalone procedures and functions
- Anonymous blocks
- Procedures and functions declared as subprograms in packages and other calling procedures, functions, and anonymous blocks
- Triggers
- Object type methods

The following are issues and restrictions related to autonomous transactions:

- Each autonomous transaction consumes a connection slot for as long as it’s in progress. In some cases, this might mean that you need to raise the max_connections parameter in the postgresql.conf file.
- In most respects, an autonomous transaction behaves as if it were a completely separate session, but GUCs (settings established with SET) are a deliberate exception. Autonomous transactions absorb the surrounding values and can propagate values they commit to the outer transaction.
- Autonomous transactions can be nested, but there is a limit of 16 levels of autonomous transactions in a single session.
- Parallel query isn’t supported in autonomous transactions.
- The EDB Postgres Advanced Server implementation of autonomous transactions isn’t entirely compatible with Oracle databases. The EDB Postgres Advanced Server autonomous transaction doesn’t produce an error if there’s an uncommitted transaction at the end of an SPL block.

About the examples

The following set of examples use autonomous transactions. This first set of scenarios shows the default behavior when there are no autonomous transactions.

Before each scenario, the dept table is reset to the following initial values:

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------</td>
<td>-------</td>
<td>-----</td>
</tr>
</tbody>
</table>

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Scenario 1a: No autonomous transactions with only a final COMMIT

This first set of scenarios shows the insertion of three rows:

- Starting just after the initial BEGIN command of the transaction
- From an anonymous block in the starting transactions
- From a stored procedure executed from the anonymous block

The stored procedure is the following:

```sql
CREATE OR REPLACE PROCEDURE insert_dept_70
IS
    BEGIN
        INSERT INTO dept VALUES (70,'MARKETING','LOS ANGELES');
    END;
END;
```

The PSQL session is the following:

```sql
BEGIN;
    INSERT INTO dept VALUES (50,'HR','DENVER');
    BEGIN
        INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
        insert_dept_70;
    END;
COMMIT;
```

After the final commit, all three rows are inserted:

```sql
SELECT * FROM dept ORDER BY 1;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>50</td>
<td>HR</td>
<td>DENVER</td>
</tr>
<tr>
<td>60</td>
<td>FINANCE</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>70</td>
<td>MARKETING</td>
<td>LOS ANGELES</td>
</tr>
</tbody>
</table>

(7 rows)

Scenario 1b: No autonomous transactions but a final ROLLBACK

The next scenario shows that a final ROLLBACK command after all inserts results in the rollback of all three insertions:

```sql
BEGIN;
    INSERT INTO dept VALUES (50,'HR','DENVER');
    BEGIN
        INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
        insert_dept_70;
    END;
ROLLBACK;
```

```sql
SELECT * FROM dept ORDER BY 1;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>
Scenario 1c: No autonomous transactions, but anonymous block ROLLBACK

A **ROLLBACK** command given at the end of the anonymous block also eliminates all three prior insertions:

```
BEGIN;
INSERT INTO dept VALUES (50, 'HR', 'DENVER');
BEGIN
INSERT INTO dept VALUES (60, 'FINANCE', 'CHICAGO');
  insert_dept_70;
ROLLBACK;
END;
COMMIT;
SELECT * FROM dept ORDER BY 1;
```

```
<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>60</td>
<td>FINANCE</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>70</td>
<td>MARKETING</td>
<td>LOS ANGELES</td>
</tr>
</tbody>
</table>
```

(4 rows)

The next set of scenarios shows the effect of using autonomous transactions with **PRAGMA AUTONOMOUS_TRANSACTION** in various locations.

Scenario 2a: Autonomous transaction of anonymous block with COMMIT

The procedure remains as initially created:

```
CREATE OR REPLACE PROCEDURE insert_dept_70
IS
BEGIN
  INSERT INTO dept VALUES (70, 'MARKETING', 'LOS ANGELES');
END;
END;
```

```
BEGIN;
INSERT INTO dept VALUES (50, 'HR', 'DENVER');
DECLARE
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
INSERT INTO dept VALUES (60, 'FINANCE', 'CHICAGO');
  insert_dept_70;
COMMIT;
END;
ROLLBACK;
SELECT * FROM dept ORDER BY 1;
```

After the **ROLLBACK** at the end of the transaction, only the first row insertion at the beginning of the transaction is discarded. The other two row insertions in the anonymous block with **PRAGMA AUTONOMOUS_TRANSACTION** were independently committed.
Scenario 2b: Autonomous transaction anonymous block with COMMIT, including procedure with ROLLBACK but not an autonomous transaction procedure

This procedure has the **ROLLBACK** command at the end. However, the **PRAGMA ANONYMOUS_TRANSACTION** isn’t included in this procedure.

```
CREATE OR REPLACE PROCEDURE insert_dept_70
IS
  BEGIN
    INSERT INTO dept VALUES (78, 'MARKETING', 'LOS ANGELES');
    ROLLBACK;
  END;
END;
```

The rollback in the procedure removes the two rows inserted in the anonymous block (deptno 60 and 70) before the final **COMMIT** command in the anonymous block:

```
BEGIN;
  INSERT INTO dept VALUES (59, 'HR', 'DENVER');
DECLARE PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO dept VALUES (69, 'FINANCE', 'CHICAGO');
  insert_dept_70;
END;
COMMIT;
```

After the final commit at the end of the transaction, the only row inserted is the first one from the beginning of the transaction. Since the anonymous block is an autonomous transaction, the rollback in the enclosed procedure has no effect on the insertion that occurs before the anonymous block is executed.

```
SELECT * FROM dept ORDER by 1;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>50</td>
<td>HR</td>
<td>DENVER</td>
</tr>
</tbody>
</table>

(5 rows)

Scenario 2c: Autonomous transaction anonymous block with COMMIT, including procedure with ROLLBACK that is also an autonomous transaction procedure

The procedure with the **ROLLBACK** command at the end also has **PRAGMA ANONYMOUS_TRANSACTION** included. This isolates the effect of the **ROLLBACK** command in the procedure.

```
CREATE OR REPLACE PROCEDURE insert_dept_70
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO dept VALUES (78, 'MARKETING', 'LOS ANGELES');
  ROLLBACK;
END;
END;
```

The rollback in the procedure removes the row inserted by the procedure but not the other row inserted in the anonymous block:

```
BEGIN;
  INSERT INTO dept VALUES (59, 'HR', 'DENVER');
DECLARE PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO dept VALUES (69, 'FINANCE', 'CHICAGO');
  insert_dept_70;
END;
COMMIT;
```

After the final commit at the end of the transaction, the row inserted is the first one from the beginning of the transaction as well as the row inserted at the beginning of the anonymous block. The only insertion rolled back is the one in the procedure.

```
SELECT * FROM dept ORDER by 1;
```
The following examples show `PRAGMA AUTONOMOUS_TRANSACTION` in a couple of other SPL program types.

Autonomous transaction trigger

This example shows the effect of declaring a trigger with `PRAGMA AUTONOMOUS_TRANSACTION`.

The following table is created to log changes to the `emp` table:

```
CREATE TABLE empauditlog
(
  audit_date DATE,
  audit_user VARCHAR2(20),
  audit_desc VARCHAR2(20)
);
```

The trigger attached to the `emp` table that inserts these changes into the `empauditlog` table is the following. `PRAGMA AUTONOMOUS_TRANSACTION` is included in the declaration section.

```
CREATE OR REPLACE TRIGGER emp_audit_trig
  AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
  v_action VARCHAR2(20);
BEGIN
  IF INSERTING THEN
    v_action := 'Added employee(s)';
  ELSIF UPDATING THEN
    v_action := 'Updated employee(s)';
  ELSIF DELETING THEN
    v_action := 'Deleted employee(s)';
  END IF;
  INSERT INTO empauditlog VALUES (SYSDATE, USER, v_action);
END;
```

The following two inserts are made into the `emp` table in a transaction started by the `BEGIN` command:

```
BEGIN;
INSERT INTO emp VALUES (9001,'SMITH','ANALYST',7782,SYSDATE,NULL,NULL,10);
INSERT INTO emp VALUES (9002,'JONES','CLERK',7782,SYSDATE,NULL,NULL,10);
```

The following shows the two new rows in the `emp` table as well as the two entries in the `empauditlog` table:

```
SELECT * FROM emp WHERE empno > 9000;
```

The following shows two new rows in the `emp` table as well as the two entries in the `empauditlog` table:

```
SELECT TO_CHAR(AUDIT_DATE,'DD-MON-YY HH24:MI:SS') AS "audit date",
       audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;
```
But then the `ROLLBACK` command is given during this session. The `emp` table no longer contains the two rows, but the `empauditlog` table still contains its two entries. The trigger implicitly performed a commit, and `PRAGMA AUTONOMOUS_TRANSACTION` commits those changes independent from the rollback given in the calling transaction.

```
ROLLBACK;
```

```
SELECT * FROM emp WHERE empno > 9000;
```

```
empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 rows)
```

```
SELECT TO_CHAR(AUDIT_DATE, 'DD-MON-YY HH24:MI:SS') AS "audit date",
audit_user, audit_desc FROM empauditlog ORDER BY 1 ASC;
```

```
audit date |  audit_user  |    audit_desc
--------------------+--------------+-------------------
23-AUG-18 07:12:27 | enterprisedb | Added employee(s)
23-AUG-18 07:12:27 | enterprisedb | Added employee(s)
(2 rows)
```

Autonomous transaction object type method

This example shows the effect of declaring an object method with `PRAGMA AUTONOMOUS_TRANSACTION`.

The following object type and object type body are created. The member procedure in the object type body contains the `PRAGMA AUTONOMOUS_TRANSACTION` in the declaration section along with `COMMIT` at the end of the procedure.

```
CREATE OR REPLACE TYPE insert_dept_typ AS OBJECT
(
  deptno NUMBER(2),
  dname VARCHAR2(14),
  loc VARCHAR2(13),
  MEMBER PROCEDURE
insert_dept );

CREATE OR REPLACE TYPE BODY insert_dept_typ AS
MEMBER PROCEDURE
insert_dept
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO dept VALUES (SELF.deptno,SELF.dname,SELF.loc);
  COMMIT;
END;
END;
```

In the following anonymous block, an insert is performed into the `dept` table, followed by invoking the `insert_dept` method of the object and ending with a `ROLLBACK` command in the anonymous block.

```
BEGIN;
DECLARE
  v_dept INSERT_DEPT_TYP :=
insert_dept_typ(60,'FINANCE','CHICAGO');
BEGIN
  INSERT INTO dept VALUES (50,'HR','DENVER');
  v_dept.insert_dept;
  ROLLBACK;
END;
END;
```
Since `insert_dept` was declared as an autonomous transaction, its insert of department number 60 remains in the table, but the rollback removes the insertion of department 50:

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>60</td>
<td>FINANCE</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>

(5 rows)

11.5.4 Using dynamic SQL

Dynamic SQL is a technique that lets you execute SQL commands that aren't known until the commands are about to be executed. In static SQL commands, the full command (with the exception of variables) must be known and coded into the program before the program can begin to execute. Using dynamic SQL, the executed SQL can change during program runtime.

In addition, dynamic SQL is the only method by which data definition commands, such as `CREATE TABLE`, can be executed from an SPL program.

However, the runtime performance of dynamic SQL is slower than static SQL.

Syntax

The `EXECUTE IMMEDIATE` command is used to run SQL commands dynamically:

```sql
EXECUTE IMMEDIATE '<sql_expression>;' 
[ INTO { <variable> [, ...] | <record> } ] 
[ USING { | <bind_type> <bind_argument> } [, ...} ]
```

Where:

- `sql_expression` is a string expression containing the SQL command to dynamically execute.
- `variable` receives the output of the result set, typically from a `SELECT` command, created as a result of executing the SQL command in `sql_expression`. The number, order, and type of variables must match the number and order and be type-compatible with the fields of the result set.

When using the `USING` clause, the value of `expression` is passed to a placeholder. Placeholders appear embedded in the SQL command in `sql_expression` where you can use variables. Placeholders are denoted by an identifier with a colon (:) prefix, for example, `:name`. The number and order of the evaluated expressions must match the number, order, and the placeholders in `sql_expression`. The resulting data types must also be compatible with the placeholders. You don't declare placeholders in the SPL program. They appear only in `sql_expression`.

Currently `bind_type` is ignored, and `bind_argument` is treated as `IN OUT`.

Alternatively, a `record` can be specified as long as the record's fields match the number and order and are type-compatible with the result set.

When using the `INTO` clause, exactly one row must be returned in the result set. Otherwise an exception occurs. When using the `USING` clause, the value of `expression` is passed to a placeholder. Placeholders appear embedded in the SQL command in `sql_expression` where variables can be used. Placeholders are denoted by an identifier with a colon (:) prefix, such as `:name`. The number, order, and resultant data types of the evaluated expressions must match the number and order and be type-compatible with the placeholders in `sql_expression`.

Placeholders aren't declared anywhere in the SPL program. They appear only in `sql_expression`.

Currently all options for `bind_type` are ignored and `bind_argument` is treated as `IN OUT`.

Example: SQL commands as string literals

This example shows basic dynamic SQL commands as string literals:

```sql
DECLARE v_sql VARCHAR2(50);
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE job (jobno NUMBER(3),
  jname VARCHAR2(9));
  v_sql := 'INSERT INTO job VALUES (100, 'ANALYST'))';
END;
```

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
This example uses the `USING` clause to pass values to placeholders in the SQL string:

```
EXECUTE IMMEDIATE
v_sql;
 v_sql := 'INSERT INTO job VALUES (200, ' 'CLERK')';
 EXECUTE IMMEDIATE
v_sql;
END;
```

This example uses the `USING` clause to pass values to placeholders in the SQL string:

```
DECLARE
 v_sql VARCHAR2(50) := 'INSERT INTO job VALUES ' || '(':p_jobno, :p_jname')';
 v_jobno job.jobno%TYPE;
 v_jname job.jname%TYPE;
BEGIN
 v_jobno := 300;
 v_jname := 'MANAGER';
 EXECUTE IMMEDIATE v_sql USING v_jobno,
 v_jname;
 v_jobno := 400;
 v_jname := 'SALESMAN';
 EXECUTE IMMEDIATE v_sql USING v_jobno,
 v_jname;
 v_jobno := 500;
 v_jname := 'PRESIDENT';
 EXECUTE IMMEDIATE v_sql USING v_jobno,
 v_jname;
END;
```

This example shows both the `INTO` and `USING` clauses. The last execution of the `SELECT` command returns the results into a record instead of individual variables.

```
DECLARE
 v_sql VARCHAR2(60);
 v_jobno job.jobno%TYPE;
 v_jname job.jname%TYPE;
 r_job job%ROWTYPE;
BEGIN
 DBMS_OUTPUT.PUT_LINE('JOBNO JNAME');
 DBMS_OUTPUT.PUT_LINE('----- ------- ');
 v_sql := 'SELECT jobno, jname FROM job WHERE jobno = :p_jobno';
 EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 108;
 DBMS_OUTPUT.PUT_LINE(v_jobno || ' ' || v_jname);
 EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 208;
 DBMS_OUTPUT.PUT_LINE(v_jobno || ' ' || v_jname);
 EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 308;
 DBMS_OUTPUT.PUT_LINE(v_jobno || ' ' || v_jname);
 EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 408;
 DBMS_OUTPUT.PUT_LINE(v_jobno || ' ' || v_jname);
 EXECUTE IMMEDIATE v_sql INTO r_job USING 500;
 DBMS_OUTPUT.PUT_LINE(r_job.jobno || ' ' || r_job.jname);
END;
```

The following is the output from this anonymous block:

```
<table>
<thead>
<tr>
<th>JOBNO</th>
<th>JNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>ANALYST</td>
</tr>
<tr>
<td>200</td>
<td>CLERK</td>
</tr>
<tr>
<td>300</td>
<td>MANAGER</td>
</tr>
<tr>
<td>408</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>500</td>
<td>PRESIDENT</td>
</tr>
</tbody>
</table>
```

You can use the `BULK COLLECT` clause to assemble the result set from an `EXECUTE IMMEDIATE` statement into a named collection. See Using the `BULK COLLECT` clause, `EXECUTE IMMEDIATE BULK COLLECT` for more information.
11.5.5 Working with static cursors

Rather than executing a whole query at once, you can set up a cursor that encapsulates the query and then read the query result set one row at a time. This approach allows you to create SPL program logic that retrieves a row from the result set, does some processing on the data in that row, and then retrieves the next row and repeats the process.

Cursors are most often used in the context of a `FOR` or `WHILE` loop. Include a conditional test in the SPL logic that detects when the end of the result set was reached so the program can exit the loop.

11.5.5.1 Declaring a cursor

Before you can use a cursor, you must first declare it in the declaration section of the SPL program. A cursor declaration appears as follows:

```sql
CURSOR <name> IS <query>;
```

Where:
- `<name>` is an identifier used to reference the cursor and its result set later in the program.
- `<query>` is a SQL `SELECT` command that determines the result set retrievable by the cursor.

**Note**
An extension of this syntax allows the use of parameters. For details, see Parameterized cursors.

This example shows some cursor declarations:

```sql
CREATE OR REPLACE PROCEDURE cursor_example
IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
  CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;
  CURSOR emp_cur_3 IS SELECT empno, ename FROM emp
  WHERE deptno = 10
    ORDER BY empno;
BEGIN
  ...
END;
```

11.5.5.2 Opening a cursor

Before you can use a cursor to retrieve rows, you must open it using the `OPEN` statement.

```sql
OPEN <name>;
```

`<name>` is the identifier of a cursor that was previously declared in the declaration section of the SPL program. Don't execute the `OPEN` statement on a cursor that is already open.

This example shows an `OPEN` statement with its corresponding cursor declaration:

```sql
CREATE OR REPLACE PROCEDURE cursor_example
IS
  CURSOR emp_cur_3 IS SELECT empno, ename FROM emp
  WHERE deptno = 10
    ORDER BY empno;
BEGIN
  OPEN emp_cur_3;
  ...
END;
```

11.5.5.3 Fetching rows from a cursor

Once a cursor is open, you can retrieve rows from the cursor’s result set by using the `FETCH` statement.
Syntax

`FETCH <name> INTO { <record> | <variable> [, <variable_2> ]...`;

Where:

- `<name>` is the identifier of a previously opened cursor.
- `<record>` is the identifier of a previously defined record (for example, using `table%ROWTYPE`).

`variable, variable_2...` are SPL variables that receive the field data from the fetched row. The fields in `<record>` or `variable, variable_2...` must match in number and order the fields returned in the `SELECT` list of the query given in the cursor declaration. The data types of the fields in the `SELECT` list must match or be implicitly convertible to the data types of the fields in `<record>` or the data types of `variable, variable_2...`

Note

A variation of `FETCH INTO` using the `BULK COLLECT` clause can return multiple rows at a time into a collection. See [Using the BULK COLLECT clause](#) for more information on using the `BULK COLLECT` clause with the `FETCH INTO` statement.

Example

The following shows the `FETCH` statement:

```sql
CREATE OR REPLACE PROCEDURE cursor_example IS
  v_empno     NUMBER(4);
  v_ename     VARCHAR2(10);
  CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10 ORDER BY empno;
BEGIN
  OPEN emp_cur_3;
  FETCH emp_cur_3 INTO v_empno, v_ename;
  ...
END;
```

Instead of explicitly declaring the data type of a target variable, you can use `%TYPE` instead. In this way, if the data type of the database column changes, the target variable declaration in the SPL program doesn't have to change. `%TYPE` picks up the new data type of the specified column.

```sql
CREATE OR REPLACE PROCEDURE cursor_example IS
  v_empno     emp.empno%TYPE;
  v_ename     emp.ename%TYPE;
  CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10 ORDER BY empno;
BEGIN
  OPEN emp_cur_3;
  FETCH emp_cur_3 INTO v_empno, v_ename;
  ...
END;
```

If all the columns in a table are retrieved in the order defined in the table, you can use `%ROWTYPE` to define a record into which the `FETCH` statement places the retrieved data. You can then access each field in the record using dot notation:

```sql
CREATE OR REPLACE PROCEDURE cursor_example IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  OPEN emp_cur_1;
  FETCH emp_cur_1 INTO v_emp_rec;
  DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
  DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
  ...
END;
```
11.5.5.4 Closing a cursor

Once all the desired rows are retrieved from the cursor result set, close the cursor. After you close the cursor, you can no longer access the result set.

The `CLOSE` statement appears as follows:

```
CLOSE <name>;
```

`<name>` is the identifier of a cursor that's currently open. After you close a cursor, don't close it again. However, after you close the cursor, you can use the `OPEN` statement again on the closed cursor and rebuild the query result set. After that, the `FETCH` statement can then retrieve the rows of the new result set.

This example uses the `CLOSE` statement:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
  v_emp_rec emp%ROWTYPE;
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  OPEN emp_cur_1;
  FETCH emp_cur_1 INTO v_emp_rec;
  DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
  DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
  CLOSE emp_cur_1;
END;
```

This procedure produces the following output. Employee number 7369, SMITH is the first row of the result set.

```
EXEC cursor_example;
```

Employee Number: 7369
Employee Name  : SMITH

11.5.5.5 Using %ROWTYPE with cursors

Using the `%ROWTYPE` attribute, you can define a record that contains fields corresponding to all columns fetched from a cursor or cursor variable. Each field takes on the data type of its corresponding column. The `%ROWTYPE` attribute is prefixed by a cursor name or cursor variable name.

```
<record> <cursor>%ROWTYPE;
```

`<record>` is an identifier assigned to the record. `<cursor>` is an explicitly declared cursor in the current scope.

This example shows how you can use a cursor with `%ROWTYPE` to get information about which employee works in which department:

```
CREATE OR REPLACE PROCEDURE emp_info
IS
  CURSOR empcur IS SELECT ename, deptno FROM emp;
  myvar empcur%ROWTYPE;
BEGIN
  OPEN empcur;
  LOOP
    FETCH empcur INTO myvar;
    EXIT WHEN empcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( myvar.ename || ' works in department ' || myvar.deptno );
  END LOOP;
  CLOSE empcur;
END;
```

The following is the output from this procedure:

```
EXEC emp_info;
```
11.5.5.6 Cursor attributes

Each cursor has a set of attributes associated with it that allows the program to test its state. These attributes are \%ISOPEN, \%FOUND, \%NOTFOUND, and \%ROWCOUNT.

11.5.5.6.1 \%ISOPEN

Use the \%ISOPEN attribute to test whether a cursor is open.

\<cursor_name\\>\%ISOPEN

\<cursor_name\\> is the name of the cursor for which a BOOLEAN data type of TRUE is returned if the cursor is open, FALSE otherwise.

This example uses \%ISOPEN:

```sql
CREATE OR REPLACE PROCEDURE cursor_example IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
  BEGIN
    IF emp_cur_1\%ISOPEN THEN
      NULL;
    ELSE
      OPEN emp_cur_1;
      FETCH emp_cur_1 INTO ...
    END IF;
  END;
END;
```

11.5.5.6.2 \%FOUND

The \%FOUND attribute tests whether a row is retrieved from the result set of the specified cursor after a FETCH on the cursor.

\<cursor_name\\>\%FOUND

\<cursor_name\\> is the name of the cursor for which a BOOLEAN data type of TRUE is returned if a row is retrieved from the result set of the cursor after a FETCH.

After the last row of the result set is fetched, the next FETCH results in \%FOUND returning FALSE. FALSE is also returned after the first FETCH if the result set has no rows to begin with.

Referencing \%FOUND on a cursor before it's opened or after it's closed results in an INVALID_CURSOR exception.

\%FOUND returns null if it's referenced when the cursor is open but before the first FETCH.

This example uses \%FOUND:

```sql
CREATE OR REPLACE PROCEDURE cursor_example
```
The following is the output from this example:

```
EMPNO ENAME
----- -----
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
7839 ADAMS
7844 TURNER
7876 MILLER
7900 JAMES
7902 FORD
7934 MILLER
```

11.5.5.6.3 %NOTFOUND

The %NOTFOUND attribute is the logical opposite of %FOUND.

```sql
<cursor_name>%NOTFOUND
```

cursor_name is the name of the cursor for which a BOOLEAN data type of FALSE is returned if a row is retrieved from the result set of the cursor after a FETCH.

After the last row of the result set is fetched, the next FETCH results in %NOTFOUND returning TRUE. TRUE is also returned after the first FETCH if the result set has no rows to begin with.

Referencing %NOTFOUND on a cursor before it's opened or after it's closed results in an INVALID_CURSOR exception.

%NOTFOUND returns null if it's referenced when the cursor is open but before the first FETCH.

This example uses %NOTFOUND:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
  v_emp_rec emp%ROWTYPE;
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  OPEN emp_cur_1;
  DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
  DBMS_OUTPUT.PUT_LINE('----- ------');
  FETCH emp_cur_1 INTO v_emp_rec;
  WHILE emp_cur_1%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || ' ' || v_emp_rec.ename);
    FETCH emp_cur_1 INTO v_emp_rec;
  END LOOP;
  CLOSE emp_cur_1;
END;
```

EXEC cursor_example;

```
EMPNO ENAME
----- ------
7369 SMITH
7499 ALLEN
7521 WARD
7566 JONES
7654 MARTIN
7698 BLAKE
7782 CLARK
7788 SCOTT
7839 ADAMS
7844 TURNER
7876 MILLER
7900 JAMES
7902 FORD
7934 MILLER
```
The following is the output from this example:

EXEC cursor_example;

EMPNO    ENAME
-----    -----
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER

11.5.5.6.4 %ROWCOUNT

The %ROWCOUNT attribute returns an integer showing the number of rows fetched so far from the specified cursor.

<cursor_name>%ROWCOUNT

cursor_name is the name of the cursor for which %ROWCOUNT returns the number of rows retrieved thus far. After the last row is retrieved, %ROWCOUNT remains set to the total number of rows returned until the cursor is closed. At that point, %ROWCOUNT throws an INVALID_CURSOR exception if referenced.

Referencing %ROWCOUNT on a cursor before it's opened or after it's closed results in an INVALID_CURSOR exception.

%ROWCOUNT returns 0 if it's referenced when the cursor is open but before the first FETCH. %ROWCOUNT also returns 0 after the first FETCH when the result set has no rows to begin with.

This example uses %ROWCOUNT:

CREATE OR REPLACE PROCEDURE cursor_example
IS
  v_emp_rec emp%ROWTYPE;
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  OPEN emp_cur_1;
  DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  LOOP
    FETCH emp_cur_1 INTO v_emp_rec;
    EXIT WHEN emp_cur_1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || ' ' || v_emp_rec.ename);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('************************');
  DBMS_OUTPUT.PUT_LINE(emp_cur_1%ROWCOUNT || ' rows were retrieved');
  CLOSE emp_cur_1;
END;

This procedure prints the total number of rows retrieved at the end of the employee list as follows:

EXEC cursor_example;

EMPNO    ENAME
-----    -----
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
11.5.5.6 Summary of cursor states and attributes

The table summarizes the possible cursor states and the values returned by the cursor attributes.

<table>
<thead>
<tr>
<th>Cursor state</th>
<th>%ISOPEN</th>
<th>%FOUND</th>
<th>%NOTFOUND</th>
<th>%ROWCOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before OPEN</td>
<td>False</td>
<td>INVALID_CURSOR</td>
<td>exception</td>
<td>INVALID_CURSOR</td>
</tr>
<tr>
<td>After OPEN &amp; before 1st FETCH</td>
<td>True</td>
<td>Null</td>
<td>Null</td>
<td>0</td>
</tr>
<tr>
<td>After 1st successful FETCH</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>1</td>
</tr>
<tr>
<td>After nth successful FETCH (last row)</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>n</td>
</tr>
<tr>
<td>After n+1st FETCH (after last row)</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>n</td>
</tr>
</tbody>
</table>

11.5.5.7 Using a cursor FOR loop

The programming logic required to process the result set of a cursor usually includes a statement to open the cursor, a loop construct to retrieve each row of the result set, a test for the end of the result set, and a statement to close the cursor. The cursor FOR loop is a loop construct that eliminates the need to individually code these statements.

The cursor FOR loop opens a previously declared cursor, fetches all rows in the cursor result set, and then closes the cursor.

The syntax for creating a cursor FOR loop is as follows:

```
FOR <record> IN <cursor>
LOOP
  <statements>
END LOOP;
```

Where:

- `record` is an identifier assigned to an implicitly declared record with definition `cursor%ROWTYPE`.
- `cursor` is the name of a previously declared cursor.
- `statements` are one or more SPL statements. There must be at least one statement.

This example uses a cursor FOR loop:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  DBMS_OUTPUT.PUT_LINE('EMPNO | ENAME');
  DBMS_OUTPUT.PUT_LINE('----- | ------- ');
  FOR v_emp_rec IN emp_cur_1 LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || ' | ' || v_emp_rec.ename);
  END LOOP;
END;
```

The following is the output from this procedure:

```
EXEC cursor_example;
```
### 11.5.5.8 Declaring parameterized cursors

You can declare a static cursor that accepts parameters and can pass values for those parameters when opening that cursor. This example creates a parameterized cursor that displays the name and salary of all employees from the `emp` table that have a salary less than a specified value. This information is passed as a parameter.

```sql
DECLARE
  my_record emp%ROWTYPE;
  CURSOR c1 (max_wage NUMBER) IS
    SELECT * FROM emp WHERE sal < max_wage;
BEGIN
  OPEN c1 (2000);
  LOOP
    FETCH c1 INTO my_record;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = ' || my_record.sal);
  END LOOP;
  CLOSE c1;
END;
```

For example, if you pass the value 2000 as `max_wage`, then you see only the name and salary of all employees that have a salary less than 2000. The following is the result of the above query:

Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00

### 11.5.6 Working with REF CURSOR and cursor variables

The `REF CURSOR` provides greater flexibility than static cursors.

#### 11.5.6.1 REF CURSOR overview

A cursor variable is a cursor that contains a pointer to a query result set. The result set is determined by executing the `OPEN FOR` statement using the cursor variable.

A cursor variable isn’t tied to a single query like a static cursor is. You can open the same cursor variable a number of times with `OPEN FOR` statements containing different queries. Each time, a new
result set is created from that query and made available by way of the cursor variable.

You can pass `REF CURSOR` types as parameters to or from stored procedures and functions. The return type of a function can also be a `REF CURSOR` type. This ability lets you modularize the operations on a cursor into separate programs by passing a cursor variable between programs.

### 11.5.6.2 Declaring a cursor variable

SPL supports two ways of declaring a cursor variable:

- Using the `SYS_REFCURSOR` built-in data type. `SYS_REFCURSOR` is a `REF CURSOR` type that allows any result set to be associated with it. This is known as a weakly-typed `REF CURSOR`.
- Creating a type of `REF CURSOR` and then declaring a variable of that type.

Only the declaration of `SYS_REFCURSOR` and user-defined `REF CURSOR` variables differ. The remaining usage, such as opening the cursor, selecting into the cursor, and closing the cursor, is the same for both the cursor types. The examples primarily make use of the `SYS_REFCURSOR` cursors. To make the examples work for a user-defined `REF CURSOR`, change the declaration section.

#### Note

A strongly-typed `REF CURSOR` requires the result set to conform to a declared number and order of fields with compatible data types. It can also optionally return a result set.

### 11.5.6.2.1 Declaring a SYS_REFCURSOR cursor variable

The following is the syntax for declaring a `SYS_REFCURSOR` cursor variable:

```
<name> SYS_REFCURSOR;
```

Where `name` is an identifier assigned to the cursor variable.

This example shows a `SYS_REFCURSOR` variable declaration:

```
DECLARE
  emp_refcur SYS_REFCURSOR;
```

### 11.5.6.2.2 Declaring a user-defined REF CURSOR type variable

You must perform two distinct declaration steps to use a user-defined `REF CURSOR` variable:

- Create a referenced cursor `TYPE`.
- Declare the actual cursor variable based on that `TYPE`.

The syntax for creating a user-defined `REF CURSOR` type:

```
TYPE <cursor_type_name> IS REF CURSOR [RETURN <return_type>];
```

This example shows a cursor variable declaration:

```
DECLARE
  TYPE emp_cur_type IS REF CURSOR RETURN empROWTYPE;
  my_rec emp_cur_type;
```

### 11.5.6.3 Opening a cursor variable

Once you declare a cursor variable, you must open it with an associated `SELECT` command. The `OPEN FOR` statement specifies the `SELECT` command to use to create the result set:

```
OPEN <name> FOR query;
```
Where:

- `name` is the identifier of a previously declared cursor variable.
- `query` is a `SELECT` command that determines the result set when the statement is executed.

The value of the cursor variable after the `OPEN FOR` statement is executed identifies the result set.

This example shows a result set that's a list of employee numbers and names from a selected department. You can use a variable or parameter in the `SELECT` command anywhere an expression can normally appear. In this case, a parameter is used in the equality test for department number.

```sql
CREATE OR REPLACE PROCEDURE emp_by_dept
  (
    p_deptno emp.deptno%TYPE
  )
IS
  emp_refcur SYS_REFCURSOR;
BEGIN
  OPEN emp_refcur FOR
    SELECT empno, ename
    FROM emp
    WHERE deptno = p_deptno;
  LOOP
    FETCH emp_refcur INTO v_empno, v_ename;
    EXIT WHEN emp_refcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
  END LOOP;
END;
```

### 11.5.6.4 Fetching rows From a cursor variable

After you open a cursor variable, you can retrieve rows from the result set using the `FETCH` statement. For details, see [Fetching rows from a cursor](#).

This example uses a `FETCH` statement to cause the result set to be returned into two variables and then displayed. You can also use the cursor attributes used to determine cursor state of static cursors with cursor variables. For details, see [Cursor attributes](#).

```sql
CREATE OR REPLACE PROCEDURE emp_by_dept
  (
    p_deptno emp.deptno%TYPE
  )
IS
  emp_refcur SYS_REFCURSOR;
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
BEGIN
  OPEN emp_refcur FOR
    SELECT empno, ename
    FROM emp
    WHERE deptno = p_deptno;
  DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
  DBMS_OUTPUT.PUT_LINE('----- -------');
  LOOP
    FETCH emp_refcur INTO v_empno, v_ename;
    EXIT WHEN emp_refcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
  END LOOP;
END;
```

### 11.5.6.5 Closing a cursor variable

Use the `CLOSE` statement described in [Closing a cursor](#) to release the result set.

**Note**

Unlike static cursors, you don't have to close a cursor variable before you can reopen it. When you reopen it, the result set from the previous open is lost.

This example includes the `CLOSE` statement:

```sql
CREATE OR REPLACE PROCEDURE emp_by_dept
  (
    p_deptno emp.deptno%TYPE
  )
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The following is the output from this procedure:

```sql
EXEC emp_by_dept(20)
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
</tr>
<tr>
<td>7982</td>
<td>FORD</td>
</tr>
</tbody>
</table>

### 11.5.6.6 Usage restrictions

The following are restrictions on cursor variable usage:

- You can't use comparison operators to test cursor variables for equality, inequality, null, or not null.
- You can't assign null to a cursor variable.
- You can't store the value of a cursor variable in a database column.
- Static cursors and cursor variables aren't interchangeable. For example, you can't use a static cursor in an `OPEN` `FOR` statement.

The table shows the permitted parameter modes for a cursor variable used as a procedure or function parameter. This use depends on the operations on the cursor variable in the procedure or function.

<table>
<thead>
<tr>
<th>Operation</th>
<th>IN</th>
<th>IN OUT</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>No</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>FETCH</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

For example, if a procedure performs all three operations—`OPEN FOR`, `FETCH`, and `CLOSE`—on a cursor variable declared as the procedure's formal parameter, then that parameter must be declared with `IN OUT` mode.

### 11.5.6.7 Cursor variable examples

The examples that follow show cursor variable usage.

#### 11.5.6.7.1 Returning a REF CURSOR from a function

This example opens the cursor variable with a query that selects employees with a given job. The cursor variable is specified in this function's `RETURN` statement, which makes the result set available to the caller of the function.

```sql
CREATE OR REPLACE FUNCTION emp_by_job (p_job VARCHAR2) IS
  emp_refcur SYS_REFCURSOR;
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
BEGIN
  OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
  DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
  LOOP
    FETCH emp_refcur INTO v_empno, v_ename;
    EXIT WHEN emp_refcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || v_ename);
  END LOOP;
  CLOSE emp_refcur;
END;
```

The table shows the permitted parameter modes for a cursor variable used as a procedure or function parameter. This use depends on the operations on the cursor variable in the procedure or function.

<table>
<thead>
<tr>
<th>Operation</th>
<th>IN</th>
<th>IN OUT</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>No</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>FETCH</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
This function is invoked in the following anonymous block by assigning the function’s return value to a cursor variable declared in the anonymous block’s declaration section. The result set is fetched using this cursor variable, and then it is closed.

```sql
DECLARE
  emp_refcur SYS_REFCURSOR;
BEGIN
  OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
  RETURN emp_refcur;
END;

RETURN SYS_REFCURSOR
IS
  emp_refcur SYS_REFCURSOR;
BEGIN
  OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
  RETURN emp_refcur;
END;
```

This function is invoked in the following anonymous block by assigning the function’s return value to a cursor variable declared in the anonymous block’s declaration section. The result set is fetched using this cursor variable, and then it is closed.

### Modularizing cursor operations

This example shows how you can modularize the various operations on cursor variables into separate programs.

The following procedure opens the given cursor variable with a `SELECT` command that retrieves all rows:

```sql
CREATE OR REPLACE PROCEDURE open_all_emp
  (p_emp_refcur IN OUT SYS_REFCURSOR)
IS
BEGIN
  OPEN p_emp_refcur FOR SELECT empno, ename FROM emp;
END;
```

This variation opens the given cursor variable with a `SELECT` command that retrieves all rows of a given department:

```sql
CREATE OR REPLACE PROCEDURE open_emp_by_dept
  (p_emp_refcur IN OUT SYS_REFCURSOR,
   p_deptno emp.deptno%TYPE)
```
This variation opens the given cursor variable with a `SELECT` command that retrieves all rows but from a different table. The function’s return value is the opened cursor variable.

```sql
CREATE OR REPLACE FUNCTION open_dept
(  p_dept_refcur    IN OUT SYS_REFCURSOR
)  RETURN SYS_REFCURSOR
IS
  v_dept_refcur SYS_REFCURSOR;
BEGIN
  v_dept_refcur :=  p_dept_refcur;
  OPEN v_dept_refcur FOR SELECT deptno, dname FROM dept;
  RETURN v_dept_refcur;
END;
```

This procedure fetches and displays a cursor variable result set consisting of employee number and name:

```sql
CREATE OR REPLACE PROCEDURE fetch_emp
(  p_emp_refcur    IN OUT SYS_REFCURSOR
)  IS
  v_empno         emp.empno%TYPE;
  v_ename         emp.ename%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
  DBMS_OUTPUT.PUT_LINE('----- -------');
  LOOP
    FETCH p_emp_refcur INTO v_empno, v_ename;
    EXIT WHEN p_emp_refcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || v_ename);
  END LOOP;
END;
```

This procedure fetches and displays a cursor variable result set consisting of department number and name:

```sql
CREATE OR REPLACE PROCEDURE fetch_dept
(  p_dept_refcur    IN SYS_REFCURSOR
)  IS
  v_deptno         dept.deptno%TYPE;
  v_dname         dept.dname%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('DEPT DNAME');
  DBMS_OUTPUT.PUT_LINE('---- --------');
  LOOP
    FETCH p_dept_refcur INTO v_deptno, v_dname;
    EXIT WHEN p_dept_refcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_deptno || ' ' || v_dname);
  END LOOP;
END;
```

This procedure closes the given cursor variable:

```sql
CREATE OR REPLACE PROCEDURE close_refcur
(  p_refcur    IN OUT SYS_REFCURSOR
)  IS
BEGIN
  CLOSE p_refcur;
END;
```
This anonymous block executes all the previous programs:

```sql
DECLARE
    gen_refcur SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('ALL EMPLOYEES');
        open_all_emp(gen_refcur);
        fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('********************');

    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #10');
        open_emp_by_dept(gen_refcur, 10);
        fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('********************');

    DBMS_OUTPUT.PUT_LINE('DEPARTMENTS');
    fetch_dept(open_dept(gen_refcur));
    DBMS_OUTPUT.PUT_LINE('******************');
    close_refcur(gen_refcur);
END;
```

The following is the output from the anonymous block:

```
__OUTPUT__
ALL EMPLOYEES
  EMPNO  ENAME
--------
  7369  SMITH
  7499  ALLEN
  7521  WARD
  7566  JONES
  7654  MARTIN
  7698  BLAKE
  7782  CLARK
  7788  SCOTT
  7839  KING
  7844  TURNER
  7876  ADAMS
  7900  JAMES
  7902  FORD
  7934  MILLER
  7934  MILLER

****************
EMPLOYEES IN DEPT #10
  EMPNO  ENAME
--------
  7782  CLARK
  7839  KING
  7934  MILLER

****************
DEPARTMENTS
  DEPT  DNAME
--------
  10  ACCOUNTING
  20  RESEARCH
  30  SALES
  40  OPERATIONS
  40  OPERATIONS
```

### 11.5.6.8 Using dynamic queries with REF CURSOR

EDB Postgres Advanced Server supports dynamic queries by way of the `OPEN FOR USING` statement. A string literal or string variable is supplied in the `OPEN FOR USING` statement to the
**SELECT** command:

```sql
OPEN <name> FOR <dynamic_string>
[ USING <bind_arg> [, <bind_arg_2> ] ]
...;
```

Where:

- `<name>` is the identifier of a previously declared cursor variable.
- `<dynamic_string>` is a string literal or string variable containing a **SELECT** command without the terminating semi-colon.
- `<bind_arg>`, `<bind_arg_2>`... are bind arguments that pass variables to corresponding placeholders in the **SELECT** command when the cursor variable is opened. The placeholders are identifiers prefixed by a colon character.

This example shows a dynamic query using a string literal:

```sql
CREATE OR REPLACE PROCEDURE dept_query
IS
  emp_refcur SYS_REFCURSOR;
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
BEGIN
  OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30'
  || ' AND sal >= 1500';
  DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  LOOP
    FETCH emp_refcur INTO v_empno, v_ename;
    EXIT WHEN emp_refcur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
  END LOOP;
  CLOSE emp_refcur;
END;
```

The following is the output from this procedure:

```sql
dept_query;
EMPNO  ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

This example query uses bind arguments to pass the query parameters:

```sql
CREATE OR REPLACE PROCEDURE dept_query
(p_deptno emp.deptno%TYPE,
 p_sal emp.sal%TYPE)
IS
  emp_refcur SYS_REFCURSOR;
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
BEGIN
  OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = :dept'
  || ' AND sal >= :sal' USING p_deptno,
  p_sal;
  DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  LOOP
    FETCH emp_refcur INTO v_empno, v_ename;
    EXIT WHEN emp_refcur%NOTFOUND;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Finally, a string variable is used to pass the `SELECT`, providing the most flexibility:

```sql
CREATE OR REPLACE PROCEDURE dept_query
(
    p_deptno emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    p_query_string VARCHAR2(100);
BEGIN
    p_query_string := 'SELECT empno, ename FROM emp WHERE ' ||
                     'deptno = :dept AND sal >= :sal';
    OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
    DBMS_OUTPUT.PUT_LINE('----- -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
EXEC dept_query(20, 1500);
```

11.5.7 Working with collection types

A collection is a set of ordered data items with the same data type. Generally, the data item is a scalar field. It can also be a user-defined type such as a record type or an object type. In this case, the structure and the data types that make up each field of the user-defined type must be the same for each element in the set. Reference each data item in the set by using subscript notation inside a pair of parentheses.

11.5.7.1 About collection types

The most commonly known type of collection is an array. In EDB Postgres Advanced Server, the supported collection types are:

- Associative arrays (formerly called index-by-tables in Oracle)
- Nested tables
Defining the collection type

To set up a collection:

1. Define a collection of the desired type. You can do this in the declaration section of an SPL program, which results in a local type that you can access only in that program. For nested table and varray types, you can also do this using the `CREATE TYPE` command, which creates a persistent, standalone type that any SPL program in the database can reference.

2. Declare variables of the collection type. The collection associated with the declared variable is uninitialized at this point if no value assignment is made as part of the variable declaration.

Initializing a null collection

- Uninitialized collections of nested tables and varrays are null. A null collection doesn't yet exist. Generally, a `COLLECTION_IS_NULL` exception is thrown if a collection method is invoked on a null collection.
- To initialize a null collection, you must either make it an empty collection or assign a non-null value to it. Generally, a null collection is initialized by using its constructor.

Adding elements to an associative array

- Uninitialized collections of associative arrays exist but have no elements. An existing collection with no elements is called an empty collection.
- To add elements to an empty associative array, you can assign values to its keys. For nested tables and varrays, generally its constructor is used to assign initial values to the nested table or varray.
- For nested tables and varrays, you then use the `EXTEND` method to grow the collection beyond its initial size set by the constructor.

Limitations

- Multilevel collections (that is, where the data item of a collection is another collection) aren't supported.
- Columns of collection types aren't supported.

For example, you can create an array `varchar2_t`, but you can't create a table using array `varchar2_t` as a column data type.

```
-- Create an array
create type varchar2_t AS TABLE OF character varying;

-- Create a table using array as the column data type
create table t(a varchar2_t);
```

```
ERROR: column "a" has collection type varchar2_t, columns of collection types are not supported.
```

11.5.7.2 Using associative arrays

An associative array is a type of collection that associates a unique key with a value. The key doesn’t have to be numeric. It can be character data as well.

Associative array overview

An associative array has the following characteristics:

- You must define an associative array type after which you can declare array variables of that array type. Data manipulation occurs using the array variable.
- When an array variable is declared, the associative array is created, but it is empty. Start assigning values to keys.
- The key can be any negative integer, positive integer, or zero if you specify `INDEX BY BINARY_INTEGER` or `PLS_INTEGER`.
- The key can be character data if you specify `INDEX BY VARCHAR2`.
- There's no predefined limit on the number of elements in the array, it grows dynamically as elements are added.
- The array can be sparse. There can be gaps in the assignment of values to keys.
- An attempt to reference an array element that hasn't been assigned a value results in an exception.

```
-- Create an associative array
define varchar2_t as table of character varying;
```

```
ERROR: column "a" has collection type varchar2_t, columns of collection types are not supported.
```
The `TYPE IS TABLE OF ... INDEX BY` statement is used to define an associative array type:

```
TYPE <assoctype> IS TABLE OF { <datatype> | <rectype> | <objtype>

INDEX BY { BINARY_INTEGER | PLS_INTEGER | VARCHAR2(<n>)

};
```

Where:

- `<assoctype>` is an identifier assigned to the array type.
- `<datatype>` is a scalar data type such as `VARCHAR2` or `NUMBER`.
- `<rectype>` is a previously defined record type.
- `<objtype>` is a previously defined object type.
- `<n>` is the maximum length of a character key.

### Declaring a variable

To make use of the array, you must declare a variable with that array type. The following is the syntax for declaring an array variable:

```
<array> <assoctype>
```

Where:

- `<array>` is an identifier assigned to the associative array.
- `<assoctype>` is the identifier of a previously defined array type.

### Referencing an element of the array

Reference an element of the array using the following syntax:

```
<array>(<n>)[<.field>]
```

- `<array>` is the identifier of a previously declared array.
- `<n>` is the key value, type-compatible with the data type given in the `INDEX BY` clause.

If the array type of `<array>` is defined from a record type or object type, then `[<.field>]` must reference an individual field in the record type or attribute in the object type from which the array type is defined. Alternatively, you can reference the entire record by omitting `[<.field>]`.

### Examples

This example reads the first 10 employee names from the `emp` table, stores them in an array, and then displays the results from the array:

```
DECLARE
  TYPE emp_arr_typ IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
  emp_arr emp_arr_typ;
  CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
  i INTEGER := 0;
BEGIN
  FOR r_emp IN emp_cur LOOP
    i := i + 1;
    emp_arr(i) := r_emp.ename;
  END LOOP;
  FOR j IN 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE(emp_arr(j));
  END LOOP;
END;
```
This example uses a record type in the array definition:

```sql
DECLARE
    TYPE emp_rec_typ IS RECORD
    (
        empno NUMBER(4),
        ename VARCHAR2(10)
    );
    TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
    emp_arr emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
    DBMS_OUTPUT.PUT_LINE('----- -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' || emp_arr(j).ename);
    END LOOP;
END;
```

This example uses the `emp%ROWTYPE` attribute to define `emp_arr_typ` instead of using the `emp_rec_typ` record type:

```sql
DECLARE
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    emp_arr emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
    DBMS_OUTPUT.PUT_LINE('----- -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' || emp_arr(j).ename);
    END LOOP;
END;
```
The results are the same as using a record type in the array definition.

Instead of assigning each field of the record individually, you can make a record-level assignment from `r_emp` to `emp_arr`:

```sql
DECLARE
    TYPE emp_rec IS RECORD
    (empno NUMBER(4), ename VARCHAR2(10));
    TYPE emp_arr_typ IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    emp_arr emp_arr_typ;
CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
    DBMS_OUTPUT.PUT_LINE('----- ------- ');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
FOR j IN 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '  ' || emp_arr(j).ename);
END LOOP;
END;
```

This example uses the key of an associative array as character data:

```sql
DECLARE
    TYPE job_arr_typ IS TABLE OF NUMBER INDEX BY VARCHAR2(9);
    job_arr job_arr_typ;
BEGIN
    job_arr('ANALYST') := 100;
    job_arr('CLERK') := 200;
    job_arr('MANAGER') := 300;
    job_arr('SALESMAN') := 400;
    job_arr('PRESIDENT') := 500;
    DBMS_OUTPUT.PUT_LINE('ANALYST  : ' || job_arr('ANALYST'));
    DBMS_OUTPUT.PUT_LINE('CLERK    : ' || job_arr('CLERK'));
    DBMS_OUTPUT.PUT_LINE('MANAGER  : ' || job_arr('MANAGER'));
    DBMS_OUTPUT.PUT_LINE('SALESMAN : ' || job_arr('SALESMAN'));
    DBMS_OUTPUT.PUT_LINE('PRESIDENT: ' || job_arr('PRESIDENT'));
END;
```
11.5.7.3 Working with nested tables

A nested table is a type of collection that associates a positive integer with a value.

Nested tables overview

A nested table has the following characteristics:

- You must define a nested table type. After that, you can declare nested table variables of that nested table type. Data manipulation occurs using the nested table variable, also known simply as a table.
- When you declare a nested table variable, the nested table doesn't yet exist. It is a null collection. You must initialize the null table with a constructor. You can also initialize the table by using an assignment statement where the right-hand side of the assignment is an initialized table of the same type.

  Note

  Initialization of a nested table is mandatory in Oracle but optional in SPL.

- The key is a positive integer.
- The constructor establishes the number of elements in the table. The EXTEND method adds elements to the table. For details, see Collection methods.

  Note

  Using the constructor to establish the number of elements in the table and using the EXTEND method to add elements to the table are mandatory in Oracle but optional in SPL.

- The table can be sparse. There can be gaps in assigning values to keys.
- An attempt to reference a table element beyond its initialized or extended size results in a SUBSCRIPT_BEYOND_COUNT exception.

Defining a nested table

Use the TYPE IS TABLE statement to define a nested table type in the declaration section of an SPL program:

```sql
TYPE <tbltype> IS TABLE OF {<datatype> | <rectype> | <objtype>};
```

Where:

- `<tbltype>` is an identifier assigned to the nested table type.
- `<datatype>` is a scalar data type such as VARCHAR2 or NUMBER.
- `<rectype>` is a previously defined record type.
- `<objtype>` is a previously defined object type.

  Note

  You can use the CREATE TYPE command to define a nested table type that's available to all SPL programs in the database. See SQL reference for more information about the CREATE TYPE command.

Declaring a variable

To use the table, you must declare a variable of that nested table type. The following is the syntax for declaring a table variable:

```sql
<table> <tbltype>
```
Where:
	**table** is an identifier assigned to the nested table.
	**tbltype** is the identifier of a previously defined nested table type.

**Initializing the nested table**

Initialize a nested table using the nested table type's constructor:

```sql
<tbltype>([ ( <expr1> | NULL ) [, ( <expr2> | NULL ) ] [, ... ] ])
```

Where:
	**tbltype** is the identifier of the nested table type's constructor, which has the same name as the nested table type.
	x, y, z, ... are expressions that are type-compatible with the element type of the table. If you specify `NULL`, the corresponding element is set to null. If the parameter list is empty, then an empty nested table is returned, which means the table has no elements. If the table is defined from an object type, then `exprn` must return an object of that object type. The object can be the return value of a function or the object type's constructor. Or the object can be an element of another nested table of the same type.

If you apply a collection method other than `EXISTS` to an uninitialized nested table, a `COLLECTION_IS_NULL` exception is thrown. For details, see **Collection methods**.

This example shows a constructor for a nested table:

```sql
DECLARE TYPE nested_typ IS TABLE OF CHAR(1);
  v_nested nested_typ := nested_typ('A','B');
```

**Referencing an element of the table**

Reference an element of the table using the following syntax:

```sql
<table>({n})[<.element>]
```

Where:
	**table** is the identifier of a previously declared table.
	**n** is a positive integer.

If the table type of **table** is defined from a record type or object type, then `[.element]` must reference an individual field in the record type or attribute in the object type from which the nested table type is defined. Alternatively, you can reference the entire record or object by omitting `[.element]`.

**Examples**

This example shows a nested table where it's known that there are four elements:

```sql
DECLARE

  TYPE dname_tbl_typ IS TABLE OF VARCHAR2(14);
  dname_tbl dname_tbl_typ;
  CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
  i INTEGER := 0;
BEGIN
  dname_tbl := dname_tbl_typ(NULL, NULL, NULL, NULL);
  FOR r_dept IN dept_cur LOOP
    i := i + 1;
    dname_tbl(i) := r_dept.dname;
  END LOOP;
```
The following is the output from the example:

```
ANOTHERflatten
----------
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```

This example reads the first 10 employee names from the `emp` table, stores them in a nested table, and then displays the results from the table. The SPL code is written to assume that the number of employees to return isn’t known beforehand.

```
DECLARE
  TYPE emp_rec_typ IS RECORD
  (
    empno NUMBER(4),
    ename VARCHAR2(10)
  );
  TYPE emp_tbl_typ IS TABLE OF
    emp_rec_typ;
  emp_tbl emp_tbl_typ;
  CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
  i INTEGER := 0;
BEGIN
  emp_tbl := emp_tbl_typ();
  DBMS_OUTPUT.PUT_LINE('EMPNO     ENAME');
  FOR r_emp IN emp_cur LOOP
    i := i + 1;
    emp_tbl.EXTEND;
    emp_tbl(i) := r_emp;
  END LOOP;
  FOR j IN 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE(emp_tbl(j).empno || '     ' || emp_tbl(j).ename);
  END LOOP;
END;
```

An empty table with the constructor `emp_tbl_typ()` is created as the first statement in the executable section of the anonymous block. The `EXTEND` collection method is then used to add an element to the table for each employee returned from the result set. See `Extend`.

The following is the output:

```
EMPNO     ENAME
-----     -------
7369      SMITH
7499      ALLEN
7521      WARD
7566      JONES
7654      MARTIN
7698      BLAKE
7782      CLARK
7788      SCOTT
7839      KING
7844      TURNER
```

This example shows how you can use a nested table of an object type. First, create an object type with attributes for the department name and location:
This anonymous block defines a nested table type whose element consists of the `dept_obj_typ` object type. A nested table variable is declared, initialized, and then populated from the `dept` table. Finally, the elements from the nested table are displayed.

```sql
CREATE TYPE dept_obj_typ AS OBJECT
(
    dname VARCHAR2(14),
    loc VARCHAR2(13)
);
```

The parameters that make up the nested table’s constructor, `dept_tbl_typ`, are calls to the object type’s constructor `dept_obj_typ`. The following is the output from the anonymous block:

```
--- OUTPUT ---
DNAME LOC
----------     ----------
ACCOUNTING NEW YORK
OPERATIONS BOSTON
RESEARCH DALLAS
SALES        CHICAGO
```

### 11.5.7.4 Using varrays

A varray or variable-size array is a type of collection that associates a positive integer with a value. In many respects, it’s similar to a nested table.

**Varray overview**

A varray has the following characteristics:

- You must define a varray type with a maximum size limit. After you define the varray type, you can declare varray variables of that varray type. Data manipulation occurs using the varray variable, also known simply as a varray. The number of elements in the varray can’t exceed the maximum size limit set in the varray type definition.
- When you declare a varray variable, the varray at first is a null collection. You must initialize the null varray with a constructor. You can also initialize the varray by using an assignment statement where the right-hand side of the assignment is an initialized varray of the same type.
- The key is a positive integer.
- The constructor sets the number of elements in the varray, which must not exceed the maximum size limit. The `EXTEND` method can add elements to the varray up to the maximum size limit. For details, see Collection methods.
- Unlike a nested table, a varray cannot be sparse. There are no gaps when assigning values to keys.
An attempt to reference a varray element beyond its initialized or extended size but within the maximum size limit results in a **SUBSCRIPT_BEYOND_COUNT** exception.

An attempt to reference a varray element beyond the maximum size limit or extend a varray beyond the maximum size limit results in a **SUBSCRIPT_OUTSIDE_LIMIT** exception.

## Defining a varray type

The **TYPE IS VARRAY** statement is used to define a varray type in the declaration section of an SPL program:

```
TYPE <varraytype> IS { VARRAY | VARYING ARRAY }
{<maxsize>} OF { <datatype> | <objtype> }
```

Where:

- `<varraytype>` is an identifier assigned to the varray type.
- `<datatype>` is a scalar data type such as `VARCHAR2` or `NUMBER`.
- `<maxsize>` is the maximum number of elements permitted in varrays of that type.
- `<objtype>` is a previously defined object type.

You can use the **CREATE TYPE** command to define a varray type that's available to all SPL programs in the database. To make use of the varray, you must declare a variable of that varray type. The following is the syntax for declaring a varray variable:

```
<varray> <varraytype>
```

Where:

- `<varray>` is an identifier assigned to the varray.
- `<varraytype>` is the identifier of a previously defined varray type.

## Initializing a varray

Initialize a varray using the varray type’s constructor:

```
<varraytype> ([ [ <expr1> | NULL ] [, [ <expr2> | NULL ] ] [ , ... ] ])
```

Where:

- `<varraytype>` is the identifier of the varray type’s constructor, which has the same name as the varray type.
- `<expr1>`, `<expr2>`, ... are expressions that are type-compatible with the element type of the varray. If you specify `NULL`, the corresponding element is set to null. If the parameter list is empty, then an empty varray is returned, which means there are no elements in the varray. If the varray is defined from an object type, then `<exprn>` must return an object of that object type. The object can be the return value of a function or the return value of the object type’s constructor. The object can also be an element of another varray of the same varray type.

If you apply a collection method other than **EXISTS** to an uninitialized varray, a **COLLECTION_IS_NULL** exception is thrown. For details, see Collection methods.

The following is an example of a constructor for a varray:

```
DECLARE
    TYPE varray_typ IS VARRAY(2) OF CHAR(1);
    v_varray varray_typ :=
    varray_typ('A','B');
```

## Referencing an element of the varray

Reference an element of the varray using this syntax:

```
<varray>(<n>)[<.element>]
```
Where:

- **varray** is the identifier of a previously declared varray.
- **n** is a positive integer.

If the varray type of **varray** is defined from an object type, then **[].element** must reference an attribute in the object type from which the varray type is defined. Alternatively, you can reference the entire object by omitting **[].element**.

This example shows a varray where it is known that there are four elements:

```sql
DECLARE
  TYPE dname_varray_typ IS VARRAY(4) OF VARCHAR2(14);
  dname_varray dname_varray_typ;
  CURSOR dept_cur IS SELECT dname FROM dept ORDER BY dname;
  i INTEGER := 0;
BEGIN
  dname_varray := dname_varray_typ(NULL, NULL, NULL, NULL);
  FOR r_dept IN dept_cur LOOP
    i := i + 1;
    dname_varray(i) := r_dept.dname;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('DNAME');
  DBMS_OUTPUT.PUT_LINE('----------');
  FOR j IN 1..i LOOP
    DBMS_OUTPUT.PUT_LINE(dname_varray(j));
  END LOOP;
END;
```

The following is the output from this example:

```
DNAME
----------
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```

**11.5.8 Working with collections**

Collection operators allow you to transform, query, and manipulate the contents of a collection.

**11.5.8.1 Using the TABLE function**

Use the **TABLE()** function to transform the members of an array into a set of rows. The signature is:

```sql
TABLE(<collection_value>)
```

Where **collection_value** is an expression that evaluates to a value of collection type.

The **TABLE()** function expands the nested contents of a collection into a table format. You can use the **TABLE()** function anywhere you use a regular table expression.

The **TABLE()** function returns a **SETOF ANYELEMENT**, which is a set of values of any type. For example, if the argument passed to this function is an array of **dates**, **TABLE()** returns a **SETOF dates**. If the argument passed to this function is an array of **paths**, **TABLE()** returns a **SETOF paths**.

You can use the **TABLE()** function to expand the contents of a collection into table form:

```sql
postgres=# SELECT * FROM TABLE(monthly_balance(445.00, 988.20, 552.00));
monthly_balance
----------------
```

*EDB Postgres Advanced Server (EPAS)*

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
11.5.8.2 Using the MULTISET operators

The **MULTISET** operators combine two collections to form a third collection.

**Syntax**

```sql
<coll_1> MULTISET [ UNION | INTERSECT | EXCEPT ] [ ALL | DISTINCT | UNIQUE ] <coll_2>
```

Where `coll_1` and `coll_2` specify the names of the collections to combine.

Include the **ALL** keyword to specify to represent duplicate elements (elements that are present in both `coll_1` and `coll_2`) in the result, once for each time they're present in the original collections. This is the default behavior.

Include the **DISTINCT** or **UNIQUE** keyword to include duplicate elements in the result only once. There is no difference between the **DISTINCT** and **UNIQUE** keywords.

There are three forms of MULTISET operators:

- **MULTISET UNION**
- **MULTISET INTERSECT**
- **MULTISET EXCEPT**

**MULTISET UNION**

**MULTISET UNION** takes as arguments two nested tables and returns a nested table whose values from both the input nested tables. The two input nested tables must be of the same type, and the returned nested table is of the same type as well.

This example uses the **MULTISET UNION** operator to combine `collection_1` and `collection_2` into a third collection, `collection_3`:

```sql
DECLARE
  TYPE int_arr_typ IS TABLE OF NUMBER(2);
  collection_1  int_arr_typ;
  collection_2  int_arr_typ;
  collection_3  int_arr_typ;
  v_results VARCHAR2(50);
BEGIN
  collection_1 := int_arr_typ(10,20,30);
  collection_2 := int_arr_typ(30,40);
  collection_3 := collection_1 MULTISET UNION ALL collection_2;
  DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
  FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
    IF collection_3(i) IS NULL THEN
      v_results := v_results || 'NULL' || ' ';
    ELSE
      v_results := v_results || collection_3(i) || ' ';
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
```

The resulting collection includes one entry for each element in `collection_1` and `collection_2`. If you use the **DISTINCT** keyword, the results are as follows:

```sql
DECLARE
  TYPE int_arr_typ IS TABLE OF NUMBER(2);
```
The resulting collection includes only those members with distinct values.

In this example, the **MULTISET UNION DISTINCT** operator removes duplicate entries that are stored in the same collection:

```sql
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1 int_arr_typ;
    collection_2 int_arr_typ;
    collection_3 int_arr_typ;
    v_results VARCHAR2 (50);
BEGIN
    collection_1 := int_arr_typ(10,20,30);
    collection_2 := int_arr_typ(30,40);
    collection_3 := collection_1 MULTISET UNION DISTINCT collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL' || '
        ELSE
            v_results := v_results || collection_3(i) || ' ''
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
COUNT: 4
Results: 10 20 30 48
```

In this example, the **MULTISET UNION DISTINCT** operator removes duplicate entries that are stored in the same collection:

```sql
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1 int_arr_typ;
    collection_2 int_arr_typ;
    collection_3 int_arr_typ;
    v_results VARCHAR2 (50);
BEGIN
    collection_1 := int_arr_typ(10,20,30,30);
    collection_2 := int_arr_typ(40,50);
    collection_3 := collection_1 MULTISET UNION DISTINCT collection_2;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || collection_3.COUNT);
    FOR i IN collection_3.FIRST .. collection_3.LAST LOOP
        IF collection_3(i) IS NULL THEN
            v_results := v_results || 'NULL' || '
        ELSE
            v_results := v_results || collection_3(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
COUNT: 5
Results: 10 20 30 40 50
```

**MULTISET INTERSECT**

**MULTISET INTERSECT** takes as arguments two nested tables and returns a nested table whose values are common in the two input nested tables. The two input nested tables must be of the same type, and the returned nested table is of the same type as well.

This example uses the **MULTISET INTERSECT** operator to combine **color_name** and **fruit_name** into a third collection, **common_name**:

```sql
DECLARE
    TYPE name_typ IS TABLE OF VARCHAR(50);
    color_name name_typ;
    fruit_name name_typ;
    common_name name_typ;
BEGIN
    color_name := name_typ('Red', 'Green', 'Blue', 'Orange', 'Peach', 'Yellow', 'Peach');
    fruit_name := name_typ('Mango', 'Orange', 'Grapes', 'Banana', 'Peach', 'Peach');
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
This example shows the use of `MULTISET INTERSECT DISTINCT`:

```sql
DECLARE
    TYPE name_typ IS TABLE OF VARCHAR(50);
    color_name name_typ;
    fruit_name name_typ;
    common_name name_typ;
BEGIN
    color_name := name_typ('Red', 'Green', 'Blue', 'Orange', 'Yellow', 'Peach');
    fruit_name := name_typ('Mango', 'Orange', 'Grapes', 'Banana', 'Peach', 'Peach');
    common_name := color_name MULTISET INTERSECT DISTINCT fruit_name;
    FOR i IN common_name.FIRST .. common_name.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(common_name(i));
    END LOOP;
END;
```

This example shows the use of `MULTISET INTERSECT ALL`:

```sql
DECLARE
    TYPE name_typ IS TABLE OF VARCHAR(50);
    color_name name_typ;
    fruit_name name_typ;
    common_name name_typ;
BEGIN
    color_name := name_typ('Red', 'Green', 'Blue', 'Orange', 'Peach', 'Yellow', 'Peach');
    fruit_name := name_typ('Mango', 'Orange', 'Grapes', 'Banana', 'Peach', 'Peach');
    common_name := color_name MULTISET INTERSECT ALL fruit_name;
    FOR i IN common_name.FIRST .. common_name.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(common_name(i));
    END LOOP;
END;
```

**MULTISET EXCEPT**

`MULTISET EXCEPT` takes two nested tables as arguments and returns a nested table whose elements are in the first nested table but not in the second nested table. The two input nested tables must be of the same type, and the returned nested table is of the same type as well.

This example shows the use of `MULTISET EXCEPT UNIQUE`:

```sql
DECLARE
    TYPE name_typ IS TABLE OF VARCHAR(50);
    color_name name_typ;
    fruit_name name_typ;
    common_name name_typ;
BEGIN
    color_name := name_typ('Red', 'Green', 'Blue', 'Blue', 'Orange', 'Peach', 'Yellow');
    fruit_name := name_typ('Mango', 'Orange', 'Grapes', 'Banana', 'Peach');
```
This example shows the use of **MULTISET EXCEPT DISTINCT**:

```sql
DECLARE
    TYPE name_typ IS TABLE OF VARCHAR(50);
    color_name  name_typ;
    fruit_name  name_typ;
    common_name name_typ;
BEGIN
    color_name := name_typ('Red', 'Green', 'Blue', 'Blue', 'Orange', 'Peach', 'Yellow');
    fruit_name := name_typ('Mango', 'Orange', 'Grapes', 'Banana', 'Peach');
    common_name := color_name MULTISET EXCEPT DISTINCT fruit_name;
    FOR i IN common_name.FIRST .. common_name.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(common_name(i));
    END LOOP;
END;
```

This example shows the use of **MULTISET EXCEPT DISTINCT**:

```sql
DECLARE
    TYPE name_typ IS TABLE OF VARCHAR(50);
    color_name  name_typ;
    fruit_name  name_typ;
    common_name name_typ;
BEGIN
    color_name := name_typ('Red', 'Green', 'Blue', 'Blue', 'Orange', 'Peach', 'Yellow');
    fruit_name := name_typ('Mango', 'Orange', 'Grapes', 'Banana', 'Peach');
    common_name := color_name MULTISET EXCEPT DISTINCT fruit_name;
    FOR i IN common_name.FIRST .. common_name.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(common_name(i));
    END LOOP;
END;
```

11.5.8.3 Using the FORALL statement

You can use collections to process DML commands more efficiently by passing all the values to be used for repetitive execution of a `DELETE`, `INSERT`, or `UPDATE` command in one pass to the database server. The alternative is to iteratively invoke the DML command with new values. Specify the DML command to process this way with the `FORALL` statement. In addition, provide one or more collections in the DML command where you want to substitute different values each time the command is executed.
## Syntax

`FORALL <index> IN <lower_bound> .. <upper_bound>
{
  <insert_stmt> | <update_stmt> | <delete_stmt>
}
`

- **<index>** is the position in the collection given in the `insert_stmt`, `update_stmt`, or `delete_stmt` DML command that iterates from the integer value given as `lower_bound` up to and including `upper_bound`.

## How it works

If an exception occurs during any iteration of the `FORALL` statement, all updates that occurred since the start of the execution of the `FORALL` statement are rolled back.

### Note

This behavior isn't compatible with Oracle databases. Oracle allows explicit use of the `COMMIT` or `ROLLBACK` commands to control whether to commit or roll back updates that occurred prior to the exception.

The `FORALL` statement creates a loop. Each iteration of the loop increments the `index` variable. You typically use the `index` in the loop to select a member of a collection. Control the number of iterations with the `lower_bound .. upper_bound` clause. The loop executes once for each integer between the `lower_bound` and `upper_bound` (inclusive), and the index increments by one for each iteration.

For example:

```
FORALL i IN 2 .. 5
```

This expression creates a loop that executes four times. In the first iteration, `index (i)` is set to the value 2. In the second iteration, the index is set to the value 3, and so on. The loop executes for the value 5 and then terminates.

### Using FORALL with CREATE

This example creates a table `emp_copy` that's an empty copy of the `emp` table. The example declares a type `emp_tbl` that's an array. Each element in the array is of composite type, composed of the column definitions used to create the table `emp`. The example also creates an index on the `emp_tbl` type.

```sql
CREATE TABLE emp_copy(LIKE emp);
DECLARE
  TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
  t_emp emp_tbl;
BEGIN
  SELECT * FROM emp BULK COLLECT INTO t_emp;
  FORALL i IN t_emp.FIRST .. t_emp.LAST
  INSERT INTO emp_copy VALUES t_emp(i);
END;
```

### Using FORALL with UPDATE

This example uses a `FORALL` statement to update the salary of three employees:

```
DECLARE
  TYPE empno_tbl IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
  TYPE sal_tbl IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
  t_empno EMPNO_TBL;
```

The `FORALL` statement iterates through the `t_empno` array and inserts a row for each record into `emp_copy`.

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Using FORALL with DELETE

This example deletes three employees in a FORALL statement:

```
DECLARE
    TYPE empno_tbl IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    t_empno EMPNO_TBL;
BEGIN
    t_empno(1) := 9001;
    t_empno(2) := 9002;
    t_empno(3) := 9003;
    FORALL i IN t_empno.FIRST..t_empno.LAST
        DELETE FROM emp WHERE empno = t_empno(i);
END;
```

```
empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
9001   | JONES | ANALYST |     |          | 3350.00 |      |     40
9002   | LARSEN | CLERK |     |          | 2000.00 |      |     40
9003   | WILSON | MANAGER |     |          | 4100.00 |      |     40
(3 rows)
```

11.5.8.4 Using the BULK COLLECT clause

SQL commands that return a result set consisting of a large number of rows might not operate efficiently. This is due to the constant context switching to transfer the entire result set that occurs between the database server and the client.

You can mitigate this inefficiency by using a collection to gather the entire result set in memory, which the client can then access. You use the BULK COLLECT clause to specify the aggregation of the result set into a collection.

You can use the BULK COLLECT clause with the SELECT INTO, FETCH INTO, and EXECUTE IMMEDIATE commands. You can also use it with the RETURNING INTO clause of the DELETE, INSERT, and UPDATE commands.

11.5.8.4.1 SELECT BULK COLLECT

The following shows the syntax for the BULK COLLECT clause with the SELECT INTO statement. For details on the SELECT INTO statement, see SELECT INTO.
This example produces the same result but uses an associative array on a record type defined with the

**NOMTYPE** attribute:

```sql
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
BEGIN
    SELECT empno, ename, job, hiredate, sal, comm, deptno BULK COLLECT INTO emp
    FROM emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO ENAME   JOB    HIREDATE   SAL     COMM     DEPTNO ');
    DBMS_OUTPUT.PUT_LINE('-------- ------- ------ -------- ------ ------ -------- ');
    FOR i IN 1..emp.empno.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(emp.empno(i) || ' ' || ' ' ||
            RPAD(emp.ename(i),8) || ' ' ||
            RPAD(emp.job(i),10) || ' ' ||
            TO_CHAR(emp.hiredate(i),'DD-MON-YY') || ' ' ||
            TO_CHAR(NVL(emp.comm(i),0),'.99,999.99') || ' ' ||
            t_deptno(i));
    END LOOP;
END;
```

This example uses the **BULK COLLECT** clause where the target collections are associative arrays consisting of a single field:
### 11.5.8.4.2 **FETCH BULK COLLECT**

You can use the `BULK COLLECT` clause with a `FETCH` statement. Instead of returning a single row at a time from the result set, the `FETCH BULK COLLECT` returns all rows at once from the result set into one or more specified collections (both scalar and composite types) unless restricted by the `LIMIT` clause:

```sql
FETCH <name> BULK COLLECT INTO <collection> [, ...] [ LIMIT <n> ];
```

For information on the `FETCH` statement, see *Fetching rows from a cursor*.

If you specify a single collection, then `collection` can be a collection of a single field, or it can be a collection of a record type. If you specify more than one collection, then each `collection` must consist of a single field or a record type. The expressions in the `SELECT` list of the cursor identified by `name` must match all fields in the target collections in number, order, and type-compatibility. If you specify `LIMIT n`, the number of rows returned into the collection on each `FETCH` doesn't exceed `n`.

This example uses the `FETCH BULK COLLECT` statement to retrieve rows into an associative array:

```sql
DECLARE
  TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
  t_emp EMP_TBL;
BEGIN
  CURSOR emp_cur IS SELECT * FROM emp;
  OPEN emp_cur;
  FETCH emp_cur BULK COLLECT INTO t_emp;
  CLOSE emp_cur;
  DBMS_OUTPUT.PUT_LINE('EMPNO ENAME JOB HIREDATE ' ||
                        t_emp.empno || ' ' ||
                        RPAD(t_emp.ename,8) || ' ' ||
                        RPAD(t_emp.job,10) || ' ' ||
                        TO_CHAR(t_emp.hiredate,'DD-MON-YY') || ' ' ||
                        TO_CHAR(t_emp.sal,'99,999.99') || ' ' ||
                        TO_CHAR(NVL(t_emp.comm,'99,999.99'),'99,999.99') || ' ' ||
                        t_emp.deptno);
END;
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>17-DEC-80</td>
<td>800.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>20-FEB-81</td>
<td>1,600.00</td>
<td>300.00</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>22-FEB-81</td>
<td>1,250.00</td>
<td>500.00</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>02-APR-81</td>
<td>2,975.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>28-SEP-81</td>
<td>1,250.00</td>
<td>1,400.00</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>01-MAY-81</td>
<td>2,850.00</td>
<td>.00</td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>09-JUN-81</td>
<td>2,450.00</td>
<td>.00</td>
<td>30</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>19-APR-87</td>
<td>3,000.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>17-NOV-81</td>
<td>5,000.00</td>
<td>.00</td>
<td>10</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>08-SEP-81</td>
<td>1,500.00</td>
<td>.00</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>23-MAY-87</td>
<td>1,100.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>03-DEC-81</td>
<td>950.00</td>
<td>.00</td>
<td>30</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>03-DEC-81</td>
<td>3,000.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>23-JAN-82</td>
<td>1,300.00</td>
<td>.00</td>
<td>20</td>
</tr>
</tbody>
</table>

---

For more information on `BULK COLLECT` and `FETCH`, see the documentation for **11.5.8.2** and **11.5.8.3**.
<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>17-DEC-80</td>
<td>800.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>20-FEB-81</td>
<td>1,600.00</td>
<td>300.00</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>22-FEB-81</td>
<td>1,250.00</td>
<td>500.00</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>02-APR-81</td>
<td>2,975.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>28-SEP-81</td>
<td>1,250.00</td>
<td>1,400.00</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>01-MAY-81</td>
<td>2,850.00</td>
<td>.00</td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>09-JUN-81</td>
<td>2,450.00</td>
<td>.00</td>
<td>10</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>19-APR-87</td>
<td>3,000.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>17-NOV-81</td>
<td>5,000.00</td>
<td>.00</td>
<td>10</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>08-SEP-81</td>
<td>1,500.00</td>
<td>.00</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>23-MAY-87</td>
<td>1,100.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>03-DEC-81</td>
<td>950.00</td>
<td>.00</td>
<td>30</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>03-DEC-81</td>
<td>3,000.00</td>
<td>.00</td>
<td>20</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>23-JAN-82</td>
<td>1,380.00</td>
<td>.00</td>
<td>10</td>
</tr>
</tbody>
</table>

```
-- Create two object types of composite data
types
CREATE TYPE db_type1 as OBJECT(a INT, b VARCHAR2(10));
CREATE TYPE db_type2 as OBJECT(c VARCHAR2(10), d INT);

-- Create a table using above object types
CREATE TABLE db_tab(x DB_TYPE1, s1 NUMBER, y DB_TYPE2, s2 TIMESTAMP);

-- Insert the rows into the table
INSERT INTO db_tab values(DB_TYPE1(1, '10'), 1.1, DB_TYPE2('100', 1000), '1-Jan-2021 12:30:11 PM');
INSERT INTO db_tab values(DB_TYPE1(2, '20'), 2.2, DB_TYPE2('200', 2000), '2-Feb-2022 08:40:52 AM');
INSERT INTO db_tab values(DB_TYPE1(3, '30'), 3.3, DB_TYPE2('300', 3000), '3-Jan-2023 04:20:33 PM');

-- Use FETCH BULK COLLECT INTO clause for fetching both scalar and composite types
DECLARE
  type1_tbl IS TABLE OF db_type1 INDEX BY BINARY_INTEGER;
  type2_tbl IS TABLE OF db_type2 INDEX BY BINARY_INTEGER;
  x type1_tbl;
  y type2_tbl;
BEGIN
  CURSOR c1 is SELECT * FROM db_tab ORDER BY s1;
  OPEN c1;
  FETCH c1 BULK COLLECT INTO x, s1, y, s2;
END;
```
11.5.8.3  EXECUTE IMMEDIATE BULK COLLECT

You can use the **BULK COLLECT** clause with an **EXECUTE IMMEDIATE** statement to specify a collection to receive the returned rows:

```sql
EXECUTE IMMEDIATE '<sql_expression>;'
BULK COLLECT INTO <collection> [, ...]
[USING {[bind_type] <bind_argument>} [, ...]];```

Where:

- **collection** specifies the name of a collection.
- **bind_type** specifies the parameter mode of the **bind_argument**.
- **bind_argument** specifies a parameter that contains a value that either:
  - Is passed to the **sql_expression** (specified with a **bind_type** of **IN**)
  - Receives a value from the **sql_expression** (specified with a **bind_type** of **OUT**)
  - Does both (specified with a **bind_type** of **IN OUT**). Currently **bind_type** is ignored and **bind_argument** is treated as **IN OUT**.

If you specify a single collection, then **collection** can be a collection of a single field or a collection of a record type. If you specify more than one collection, each **collection** must consist of a single field.

11.5.8.4  RETURNING BULK COLLECT

**Syntax**

You can add **BULK COLLECT** to the **RETURNING INTO** clause of a **DELETE**, **INSERT**, or **UPDATE** command:

```sql
[<insert> | <update> | <delete>]
RETURNING
  ( [ * | <expr_1> ] [, <expr_2> ]
  )
BULK COLLECT INTO <collection> [, ...];```

For information on the **RETURNING INTO clause**, see Using the RETURNING INTO clause. **insert**, **update**, and **delete** are the same as the **INSERT**, **UPDATE**, and **DELETE** commands described in **INSERT**, **UPDATE**, and **DELETE**.

If you specify a single collection, then **collection** can be a collection of a single field, or it can be a collection of a record type. If you specify more than one collection, then each **collection** must consist of a single field. The expressions following the **RETURNING** keyword must match all fields in the target collections in number, order, and type-compatibility. Specifying `*` returns all columns in the affected table.
Note

The use of \* is an EDB Postgres Advanced Server extension and isn’t compatible with Oracle databases.

The `clerkemp` table created by copying the `emp` table is used in the examples that follow.

```sql
CREATE TABLE clerkemp AS
    SELECT * 
    FROM emp
    WHERE job = 'CLERK';
```

```sql
SELECT * 
FROM clerkemp;
```

```sql
DECLARE
    TYPE empno_tbl IS TABLE OF emp.empno%TYPE 
        INDEX BY BINARY_INTEGER;
    TYPE ename_tbl IS TABLE OF emp.ename%TYPE 
        INDEX BY BINARY_INTEGER;
    TYPE sal_tbl IS TABLE OF emp.sal%TYPE 
        INDEX BY BINARY_INTEGER;
    t_empno EMPNO_TBL;
    t_ename ENAME_TBL;
    t_sal SAL_TBL;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 
    RETURNING empno, ename, sal
    BULK COLLECT INTO t_empno, t_ename, t_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL');
    FOR i IN 1..t_empno.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_empno(i) || '   ' || RPAD(t_ename(i), 8) || '   ' || TO_CHAR(t_sal(i), '99,999.99'));
    END LOOP;
END;
```

### Examples

This example increases all employee salaries by 1.5, stores the employees' numbers, names, and new salaries in three associative arrays, and displays the contents of these arrays:

```sql
DECLARE
    TYPE emp_rec IS RECORD
    (
        empno emp.empno%TYPE,
        ename emp.ename%TYPE,
        sal emp.sal%TYPE
    );
    TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
    t_emp EMP_TBL;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 
    RETURNING empno, ename, sal
    BULK COLLECT INTO t_emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL');
END;
```

This example uses a single collection defined with a record type to store the employees' numbers, names, and new salaries:

```sql
DECLARE
    TYPE empno_tbl IS TABLE OF emp.empno%TYPE INDEX BY BINARY_INTEGER;
    TYPE ename_tbl IS TABLE OF emp.ename%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl IS TABLE OF emp.sal%TYPE INDEX BY BINARY_INTEGER;
    t_empno EMPNO_TBL;
    t_ename ENAME_TBL;
    t_sal SAL_TBL;
BEGIN
    UPDATE clerkemp SET sal = sal * 1.5 
    RETURNING empno, ename, sal
    BULK COLLECT INTO t_empno, t_ename, t_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME      SAL');
    FOR i IN 1..t_empno.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_empno(i) || '   ' || RPAD(t_ename(i), 8) || '   ' || TO_CHAR(t_sal(i), '99,999.99'));
    END LOOP;
END;
```
This example deletes all rows from the `clerkemp` table and returns information on the deleted rows into an associative array. It then displays the array.

```sql
DECLARE
  TYPE emp_rec IS RECORD
  (    empno emp.empno%TYPE,
       ename emp.ename%TYPE,
       job emp.job%TYPE,
       hiredate emp.hiredate%TYPE,
       sal emp.sal%TYPE,
       comm emp.comm%TYPE,
       deptno emp.deptno%TYPE);
  TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
  r_emp EMP_TBL;
BEGIN
  DELETE FROM clerkemp RETURNING empno, ename, job, hiredate, sal, comm, deptno BULK COLLECT INTO r_emp;
  DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ');
  DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   --------   ');
  FOR i IN 1..r_emp.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(r_emp(i).empno || '   ' || RPAD(r_emp(i).ename,8) || '   ' || TO_CHAR(r_emp(i).hiredate,'DD-MON-YY') || '   ' || TO_CHAR(r_emp(i).sal,'99,999.99') || '   ' || r_emp(i).deptno);
  END LOOP;
END;
```

This example deletes all rows from the `clerkemp` table and returns information on the deleted rows into an associative array. It then displays the array.

```
DECLARE
  TYPE emp_rec IS RECORD
  (    empno emp.empno%TYPE,
       ename emp.ename%TYPE,
       job emp.job%TYPE,
       hiredate emp.hiredate%TYPE,
       sal emp.sal%TYPE,
       comm emp.comm%TYPE,
       deptno emp.deptno%TYPE);
  TYPE emp_tbl IS TABLE OF emp_rec INDEX BY BINARY_INTEGER;
  r_emp EMP_TBL;
BEGIN
  DELETE FROM clerkemp RETURNING empno, ename, job, hiredate, sal, comm, deptno BULK COLLECT INTO r_emp;
  DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ');
  DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   --------   ');
  FOR i IN 1..r_emp.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(r_emp(i).empno || '   ' || RPAD(r_emp(i).ename,8) || '   ' || TO_CHAR(r_emp(i).hiredate,'DD-MON-YY') || '   ' || TO_CHAR(r_emp(i).sal,'99,999.99') || '   ' || r_emp(i).deptno);
  END LOOP;
END;
```
Reporting messages

Use the `DBMS_OUTPUT.PUT_LINE` statement to report messages:

```sql
DBMS_OUTPUT.PUT_LINE ( '<message>' );
```

Where `message` is any expression evaluating to a string.

This example displays the message on the user's output display:

```sql
DBMS_OUTPUT.PUT_LINE('My name is John');
```

The special variables `SQLCODE` and `SQLERRM` contain a numeric code and a text message, respectively, that describe the outcome of the last SQL command issued. If any other error occurs in the program such as division by zero, these variables contain information pertaining to the error.

**SQLCODE and SQLERRM functions**

SQLCODE and SQLERRM functions are now available in EDB Postgres Advanced Server.

In an exception handler, the `SQLCODE` function returns the numeric code of the exception being handled. Outside an exception handler, `SQLCODE` returns 0.

The `SQLERRM` function returns the error message associated with an `SQLCODE` variable value. If the error code value is passed to the `SQLERRM` function, it returns an error message associated with the passed error code value, regardless of the current error raised.

A SQL statement can't invoke `SQLCODE` and `SQLERRM` functions.

Examples:

```sql
declare
  l_var number;
begin
  l_var := -1476;
  dbms_output.put_line(sqlerrm(l_var::int));
  l_var := 8;
  dbms_output.put_line(sqlerrm(l_var::int));
  l_var := 12;
  dbms_output.put_line(sqlerrm(l_var::int));
  l_var := 81403;
  dbms_output.put_line(sqlerrm(l_var::int));
end;
```

```sql
DECLARE
  Balance integer := 24;
BEGIN
  IF (Balance <= 100) THEN
    Raise ApplicationException (-20343, 'The balance is too low.');
  END IF;
  exception
  when others
  then
    dbms_output.put_line('sqlcode ==>'||sqlcode);
    dbms_output.put_line('sqlerrm ==>'||sqlerrm);
    dbms_output.put_line('sqlerrm(sqlcode) ==>'||sqlerrm(sqlcode));
  END;
  sqlcode := -20343
  sqlerrm => EDB-20343: The balance is too low.
  sqlerrm(sqlcode) => EDB-20343: The balance is too low.
```

11.5.9 Working with triggers
As with procedures and functions, you write triggers in the SPL language.

### 11.5.9.1 Trigger overview

A *trigger* is a named SPL code block that's associated with a table and stored in the database. When a specified event occurs on the associated table, the SPL code block executes. The trigger is said to be *fired* when the code block executes.

The event that causes a trigger to fire can be any combination of an insert, update, or deletion carried out on the table, either directly or indirectly. If the table is the object of a SQL *INSERT*, *UPDATE*, *DELETE*, or *TRUNCATE* command, the trigger is directly fired when the corresponding insert, update, delete, or truncate event is defined as a triggering event. The events that fire the trigger are defined in the *CREATE TRIGGER* command.

A trigger can fire indirectly if a triggering event occurs on the table as a result of an event initiated on another table. For example, suppose a trigger is defined on a table containing a foreign key defined with the *ON DELETE CASCADE* clause, and a row in the parent table is deleted. In this case, all children of the parent are deleted as well. If deletion is a triggering event on the child table, deleting the children causes the trigger to fire.

### 11.5.9.2 Types of triggers

EDB Postgres Advanced Server supports *row-level* and *statement-level* triggers.

- A row-level trigger fires once for each row that's affected by a triggering event. For example, suppose deletion is defined as a triggering event on a table, and a single *DELETE* command is issued that deletes five rows from the table. In this case, the trigger fires five times, once for each row.

- A statement-level trigger fires once per triggering statement, regardless of the number of rows affected by the triggering event. In the previous example of a single *DELETE* command deleting five rows, a statement-level trigger fires only once.

You can define the sequence of actions regarding whether the trigger code block executes before or after the triggering statement for statement-level triggers. For row-level triggers, you can define whether the trigger code block executes before or after each row is affected by the triggering statement.

- In a *before* row-level trigger, the trigger code block executes before the triggering action is carried out on each affected row. In a *before* statement-level trigger, the trigger code block executes before the action of the triggering statement is carried out.

- In an *after* row-level trigger, the trigger code block executes after the triggering action is carried out on each affected row. In an *after* statement-level trigger, the trigger code block executes after the action of the triggering statement is carried out.

In a *compound trigger*, you can define a statement-level and a row-level trigger in a single trigger and fire it at more than one timing point. For details, see Compound triggers.

### 11.5.9.3 Creating triggers

The *CREATE TRIGGER* command defines and names a trigger that's stored in the database. You can create a simple trigger or a compound trigger.

#### Creating a simple trigger

```sql
CREATE [ OR REPLACE ] TRIGGER
<name>
{ 
  BEFORE | AFTER | INSTEAD OF 
} 
{ 
  INSERT | UPDATE | DELETE | TRUNCATE 
} 
{ 
  [ OR [ INSERT | UPDATE | DELETE | TRUNCATE ] ] [, 
  } 
[ ON <table> 
  [ REFERENCING [ OLD AS <old> | NEW AS <new> ] ] 
  ... 
  ] 
  { FOR EACH ROW 
  ] 
  { WHEN <condition> 
  } 
DECLARE 
  [ PRAGMA AUTONOMOUS_TRANSACTION; 
  ] 
<declaration>; [, ...]
BEGIN
```
Creating a compound trigger

**CREATE TRIGGER** — Define a compound trigger.

```
CREATE TRIGGER <name>
FOR <insert | update | delete | truncate>
[ OR ( <insert | update | delete | truncate > ] [,
...]
ON <table>
[ REFERENCING <old AS <old> | NEW AS <new> ]
[ WHEN <condition>
] COMPOUND TRIGGER
[ <private_declaration>; ]
...
[ <procedure_or_function_definition> ]
END
```

Where **private_declaration** is an identifier of a private variable that can be accessed by any procedure or function. There can be zero, one, or more private variables. **private_declaration** can be any of the following:

- Variable declaration
- Record declaration
- Collection declaration
- **REF CURSOR** and cursor variable declaration
- **TYPE** definitions for records, collections, and **REF CURSOR**
- Exception
- Object variable declaration

Where **procedure_or_function_definition** :=

```
procedureddefinition | function_definition
```

Where **procedure_definition** :=

```
PROCEDURE proc_name[ argument_list
] [ options_list
] [ IS | AS
]
procedure_body
END [ proc_name ]
```

Where **procedure_body** :=

```
[ <declaration>; ] [, 
...]
BEGIN
<statement>; [, 
...]
[ EXCEPTION
[ WHEN <exception> [ OR <exception> ] [...] THEN <statement>; 
...]
[ [...] 
]
```

Where **function_definition** :=
CREATE TRIGGER defines a new trigger. CREATE OR REPLACE TRIGGER creates a new trigger or replaces an existing definition.

If you're using the CREATE TRIGGER keywords to create a trigger, the name of the new trigger must not match any existing trigger defined on the same table. New triggers are created in the same schema as the table on which the triggering event is defined.

If you're updating the definition of an existing trigger, use the CREATE OR REPLACE TRIGGER keywords.

When you use syntax compatible with Oracle databases to create a trigger, the trigger runs as a SECURITY DEFINER function.

Parameters

name

The name of the trigger to create.

BEFORE | AFTER

Determines whether the trigger is fired before or after the triggering event.
INSTEAD OF

Trigger that modifies an updatable view. The trigger executes to update the underlying tables appropriately. The INSTEAD OF trigger executes for each row of the view that’s updated or modified.

INSERT | UPDATE | DELETE | TRUNCATE

Defines the triggering event.

table

The name of the table or view on which the triggering event occurs.

condition

A Boolean expression that determines if the trigger actually executes. If condition evaluates to TRUE, the trigger fires.

- If the simple trigger definition includes the FOR EACH ROW keywords, the WHEN clause can refer to columns of the old or new row values by writing OLD.column_name or NEW.column_name respectively. INSERT triggers can’t refer to OLD, and DELETE triggers can’t refer to NEW.

- If the compound trigger definition includes a statement-level trigger having a WHEN clause, then the trigger executes without evaluating the expression in the WHEN clause. Similarly, if a compound trigger definition includes a row-level trigger having a WHEN clause, then the trigger executes if the expression evaluates to TRUE.

- If the trigger includes the INSTEAD OF keywords, it can’t include a WHEN clause. A WHEN clause can’t contain subqueries.

REFERENCING { OLD AS old | NEW AS new } ...

REFERENCING clause to reference old rows and new rows but restricted in that old can be replaced only by an identifier named old or any equivalent that’s saved in all lowercase. Examples include REFERENCING OLD AS old, REFERENCING OLD AS OLD, or REFERENCING OLD AS "old". Also, new can be replaced only by an identifier named new or any equivalent that’s saved in all lowercase. Examples include REFERENCING NEW AS new, REFERENCING NEW AS NEW, or REFERENCING NEW AS "new".

You can specify one or both phrases OLD AS old and NEW AS new in the REFERENCING clause, such as REFERENCING NEW AS New OLD AS Old. This clause isn’t compatible with Oracle databases in that you can’t use identifiers other than old or new.

FOR EACH ROW

Determines whether to fire the trigger once for every row affected by the triggering event or once per SQL statement. If specified, the trigger is fired once for every affected row (row-level trigger). Otherwise the trigger is a statement-level trigger.

PRAGMA AUTONOMOUS_TRANSACTION

PRAGMA AUTONOMOUS_TRANSACTION is the directive that sets the trigger as an autonomous transaction.

declaration

A variable, type, REF_CURSOR, or subprogram declaration. If subprogram declarations are included, you must declare them after all other variable, type, and REF_CURSOR declarations.

statement

An SPL program statement. A DECLARE - BEGIN - END block is considered an SPL statement. Thus, the trigger body can contain nested blocks.

exception

An exception condition name such as NO_DATA_FOUND.

11.5.9.4 Trigger variables

In the trigger code block, several special variables are available for use.

NEW

NEW is a pseudo-record name that refers to the new table row for insert and update operations in row-level triggers. This variable doesn’t apply to statement-level triggers and delete operations of row-level triggers.

Its usage is:
WHERE column is the name of a column in the table where the trigger is defined.

The initial content of :NEW.column is the value in the named column of the new row to insert. Or, when used in a before row-level trigger, it's the value of the new row that replaces the old one. When used in an after row-level trigger, this value is already stored in the table since the action already occurred on the affected row.

In the trigger code block, you can use :NEW.column like any other variable. If a value is assigned to :NEW.column in the code block of a before row-level trigger, the assigned value is used in the new inserted or updated row.

OLD

OLD is a pseudo-record name that refers to the old table row for update and delete operations in row-level triggers. This variable doesn't apply in statement-level triggers and in insert operations of row-level triggers.

Its usage is: :OLD.column, where column is the name of a column in the table on which the trigger is defined.

The initial content of :OLD.column is the value in the named column of the row to delete or of the old row to replace with the new one when used in a before row-level trigger. When used in an after row-level trigger, this value is no longer stored in the table since the action already occurred on the affected row.

In the trigger code block, you can use :OLD.column like any other variable. Assigning a value to :OLD.column has no effect on the action of the trigger.

INSERTING

INSERTING is a conditional expression that returns TRUE if an insert operation fired the trigger. Otherwise it returns FALSE.

UPDATING

UPDATING is a conditional expression that returns TRUE if an update operation fired the trigger. Otherwise it returns FALSE.

DELETING

DELETING is a conditional expression that returns TRUE if a delete operation fired the trigger. Otherwise it returns FALSE.

11.5.9.5 Transactions and exceptions

A trigger is always executed as part of the same transaction in which the triggering statement is executing. When no exceptions occur in the trigger code block, the effects of any triggering commands in the trigger are committed only if the transaction containing the triggering statement is committed. Therefore, if the transaction is rolled back, the effects of any triggering commands in the trigger are also rolled back.

If an exception does occur in the trigger code block, but it is caught and handled in an exception section, the effects of any triggering commands in the trigger are still rolled back. The triggering statement, however, is rolled back only if the application forces a rollback of the containing transaction.

If an unhandled exception occurs in the trigger code block, the transaction that contains the trigger is aborted and rolled back. Therefore, the effects of any triggering commands in the trigger and the triggering statement are all rolled back.

11.5.9.6 Compound triggers

EDB Postgres Advanced Server has compatible syntax to support compound triggers.

Compound trigger overview

A compound trigger combines all the triggering timings under one trigger body that you can invoke at one or more timing points. A timing point is a point in time related to a triggering statement, which is an INSERT, UPDATE, DELETE, or TRUNCATE statement that modifies data. The supported timing points are:
**Before Statement** — Before the triggering statement executes.

**Before Each Row** — Before each row that the triggering statement affects.

**After Each Row** — After each row that the triggering statement affects.

**After Statement** — After the triggering statement executes.

**Instead Of Each Row** — Trigger fires once for every row affected by the triggering statement.

A compound trigger can include any combination of timing points defined in a single trigger.

The optional declaration section in a compound trigger allows you to declare trigger-level variables and subprograms. The content of the declaration is accessible to all timing points referenced by the trigger definition. The variables and subprograms created by the declaration persist only for the duration of the triggering statement.

### Syntax

A compound trigger contains a declaration followed by a PL block for each timing point:

```sql
CREATE OR REPLACE TRIGGER
    compound_trigger_name
FOR INSERT OR UPDATE OR DELETE ON table_name
COMPOUND TRIGGER
    -- Global Declaration Section
    (optional)
    -- Variables declared here can be used inside any timing-point blocks.
    BEFORE STATEMENT IS
        BEGIN
            NULL;
        END BEFORE STATEMENT;
    BEFORE EACH ROW IS
        BEGIN
            NULL;
        END BEFORE EACH ROW;
    AFTER EACH ROW IS
        BEGIN
            NULL;
        END AFTER EACH ROW;
    AFTER STATEMENT IS
        BEGIN
            NULL;
        END AFTER STATEMENT;
END compound_trigger_name;
/
```

**Note**

You don't have to have all the four timing blocks. You can create a compound trigger for any of the required timing points.

### Restrictions

A compound trigger has the following restrictions:

- A compound trigger body is made up of a compound trigger block.
- You can define a compound trigger on a table or a view.
- You can't transfer exceptions to another timing-point section. They must be handled separately in that section only by each compound trigger block.
- If a `GOTO` statement is specified in a timing-point section, then the target of the `GOTO` statement must also be specified in the same timing-point section.
- `:OLD` and `:NEW` variable identifiers can't exist in the declarative section, the `BEFORE STATEMENT` section, or the `AFTER STATEMENT` section.
- `:NEW` values are modified only by the `BEFORE EACH ROW` block.
- The sequence of compound trigger timing-point execution is specific. However, if a simple trigger is in the same timing point, then the simple trigger is fired first, followed by the compound triggers.

### 11.5.9.7 Trigger examples

The examples that follow show each type of trigger.
11.5.9.7.1 Before statement-level trigger

This example shows a simple before statement-level trigger that displays a message before an insert operation on the `emp` table:

```sql
CREATE OR REPLACE TRIGGER emp_alert_trig
BEFORE INSERT ON emp
BEGIN
  DBMS_OUTPUT.PUT_LINE('New employees are about to be added');
END;
```

The following `INSERT` is constructed so that several new rows are inserted upon a single execution of the command. For each row that has an employee id between 7900 and 7999, a new row is inserted with an employee id incremented by 1000. The following are the results of executing the command when three new rows are inserted:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>8900</td>
<td>JAMES</td>
<td>40</td>
</tr>
<tr>
<td>8902</td>
<td>FORD</td>
<td>40</td>
</tr>
<tr>
<td>8934</td>
<td>MILLER</td>
<td>40</td>
</tr>
</tbody>
</table>

The message `New employees are about to be added` is displayed once by the firing of the trigger even though the result adds three rows.

11.5.9.7.2 After statement-level trigger

This example shows an after statement-level trigger. When an insert, update, or delete operation occurs on the `emp` table, a row is added to the `empauditlog` table recording the date, user, and action.

```sql
CREATE TABLE empauditlog
(
  audit_date DATE,
  audit_user VARCHAR2(20),
  audit_desc VARCHAR2(20)
); CREATE OR REPLACE TRIGGER emp_audit_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
  v_action VARCHAR2(20);
BEGIN
  IF INSERTING THEN
    v_action := 'Added employee(s)';
  ELSIF UPDATING THEN
    v_action := 'Updated employee(s)';
  ELSIF DELETING THEN
    v_action := 'Deleted employee(s)';
  END IF;
  INSERT INTO empauditlog VALUES (SYSDATE, USER, v_action);
END;
```

In the following sequence of commands, two rows are inserted into the `emp` table using two `INSERT` commands. One `UPDATE` command updates the `sal` and `comm` columns of both rows. Then, one `DELETE` command deletes both rows.

```sql
INSERT INTO emp VALUES (
  9801,  'SMITH',  'ANALYST',  7782, SYSDATE, NULL, NULL, 10);
```
The contents of the `empauditlog` table show how many times the trigger was fired:

- Once each for the two inserts
- Once for the update (even though two rows were changed)
- Once for the deletion (even though two rows were deleted)

### 11.5.9.7.3 Before row-level trigger

This example shows a before row-level trigger that calculates the commission of every new employee belonging to department 30 that’s inserted into the `emp` table:

```sql
CREATE OR REPLACE TRIGGER emp_comm_trig
  BEFORE INSERT ON emp
  FOR EACH ROW
BEGIN
  IF :NEW.deptno = 30 THEN
    :NEW.comm := :NEW.sal * .4;
  END IF;
END;
```

The listing following the addition of the two employees shows that the trigger computed their commissions and inserted it as part of the new employee rows:

```sql
INSERT INTO emp VALUES (9005,'ROBERS','SALESMAN',7782,SYSDATE,3000.00,NULL,30);
INSERT INTO emp VALUES (9006,'ALLEN','SALESMAN',7782,SYSDATE,4500.00,NULL,30);
SELECT * FROM emp WHERE empno IN (9005, 9006);
```

### 11.5.9.7.4 After row-level trigger

This example shows an after row-level trigger. When a new employee row is inserted, the trigger adds a row to the `jobhist` table for that employee. When an existing employee is updated, the trigger sets the `enddate` column of the latest `jobhist` row (assumed to be the one with a null `enddate`) to the current date and inserts a new `jobhist` row with the employee’s new information.

Then, the trigger adds a row to the `empchglog` table with a description of the action.

```sql
CREATE TABLE empchglog
(
  chg_date DATE,
  chg_desc VARCHAR2(30)
);
CREATE OR REPLACE TRIGGER emp_chg_trig
```
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW
DECLARE
  v_empno emp.empno%TYPE;
  v_deptno emp.deptno%TYPE;
  v_dname dept.dname%TYPE;
  v_action VARCHAR2(7);
  v_chgdesc jobhist.chgdesc%TYPE;
BEGIN
  IF INSERTING THEN
    v_action := 'Added';
    v_empno := :NEW.empno;
    v_deptno := :NEW.deptno;
    v_chgdesc := ' ';
    IF NVL(OLD.empno, '-null-') != NVL(:NEW.empno, '-null-') THEN
      v_chgdesc := v_chgdesc || 'name,';
    END IF;
    IF NVL(OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
      v_chgdesc := v_chgdesc || 'job,';
    END IF;
    IF NVL(OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
      v_chgdesc := v_chgdesc || 'salary,';
    END IF;
    IF NVL(OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
      v_chgdesc := v_chgdesc || 'commission,'
    END IF;
    IF NVL(OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
      v_chgdesc := v_chgdesc || 'department,'
    END IF;
    v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ',');  
  UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
  AND enddate IS NULL;
  INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE,
ELSIF UPDATING THEN
  v_action := 'Updated';
  v_empno := :NEW.empno;
  v_deptno := :NEW.deptno;
  v_chgdesc := ' ';  
  IF NVL(OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
    v_chgdesc := v_chgdesc || 'job,';
  END IF;
  IF NVL(OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
    v_chgdesc := v_chgdesc || 'salary,';
  END IF;
  IF NVL(OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
    v_chgdesc := v_chgdesc || 'commission,'
  END IF;
  IF NVL(OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
    v_chgdesc := v_chgdesc || 'department,'
  END IF;
  v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ',');
  UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
  AND enddate IS NULL;
  INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE,
  :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno,
  v_chgdesc);
ELSIF DELETING THEN
  v_action := 'Deleted';
  v_empno := :OLD.empno;
  v_deptno := :OLD.deptno;
  v_chgdesc := ' ';  
  IF NVL(OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
    v_chgdesc := v_chgdesc || 'job,';
  END IF;
  IF NVL(OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
    v_chgdesc := v_chgdesc || 'salary,';
  END IF;
  IF NVL(OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
    v_chgdesc := v_chgdesc || 'commission,'
  END IF;
  IF NVL(OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
    v_chgdesc := v_chgdesc || 'department,'
  END IF;
  v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ',');
  UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
  AND enddate IS NULL;
  INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE,
  :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno,
  v_chgdesc);
END;

In the first sequence of the following commands, two employees are added using two separate INSERT commands. Then both are updated using a single UPDATE command. The contents of the jobhist table show the action of the trigger for each affected row: two new-hire entries for the two new employees and two changed commission records for the updated commissions on the two employees. The empchglog table also shows the trigger was fired a total of four times, once for each action on the two rows.

```
INSERT INTO emp VALUES
(9003,'PETERS','ANALYST',7782,SYSDATE,5800.00,NULL,40);
INSERT INTO emp VALUES
(9004,'AIKENS','ANALYST',7782,SYSDATE,4500.00,NULL,40);
UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
### 11.5.9.7.5 INSTEAD OF trigger

This example shows an INSTEAD OF trigger for inserting a new employee row into the `emp_vw` view. The `CREATE VIEW` statement creates the `emp_vw` view by joining the two tables. The trigger adds the corresponding new rows into the `emp` and `dept` tables, respectively, for a specific employee.

```sql
CREATE VIEW emp_vw AS SELECT * FROM emp e JOIN dept d USING(deptno);
CREATE VIEW empvw_instead_of_trig INSTEAD OF INSERT ON emp_vw FOR EACH ROW
DECLARE
  v_empno         emp.empno%TYPE;
  v_ename         emp.ename%TYPE;
  v_deptno        emp.deptno%TYPE;
  v_dname         dept.dname%TYPE;
  v_loc           dept.loc%TYPE;
  v_action        VARCHAR2(7);
BEGIN
  v_empno  := :NEW.empno;
  v_ename  := :NEW.ename;
  v_deptno := :NEW.deptno;
  v_dname  := :NEW.dname;
  v_loc    := :NEW.loc;
  INSERT INTO emp(empno, ename, deptno) VALUES(v_empno, v_ename, v_deptno);
  INSERT INTO dept(deptno, dname, loc) VALUES(v_deptno, v_dname, v_loc);
END;
CREATE TRIGGER
```

Next, insert the values into the `emp_vw` view. The insert action inserts a new row and produces the following output:
Similarly, if you specify an `UPDATE` or `DELETE` statement, the trigger performs the appropriate actions for `UPDATE` or `DELETE` events.

### 11.5.9.7.6 Compound triggers

#### Defining a compound trigger on a table

This example shows a compound trigger that records a change to the employee salary by defining a compound trigger named `hr_trigger` on the `emp` table.

1. Create a table named `emp`:

   ```sql
   CREATE TABLE emp (EMPNO INT, ENAME TEXT, SAL INT, DEPTNO INT);
   CREATE TABLE
   ```

2. Create a compound trigger named `hr_trigger`. The trigger uses each of the four timing points to modify the salary with an `INSERT`, `UPDATE`, or `DELETE` statement. In the global declaration section, the initial salary is declared as `10,000`.

   ```sql
   CREATE OR REPLACE TRIGGER hr_trigger
   FOR INSERT OR UPDATE OR DELETE ON
   emp
   COMPOUND TRIGGER
   -- Global declaration.
   var_sal NUMBER := 10000;

   BEFORE STATEMENT IS
   BEGIN
   var_sal := var_sal + 1000;
   DBMS_OUTPUT.PUT_LINE('Before Statement: ' || var_sal);
   END BEFORE STATEMENT;

   BEFORE EACH ROW IS
   BEGIN
   var_sal := var_sal + 1000;
   DBMS_OUTPUT.PUT_LINE('Before Each Row: ' || var_sal);
   END BEFORE EACH ROW;

   AFTER EACH ROW IS
   BEGIN
   var_sal := var_sal + 1000;
   DBMS_OUTPUT.PUT_LINE('After Each Row: ' || var_sal);
   END AFTER EACH ROW;

   AFTER STATEMENT IS
   BEGIN
   var_sal := var_sal + 1000;
   DBMS_OUTPUT.PUT_LINE('After Statement: ' || var_sal);
   END AFTER STATEMENT;
   ```
3. Insert the record into table `emp`:

```sql
INSERT INTO emp (EMPNO, ENAME, SAL, DEPTNO) VALUES(1111, 'SMITH', 10000, 20);
```

The `INSERT` statement produces the following output:

```sql
__OUTPUT__
Before Statement: 11000
Before each row: 12000
After each row: 13800
After statement: 14000
INSERT 0 1
```

4. The `UPDATE` statement updates the employee salary record, setting the salary to `15000` for a specific employee number:

```sql
UPDATE emp SET SAL = 15000 where EMPNO = 1111;
```

The `UPDATE` statement produces the following output:

```sql
Before Statement: 11000
Before each row: 12000
After each row: 13800
After statement: 14000
UPDATE 1
SELECT * FROM emp;
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>SAL</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>SMITH</td>
<td>15000</td>
<td>20</td>
</tr>
</tbody>
</table>

(1 row)

DELETE

The `DELETE` statement deletes the employee salary record:

```sql
DELETE from emp where EMPNO = 1111;
```

The `DELETE` statement produces the following output:

```sql
DELETE 1
SELECT * FROM emp;
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>SAL</th>
<th>DEPTNO</th>
</tr>
</thead>
</table>

(0 rows)

TRUNCATE

The `TRUNCATE` statement removes all the records from the `emp` table:

```sql
CREATE OR REPLACE TRIGGER hr_trigger
FOR TRUNCATE ON emp
COMPOUND
TRIGGER
-- Global declaration.
var_sal NUMBER := 10080;
```
TRUNCATE statement produces the following output:

Before Statement: 11000
After statement: 12000

Note

You can use the TRUNCATE statement only at a BEFORE STATEMENT or AFTER STATEMENT timing point.

Creating a compound trigger on a table with a WHEN condition

This example creates a compound trigger named hr_trigger on the emp table with a WHEN condition. The WHEN condition checks and prints the employee salary when an INSERT, UPDATE, or DELETE statement affects the emp table. The database evaluates the WHEN condition for a row-level trigger, and the trigger executes once per row if the WHEN condition evaluates to TRUE. The statement-level trigger executes regardless of the WHEN condition.

```sql
CREATE OR REPLACE TRIGGER hr_trigger
   FOR INSERT OR UPDATE OR DELETE ON emp
   REFERENCING NEW AS new OLD AS old
   WHEN (old.sal > 5000 OR new.sal < 8000) COMPOUND TRIGGER
   BEFORE STATEMENT IS
       BEGIN
           DBMS_OUTPUT.PUT_LINE('Before Statement: '); var_sal := var_sal + 1000;
       END BEFORE STATEMENT;
   BEFORE EACH ROW IS
       BEGIN
           DBMS_OUTPUT.PUT_LINE('Before Each Row: '); var_sal := var_sal + 1000;
       END BEFORE EACH ROW;
   AFTER EACH ROW IS
       BEGIN
           DBMS_OUTPUT.PUT_LINE('After Each Row: '); var_sal := var_sal + 1000;
       END AFTER EACH ROW;
   AFTER STATEMENT IS
       BEGIN
           DBMS_OUTPUT.PUT_LINE('After Statement: '); var_sal := var_sal + 1000;
       END AFTER STATEMENT;
END hr_trigger;
```

Output: Trigger created.
Insert the record into table `emp`:

```
INSERT INTO emp(EMPNO, ENAME, SAL, DEPTNO) VALUES(1111, 'SMITH', 1600, 20);
```

The `INSERT` statement produces the following output:

```
<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>sal</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>SMITH</td>
<td>1600</td>
<td>20</td>
</tr>
</tbody>
</table>
```

```
0 rows
```

**UPDATE**

The `UPDATE` statement updates the employee salary record, setting the salary to 7500:

```
UPDATE emp SET SAL = 7500 where EMPNO = 1111;
```

The `UPDATE` statement produces the following output:

```
<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>sal</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>SMITH</td>
<td>7500</td>
<td>20</td>
</tr>
</tbody>
</table>
```

```
1 row
```

**DELETE**

The `DELETE` statement deletes the employee salary record:

```
DELETE from emp where EMPNO = 1111;
```

The `DELETE` statement produces the following output:

```
<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>sal</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>SMITH</td>
<td>7500</td>
<td>20</td>
</tr>
</tbody>
</table>
```

```
0 rows
```

11.5.10 Working with packages

EDB Postgres Advanced Server provides a collection of packages that provide compatibility with Oracle packages.

A package is a named collection of functions, procedures, variables, cursors, user-defined record types, and records that are referenced using a common qualifier, known as the package identifier. Packages have the following characteristics:

- Packages provide a convenient means of organizing the functions and procedures that perform a related purpose. Permission to use the package functions and procedures depends on one privilege granted to the entire package. All of the package programs must be referenced with a common name.
- Certain functions, procedures, variables, types, and so on in the package can be declared as public. Public entities are visible and can be referenced by other programs that are given `EXECUTE` privilege on the package. For public functions and procedures, only their signatures are visible, that is, the program names, parameters, if any, and return types of functions. The SPL code of these functions and procedures isn’t accessible to others, therefore applications that use a package depend only on the information available in the signature and not in the procedural logic itself.
You can declare other functions, procedures, variables, types, and so on in the package as private. Private entities can be referenced and used by function and procedures in the package but not by other external applications. Private entities are for use only by programs in the package.

For more information about the package support provided by EDB Postgres Advanced Server, see built-in packages.

11.5.11 Using object types and objects

You can use object-oriented programming techniques in SPL. Object-oriented programming as seen in programming languages such as Java and C++ centers on the concept of objects. An object represents a real-world entity such as a person, place, or thing. The generic description or definition of a particular object such as a person, for example, is called an object type. Specific people, such as “Joe” or “Sally”, are said to be objects of object type person. They’re also known as instances of the object type person or, simply, person objects.

You can create objects and object types in SPL.

Note

- The terms “database objects” and “objects” are different from the terms “object type” and “object” used in object-oriented programming. Database objects are the entities that can be created in a database, such as tables, views, indexes, and users. In the context of object-oriented program, object type and object refer to specific data structures supported by the SPL programming language to implement object-oriented concepts.
- In Oracle, the term abstract data type (ADT) describes object types in PL/SQL. The SPL implementation of object types is intended to be compatible with Oracle abstract data types.
- EDB Postgres Advanced Server hasn’t yet implemented support for some features of object-oriented programming languages.

11.5.11.1 Basic object concepts

An object type is a description or definition of some entity. This definition of an object type is characterized by two components:

- Attributes — Fields that describe particular characteristics of an object instance. For a person object, examples are name, address, gender, date of birth, height, weight, eye color, and occupation.
- Methods — Programs that perform some type of function or operation on or are related to an object. For a person object, examples are calculating the person’s age, displaying the person’s attributes, and changing the values assigned to the person’s attributes.

11.5.11.1.1 Attributes

Every object type must contain at least one attribute. The data type of an attribute can be any of the following:

- A base data type such as NUMBER or VARCHAR2.
- Another object type.
- A globally defined collection type (created by the CREATE TYPE command) such as a nested table or varray.

An attribute gets its initial value, which can be null, when an object instance is first created. Each object instance has its own set of attribute values.

11.5.11.1.2 Methods

Methods are SPL procedures or functions defined in an object type. Methods are categorized into three general types:

- Member methods — Procedures or functions that operate in the context of an object instance. Member methods have access to and can change the attributes of the object instance on which they’re operating.
- Static methods — Procedures or functions that operate independently of any particular object instance. Static methods don’t have access to and can’t change the attributes of an object instance.
- Constructor methods — Functions used to create an instance of an object type. A default constructor method is always provided when an object type is defined.

11.5.11.1.3 Overloading methods

In an object type you can define two or more identically named methods (that is, a procedure or function) of the same type but with different signatures. Such methods are referred to as overloaded methods.

A method’s signature consists of the number of formal parameters, the data types of its formal parameters, and their order.
11.5.11.2 Object type components

Object types are created and stored in the database by using the following two constructs of the SPL language:

- The object type specification. This construct is the public interface specifying the attributes and method signatures of the object type.
- The object type body. This construct contains the implementation of the methods specified in the object type specification.

11.5.11.2.1 Object type specification syntax

The following is the syntax of the object type specification:

```
CREATE [ OR REPLACE ] TYPE <name>
[ AUTHID { DEFINER | CURRENT_USER } ]
[ IS | AS ]
OBJECT
{ { <attribute> [ <datatype> | <objtype> ] } }
{ [ , . . . ]
{ <method_spec> [ , . . . ]
{ <constructor> [ , . . . ]
[ NOT ] { FINAL | INSTANTIABLE } ]
... ;
```

Where `method_spec` is the following:

```
{ [ NOT ] { FINAL | INSTANTIABLE } ]
... OVERRIDING
{ <subprogram_spec>
```

Where `subprogram_spec` is the following:

```
{ MEMBER | STATIC
{ PROCEDURE <proc_name>
{ ( [ { SELF [ IN | IN OUT ] } <name>
{ [ , <parm1> [ IN | IN OUT | OUT ]
{ <datatype1> [ DEFAULT <value1> ]
{ [ , <parm2> [ IN | IN OUT | OUT ]
{ <datatype2> [ DEFAULT <value2> ]
{ ... ]
{ ]
{ FUNCTION <func_name>
{ ( [ SELF [ IN | IN OUT ] } <name>
{ [ , <parm1> [ IN | IN OUT | OUT ]
{ <datatype1> [ DEFAULT <value1> ]
{ [ , <parm2> [ IN | IN OUT | OUT ]
{ <datatype2> [ DEFAULT <value2> ]
{ ... ]
{ ]
RETURN <return_type>
```

Where `constructor` is the following:

```
CONSTRUCTOR FUNCTION <func_name>
{ ( [ <parm1> [ IN | IM OUT | OUT ]
{ <datatype1>
```
You can't use the OR REPLACE option to add, delete, or modify the attributes of an existing object type. Use the DROP TYPE command to first delete the existing object type. You can use the OR REPLACE option to add, delete, or modify the methods in an existing object type. You can use the PostgreSQL form of the ALTER TYPE ALTER ATTRIBUTE command to change the data type of an attribute in an existing object type. However, the ALTER TYPE command can't add or delete attributes in the object type.

name is an identifier (optionally schema-qualified) assigned to the object type. If you omit the AUTHID clause or specify DEFINER, the rights of the object type owner are used to determine access privileges to database objects. If you specify CURRENT_USER, the rights of the current user executing a method in the object determine access privileges.

Syntax

attribute is an identifier assigned to an attribute of the object type.

datatype is a base data type.

objtype is a previously defined object type.

collecttype is a previously defined collection type.

Following the closing parenthesis of the CREATE TYPE definition, [ NOT ] FINAL specifies whether a subtype can be derived from this object type. FINAL, which is the default, means that no subtypes can be derived from this object type. Specify NOT FINAL if you want to allow subtypes to be defined under this object type.

Note

Even though the specification of NOT FINAL is accepted in the CREATE TYPE command, SPL doesn't currently support creating subtypes.

Following the closing parenthesis of the CREATE TYPE definition, [ NOT ] INSTANTIABLE specifies whether an object instance of this object type can be created. INSTANTIABLE, which is the default, means that an instance of this object type can be created. Specify NOT INSTANTIABLE if this object type is to be used only as a parent "template" from which other specialized subtypes are defined. If NOT INSTANTIABLE is specified, then you must specify NOT FINAL as well. If any method in the object type contains the NOT INSTANTIABLE qualifier, then the object type must be defined with NOT INSTANTIABLE and NOT FINAL.

Note

Even though specifying NOT INSTANTIABLE is accepted in the CREATE TYPE command, SPL doesn't currently support creating subtypes.

method_spec

method_spec denotes the specification of a member method or static method.

Before defining a method, use [ NOT ] FINAL to specify whether the method can be overridden in a subtype. NOT FINAL is the default, meaning the method can be overridden in a subtype.

Before defining a method, specify OVERRIDING if the method overrides an identically named method in a supertype. The overriding method must have the same number of identically named method parameters with the same data types and parameter modes, in the same order, and with the same return type (if the method is a function) as defined in the supertype.

Before defining a method, use [ NOT ] INSTANTIABLE to specify whether the object type definition provides an implementation for the method. If you specify INSTANTIABLE, then the CREATE TYPE BODY command for the object type must specify the implementation of the method. If you specify NOT INSTANTIABLE, then the CREATE TYPE BODY command for the object type must not contain the implementation of the method. In this latter case, it is assumed a subtype contains the implementation of the method, overriding the method in this object type. If there are any NOT INSTANTIABLE methods in the object type, then the object type definition must specify NOT INSTANTIABLE and NOT FINAL following the closing parenthesis of the object type specification. The default is INSTANTIABLE.

subprogram_spec
subprogram_spec denotes the specification of a procedure or function and begins with the specification of either MEMBER or STATIC. A member subprogram must be invoked with respect to a particular object instance while a static subprogram isn’t invoked with respect to any object instance.

proc_name is an identifier of a procedure. If you specify the SELF parameter, name is the object type name given in the CREATE TYPE command. If specified, parm1, parm2, … are the formal parameters of the procedure. datatype1, datatype2, … are the data types of parm1, parm2, … respectively. IN, IN OUT, and OUT are the possible parameter modes for each formal parameter. The default is IN. value1, value2, … are default values that you can specify for IN parameters.

CONSTRUCTOR

Include the CONSTRUCTOR FUNCTION keyword and function definition to define a constructor function.

func_name is an identifier of a function. If specified, parm1, parm2, … are the formal parameters of the function. datatype1, datatype2, … are the data types of parm1, parm2, … respectively. IN, IN OUT, and OUT are the possible parameter modes for each formal parameter. The default is IN. value1, value2, … are default values that you can specify for IN parameters. return_type is the data type of the value the function returns.

Note the following about an object type specification:

- There must be at least one attribute defined in the object type.
- There can be zero, one, or more methods defined in the object type.
- A static method can’t be overridden. You can’t specify OVERRIDING and STATIC together in method_spec.
- A static method must be instantiable. You can’t specify NOT INSTANTIABLE and STATIC together in method_spec.

11.5.11.2 Object type body syntax

The following is the syntax of the object type body:

CREATE [ OR REPLACE ] TYPE BODY
<name>
( IS | AS )
<method_spec>
BEGIN
EXCEPTION
    WHEN...
    THEN...
END;

PRAGMA AUTONOMOUS_TRANSACTION;
<declarations>
BEGIN
<statement>; ...
[ EXCEPTION WHEN ...
    THEN ...
<statement>; ... ]
END;
FUNCTION <func_name>
( IS | AS )
<method_spec>
BEGIN
<statement>; ...
[ EXCEPTION WHEN ...
    THEN ...
<statement>; ... ]
END;
Where constructor is:

```sql
CONSTRUCTOR FUNCTION <func_name>
  [ ( [ [ <parm1> [ IN | IN OUT | OUT ]
          <datatype1> ]
       [ DEFAULT <value1> ]
   )
   [ , <parm2> [ IN | IN OUT | OUT ]
          <datatype2> ]
       [ DEFAULT <value2> ]
   ]
   ... )
  RETURN <return_type>
  ( IS | AS )
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  [ <declarations> ]
  BEGIN
  <statement>; ...
  [ EXCEPTION
     WHEN ... THEN
     <statement>; ...]
  END;
```

Where:

- **name** is an identifier (optionally schema-qualified) assigned to the object type.
- **method_spec** denotes the implementation of an instantiable method that was specified in the `CREATE TYPE` command.

If **INSTANTIABLE** was specified or omitted in **method_spec** of the `CREATE TYPE` command, then there must be a **method_spec** for this method in the `CREATE TYPE BODY` command.

If **NOT INSTANTIABLE** was specified in **method_spec** of the `CREATE TYPE` command, then there must be no **method_spec** for this method in the `CREATE TYPE BODY` command.

**subprogram_spec** denotes the specification of a procedure or function and begins with the specification of either **MEMBER** or **STATIC**. The same qualifier must be used as specified in **subprogram_spec** of the `CREATE TYPE` command.

**proc_name** is an identifier of a procedure specified in the `CREATE TYPE` command. The parameter declarations have the same meaning as described for the `CREATE TYPE` command. They must be specified in the `CREATE TYPE BODY` command in the same manner as in the `CREATE TYPE` command.

Include the **CONSTRUCTOR FUNCTION** keyword and function definition to define a constructor function.

**func_name** is an identifier of a function specified in the`CREATE TYPE` command. The parameter declarations have the same meaning as described for the `CREATE TYPE` command and must be specified in the `CREATE TYPE BODY` command in the same manner as in the `CREATE TYPE` command. **return_type** is the data type of the value the function returns and must match the **return_type** given in the `CREATE TYPE` command.

**PRAGMA AUTONOMOUS_TRANSACTION** is the directive that sets the procedure or function as an autonomous transaction.

**declarations** are variable, cursor, type, or subprogram declarations. If subprogram declarations are included, they must be declared after all other variable, cursor, and type declarations.

**statement** is an SPL program statement.
11.5.11.3 Creating object types

You can use the `CREATE TYPE` command to create an object type specification and the `CREATE TYPE BODY` command to create an object type body. The examples that follow use the `CREATE TYPE` and `CREATE TYPE BODY` commands.

The first example creates the `addr_object_type` object type that contains only attributes and no methods:

```sql
CREATE OR REPLACE TYPE addr_object_type AS
OBJECT
  (street VARCHAR2(30),
   city VARCHAR2(20),
   state CHAR(2),
   zip NUMBER(5))
;
```

Since there are no methods in this object type, an object type body isn't required. This example creates a composite type, which allows you to treat related objects as a single attribute.

### 11.5.11.3.1 Member methods

A member method is a function or procedure that is defined in an object type and can be invoked only through an instance of that type. Member methods have access to, and can change the attributes of, the object instance on which they're operating.

This object type specification creates the `emp_obj_typ` object type:

```sql
CREATE OR REPLACE TYPE emp_obj_typ AS
OBJECT
  (empno NUMBER(4),
   ename VARCHAR2(20),
   addr ADDR_OBJ_TYP,
   MEMBER PROCEDURE display_emp(SELF IN OUT emp_obj_typ))
;
```

Object type `emp_obj_typ` contains a member method named `display_emp`. `display_emp` uses a `SELF` parameter, which passes the object instance on which the method is invoked.

A `SELF` parameter is a parameter whose data type is that of the object type being defined. `SELF` always refers to the instance that's invoking the method. A `SELF` parameter is the first parameter in a member procedure or function regardless of whether it's explicitly declared in the parameter list.

The following code defines an object type body for `emp_obj_typ`:

```sql
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
  MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
  IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Employee No   : ' || SELF.empno);
      DBMS_OUTPUT.PUT_LINE('Name          : ' || SELF.ename);
      DBMS_OUTPUT.PUT_LINE('Street        : ' || addr.street);
      DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || addr.city || ', ' ||
                          addr.state || ' ' ||
                          LPAD(addr.zip, 5, '0'));
    END;
END;
```

You can also use the `SELF` parameter in an object type body. Using the `SELF` parameter in the `CREATE TYPE BODY` command, you can write the same object type body as follows:

```sql
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
  MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
  IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Employee No   : ' ||
                          SELF.empno);
      DBMS_OUTPUT.PUT_LINE('Name          : ' ||
                          SELF.ename);
    END;
END;
```
Both versions of the `emp_obj_typ` body are equivalent.

### 11.5.11.3.2 Static methods

Like a member method, a static method belongs to a type. A static method, however, is invoked not by an instance of the type, but by using the name of the type. For example, to invoke a static function named `get_count`, defined in the `emp_obj_type` type, you can write:

```sql
emp_obj_type.get_count();
```

A static method doesn't have access to and can't change the attributes of an object instance. It doesn't typically work with an instance of the type.

The following object type specification includes a static function `get_dname` and a member procedure `display_dept`:

```sql
CREATE OR REPLACE TYPE dept_obj_typ AS OBJECT
(
  deptno NUMBER(2),
  STATIC FUNCTION get_dname(p_deptno IN NUMBER) RETURN VARCHAR2,
  MEMBER PROCEDURE display_dept
);
```

The object type body for `dept_obj_typ` defines a static function named `get_dname` and a member procedure named `display_dept`:

```sql
CREATE OR REPLACE TYPE BODY dept_obj_typ AS

  STATIC FUNCTION get_dname(p_deptno IN NUMBER) RETURN VARCHAR2
  IS
    v_dname VARCHAR2(14);
    BEGIN
      CASE p_deptno
        WHEN 10 THEN v_dname := 'ACCOUNTING';
        WHEN 20 THEN v_dname := 'RESEARCH';
        WHEN 30 THEN v_dname := 'SALES';
        WHEN 40 THEN v_dname := 'OPERATIONS';
        ELSE v_dname := 'UNKNOWN';
      END CASE;
      RETURN v_dname;
    END;

  MEMBER PROCEDURE display_dept
  IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Dept No    : ' || SELF.deptno);
      DBMS_OUTPUT.PUT_LINE('Dept Name  : ' || dept_obj_typ.get_dname(SELF.deptno));
    END;
  END;

The static function `get_dname` can't reference `SELF`. Since a static function is invoked independently of any object instance, it has no implicit access to any object attribute.

Member procedure `display_dept` can access the `deptno` attribute of the object instance passed in the `SELF` parameter. It isn't necessary to explicitly declare the `SELF` parameter in the `display_dept` parameter list.

The last `DBMS_OUTPUT.PUT_LINE` statement in the `display_dept` procedure includes a call to the static function `get_dname`, qualified by its object type name `dept_obj_typ`.

### 11.5.11.3.3 Constructor methods

A constructor method is a function that creates an instance of an object type, typically by assigning values to the members of the object. An object type can define several constructors to accomplish different tasks. A constructor method is a member function invoked with a `SELF` parameter whose name matches the name of the type.
For example, if you define a type named `address`, each constructor is named `address`. You can overload a constructor by creating one or more different constructor functions with the same name but with different argument types.

The SPL compiler provides a default constructor for each object type. The default constructor is a member function whose name matches the name of the type and whose argument list matches the type members in order. For example, given an object type such as:

```
CREATE TYPE address AS OBJECT
{
  street_address VARCHAR2(40),
  postal_code VARCHAR2(10),
  city VARCHAR2(40),
  state VARCHAR2(2)
}
```

The SPL compiler provides a default constructor with the following signature:

```
CONSTRUCTOR FUNCTION address
{
  street_address VARCHAR2(40),
  postal_code VARCHAR2(10),
  city VARCHAR2(40),
  state VARCHAR2(2)
}
```

The body of the default constructor sets each member to `NULL`.

To create a custom constructor, using the keyword constructor, declare the constructor function in the `CREATE TYPE` command, and define the construction function in the `CREATE TYPE BODY` command. For example, you might want to create a custom constructor for the `address` type that computes the city and state given a `street_address` and `postal_code`:

```
CREATE TYPE address AS OBJECT
{
  street_address VARCHAR2(40),
  postal_code VARCHAR2(10),
  city VARCHAR2(40),
  state VARCHAR2(2),
}
```

```
CONSTRUCTOR FUNCTION address
{
  street_address VARCHAR2,  
  postal_code VARCHAR2
}  
RETURN self AS RESULT
}
```

```
CREATE TYPE BODY address AS
CONSTRUCTOR FUNCTION address
{
  street_address VARCHAR2,
  postal_code VARCHAR2
}  
RETURN self AS RESULT
IS BEGIN
  self.street_address := street_address;
  self.postal_code := postal_code;
  self.city := postal_code_to_city(postal_code);
  self.state := postal_code_to_state(postal_code);
  RETURN;
END;
END;
```

To create an instance of an object type, you invoke one of the constructor methods for that type. For example:

```
DECLARE
cust_addr address := address('100 Main Street',
02203');
BEGIN
```

Custom constructor functions are:

- Typically used to compute member values when given incomplete information. The example computes the values for city and state when given a postal code.

- Also used to enforce business rules that restrict the state of an object. For example, if you define an object type to represent a payment, you can use a custom constructor to ensure that no object of type payment can be created with an amount that is NULL, negative, or zero. The default constructor sets payment.amount to NULL, so you must create a custom constructor whose signature matches the default constructor to prohibit NULL amounts.

11.5.1.4 Creating object instances

Creating an instance

To create an instance of an object type, you must first declare a variable of the object type and then initialize the declared variable. The syntax for declaring an object variable is:

```
<object> <obj_type>
```

Where:

- object is an identifier assigned to the object variable.
- obj_type is the identifier of a previously defined object type.

Invoking a constructor method

After declaring the object variable, you must invoke a constructor method to initialize the object with values. Use the following syntax to invoke the constructor method:

```
[NEW] <obj_type> ({{expr1} | NULL} [, {{expr2} | NULL} ] [, ...
```

Where:

- obj_type is the identifier of the object type's constructor method. The constructor method has the same name as the previously declared object type.
- expr1, expr2, ... are expressions that are type-compatible with the first attribute of the object type, the second attribute of the object type, and so on. If an attribute is of an object type, then the corresponding expression can be NULL, an object initialization expression, or any expression that returns that object type.

This anonymous block declares and initializes a variable:

```
DECLARE
    v_emp           EMP_OBJ_TYP;
BEGIN
    v_emp := emp_obj_typ(
        9001, 'JONES',
        addr_obj_typ('123 MAIN STREET', 'EDISON', 'NJ', 08817));
END;
```

The variable v_emp is declared with a previously defined object type named EMP_OBJ_TYPE. The body of the block initializes the variable using the emp_obj_typ and addr_obj_typ constructors.

You can include the NEW keyword when creating a new instance of an object in the body of a block. The NEW keyword invokes the object constructor whose signature matches the arguments provided.

Example

This example declares two variables named mgr and emp. The variables are both of EMP_OBJ_TYPE. The mgr object is initialized in the declaration, while the emp object is initialized to NULL in the declaration and assigned a value in the body.

```
DECLARE
    mgr EMP_OBJ_TYPE :=
        (9002, 'SMITH');
```
In EDB Postgres Advanced Server, you can use the following alternate syntax in place of the constructor method.

```
ROW
is an optional keyword if two or more terms are specified in the parenthesis-enclosed, comma-delimited list. If you specify only one term, then you must specify the ROW keyword.
```

### 11.5.11.5 Referencing an object

#### Syntax

After you create and initialize an object variable, you can reference individual attributes using dot notation of the form:

```
<object>.<attribute>
```

Where:

- **object** is the identifier assigned to the object variable.
- **attribute** is the identifier of an object type attribute.

If **attribute** is of an object type, then the reference must take the form:

```
<object>.<attribute>.<attribute_inner>
```

Where **attribute_inner** is an identifier belonging to the object type to which **attribute** references in its definition of **object**.

#### Examples

This example displays the values assigned to the emp_obj_typ object:

```
DECLARE
  v_emp EMP_OBJ_TYP;
BEGIN
  v_emp := emp_obj_typ(9003,'RAY');
  DBMS_OUTPUT.PUT_LINE('Employee No   : ' || v_emp.empno);
  DBMS_OUTPUT.PUT_LINE('Name          : ' || v_emp.ename);
  DBMS_OUTPUT.PUT_LINE('Street        : ' || LPAD(v_emp.addr.street, 5, ' '));
END;
```

The following is the output from this anonymous block:

```
<table>
<thead>
<tr>
<th>Employee No</th>
<th>Name</th>
<th>Street</th>
<th>City/State/Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>9003</td>
<td>RAY</td>
<td>123 MAIN STREET</td>
<td>EDISON, NJ 08817</td>
</tr>
</tbody>
</table>
```

Methods are called in a similar manner as attributes.
Once an object variable is created and initialized, member procedures or functions are called using dot notation of the form:

<object>.<prog_name>

Where:

object is the identifier assigned to the object variable.

prog_name is the identifier of the procedure or function.

Static procedures or functions aren’t called using an object variable. Instead call the procedure or function using the object type name:

<object_type>.<prog_name>

Where:

object_type is the identifier assigned to the object type.

prog_name is the identifier of the procedure or function.

You can duplicate the results of the previous anonymous block by calling the member procedure display_emp:

```sql
DECLARE
  v_emp          EMP_OBJ_TYP;
BEGIN
  v_emp := emp_obj_typ(9001, 'JONES',
    addr_obj_typ('123 MAIN STREET', 'EDISON', 'NJ', 08817));
  v_emp.display_emp;
END
```

The following is the output from this anonymous block:

```
Employee No: 9001
Name: JONES
Street: 123 MAIN STREET
City/State/Zip: EDISON, NJ 08817
```

This anonymous block creates an instance of dept_obj_typ and calls the member procedure display_dept:

```sql
DECLARE
  v_dept          DEPT_OBJ_TYP := dept_obj_typ(20);
BEGIN
  v_dept.display_dept;
END;
```

The following is the output from this anonymous block:

```
Dept No: 20
Dept Name: RESEARCH
```

You can call the static function defined in dept_obj_typ directly by qualifying it by the object type name as follows:

```sql
BEGIN
  DBMS_OUTPUT.PUT_LINE(dept_obj_typ.get_dname(20));
END;
```

11.5.11.6 Dropping an object type

Deleting an object type
The syntax for deleting an object type is as follows:

\[ \text{DROP TYPE } \text{<objtype>}; \]

Where \text{<objtype>} is the identifier of the object type to drop. If the definition of \text{<objtype>} contains attributes that are themselves object types or collection types, you must drop these nested object types or collection types last.

If an object type body is defined for the object type, the \text{DROP TYPE} command deletes the object-type body as well as the object-type specification. To re-create the complete object type, you must reissue both the \text{CREATE TYPE} and \text{CREATE TYPE BODY} commands.

This example drops the \text{emp_obj_typ} and the \text{addr_obj_typ} object types. You must drop \text{emp_obj_typ} first since it contains \text{addr_obj_typ} in its definition as an attribute.

\[ \text{DROP TYPE } \text{emp_obj_typ}; \]
\[ \text{DROP TYPE } \text{addr_obj_typ}; \]

Dropping only the object type body

The syntax for deleting an object type body but not the object type specification is:

\[ \text{DROP TYPE BODY } \text{<objtype>}; \]

You can re-create the object type body by issuing the \text{CREATE TYPE BODY} command.

This example drops only the object type body of the \text{dept_obj_typ}.

\[ \text{DROP TYPE BODY } \text{dept_obj_typ}; \]

### 11.6 Using table partitioning

In a partitioned table, one logically large table is broken into smaller physical pieces. Partitioning can provide several benefits:

- Query performance can improve dramatically, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning allows you to omit the partition column from the front of an index. This approach reduces index size and makes it more likely that the heavily used parts of the index fit in memory.
- When a query or update accesses a large percentage of a single partition, performance might improve. This improvement happens because the server performs a sequential scan of the partition instead of using an index and random-access reads scattered across the whole table.
- If you plan the requirement into the partitioning design, you can implement a bulk load or unload by adding or removing partitions. \text{ALTER TABLE} is far faster than a bulk operation. It also avoids the \text{VACUUM} overhead caused by a bulk \text{DELETE}.
- You can migrate seldom-used data to less-expensive or slower storage media.

Table partitioning is worthwhile when a table is becoming very large. The exact point at which a table benefits from partitioning depends on the application. A good guideline is for the size of the table not to exceed the physical memory of the database server.

#### 11.6.1 Oracle table partitioning compatibility summary

EDB Postgres Advanced Server supports aspects of table partitioning that are compatible with Oracle databases.

**Note**

The declarative partitioning feature, introduced with PostgreSQL version 10, is not covered here. However, PostgreSQL declarative partitioning is supported in EDB Postgres Advanced Server 10 in addition to the table partitioning compatible with Oracle databases described here. For information about declarative partitioning, see the PostgreSQL core documentation.

The PostgreSQL \text{INSERT... ON CONFLICT DO NOTHING/UPDATE} clause, commonly known as \text{UPSERT}, isn’t supported on Oracle-styled partitioned tables. If you include the \text{ON CONFLICT DO NOTHING/UPDATE} clause when invoking the \text{INSERT} command to add data to a partitioned table, an error occurs.

**Note**

EDB Postgres Advanced Server doesn’t support global indexes, so the index isn’t inherited when you define a primary key on the partitioned table that doesn’t include partition key columns. However, all partitions defined in \text{CREATE TABLE} have an independent primary index on the column. You can re-create the primary key on all newly added partitions by using \text{ALTER TABLE ... ADD CONSTRAINT}. This primary index enforces uniqueness in each partition but not across the entire partition hierarchy. In other words, you can have the same value repeated for the primary index column in two or more partitions.

#### 11.6.2 Selecting a partition type
When you create a partitioned table, you specify LIST, RANGE, or HASH partitioning rules. The partitioning rules provide a set of constraints that define the data that's stored in each partition. As you add rows to the partitioned table, the server uses the partitioning rules to decide which partition contains each row.

EDB Postgres Advanced Server can also use the partitioning rules to enforce partition pruning, which improves performance when responding to user queries. When selecting a partitioning type and partitioning keys for a table, consider how the data that's stored in a table is queried, and include often-queried columns in the partitioning rules.

### List partitioning

When you create a list-partitioned table, you specify a single partitioning key column. When adding a row to the table, the server compares the key values specified in the partitioning rule to the corresponding column in the row. If the column value matches a value in the partitioning rule, the row is stored in the partition named in the rule.

**Note**

List partitioning doesn't support multi-column list partitioning.

See [Automatic list partitioning](#) for information about an extension to LIST partitioning that enables a database to automatically create a partition for any new distinct value of the list partitioning key.

### Range partitioning

When you create a range-partitioned table, you specify one or more partitioning key columns. When you add a row to the table, the server compares the value of the partitioning keys to the corresponding columns in a table entry. If the column values satisfy the conditions specified in the partitioning rule, the row is stored in the partition named in the rule.

See [Interval range partitioning](#) for information about an extension to range partitioning that enables a database to create a partition when the inserted data exceeds the range of an existing partition.

### Hash partitioning

When you create a hash-partitioned table, you specify one or more partitioning key columns. Data is divided into approximately equal-sized partitions among the specified partitions. When you add a row to a hash-partitioned table, the server computes a hash value for the data in the specified columns and stores the row in a partition according to the hash value.

**Note**

When upgrading EDB Postgres Advanced Server, you must rebuild each hash-partitioned table on the upgraded version server.

### Subpartitioning

Subpartitioning breaks a partitioned table into smaller subsets. You must store all subsets in the same database server cluster. A table is typically subpartitioned by a different set of columns. It can have a different subpartitioning type from the parent partition. If you subpartition one partition, then each partition has at least one subpartition.

If you subpartition a table, no data is stored in any of the partition tables. Instead, the data is stored in the corresponding subpartitions.

#### 11.6.2.1 Interval range partitioning

Interval range partitioning is an extension to range partitioning that allows a database to create a partition when the inserted data exceeds the range of an existing partition. To implement interval range partitioning, include the INTERVAL clause, and specify the range size for a new partition.

The high value of a range partition, also known as the transition point, is determined by the range partitioning key value. The database creates partitions for inserted data with values that are beyond that high value.

**Interval range partitioning example**

Suppose an interval is set to one month. If data is inserted for two months after the current transition point, only the partition for the second month is created and not the intervening partition. For example, you can create an interval-range-partitioned table with a monthly interval and a current transition point of February 15, 2023. If you try to insert data for May 10, 2023, then the required partition for April 15 to May 15, 2023 is created and data is inserted into that partition. The partition for February 15, 2023 to March 15, 2023 and March 15, 2023 to April 15, 2023 is skipped.

For information about interval range partitioning syntax, see CREATE TABLE...PARTITION BY.
Restrictions on interval range partitioning

The following restrictions apply to the `INTERVAL` clause:

- Interval range partitioning is restricted to a single partition key. That key must be a numerical or date range.
- You must define at least one range partition.
- The `INTERVAL` clause isn't supported for index-organized tables.
- You can't create a domain index on an interval-range-partitioned table.
- In composite partitioning, the interval range partitioning can be useful as a primary partitioning mechanism but isn't supported at the subpartition level.
- You can't define `DEFAULT` and `MAXVALUE` for an interval-range-partitioned table.
- You can't specify `NULL`, `Not-a-Number`, and `Infinity` values in the partitioning key column.
- Interval range partitioning expression must yield constant value and can't be a negative value.
- You must create the partitions for an interval-range-partitioned table in increasing order.

11.6.2.2 Automatic list partitioning

Automatic list partitioning is an extension to `LIST` partitioning that allows a database to create a partition for any new distinct value of the list partitioning key. A new partition is created when data is inserted into the `LIST` partitioned table and the inserted value doesn't match any of the existing table partition. Use the `AUTOMATIC` clause to implement automatic list partitioning.

For example, consider a table named `sales` with a `sales_state` column that contains the existing partition values `CALIFORNIA` and `FLORIDA`. Each of the `sales_state` values increases with a rise in the statewise sales. A sale in a new state, for example, `INDIANA` and `OHIO`, requires creating new partitions. If you implement automatic list partitioning, the new partitions `INDIANA` and `OHIO` are automatically created, and data is entered into the `sales` table.

For information about automatic list partitioning syntax, see `CREATE TABLE...PARTITION BY`.

Restrictions for automatic list partitioning

The following restrictions apply to the `AUTOMATIC` clause:

- A table that enables automatic list partitioning can't have a `DEFAULT` partition.
- Automatic list partitioning doesn't support multi-column list partitioning.
- In composite partitioning, the automatic list partitioning can be useful as a primary partitioning mechanism but isn't supported at the subpartition level.

11.6.3 Using partition pruning

EDB Postgres Advanced Server's query planner uses partition pruning to compute an efficient plan to locate any rows that match the conditions specified in the `WHERE` clause of a `SELECT` statement. To successfully prune partitions from an execution plan, the `WHERE` clause must constrain the information that's compared to the partitioning key column specified when creating the partitioned table.

When querying a... | Partition pruning is effective when...  
---|---  
List-partitioned table | The `WHERE` clause compares a literal value to the partitioning key using operators like equal (=) or `AND`.
Range-partitioned table | The `WHERE` clause compares a literal value to a partitioning key using operators such as equal (=), less than (<), or greater than (>).
Hash-partitioned table | The `WHERE` clause compares a literal value to the partitioning key using an operator such as equal (=).

Partition pruning techniques

The partition pruning mechanism uses two optimization techniques:

- Constraint exclusion
- Fast pruning

Partition pruning techniques limit the search for data only to those partitions where the values you're searching for might reside. Both pruning techniques remove partitions from a query's execution plan, improving performance.

The difference between the fast pruning and constraint exclusion is that fast pruning understands the relationship between the partitions in an Oracle-partitioned table. Constraint exclusion doesn't. For example, when a query searches for a specific value in a list-partitioned table, fast pruning can reason that only a specific partition can hold that value. Constraint exclusion must examine the constraints defined for each partition. Fast pruning occurs early in the planning process to reduce the number of partitions that the planner must consider. Constraint exclusion occurs late in the planning process.

This example shows the efficiency of partition pruning, using the `EXPLAIN` statement to confirm that EDB Postgres Advanced Server is pruning partitions from the execution plan of a query.
Using constraint exclusion

The constraint_exclusion parameter controls constraint exclusion. The constraint_exclusion parameter can have a value of on, off, or partition. To enable constraint exclusion, you must set the parameter to either partition or on. By default, the parameter is set to partition.

For more information about constraint exclusion, see the PostgreSQL documentation.

When constraint exclusion is enabled, the server examines the constraints defined for each partition to determine if that partition can satisfy a query.

When you execute a SELECT statement that doesn't contain a WHERE clause, the query planner must recommend an execution plan that searches the entire table. When you execute a SELECT statement that contains a WHERE clause, the query planner:

- Determines the partition to store the row
- Sends query fragments to that partition
- Prunes the partitions that can't contain that row from the execution plan

If you aren't using partitioned tables, disabling constraint exclusion might improve performance.

Using fast pruning

Like constraint exclusion, fast pruning can optimize only queries that include a WHERE or join clause. However, the qualifiers in the WHERE clause must match a certain form. In both cases, the query planner avoids searching for data in partitions that can't hold the data required by the query.

Fast pruning is controlled by a Boolean configuration parameter named edb_enable_pruning. Set edb_enable_pruning to ON to enable fast pruning of certain queries. Set edb_enable_pruning to OFF to disable fast pruning.

Note

Fast pruning can optimize queries against subpartitioned tables or optimize queries against range-partitioned tables only for tables that are partitioned on one column.

For LIST-partitioned tables, EDB Postgres Advanced Server can fast prune queries that contain a WHERE clause that constrains a partitioning column to a literal value. For example, given a LIST-partitioned table such as:

```sql
CREATE TABLE sales_hist(..., country text,...)
PARTITION BY LIST(country)
{
  PARTITION americas VALUES('US', 'CA', 'MX'),
  PARTITION europe VALUES('BE', 'NL', 'FR'),
  PARTITION asia VALUES('JP', 'PK', 'CN'),
  PARTITION others VALUES(DEFAULT)
}
```

Fast pruning can reason about WHERE clauses such as:

```
WHERE country = 'US'
```

```
WHERE country IS NULL;
```

With the first WHERE clause, fast pruning eliminates partitions europe, asia, and others because those partitions can't hold rows that satisfy the qualifier WHERE country = 'US'.

With the second WHERE clause, fast pruning eliminates partitions americas, europe, and asia because those partitions can't hold rows where country IS NULL.

The operator specified in the WHERE clause must be an equals sign (=) or the equality operator appropriate for the data type of the partitioning column.

For range-partitioned tables, EDB Postgres Advanced Server can fast prune queries that contain a WHERE clause that constrains a partitioning column to a literal value. However, the operator can be any of the following:

```
>
>=
=
<=
<
```
Fast pruning also reasons about more complex expressions involving \texttt{AND} and \texttt{BETWEEN} operators, such as:

\begin{verbatim}
WHERE size > 100 AND size <= 200
WHERE size BETWEEN 100 AND 200
\end{verbatim}

Fast pruning can't prune based on expressions involving \texttt{OR} or \texttt{IN}. For example, when querying a \texttt{RANGE}-partitioned table, such as:

\begin{verbatim}
CREATE TABLE boxes(id int, size int, color text)
    PARTITION BY RANGE(size)
    {
        PARTITION small VALUES LESS THAN(100),
        PARTITION medium VALUES LESS THAN(200),
        PARTITION large VALUES LESS THAN(300)
    }
\end{verbatim}

Fast pruning can reason about \texttt{WHERE} clauses such as:

\begin{verbatim}
WHERE size > 100  -- scan partitions 'medium' and 'large'
WHERE size >= 100 -- scan partitions 'medium' and 'large'
WHERE size = 100  -- scan partition 'medium'
WHERE size <= 100 -- scan partitions 'small' and 'medium'
WHERE size < 100  -- scan partition 'small'
WHERE size > 100 AND size < 199     -- scan partition 'medium'
WHERE size BETWEEN 100 AND 199      -- scan partition 'medium'
WHERE color = 'red' AND size = 100  -- scan 'medium'
WHERE color = 'red' AND (size > 100 AND size < 199) -- scan 'medium'
\end{verbatim}

In each case, fast pruning requires that the qualifier refer to a partitioning column and literal value (or \texttt{IS NULL}/\texttt{IS NOT NULL}).

**Note**

Fast pruning can also optimize \texttt{DELETE} and \texttt{UPDATE} statements containing these \texttt{WHERE} clauses.

### 11.6.3.1 Example: Partition pruning

The \texttt{EXPLAIN} statement displays the execution plan of a statement. You can use the \texttt{EXPLAIN} statement to confirm that EDB Postgres Advanced Server is pruning partitions from the execution plan of a query.

This example shows the efficiency of partition pruning. Create a simple table:

\begin{verbatim}
CREATE TABLE sales
    {
        dept_no number,
        part_no varchar2,
        country varchar2(20),
        date date,
        amount number
    }
    PARTITION BY LIST(country)
    {
        PARTITION europe VALUES('FRANCE', 'ITALY'),
        PARTITION asia VALUES('INDIA', 'PAKISTAN'),
        PARTITION americas VALUES('US', 'CANADA')
    }
\end{verbatim}

Perform a constrained query that includes the \texttt{EXPLAIN} statement:

\begin{verbatim}
EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'INDIA';
\end{verbatim}
The resulting query plan shows that the server scans only the `sales_asia` table. That's the table in which a row with a `country` value of `INDIA` is stored.

```
EDB# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'INDIA';
```

```
 QUERY PLAN
---------------------------------------------------
 Append
 -> Seq Scan on sales_asia
    Filter: ((country)::text = 'INDIA '::text)
(3 rows)
```

Suppose you perform a query that searches for a row that matches a value not included in the partitioning key:

```
EDB# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE dept_no = '30';
```

The resulting query plan shows that the server must look in all of the partitions to locate the rows that satisfy the query:

```
EDB# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE dept_no = '30';
```

```
 QUERY PLAN
-------------------------------------------
 Append
 -> Seq Scan on sales_americas
    Filter: (dept_no = '30 '::numeric)
 -> Seq Scan on sales_europe
    Filter: (dept_no = '30 '::numeric)
 -> Seq Scan on sales_asia
    Filter: (dept_no = '30 '::numeric)
(7 rows)
```

Constraint exclusion also applies when querying subpartitioned tables:

```
CREATE TABLE sales
(
    dept_no     number,
    part_no     varchar2,
    country     varchar2(20),
    date        date,
    amount      number
)
PARTITION BY RANGE (date) SUBPARTITION BY LIST (country)
(
    PARTITION "2011" VALUES LESS THAN ('01-JAN-2012')
    (
        SUBPARTITION europe_2011 VALUES ('ITALY',
          'FRANCE'),
        SUBPARTITION asia_2011 VALUES ('PAKISTAN',
          'INDIA'),
        SUBPARTITION americas_2011 VALUES ('US',
          'CANADA')
    ),
    PARTITION "2012" VALUES LESS THAN ('01-JAN-2013')
    (
        SUBPARTITION europe_2012 VALUES ('ITALY',
          'FRANCE'),
        SUBPARTITION asia_2012 VALUES ('PAKISTAN',
          'INDIA'),
        SUBPARTITION americas_2012 VALUES ('US',
          'CANADA')
    ),
    PARTITION "2013" VALUES LESS THAN ('01-JAN-2015')
    (
        SUBPARTITION europe_2013 VALUES ('ITALY',
          'FRANCE'),
        SUBPARTITION asia_2013 VALUES ('PAKISTAN',
          'INDIA'),
        SUBPARTITION americas_2013 VALUES ('US',
          'CANADA')
    )
);
```

When you query the table, the query planner prunes any partitions or subpartitions from the search path that can’t contain the desired result set.
11.6.4 Handling stray values in a LIST or RANGE partitioned table

A DEFAULT or MAXVALUE partition or subpartition captures any rows that don’t meet the other partitioning rules defined for a table.

11.6.4.1 Defining a DEFAULT partition

A DEFAULT partition captures any rows that don’t fit into any other partition in a LIST partitioned or subpartitioned table. If you don’t include a DEFAULT rule, any row that doesn’t match one of the values in the partitioning constraints causes an error. Each LIST partition or subpartition can have its own DEFAULT rule.

The syntax of a DEFAULT rule is:

```
PARTITION [〈partition_name〉] VALUES (DEFAULT)
```

Where 〈partition_name〉 specifies the name of the partition or subpartition that stores any rows that don’t match the rules specified for other partitions.

Adding a DEFAULT partition

You can create a list-partitioned table in which the server decides the partition for storing the data based on the value of the country column. In that case, if you attempt to add a row in which the value of the country column contains a value not listed in the rules, an error is reported:

```
edb# INSERT INTO sales VALUES
edb=# (46, '3800x', 'IRELAND', '01-Mar-2012',
450000);
ERROR: no partition of relation "sales_2012" found for row
DETAIL: Partition key of the failing row contains (country) = (IRELAND).
```

This example creates such a table but adds a DEFAULT partition. The server stores any rows that don’t match a value specified in the partitioning rules for europe, asia, or americas partitions in the others partition.

```
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
) PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA'),
  PARTITION others VALUES (DEFAULT)
);
```

Testing the DEFAULT partition

To test the DEFAULT partition, add a row with a value in the country column that doesn’t match one of the countries specified in the partitioning constraints:

```
edb# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'US' AND date = 'Dec 12, 2012';
```

```
QUERY PLAN
--------------------------------------------------------------------------------
<table>
<thead>
<tr>
<th>Append</th>
<th>Seq Scan on sales_americas_2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12 00:00:00'::timestamp without time zone))</td>
<td>(3 rows)</td>
</tr>
</tbody>
</table>
```

```
11.6.4 Handling stray values in a LIST or RANGE partitioned table

A DEFAULT or MAXVALUE partition or subpartition captures any rows that don’t meet the other partitioning rules defined for a table.

11.6.4.1 Defining a DEFAULT partition

A DEFAULT partition captures any rows that don’t fit into any other partition in a LIST partitioned or subpartitioned table. If you don’t include a DEFAULT rule, any row that doesn’t match one of the values in the partitioning constraints causes an error. Each LIST partition or subpartition can have its own DEFAULT rule.

The syntax of a DEFAULT rule is:

```
PARTITION [〈partition_name〉] VALUES (DEFAULT)
```

Where 〈partition_name〉 specifies the name of the partition or subpartition that stores any rows that don’t match the rules specified for other partitions.

Adding a DEFAULT partition

You can create a list-partitioned table in which the server decides the partition for storing the data based on the value of the country column. In that case, if you attempt to add a row in which the value of the country column contains a value not listed in the rules, an error is reported:

```
edb# INSERT INTO sales VALUES
edb=# (46, '3800x', 'IRELAND', '01-Mar-2012',
450000);
ERROR: no partition of relation "sales_2012" found for row
DETAIL: Partition key of the failing row contains (country) = (IRELAND).
```

This example creates such a table but adds a DEFAULT partition. The server stores any rows that don’t match a value specified in the partitioning rules for europe, asia, or americas partitions in the others partition.

```
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
) PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA'),
  PARTITION others VALUES (DEFAULT)
);
```

Testing the DEFAULT partition

To test the DEFAULT partition, add a row with a value in the country column that doesn’t match one of the countries specified in the partitioning constraints:
INSERT INTO sales VALUES
(40, '3000x', 'IRELAND', '01-Mar-2012', '45000');

Query the contents of the `sales` table to confirm that the previously rejected row is now stored in the `sales_others` partition:

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>9519b</td>
<td>US</td>
<td>12-APR-12 00:00:00</td>
<td>145000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>4577b</td>
<td>US</td>
<td>11-NOV-12 00:00:00</td>
<td>25000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>30</td>
<td>7588b</td>
<td>CANADA</td>
<td>14-DEC-12 00:00:00</td>
<td>50000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>30</td>
<td>9519b</td>
<td>CANADA</td>
<td>01-FEB-12 00:00:00</td>
<td>75000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>3788a</td>
<td>US</td>
<td>12-MAY-12 00:00:00</td>
<td>4950</td>
</tr>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>4788a</td>
<td>US</td>
<td>23-SEP-12 00:00:00</td>
<td>4950</td>
</tr>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>4788b</td>
<td>US</td>
<td>09-OCT-12 00:00:00</td>
<td>15000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>4519b</td>
<td>FRANCE</td>
<td>17-JAN-12 00:00:00</td>
<td>45000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>9519b</td>
<td>ITALY</td>
<td>07-JUL-12 00:00:00</td>
<td>15000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>9519a</td>
<td>FRANCE</td>
<td>18-AUG-12 00:00:00</td>
<td>650000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>9519b</td>
<td>FRANCE</td>
<td>18-AUG-12 00:00:00</td>
<td>650000</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788a</td>
<td>INDIA</td>
<td>01-MAR-12 00:00:00</td>
<td>75000</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788a</td>
<td>PAKISTAN</td>
<td>04-JUN-12 00:00:00</td>
<td>37500</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788b</td>
<td>INDIA</td>
<td>21-SEP-12 00:00:00</td>
<td>5890</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>4519a</td>
<td>INDIA</td>
<td>02-DEC-12 00:00:00</td>
<td>5890</td>
</tr>
<tr>
<td>sales_others</td>
<td>40</td>
<td>3000x</td>
<td>IRELAND</td>
<td>01-MAR-12 00:00:00</td>
<td>45000</td>
</tr>
</tbody>
</table>

(18 rows)

EDB Postgres Advanced Server provides the following methods to reassign the contents of a `DEFAULT` partition or subpartition:

- You can use the `ALTER TABLE... ADD PARTITION` command to add a partition to a table with a `DEFAULT` rule. There can't be conflicting values between existing rows in the table and the values of the partition you're adding. You can alternatively use the `ALTER TABLE... SPLIT PARTITION` command to split an existing partition.
- You can use the `ALTER TABLE... ADD SUBPARTITION` command to add a subpartition to a table with a `DEFAULT` rule. There can't be conflicting values between existing rows in the table and the values of the subpartition you're adding. You can alternatively use the `ALTER TABLE... SPLIT SUBPARTITION` command to split an existing subpartition.

Example: Adding a partition to a table with a `DEFAULT` partition

This example uses the `ALTER TABLE... ADD PARTITION` command. It assumes there's no conflicting values between the existing rows in the table and the values of the partition to add.

```sql
edb=# SELECT tableoid:regclass, * FROM sales;
```

When the following rows are inserted into the table, an error occurs, indicating that there are conflicting values:

```sql
edb=# INSERT INTO sales (dept_no, country) VALUES (1, 'FRANCE'), (2, 'INDIA'), (3, 'US'), (4, 'SOUTH AFRICA'), (5, 'NEPAL');
```

Row (4, 'SOUTH AFRICA') conflicts with the `VALUES` list in the `ALTER TABLE... ADD PARTITION` statement, thus resulting in an error:

```sql
edb=# ALTER TABLE sales ADD PARTITION africa values ('SOUTH AFRICA', 'KENYA');
```

ERROR: updated partition constraint for default partition "sales_others" would be violated by some row

Example: Splitting a `DEFAULT` partition

This example splits a `DEFAULT` partition, redistributing the partition's content between two new partitions in the table `sales`.

```sql
INSERT INTO sales VALUES (18, '4519b', 'FRANCE', '17-Jan-2012', '45000');
```
The partitions include the `DEFAULT others` partition:

```
partition_name |     high_value
----------------+---------------------
EUROPE         | 'FRANCE', 'ITALY'
ASIA           | 'INDIA', 'PAKISTAN'
AMERICAS       | 'US', 'CANADA'
OTHERS         | DEFAULT
```

This command shows the rows distributed among the partitions:

```
<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_americas</td>
<td>30</td>
<td>9519b</td>
<td>US</td>
<td>12-APR-12 00:00:00</td>
<td>145000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>30</td>
<td>7588b</td>
<td>CANADA</td>
<td>14-DEC-12 00:00:00</td>
<td>50000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>4519b</td>
<td>ITALY</td>
<td>07-JUL-12 00:00:00</td>
<td>15000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>9519b</td>
<td>FRANCE</td>
<td>17-JAN-12 00:00:00</td>
<td>45000</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788a</td>
<td>INDIA</td>
<td>01-MAR-12 00:00:00</td>
<td>75000</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788a</td>
<td>PAKISTAN</td>
<td>04-JUN-12 00:00:00</td>
<td>37500</td>
</tr>
<tr>
<td>sales_others</td>
<td>40</td>
<td>4519b</td>
<td>SOUTH AFRICA</td>
<td>08-APR-12 00:00:00</td>
<td>120000</td>
</tr>
<tr>
<td>sales_others</td>
<td>40</td>
<td>4519b</td>
<td>KENYA</td>
<td>08-APR-12 00:00:00</td>
<td>120000</td>
</tr>
<tr>
<td>sales_others</td>
<td>50</td>
<td>3788a</td>
<td>CHINA</td>
<td>12-MAY-12 00:00:00</td>
<td>4950</td>
</tr>
</tbody>
</table>
```

This command splits the `DEFAULT others` partition into partitions named `africa` and `others`:

```
ALTER TABLE sales SPLIT PARTITION others VALUES ('SOUTH AFRICA', 'KENYA') INTO (PARTITION africa, PARTITION others);
```

The partitions now include the `africa` partition along with the `DEFAULT others` partition:

```
| partition_name |       high_value |
|----------------+-----------------|
| EUROPE         | 'FRANCE', 'ITALY'
| ASIA           | 'INDIA', 'PAKISTAN'
| AMERICAS       | 'US', 'CANADA'
| AFRICA         | 'SOUTH AFRICA', 'KENYA'
| OTHERS         | DEFAULT         |
```

This command shows that the rows were redistributed across the new partitions:

```
<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_americas</td>
<td>30</td>
<td>9519b</td>
<td>US</td>
<td>12-APR-12 00:00:00</td>
<td>145000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>30</td>
<td>7588b</td>
<td>CANADA</td>
<td>14-DEC-12 00:00:00</td>
<td>50000</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788a</td>
<td>INDIA</td>
<td>01-MAR-12 00:00:00</td>
<td>75000</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788a</td>
<td>PAKISTAN</td>
<td>04-JUN-12 00:00:00</td>
<td>37500</td>
</tr>
<tr>
<td>sales_others</td>
<td>40</td>
<td>4519b</td>
<td>SOUTH AFRICA</td>
<td>08-APR-12 00:00:00</td>
<td>120000</td>
</tr>
<tr>
<td>sales_others</td>
<td>40</td>
<td>4519b</td>
<td>KENYA</td>
<td>08-APR-12 00:00:00</td>
<td>120000</td>
</tr>
<tr>
<td>sales_others</td>
<td>50</td>
<td>3788a</td>
<td>CHINA</td>
<td>12-MAY-12 00:00:00</td>
<td>4950</td>
</tr>
</tbody>
</table>
11.6.4.2 Defining a MAXVALUE partition

A **MAXVALUE** partition or subpartition captures any rows that don't fit into any other partition in a range-partitioned or subpartitioned table. If you don't include a **MAXVALUE** rule, any row that exceeds the maximum limit specified by the partitioning rules causes an error. Each partition or subpartition can have its own **MAXVALUE** partition.

The syntax of a **MAXVALUE** rule is:

```
PARTITION [<partition_name>] VALUES LESS THAN (MAXVALUE)
```

Where **partition_name** specifies the name of the partition that stores any rows that don't match the rules specified for other partitions.

This example created a range-partitioned table in which the data was partitioned based on the value of the **date** column. If you attempt to add a row with a **date** value that exceeds a date listed in the partitioning constraints, EDB Postgres Advanced Server reports an error.

```
edb$ CREATE TABLE sales (
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY RANGE(date)
(
    PARTITION q1_2012 VALUES LESS THAN('2012-Apr-01'),
    PARTITION q2_2012 VALUES LESS THAN('2012-Jul-01'),
    PARTITION q3_2012 VALUES LESS THAN('2012-Oct-01'),
    PARTITION q4_2012 VALUES LESS THAN('2013-Jan-01'),
    PARTITION others VALUES LESS THAN (MAXVALUE)
)
```

To test the **MAXVALUE** partition, add a row with a value in the **date** column that exceeds the last date value listed in a partitioning rule. The server stores the row in the **others** partition.

```
edb$ INSERT INTO sales VALUES
    (48, '3000x', 'IRELAND', '01-Mar-2013',
    'IRELAND', '3000x');
```

```
ERROR: no partition of relation "sales" found for input values
DETAIL: Partition key of the failing row contains (date) = (01-MAR-13 00:00:00).
```

This **CREATE TABLE** command creates the same table but with a **MAXVALUE** partition. Instead of throwing an error, the server stores any rows that don’t match the previous partitioning constraints in the **others** partition.

```
edb$ CREATE TABLE sales
    (dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number)
PARTITION BY RANGE(date)
(
    PARTITION q1_2012 VALUES LESS THAN('2012-Apr-01'),
    PARTITION q2_2012 VALUES LESS THAN('2012-Jul-01'),
    PARTITION q3_2012 VALUES LESS THAN('2012-Oct-01'),
    PARTITION q4_2012 VALUES LESS THAN('2013-Jan-01'),
    PARTITION others VALUES LESS THAN (MAXVALUE)
)
```

To test the **MAXVALUE** partition, add a row with a value in the **date** column that exceeds the last date value listed in a partitioning rule. The server stores the row in the **others** partition.

```
edb$ SELECT tableoid::regclass, * FROM sales;
```

Query the contents of the **sales** table to confirm that the previously rejected row is now stored in the **sales_others** partition:

```
edb$ SELECT tableoid::regclass, * FROM sales;
```

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_q1_2012</td>
<td>18</td>
<td>4510b</td>
<td>FRANCE</td>
<td>17-JAN-12 00:00:00</td>
<td>45080</td>
</tr>
<tr>
<td>sales_q1_2012</td>
<td>28</td>
<td>3788a</td>
<td>INDIA</td>
<td>01-MAR-12 00:00:00</td>
<td>75080</td>
</tr>
<tr>
<td>sales_q2_2012</td>
<td>38</td>
<td>9510b</td>
<td>CANADA</td>
<td>01-FEB-12 00:00:00</td>
<td>75080</td>
</tr>
<tr>
<td>sales_q2_2012</td>
<td>48</td>
<td>9510b</td>
<td>US</td>
<td>12-APR-12 00:00:00</td>
<td>145080</td>
</tr>
<tr>
<td>sales_q2_2012</td>
<td>28</td>
<td>3788a</td>
<td>PAKISTAN</td>
<td>04-JUN-12 00:00:00</td>
<td>375080</td>
</tr>
<tr>
<td>sales_q2_2012</td>
<td>38</td>
<td>4510b</td>
<td>CANADA</td>
<td>08-APR-12 00:00:00</td>
<td>120800</td>
</tr>
<tr>
<td>sales_q2_2012</td>
<td>48</td>
<td>3788a</td>
<td>US</td>
<td>12-MAY-12 00:00:00</td>
<td>49500</td>
</tr>
</tbody>
</table>
EDB Postgres Advanced Server doesn't have a way to reassign the contents of a MAXVALUE partition or subpartition.

- You can't use the `ALTER TABLE... ADD PARTITION` statement to add a partition to a table with a MAXVALUE rule. However, you can use the `ALTER TABLE... SPLIT PARTITION` statement to split an existing partition.
- You can't use the `ALTER TABLE... ADD SUBPARTITION` statement to add a subpartition to a table with a MAXVALUE rule. However, you can split an existing subpartition with the `ALTER TABLE... SPLIT SUBPARTITION` statement.

### 11.6.5 Specifying multiple partitioning keys in a RANGE partitioned table

You can often improve performance by specifying multiple key columns for a RANGE partitioned table. If you often select rows using comparison operators on a small set of columns based on a greater-than or less-than value, consider using those columns in RANGE partitioning rules.

Range-partitioned table definitions can include multiple columns in the partitioning key. To specify multiple partitioning keys for a range-partitioned table, include the column names in a comma-separated list after the `PARTITION BY RANGE` clause:

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  sale_year number,
  sale_month number,
  sale_day number,
  amount number
) PARTITION BY RANGE (sale_year, sale_month)
(
  PARTITION p1_2012 VALUES LESS THAN(2012, 4),
  PARTITION p2_2012 VALUES LESS THAN(2012, 7),
  PARTITION p3_2012 VALUES LESS THAN(2012, 10),
  PARTITION p4_2012 VALUES LESS THAN(2013, 1)
);
```

If a table is created with multiple partitioning keys, you must specify multiple key values when querying the table to take full advantage of partition pruning:

```sql
edb=# EXPLAIN SELECT * FROM sales WHERE sale_year = 2012 AND sale_month = 8;
```

```
<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append (cost=0.00..14.35 rows=1 width=256)</td>
</tr>
<tr>
<td>-&gt; Seq Scan on sales_q3_2012 (cost=0.00..14.35 rows=1 width=256)</td>
</tr>
<tr>
<td>Filter: ((sale_year = '2012':numeric) AND (sale_month = '8':numeric))</td>
</tr>
</tbody>
</table>
```

Since all rows with a value of 8 in the `sale_month` column and a value of 2012 in the `sale_year` column are stored in the `q3_2012` partition, EDB Postgres Advanced Server searches only that partition.

### 11.6.6 Retrieving information about a partitioned table

EDB Postgres Advanced Server provides five system catalog views that you can use to view information about the structure of partitioned tables.
Querying the partitioning views

You can query the following views to retrieve information about partitioned and subpartitioned tables:

- `ALL_PART_TABLES`
- `ALL_TAB_PARTITIONS`
- `ALL_TAB_SUBPARTITIONS`
- `ALL_PART_KEY_COLUMNS`
- `ALL_SUBPART_KEY_COLUMNS`

The structure of each view is explained in Table partitioning views reference. If you're using the EDB-PSQL client, you can also learn about the structure of a view by entering:

```
\d <view_name>
```

Where `view_name` specifies the name of the table partitioning view.

Querying a view can provide information about the structure of a partitioned or subpartitioned table. For example, this code displays the names of a subpartitioned table:

```
subpartition_name | partition_name
-------------------+----------------
EUROPE_2011       | EUROPE
EUROPE_2012       | EUROPE
ASIA_2011         | ASIA
ASIA_2012         | ASIA
AMERICAS_2011     | AMERICAS
AMERICAS_2012     | AMERICAS
(6 rows)
```

11.7 Optimizing code

EDB Postgres Advanced Server includes features designed to help application programmers address database performance problems. SQL Profiler helps you locate and optimize poorly running SQL code. You can use optimizer hints to influence the server as it selects a query plan when you invoke a DELETE, INSERT, SELECT, or UPDATE command.

11.7.1 Using optimizer hints

When you invoke a `DELETE`, `INSERT`, `SELECT`, or `UPDATE` command, the server generates a set of execution plans. After analyzing those execution plans, the server selects a plan that generally returns the result set in the least amount of time. The server's choice of plan depends on several factors:

- The estimated execution cost of data handling operations
- Parameter values assigned to parameters in the Query Tuning section of the `postgresql.conf` file
- Column statistics that were gathered by the ANALYZE command

As a rule, the query planner selects the least expensive plan. You can use an optimizer hint to influence the server as it selects a query plan.

11.7.1.1 About optimizer hints

An optimizer hint is one or more directives embedded in a comment-like syntax that immediately follows a `DELETE`, `INSERT`, `SELECT`, or `UPDATE` command. Keywords in the comment instruct the server to use or avoid a specific plan when producing the result set.

**Synopsis**

```
{ DELETE | INSERT | SELECT | UPDATE } /*+ { <hint> [ <comment> ] } [ ... ] */
```

```
{ DELETE | INSERT | SELECT | UPDATE } --> { <hint> [ <comment> ] }
```

In both forms, a plus sign (+) must immediately follow the `/*` or `-->` opening comment symbols, with no intervening space. Otherwise the server doesn't interpret the tokens that follow as hints.
If you’re using the first form, the hint and optional comment might span multiple lines. In the second form, all hints and comments must occupy a single line. The rest of the statement must start on a new line.

Description

Note:

- The database server always tries to use the specified hints if at all possible.
- If a planner method parameter is set so as to disable a certain plan type, then this plan isn’t used even if it’s specified in a hint, unless there are no other possible options for the planner. Examples of planner method parameters are `enable_indexscan`, `enable_seqscan`, `enable_hashjoin`, `enable_mergejoin`, and `enable_nestloop`. These are all Boolean parameters.
- The hint is embedded in a comment. As a consequence, if the hint is misspelled or if any parameter to a hint, such as view, table, or column name, is misspelled or nonexistent in the SQL command, there’s no indication that an error occurred. No syntax error is given. The entire hint is silently ignored.
- If an alias is used for a table name in the SQL command, then you must use the alias name in the hint, not the original table name. For example, in the command `SELECT /*+ FULL(acct) */ * FROM accounts acct ..., acct`, you must specify the alias for `accounts` in the `FULL` hint, not in the table name `accounts`.

Use the `EXPLAIN` command to ensure that the hint is correctly formed and the planner is using the hint.

In general, don’t use optimizer hints in a production application, where table data changes throughout the life of the application. By ensuring that dynamic columns are analyzed frequently via the `ANALYZE` command, the column statistics are updated to reflect value changes. The planner uses such information to produce the lowest-cost plan for any given command execution. Use of optimizer hints defeats the purpose of this process and results in the same plan regardless of how the table data changes.

Parameters

- `hint`: An optimizer hint directive.
- `comment`: A string with additional information. Comments have restrictions as to what characters you can include. Generally, `comment` can consist only of alphabetic, numeric, the underscore, dollar sign, number sign, and space characters. These must also conform to the syntax of an identifier. Any subsequent hint is ignored if the comment isn’t in this form.
- `statement_body`: The remainder of the `DELETE`, `INSERT`, `SELECT`, or `UPDATE` command.

11.7.1.2 Default optimization modes

You can choose an optimization mode as the default setting for an EDB Postgres Advanced Server database cluster. You can also change this setting on a per-session basis by using the `ALTER SESSION` command as well as in individual `DELETE`, `SELECT`, and `UPDATE` commands in an optimizer hint. The configuration parameter that controls these default modes is `OPTIMIZER_MODE`.

The table shows the possible values.

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ALL_ROWS</code></td>
<td>Optimizes for retrieving all rows of the result set.</td>
</tr>
<tr>
<td><code>CHOOSE</code></td>
<td>Does no default optimization based on assumed number of rows to retrieve from the result set. This is the default.</td>
</tr>
<tr>
<td><code>FIRST_ROWS</code></td>
<td>Optimizes for retrieving only the first row of the result set.</td>
</tr>
<tr>
<td><code>FIRST_ROWS_10</code></td>
<td>Optimizes for retrieving the first 10 rows of the results set.</td>
</tr>
<tr>
<td><code>FIRST_ROWS_100</code></td>
<td>Optimizes for retrieving the first 100 rows of the result set.</td>
</tr>
<tr>
<td><code>FIRST_ROWS_1000</code></td>
<td>Optimizes for retrieving the first 1000 rows of the result set.</td>
</tr>
<tr>
<td><code>FIRST_ROWS(n)</code></td>
<td>Optimizes for retrieving the first $n$ rows of the result set. You can’t use this form as the object of the <code>ALTER SESSION SET OPTIMIZER_MODE</code> command. You can use it only in the form of a hint in a SQL command.</td>
</tr>
</tbody>
</table>

These optimization modes are based on the assumption that the client submitting the SQL command is interested in viewing only the first $n$ rows of the result set and not the remainder of the result set. Resources allocated to the query are adjusted as such.

Example: Specifying the number of rows to retrieve in the result set
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10;

### Example: Showing the current value of the OPTIMIZER_MODE parameter

You can show the current value of the OPTIMIZER_MODE parameter by using the SHOW command. This command depends on the utility. In PSQL, use the SHOW command as follows:

```sql
SHOW OPTIMIZER_MODE;
```

<table>
<thead>
<tr>
<th>optimizer_mode</th>
<th>first_rows_10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1 row)</td>
</tr>
</tbody>
</table>

The SHOW command compatible with Oracle databases has the following syntax:

```sql
SHOW PARAMETER OPTIMIZER_MODE;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimizer_mode</td>
<td>first_rows_10</td>
</tr>
</tbody>
</table>

This example shows an optimization mode used in a SELECT command as a hint:

```sql
SELECT /*+ FIRST_ROWS(7) */ * FROM emp;
```

11.7.1.3 Access method hints

The following hints influence how the optimizer accesses relations to create the result set.

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL(table)</td>
<td>Perform a full sequential scan on table.</td>
</tr>
<tr>
<td>INDEX(table [ index ] [...])</td>
<td>Use index on table to access the relation.</td>
</tr>
<tr>
<td>NO_INDEX(table [ index ] [...])</td>
<td>Don't use index on table to access the relation.</td>
</tr>
</tbody>
</table>

In addition, you can use the ALL_ROWS, FIRST_ROWS, and FIRST_ROWS(n) hints.

- **INDEX** and **NO_INDEX** hints for the partitioned table internally expand to include the corresponding inherited child indexes and apply in later processing.

About the examples
The sample application doesn't have enough data to show the effects of optimizer hints. Thus the remainder of these examples use a banking database created by the `pgbench` application located in the EDB Postgres Advanced Server `bin` subdirectory.

Example: Create a sample database and tables

The following steps create a database named, `bank` populated by the tables `pgbench_accounts`, `pgbench_branches`, `pgbench_tellers`, and `pgbench_history`. The `-s 20` option specifies a scaling factor of 20, which results in the creation of 20 branches. Each branch has 100,000 accounts. The result is a total of 2,000,000 rows in the `pgbench_accounts` table and 20 rows in the `pgbench_branches` table. Ten tellers are assigned to each branch resulting, in a total of 200 rows in the `pgbench_tellers` table.

The following initializes the `pgbench` application in the `bank` database.

```
createdb -U enterprisedb bank
```

```
CREATE DATABASE
pgbench -i -s 20 -U enterprisedb bank
```

```
NOTICE: table "pgbench_history" does not exist,
skipping
NOTICE: table "pgbench_tellers" does not exist,
skipping
NOTICE: table "pgbench_accounts" does not exist,
skipping
NOTICE: table "pgbench_branches" does not exist,
skipping
creating tables...
100000 of 2000000 tuples (5%) done (elapsed 8.11 s, remaining 2.10 s)
200000 of 2000000 tuples (10%) done (elapsed 8.22 s, remaining 1.98 s)
300000 of 2000000 tuples (15%) done (elapsed 8.33 s, remaining 1.84 s)
400000 of 2000000 tuples (20%) done (elapsed 8.42 s, remaining 1.67 s)
500000 of 2000000 tuples (25%) done (elapsed 8.52 s, remaining 1.57 s)
600000 of 2000000 tuples (30%) done (elapsed 8.62 s, remaining 1.45 s)
700000 of 2000000 tuples (35%) done (elapsed 8.73 s, remaining 1.35 s)
800000 of 2000000 tuples (40%) done (elapsed 8.87 s, remaining 1.31 s)
900000 of 2000000 tuples (45%) done (elapsed 8.98 s, remaining 1.19 s)
1000000 of 2000000 tuples (50%) done (elapsed 9.09 s, remaining 1.09 s)
1100000 of 2000000 tuples (55%) done (elapsed 9.22 s, remaining 1.08 s)
1200000 of 2000000 tuples (60%) done (elapsed 9.36 s, remaining 0.91 s)
1300000 of 2000000 tuples (65%) done (elapsed 9.51 s, remaining 0.82 s)
1400000 of 2000000 tuples (70%) done (elapsed 9.65 s, remaining 0.71 s)
1500000 of 2000000 tuples (75%) done (elapsed 9.78 s, remaining 0.59 s)
1600000 of 2000000 tuples (80%) done (elapsed 9.93 s, remaining 0.48 s)
1700000 of 2000000 tuples (85%) done (elapsed 2.18 s, remaining 0.37 s)
1800000 of 2000000 tuples (90%) done (elapsed 2.23 s, remaining 0.25 s)
1900000 of 2000000 tuples (95%) done (elapsed 2.37 s, remaining 0.12 s)
2000000 of 2000000 tuples (100%) done (elapsed 2.48 s, remaining 0.00 s)
vacuum...
set primary keys...
done.
```

A total of 500,000 transactions are then processed. These transactions populate the `pgbench_history` table with 500,000 rows.

```
pbench -U enterprisedb -t 500000 bank
```

```
starting vacuum..end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 20
query mode: simple
number of clients: 1
```
The following are the table definitions:

```
\d pgbench_accounts

  Table "public pgbench_accounts"
  Column | Type     | Modifiers
          |-----------+-----------
  aid     | integer   | not null
  bid     | integer
  abalance | integer
  filler  | character(84)

Indexes:
  "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
```

```
\d pgbench_branches

  Table "public pgbench_branches"
  Column | Type     | Modifiers
          |-----------+-----------
  bid     | integer   | not null
  bbalance | integer
  filler  | character(88)

Indexes:
  "pgbench_branches_pkey" PRIMARY KEY, btree (bid)
```

```
\d pgbench_tellers

  Table "public pgbench_tellers"
  Column | Type     | Modifiers
          |-----------+-----------
  tid     | integer   | not null
  bid     | integer
  tbalance | integer
  filler  | character(84)

Indexes:
  "pgbench_tellers_pkey" PRIMARY KEY, btree (tid)
```

```
\d pgbench_history

  Table "public pgbench_history"
  Column | Type
          |-----------
  tid     | integer
  bid     | integer
  aid     | integer
  delta   | integer
  mtime   | timestamp without time zone
  filler  | character(22)
```

The `EXPLAIN` command shows the plan selected by the query planner. In this example, `aid` is the primary key column, so an indexed search is used on index `pgbench_accounts_pkey`.
Example: FULL hint

The `FULL` hint forces a full sequential scan instead of using the index:

```sql
EXPLAIN SELECT * FROM pgbench_accounts WHERE aid = 100;
```

```
QUERY PLAN

Index Scan using pgbench_accounts_pkey on pgbench_accounts
(cost=8.43..8.45 rows=1 width=97)
  Index Cond: (aid = 100)
(2 rows)
```

Example: NO_INDEX hint

The `NO_INDEX` hint forces a parallel sequential scan instead of using the index:

```sql
EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_pkey) */ * FROM pgbench_accounts WHERE aid = 100;
```

```
QUERY PLAN

Gather (cost=1000.00..45094.80 rows=1 width=97)
Workers Planned: 2
  -> Parallel Seq Scan on pgbench_accounts (cost=0.00..44094.70 rows=1 width=97)
    Filter: (aid = 100)
(4 rows)
```

Example: Tracing optimizer hints

You can obtain more detailed information than the `EXPLAIN` command provides about whether the planner used a hint. To do so, set the `trace_hints` configuration parameter as follows:

```sql
SET trace_hints TO on;
```

The `SELECT` command with the `NO_INDEX` hint shows the additional information produced when you set the `trace_hints` configuration parameters:

```sql
EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_pkey) */ * FROM pgbench_accounts WHERE aid = 108;
```

```
INFO: [HINTS] Index Scan of [pgbench_accounts]. [pgbench_accounts_pkey] rejected due to NO_INDEX hint.
QUERY PLAN

Gather (cost=1000.00..45094.88 rows=1 width=97)
```
Example: Hint ignored

A hint is ignored, the `INFO: [HINTS]` line doesn't appear. This might indicate a syntax error or some other misspelling in the hint. In this example, the index name is misspelled.

```sql
EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_xxx) */ * FROM pgbench_accounts WHERE aid = 108;
```

Example: INDEX hint for the partitioned table

```sql
CREATE TABLE t_1384(col1 int, col2 int, col3 int) PARTITION BY RANGE (col1) (PARTITION p1 VALUES LESS THAN (500), PARTITION p2 VALUES LESS THAN (1000));
ALTER TABLE t_1384 ADD PRIMARY KEY(col1);
CREATE INDEX idx1 ON t_1384(col2);
CREATE INDEX idx2 ON t_1384(col1, col2);
SET enable_hints = true;
SET trace_hints TO on;
-- Use primary index
EXPLAIN (COSTS OFF) SELECT /*+ INDEX(s t_1384_pkey) */ * FROM t_1384 s WHERE col2 = 10;
```

INFO: [HINTS] SeqScan of [s] rejected due to INDEX hint.
INFO: [HINTS] Parallel SeqScan of [s] rejected due to INDEX hint.
INFO: [HINTS] Index Scan of [s].[t_1384_p1_col1_col2_idx] rejected due to INDEX hint.
INFO: [HINTS] Index Scan of [s].[t_1384_p1_col1_col2_idx] rejected due to INDEX hint.
INFO: [HINTS] SeqScan of [s] rejected due to INDEX hint.
INFO: [HINTS] Parallel SeqScan of [s] rejected due to INDEX hint.
INFO: [HINTS] Index Scan of [s].[t_1384_p2_col1_col2_idx] rejected due to INDEX hint.
INFO: [HINTS] Index Scan of [s].[t_1384_p2_col1_col2_idx] rejected due to INDEX hint.
INFO: [HINTS] Index Scan of [s].[t_1384_p2_pkey] accepted.

QUERY PLAN

Append
-> Bitmap Heap Scan on t_1384_p1
s_1
  Recheck Cond: (col2 = 10)
-> Bitmap Index Scan on t_1384_p1_col2_idx
11.7.1.4 Specifying a join order

Include the `ORDERED` directive to instruct the query optimizer to join tables in the order in which they're listed in the `FROM` clause. If you don't include the `ORDERED` keyword, the query optimizer chooses the order in which to join the tables.

For example, the following command allows the optimizer to choose the order in which to join the tables listed in the `FROM` clause:

```sql
SELECT e.ename, d.dname, h.startdate
FROM emp e, dept d, jobhist h
WHERE d.deptno = e.deptno
AND h.empno = e.empno;
```

The following command instructs the optimizer to join the tables in the order specified:

```sql
SELECT e.ename, d.dname, h.startdate
FROM emp e, dept d, jobhist h
WHERE d.deptno = e.deptno
AND h.empno = e.empno;
```

In the `ORDERED` version of the command, EDB Postgres Advanced Server first joins `emp e` with `dept d` before joining the results with `jobhist h`. Without the `ORDERED` directive, the query optimizer selects the join order.

**Note**

The `ORDERED` directive doesn't work for Oracle-style outer joins (joins that contain a `+` sign).

11.7.1.5 Joining relations hints

When you join two tables, you can use any of three plans to perform the join.

- Nested loop join — A table is scanned once for every row in the other joined table.
- Merge sort join — Each table is sorted on the join attributes before the join starts. The two tables are then scanned in parallel, and the matching rows are combined to form the join rows.
- Hash join — A table is scanned and its join attributes are loaded into a hash table using its join attributes as hash keys. The other joined table is then scanned and its join attributes are used as hash keys to locate the matching rows from the first table.

List of optimizer hints for join plans

The following table lists the optimizer hints that you can use to influence the planner to use one type of join plan over another.

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USE_HASH(table [...] )</td>
<td>Use a hash join for table.</td>
</tr>
<tr>
<td>NO_USE_HASH(table [...])</td>
<td>Don't use a hash join for table.</td>
</tr>
<tr>
<td>USE_MERGE(table [...] )</td>
<td>Use a merge sort join for table.</td>
</tr>
<tr>
<td>NO_USE_MERGE(table [...])</td>
<td>Don't use a merge sort join for table.</td>
</tr>
<tr>
<td>USE_NL(table [...] )</td>
<td>Use a nested loop join for table.</td>
</tr>
<tr>
<td>NO_USE_NL(table [...] )</td>
<td>Don't use a nested loop join for table.</td>
</tr>
</tbody>
</table>
Example: Hash join

In this example, the **USE_HASH** hint is used for a join on the `pgbench_branches` and `pgbench_accounts` tables. The query plan shows that a hash join is used by creating a hash table from the join attribute of the `pgbench_branches` table:

```sql
EXPLAIN SELECT /*+ USE_HASH(b) */ b.bid, a.aid, abalance FROM pgbench_branches b, pgbench_accounts a WHERE b.bid = a.bid;
```

**QUERY PLAN**

```
Hash Join (cost=21.45..81463.06 rows=2014215 width=12)
Hash Cond: (a.bid = b.bid)
-> Seq Scan on pgbench_accounts a (cost=0.00..53746.15 rows=2014215 width=12)
-> Hash (cost=21.20..21.20 rows=20 width=4)
  -> Seq Scan on pgbench_branches b (cost=0.00..21.20 rows=20 width=4)
```

Next, the **NO_USE_HASH(a b)** hint forces the planner to use an approach other than hash tables. The result is a merge join.

```sql
EXPLAIN SELECT /*+ NO_USE_HASH(a b) */ b.bid, a.aid, abalance FROM pgbench_branches b, pgbench_accounts a WHERE b.bid = a.bid;
```

**QUERY PLAN**

```
Merge Join (cost=333526.08..368774.94 rows=2014215 width=12)
Merge Cond: (b.bid = a.bid)
  -> Sort (cost=21.63..21.68 rows=20 width=4)
    Sort Key: b.bid
  -> Seq Scan on pgbench_branches b (cost=0.00..21.20 rows=20 width=4)
  -> Materialize (cost=333504.45..338539.99 rows=2014215 width=12)
    Sort Key: a.bid
    -> Seq Scan on pgbench_accounts a (cost=0.00..53746.15 rows=2014215 width=12)
```

Finally, the **USE_MERGE** hint forces the planner to use a merge join:

```sql
EXPLAIN SELECT /*+ USE_MERGE(a) */ b.bid, a.aid, abalance FROM pgbench_branches b, pgbench_accounts a WHERE b.bid = a.bid;
```

**QUERY PLAN**

```
Merge Join (cost=333526.08..368774.94 rows=2014215 width=12)
Merge Cond: (b.bid = a.bid)
  -> Sort (cost=21.63..21.68 rows=20 width=4)
    Sort Key: b.bid
  -> Seq Scan on pgbench_branches b (cost=0.00..21.20 rows=20 width=4)
  -> Materialize (cost=333504.45..338539.99 rows=2014215 width=12)
    Sort Key: a.bid
    -> Seq Scan on pgbench_accounts a (cost=0.00..53746.15 rows=2014215 width=12)
```

Example: Three-table join

In this three-table join example, the planner first performs a hash join on the `pgbench_branches` and `pgbench_history` tables. Then it performs a hash join of the result with the `pgbench_accounts` table.

```sql
EXPLAIN SELECT h.mtime, h.delta, b.bid, a.aid FROM pgbench_history h, pgbench_branches b, pgbench_accounts a WHERE h.bid = b.bid AND h.aid = a.aid;
```

**QUERY PLAN**

```
--------
Hash Join (cost=426667.53..1888344.31 rows=2014215 width=24)
Hash Cond: (a.aid = h.aid)
  -> Sort (cost=21.63..21.68 rows=20 width=4)
    Sort Key: a.aid
  -> Seq Scan on pgbench_accounts a (cost=0.00..53746.15 rows=2014215 width=12)
  -> Hash (cost=19.99..19.99 rows=20 width=4)
    -> Seq Scan on pgbench_history h (cost=0.00..19.99 rows=20 width=12)
    -> Hash (cost=1.99..1.99 rows=20 width=4)
      -> Seq Scan on pgbench_branches b (cost=0.00..1.99 rows=20 width=4)
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
This plan is altered by using hints to force a combination of a merge sort join and a hash join:

```sql
EXPLAIN SELECT /*+ USE_MERGE(h b) USE_HASH(a) */ h.mtime, h.delta, b.bid, a.aid FROM pgbench_history h, pgbench_branches b, pgbench_accounts a WHERE h.bid = b.bid AND h.aid = a.aid;
```

### 11.7.1.6 Global hints

In addition to applying hints directly to tables that are referenced in the SQL command, you can apply hints to tables that appear in a view when the view is referenced in the SQL command. The hint doesn't appear in the view but in the SQL command that references the view.

When specifying a hint that applies to a table in a view, give the view and table names in dot notation in the hint argument list.

#### Synopsis

```
<hint>(<view>.<table>)
```

#### Parameters

- **hint**
  
  Any of the hints in the table Access method hints, Joining relations hints.

- **view**
  
  The name of the view containing table.

- **table**
  
  The table on which to apply the hint.

#### Example: Applying hints to a stored view

A view named `tx` is created from the three-table join of `pgbench_history`, `pgbench_branches`, and `pgbench_accounts`, shown in the last example of Joining relations hints.
CREATE VIEW tx AS SELECT h.mtime, h.delta, b.bid, a.aid FROM pgbench_history h, pgbench_branches b, pgbench_accounts a WHERE h.bid = b.bid AND h.aid = a.aid;

The query plan produced by selecting from this view is:

```
EXPLAIN SELECT * FROM tx;
```

```
<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Join (cost=86814.29..123103.29 rows=500000 width=20)</td>
</tr>
<tr>
<td>Hash Cond: (h.aid = a.aid)</td>
</tr>
<tr>
<td>-&gt; Hash Join (cost=21.45..15081.45 rows=500000 width=20)</td>
</tr>
<tr>
<td>Hash Cond: (h.bid = b.bid)</td>
</tr>
<tr>
<td>-&gt; Hash (cost=21.20..21.20 rows=20 width=4)</td>
</tr>
<tr>
<td>-&gt; Seq Scan on pgbench_history h (cost=0.00..8185.00 rows=500000 width=20)</td>
</tr>
<tr>
<td>-&gt; Seq Scan on pgbench_branches b (cost=0.00..21.20 rows=20 width=4)</td>
</tr>
<tr>
<td>-&gt; Hash (cost=53746.15..53746.15 rows=2014215 width=4)</td>
</tr>
<tr>
<td>-&gt; Seq Scan on pgbench_accounts a (cost=0.08..53746.15 rows=2014215 width=4)</td>
</tr>
</tbody>
</table>
```

The same hints that were applied to this join at the end of Joining relations hints can be applied to the view:

```
EXPLAIN SELECT /*+ USE_MERGE(tx.h tx.b) USE_HASH(tx.a) */ * FROM tx;
```

```
<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Join (cost=152583.39..182562.49 rows=500000 width=20)</td>
</tr>
<tr>
<td>Hash Cond: (h.aid = a.aid)</td>
</tr>
<tr>
<td>-&gt; Merge Join (cost=65790.55..74540.65 rows=500000 width=20)</td>
</tr>
<tr>
<td>Merge Cond: (b.bid = h.bid)</td>
</tr>
<tr>
<td>-&gt; Sort (cost=21.63..21.68 rows=20 width=4)</td>
</tr>
<tr>
<td>Sort Key: b.bid</td>
</tr>
<tr>
<td>-&gt; Seq Scan on pgbench_branches b (cost=0.00..21.20 rows=20 width=4)</td>
</tr>
<tr>
<td>-&gt; Materialize (cost=65768.92..68268.92 rows=500000 width=20)</td>
</tr>
<tr>
<td>-&gt; Sort (cost=65768.92..67018.92 rows=500000 width=20)</td>
</tr>
<tr>
<td>Sort Key: h.bid</td>
</tr>
<tr>
<td>-&gt; Seq Scan on pgbench_history h (cost=0.00..8185.00 rows=500000 width=20)</td>
</tr>
<tr>
<td>-&gt; Seq Scan on pgbench_accounts a (cost=0.08..53746.15 rows=2014215 width=4)</td>
</tr>
</tbody>
</table>
```

Applying hints to tables in subqueries

In addition to applying hints to tables in stored views, you can apply hints to tables in subqueries. In this query on the sample application emp table, employees and their managers are listed by joining the emp table with a subquery of the emp table identified by the alias b:

```
SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename" FROM emp a,
(SELECT * FROM emp) b WHERE a.mgr = b.empno;
```

```
<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>mgr empno</th>
<th>mgr ename</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>7902</td>
<td>FORD</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>7788</td>
<td>SCOTT</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>7782</td>
<td>CLARK</td>
</tr>
</tbody>
</table>
```

This code shows the plan chosen by the query planner:
You can apply a hint to the `emp` table in the subquery to perform an index scan on index `emp_pk` instead of a table scan. Note the difference in the query plans.

EXPLAIN SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename" FROM emp a, (SELECT * FROM emp) b WHERE a.mgr = b.empno;

QUERY PLAN
Hash Join (cost=1.32..2.64 rows=13 width=22)
Hash Cond: (a.mgr = emp.empno)
-> Seq Scan on emp a (cost=0.00..1.14 rows=14 width=16)
-> Hash (cost=1.14..1.14 rows=14 width=11)
  -> Seq Scan on emp (cost=0.00..1.14 rows=14 width=11)
(5 rows)

EXPLAIN SELECT /*+ INDEX(b.emp emp_pk) */ a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename" FROM emp a, (SELECT * FROM emp) b WHERE a.mgr = b.empno;

QUERY PLAN
Merge Join (cost=4.17..13.11 rows=13 width=22)
Merge Cond: (a.mgr = emp.empno)
-> Sort (cost=1.41..1.44 rows=14 width=16)
  Sort Key: a.mgr
  -> Seq Scan on emp a (cost=0.00..1.14 rows=14 width=16)
-> Index Scan using emp_pk on emp (cost=0.14..12.35 rows=14 width=11)
(6 rows)

11.7.1.7 APPEND optimizer hint

By default, EDB Postgres Advanced Server adds new data into the first available free space in a table vacated by vacuumed records. Include the `APPEND` directive after an `INSERT` or `SELECT` command to bypass midtable free space and affix new rows to the end of the table. This optimizer hint can be particularly useful when bulk loading data.

The syntax is:

```sql
/*+APPEND*/
```

For example, the following command, compatible with Oracle databases, instructs the server to append the data in the `INSERT` statement to the end of the `sales` table:

```sql
INSERT /*+APPEND*/ INTO sales VALUES
(10, 10, '01-Mar-2011', 10, 'OR');
```

EDB Postgres Advanced Server supports the `APPEND` hint when adding multiple rows in a single `INSERT` statement:

```sql
INSERT /*+APPEND*/ INTO sales VALUES
(20, 20, '01-Aug-2011', 20, 'NY'),
(30, 30, '01-Feb-2011', 30, 'FL'),
(40, 40, '01-Nov-2011', 40, 'TX');
```

You can also include the `APPEND` hint in the `SELECT` clause of an `INSERT INTO` statement:

```sql
INSERT INTO sales_history SELECT /*+APPEND*/ FROM sales;
```

11.7.1.8 Parallelism hints

Parallel scanning is the use of multiple background workers to simultaneously perform a scan of a table, that is, in parallel, for a given query. This process provides performance improvement over other methods such as the sequential scan.

- The `PARALLEL` optimizer hint forces parallel scanning.
- The `NO_PARALLEL` optimizer hint prevents use of a parallel scan.

Synopsis
Parameters

**table**

The to which to apply the parallel hint.

**parallel_degree | DEFAULT**

*parallel_degree* is a positive integer that specifies the desired number of workers to use for a parallel scan. If specified, the lesser of *parallel_degree* and configuration parameter *max_parallel_workers_per_gather* is used as the planned number of workers. For information on the *max_parallel_workers_per_gather* parameter, see Asynchronous Behavior under Resource Consumption in the PostgreSQL core documentation.

- If you specify *DEFAULT*, then the maximum possible parallel degree is used.
- If you omit both *parallel_degree* and *DEFAULT*, then the query optimizer determines the parallel degree. In this case, if *table* was set with the *parallel_workers* storage parameter, then this value is used as the parallel degree. Otherwise, the optimizer uses the maximum possible parallel degree as if *DEFAULT* were specified. For information on the *parallel_workers* storage parameter, see Storage Parameters under CREATE TABLE in the PostgreSQL core documentation.

Regardless of the circumstance, the parallel degree never exceeds the setting of configuration parameter *max_parallel_workers_per_gather*.

About the examples

For these examples, the following configuration parameter settings are in effect:

```sql
SHOW max_worker_processes;
```

```sql
<table>
<thead>
<tr>
<th>max_worker_processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

```sql
SHOW max_parallel_workers_per_gather;
```

```sql
<table>
<thead>
<tr>
<th>max_parallel_workers_per_gather</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

**Example: Default scan**

This example shows the default scan on table *pgbench_accounts*. A sequential scan is shown in the query plan.

```sql
SET trace_hints TO on;
EXPLAIN SELECT * FROM pgbench_accounts;
```

```
<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq Scan on pgbench_accounts  (cost=0.00..53746.15 rows=2014215 width=97)</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

**Example: PARALLEL hint**

This example uses the **PARALLEL** hint. In the query plan, the Gather node, which launches the background workers, indicates the plan to use two workers:
If `trace_hints` is set to `on`, the `INFO: [HINTS]` lines appear stating that `PARALLEL` was accepted for `pgbench_accounts` and other hint information. For the remaining examples, these lines aren't displayed as they generally show the same output, that is, `trace_hints` was reset to `off`.

```sql
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;
```

```sql
INFO: [HINTS] SeqScan of [pgbench_accounts] rejected due to PARALLEL hint. INFO: [HINTS] PARALLEL on [pgbench_accounts] accepted. INFO: [HINTS] Index Scan of [pgbench_accounts].[pgbench_accounts_pkey] rejected due to PARALLEL hint.
```

QUERY PLAN
```
---------------------------------------------
| Gather (cost=1800.00..244418.06 rows=2814215 width=97) |
| Workers Planned: 2 |
|   -> Parallel Seq Scan on pgbench_accounts (cost=0.00..41996.56 rows=839256 width=97) |
| (3 rows) |
```

Now, the `max_parallel_workers_per_gather` setting is increased:

```sql
SET max_parallel_workers_per_gather TO 6;
SHOW max_parallel_workers_per_gather;
```

```
max_parallel_workers_per_gather
---
6 (1 row)
```

The same query on `pgbench_accounts` is issued again with no parallel degree specification in the `PARALLEL` hint. The number of planned workers has increased to 4, as determined by the optimizer.

```sql
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;
```

QUERY PLAN
```
---------------------------------------------
| Gather (cost=1800.00..241061.04 rows=2814215 width=97) |
| Workers Planned: 4 |
|   -> Parallel Seq Scan on pgbench_accounts (cost=0.00..38639.54 rows=503554 width=97) |
| (3 rows) |
```

Now, a value of `6` is specified for the parallel degree parameter of the `PARALLEL` hint. The planned number of workers is returned as this specified value:

```sql
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts 6) */ * FROM pgbench_accounts;
```

QUERY PLAN
```
---------------------------------------------
| Gather (cost=1800.00..239382.52 rows=2814215 width=97) |
| Workers Planned: 6 |
|   -> Parallel Seq Scan on pgbench_accounts (cost=0.00..36961.03 rows=335702 width=97) |
| (3 rows) |
```

The same query is now issued with the `DEFAULT` setting for the parallel degree. The results indicate that the maximum allowable number of workers is planned.

```sql
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts DEFAULT) */ * FROM pgbench_accounts;
```

QUERY PLAN
```
---------------------------------------------
| Gather (cost=1800.00..239382.52 rows=2814215 width=97) |
| Workers Planned: 6 |
|   -> Parallel Seq Scan on pgbench_accounts (cost=0.00..36961.03 rows=335702 width=97) |
| (3 rows) |
```

Table `pgbench_accounts` is now altered so that the `parallel_workers` storage parameter is set to `3`.

Note
This format of the `ALTER TABLE` command to set the `parallel_workers` parameter isn't compatible with Oracle databases.

The `parallel_workers` setting is shown by the PSQL `\d+` command.

```
ALTER TABLE pgbench_accounts SET
(parallel_workers=3);
```

```
\d+ pgbench_accounts

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
<th>Storage</th>
<th>Stats target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aid</td>
<td>integer</td>
<td>not null</td>
<td>plain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bid</td>
<td>integer</td>
<td></td>
<td>plain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abalance</td>
<td>integer</td>
<td></td>
<td>plain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>filler</td>
<td>character(84)</td>
<td></td>
<td>extended</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Indexes:
  "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Options: fillfactor=100, parallel_workers=3
```

Example: PARALLEL hint is given with no parallel degree

When the `PARALLEL` hint is given with no parallel degree, the resulting number of planned workers is the value from the `parallel_workers` parameter:

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;
```

```
QUERY PLAN
--------------------------------------------------------------------------------------
| Gather (cost=1800.00..242522.97 rows=2814215 width=97) Workers Planned: 3 |
| -> Parallel Seq Scan on pgbench_accounts (cost=0.00..40101.47 rows=649747 width=97) |

(3 rows)
```

Specifying a parallel degree value or `DEFAULT` in the `PARALLEL` hint overrides the `parallel_workers` setting.

Example: NO_PARALLEL hint

This example shows the `NO_PARALLEL` hint. With `trace_hints` set to `on`, the `INFO: [HINTS]` message states that the parallel scan was rejected due to the `NO_PARALLEL` hint.

```
EXPLAIN SELECT /*+ NO_PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;
```

```
INFO: [HINTS] Parallel SeqScan of [pgbench_accounts] rejected due to NO_PARALLEL hint.
QUERY PLAN
--------------------------------------------------------------------------------------
| Seq Scan on pgbench_accounts (cost=0.00..53746.15 rows=2814215 width=97) |

(1 row)
```

### 11.7.1.9 Conflicting hints

If a command includes two or more conflicting hints, the server ignores the contradicting hints. The following table lists hints that contradict each other.

<table>
<thead>
<tr>
<th>Hint</th>
<th>Conflicting hint</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ALL_ROWS</code></td>
<td><code>FIRST_ROWS</code> - all formats</td>
</tr>
<tr>
<td><code>FULL(table)</code></td>
<td><code>INDEX(table [ index ])</code></td>
</tr>
<tr>
<td></td>
<td><code>PARALLEL(table [ degree ])</code></td>
</tr>
</tbody>
</table>
11.7.2 Optimizing inefficient SQL code

Inefficient SQL code is a leading cause of database performance problems. The challenge for database administrators and developers is locating and then optimizing this code in large, complex systems.

See SQL Profiler to see how the utility can help you locate and optimize poorly running SQL code.

12 Working with Oracle data

EDB Postgres Advanced Server makes Postgres look, feel, and operate more like Oracle. So when you migrate, there’s less code to rewrite, and you can be up and running quickly. The Oracle compatibility features allow you to run many applications written for Oracle in EDB Postgres Advanced Server with minimal to no changes.

12.1 Enhanced compatibility features

EDB Postgres Advanced Server includes extended functionality that provides compatibility for syntax supported by Oracle applications. See Database compatibility for Oracle developers for additional information about the compatibility features supported by EDB Postgres Advanced Server.

Enabling compatibility features

You can install EDB Postgres Advanced Server in several ways to enable compatibility features:

- Before initializing your cluster, use the INITDBOPTS variable in the EDB Postgres Advanced Server service configuration file to specify --redwood-like.
- Include the --redwood-like parameter when using initdb to initialize your cluster.

See Configuration parameters compatible with Oracle databases and Managing an EDB Postgres Advanced Server installation for more information about the installation options supported by the EDB Postgres Advanced Server installers.

Stored procedural language

EDB Postgres Advanced Server supports a highly productive procedural language that allows you to write custom procedures, functions, triggers, and packages. This procedural language:

- Complements the SQL language and built-in packages.
- Provides a seamless development and testing environment.
- Allows you to create reusable code.

See Stored procedural language for more information.
Optimizer hints

When you invoke a DELETE, INSERT, SELECT, or UPDATE command, the server generates a set of execution plans. After analyzing those execution plans, the server selects a plan that generally returns the result set in the least amount of time. The server’s choice of plan depends on several factors:

- The estimated execution cost of data-handling operations
- Parameter values assigned in the Query Tuning section of the postgresql.conf file
- Column statistics that are gathered by the ANALYZE command

As a rule, the query planner selects the least expensive plan. You can use an optimizer hint to influence the server as it selects a query plan. An optimizer hint is one or more directives embedded in a comment-like syntax that immediately follows a DELETE, INSERT, SELECT, or UPDATE command. Keywords in the comment instruct the server to use or avoid a specific plan when producing the result set. See Optimizer hints for more information.

Data dictionary views

EDB Postgres Advanced Server includes a set of views that provide information about database objects in a manner compatible with the Oracle data dictionary views. See Database compatibility for Oracle developers: catalog views for detailed information about the views available with EDB Postgres Advanced Server.

dblink_ora

dblink_ora provides an OCI-based database link that allows you to SELECT, INSERT, UPDATE, or DELETE data stored on an Oracle system from EDB Postgres Advanced Server. See dblink_ora for detailed information about using dblink_ora and the supported functions and procedures.

Profile management

EDB Postgres Advanced Server supports compatible SQL syntax for profile management. Profile management commands allow a database superuser to create and manage named profiles. A profile is a named set of attributes that allow you to easily manage a group of roles that share comparable authentication requirements. Each profile defines rules for password management that augment password and md5 authentication. The rules in a profile can:

- Count failed login attempts
- Lock an account due to excessive failed login attempts
- Mark a password for expiration
- Define a grace period after a password expiration
- Define rules for password complexity
- Define rules that limit password reuse

After creating the profile, you can associate the profile with one or more users. If password requirements change, you can modify the profile to apply the new requirements to each user that’s associated with that profile.

When a user connects to the server, the server enforces the profile that’s associated with their login role.

Profiles are shared by all databases in a cluster, but each cluster can have multiple profiles. A single user with access to multiple databases uses the same profile when connecting to each database in the cluster.

See Profile management for information about using profile management commands.

Built-in packages

EDB Postgres Advanced Server supports a number of built-in packages that provide compatibility with Oracle procedures and functions.

<table>
<thead>
<tr>
<th>Package name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_ALERT</td>
<td>Lets you register for, send, and receive alerts.</td>
</tr>
<tr>
<td>DBMS_AQ</td>
<td>Provides message queuing and processing for EDB Postgres Advanced Server.</td>
</tr>
<tr>
<td>DBMS_AQDK</td>
<td>Provides supporting procedures for Advanced Queueing functionality.</td>
</tr>
<tr>
<td>DBMS_CRYPTO</td>
<td>Provides functions and procedures that allow you to encrypt or decrypt RAW, BLOB, or CLOB data. You can also use DBMS_CRYPTO functions to generate cryptographically strong random values.</td>
</tr>
<tr>
<td>DBMS_JOB</td>
<td>Provides for creating, scheduling, and managing jobs.</td>
</tr>
<tr>
<td>DBMS_LOB</td>
<td>Lets you operate on large objects.</td>
</tr>
<tr>
<td>DBMS_LOCK</td>
<td>Provides support for the DBMS_LOCK.SLEEP procedure.</td>
</tr>
<tr>
<td>DBMS_MVIEW</td>
<td>Use to manage and refresh materialized views and their dependencies.</td>
</tr>
</tbody>
</table>
### Built-in packages

<table>
<thead>
<tr>
<th>Package name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_OUTPUT</td>
<td>Lets you send messages to a message buffer or get messages from the message buffer.</td>
</tr>
<tr>
<td>DBMS_PIPE</td>
<td>Lets you send messages through a pipe in or between sessions connected to the same database cluster.</td>
</tr>
<tr>
<td>DBMS_PROFILE</td>
<td>Collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a performance profiling session.</td>
</tr>
<tr>
<td>DBMS_RANDOM</td>
<td>Provides methods to generate random values.</td>
</tr>
<tr>
<td>DBMS_REDACT</td>
<td>Enables redacting or masking data that’s returned by a query.</td>
</tr>
<tr>
<td>DBMS_RLS</td>
<td>Enables implementing Virtual Private Database on certain EDB Postgres Advanced Server database objects.</td>
</tr>
<tr>
<td>DBMS_SCHEDULER</td>
<td>Lets you create and manage jobs, programs, and job schedules.</td>
</tr>
<tr>
<td>DBMS_SESSION</td>
<td>Provides support for the <code>DBMS_SESSION.SET_ROLE</code> procedure.</td>
</tr>
<tr>
<td>DBMS_SQL</td>
<td>Provides an application interface to the EDB dynamic SQL functionality.</td>
</tr>
<tr>
<td>DBMS.Utility</td>
<td>Provides various utility programs.</td>
</tr>
<tr>
<td>UTL_ENCODE</td>
<td>Provides a way to encode and decode data.</td>
</tr>
<tr>
<td>UTL_FILE</td>
<td>Lets you read from and write to files on the operating system’s file system.</td>
</tr>
<tr>
<td>UTL_HTTP</td>
<td>Lets you use the HTTP or HTTPS protocol to retrieve information found at a URL.</td>
</tr>
<tr>
<td>UTL_MAIL</td>
<td>Lets you manage email.</td>
</tr>
<tr>
<td>UTL_RAW</td>
<td>Allows you to manipulate or retrieve the length of raw data types.</td>
</tr>
<tr>
<td>UTL_SMTP</td>
<td>Lets you send emails over the Simple Mail Transfer Protocol (SMTP).</td>
</tr>
<tr>
<td>UTL_URL</td>
<td>Provides a way to escape illegal and reserved characters in a URL.</td>
</tr>
</tbody>
</table>

See [Built-in packages](#) for detailed information about the procedures and functions available in each package.

### Open Client Library

The Open Client Library provides application interoperability with the Oracle Call Interface. An application that was formerly "locked in" can now work with either an EDB Postgres Advanced Server or an Oracle database with minimal to no changes to the application code. The EDB implementation of the Open Client Library is written in C.

The following diagram compares the Open Client Library and Oracle Call Interface application stacks.

![Open Client Library Diagram](image)

For detailed information about the functions supported by the Open Client Library, see [EDB OCL Connector](#).

### Utilities

For detailed information about the compatible syntax supported by these utilities, see [Tools, utilities, and components](#).

### EDB*Plus

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
EDB*Plus is a utility program that provides a command line user interface to the EDB Postgres Advanced Server that's familiar to Oracle developers and users. EDB*Plus accepts SQL commands, SPL anonymous blocks, and EDB*Plus commands.

EDB*Plus allows you to:

- Query certain database objects
- Execute stored procedures
- Format output from SQL commands
- Execute batch scripts
- Execute OS commands
- Record output

See EDB*Plus for more information.

EDB*Loader

EDB*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for EDB Postgres Advanced Server. The EDB*Loader command line utility loads data from an input source, typically a file, into one or more tables using a subset of the parameters offered by Oracle SQL*Loader.

EDB*Loader features include:

- Support for the Oracle SQL*Loader data loading methods—conventional path load, direct path load, and parallel direct path load
- Oracle SQL*Loader-compatible syntax for control file directives
- Input data with delimiter-separated or fixed-width fields
- Bad file for collecting rejected records
- Loading of multiple target tables
- Discard file for collecting records that don’t meet the selection criteria of any target table
- Data loading from standard input and remote loading

See Loading bulk data for information.

EDB*Wrap

The EDB*Wrap utility protects proprietary source code and programs (functions, stored procedures, triggers, and packages) from unauthorized scrutiny. The EDB*Wrap program translates a file that contains SPL or PL/pgSQL source code (the plaintext) into a file that contains the same code in a form that's nearly impossible to read. Once you have the obfuscated form of the code, you can send that code to EDB Postgres Advanced Server, which stores those programs in obfuscated form. While EDB*Wrap does obscure code, table definitions are still exposed.

Everything you wrap is stored in obfuscated form. If you wrap an entire package, the package body source as well as the prototypes contained in the package header and the functions and procedures contained in the package body are stored in obfuscated form.

See Protecting proprietary source code for information.

Dynamic Runtime Instrumentation Tools Architecture (DRITA)

DRITA allows a DBA to query catalog views to determine the wait events that affect the performance of individual sessions or the system as a whole. DRITA records the number of times each event occurs as well as the time spent waiting. You can use this information to diagnose performance problems. DRITA offers this functionality while consuming minimal system resources.

DRITA compares snapshots to evaluate the performance of a system. A snapshot is a saved set of system performance data at a given point in time. A unique ID number identifies each snapshot. You can use snapshot ID numbers with DRITA reporting functions to return system performance statistics.

See Using dynamic resource tuning for information.

ECPGPlus

EDB enhanced ECPG (the PostgreSQL precompiler) to create ECPGPlus. ECPGPlus allows you to include embedded SQL commands in C applications. When you use ECPGPlus to compile an application that contains embedded SQL commands, the SQL code is syntax checked and translated to C.

ECPGPlus supports Pro*C syntax in C programs when connected to an EDB Postgres Advanced Server database. ECPGPlus supports:

- Oracle Dynamic SQL – Method 4 (ODS-M4)
- Pro*C-compatible anonymous blocks
- A CALL statement compatible with Oracle databases

See ECPGPlus for information.

Table partitioning
In a partitioned table, one logically large table is broken into smaller physical pieces. Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning allows you to omit the partition column from the front of an index, reducing index size and making it more likely that the heavily used parts of the index fit in memory.
- When a query or update accesses a large percentage of a single partition, performance might improve because the server performs a sequential scan of the partition instead of using an index and random access reads scattered across the whole table.
- If you plan the requirement into the partitioning design, you can implement bulk load or unload by adding or removing partitions. ALTER TABLE is faster than a bulk operation. It also avoids the VACUUM overhead caused by a bulk DELETE.
- You can migrate seldom-used data to less expensive or slower storage media.

Table partitioning is worthwhile only when a table would otherwise be very large. The exact point at which a table benefits from partitioning depends on the application. A good estimation is when the size of the table exceeds the physical memory of the database server.

For information about database compatibility features supported by EDB Postgres Advanced Server, see Table partitioning.

### 12.2 Querying an Oracle server

dblink_ora enables you to issue arbitrary queries to a remote Oracle server. It provides an OCI-based database link that enables you to SELECT, INSERT, UPDATE, or DELETE data stored on an Oracle system from EDB Postgres Advanced Server.

#### 12.2.1 Calling dblink_ora functions

**Using the dblink_ora_connect function**

The following command establishes a connection using the dblink_ora_connect() function:

```sql
SELECT dblink_ora_connect('acctg', 'localhost', 'xe', 'hr', 'pwd', 1521);
```

The example connects to:

- A service named xe
- Running on port 1521 on the localhost
- With a user name of hr
- With a password of pwd

You can use the connection name `acctg` to refer to this connection when calling other dblink_ora functions.

**Using the dblink_ora_copy function**

The following command uses the dblink_ora_copy() function over a connection named `edb_conn`. It copies the `empid` and `deptno` columns from a table on an Oracle server named `ora_acctg` to a table located in the `public` schema on an instance of EDB Postgres Advanced Server named `as_acctg`. The TRUNCATE option is enforced, and a feedback count of 3 is specified:

```sql
db2= SELECT dblink_ora_copy('edb_conn','select empid, deptno
FROM
ora_acctg', 'public', 'as_acctg', true, 3);
```

```
INFO:  Row: 8
INFO:  Row: 3
INFO:  Row: 6
INFO:  Row: 9
INFO:  Row: 12

12
(1 row)
```

**Using the dblink_ora_record function**

The following SELECT statement uses the dblink_ora_record() function and the `acctg` connection to retrieve information from the Oracle server:
The command retrieves a list that includes all of the entries in the `first_name` column of the `employees` table.

### 12.2.2 Connecting to an Oracle database

To enable Oracle connectivity, download Oracle’s freely available OCI drivers from their website.

**Creating a symbolic link**

For Linux, if the Oracle instant client that you downloaded doesn’t include the `libclntsh.so` library, you must create a symbolic link named `libclntsh.so` that points to the downloaded version. Navigate to the instant client directory and execute the following command:

```
ln -s libclntsh.so.<version> libclntsh.so
```

Where `version` is the version number of the `libclntsh.so` library. For example:

```
ln -s libclntsh.so.12.1 libclntsh.so
```

**Setting the environment variable**

Before creating a link to an Oracle server, you must direct EDB Postgres Advanced Server to the correct Oracle home directory. Set the `LD_LIBRARY_PATH` environment variable on Linux or `PATH` on Windows to the `lib` directory of the Oracle client installation directory.

**Setting the oracle_home configuration parameter**

Alternatively, you can set the value of the `oracle_home` configuration parameter in the `postgresql.conf` file. The value specified in the `oracle_home` configuration parameter overrides the `LD_LIBRARY_PATH` environment variable in Linux and `PATH` environment variable in Windows.

**Note**

The `oracle_home` configuration parameter provides the correct path to the Oracle client, that is, the OCI library.

To set the `oracle_home` configuration parameter in the `postgresql.conf` file, edit the file and add the following line:

```
oracle_home = 'lib_directory'
```

In place of `<lib_directory>`, substitute the name of the `oracle_home` path to the Oracle client installation directory that contains `libclntsh.so` in Linux and `oci.dll` in Windows.

**Restarting the server**

After setting the `oracle_home` configuration parameter, you must restart the server for the changes to take effect. Restart the server:

- On Linux, using the `systemctl` command or `pg_ctl` services.
- On Windows, from the Windows Services console.

### 12.3 Using Open Client Library

The Open Client Library provides application interoperability with the Oracle Call Interface. An application that was formerly “locked in” can now work with either an EDB Postgres Advanced Server or an Oracle database with minimal to no changes to the application code. The EDB implementation of the Open Client Library is written in C.

The following diagram compares the Open Client Library and Oracle Call Interface application stacks.
For detailed usage information about the Open Client Library and the supported functions, see EDB OCL Connector.

**Note**

EDB doesn't support the use of the Open Client Library with Oracle Real Application Clusters (RAC) and Oracle Exadata. These Oracle products haven't been evaluated or certified with this EDB product.

### 12.4 Including embedded SQL commands in C applications (ECPGPlus)

EDB has enhanced ECPG (the PostgreSQL precompiler) to create ECPGPlus. ECPGPlus allows you to include embedded SQL commands in C applications. When you use ECPGPlus to compile an application that contains embedded SQL commands, the SQL code is syntax checked and translated into C.

See ECPGPlus overview for details about installing and configuring the utility.

### 12.5 Loading bulk data (EDB*Loader)

EDB*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for EDB Postgres Advanced Server. The EDB*Loader command line utility loads data from an input source, typically a file, into one or more tables using a subset of the parameters offered by Oracle SQL*Loader.

See EDB*Loader for details about using the utility.

### 12.6 Protecting proprietary source code (EDB*Wrap)

The EDB*Wrap utility protects proprietary source code and programs like functions, stored procedures, triggers, and packages from unauthorized scrutiny. The EDB*Wrap program translates a plaintext file that contains SPL or PL/pgSQL source code into a file that contains the same code in a form that's nearly impossible to read. Once you have the obfuscated form of the code, you can send that code to the PostgreSQL server, and the server stores those programs in obfuscated form. While EDB*Wrap does obscure code, table definitions are still exposed.

See EDB*Wrap for details about using the utility.

### 13 Tools, utilities, and components

Various tools and utilities are available with EDB Postgres Advanced Server.

#### 13.1 Application programmer tools

These EDB Postgres Advanced Server tools are designed to help the application programmer.
## 13.1.1 Unicode Collation Algorithm

The Unicode Collation Algorithm (UCA) is a specification (Unicode Technical Report #10) that defines a customizable method of collating and comparing Unicode data. Collation means how data is sorted, as with a `SELECT ... ORDER BY` clause. Comparison is relevant for searches that use ranges with less than, greater than, or equal to operators.

### Benefits

Customizability is an important factor for various reasons:

- Unicode supports many languages. Letters that might be common to several languages might collate in different orders depending on the language.
- Characters that appear with letters in certain languages, such as accents or umlauts, have an impact on the expected collation depending on the language.
- In some languages, combinations of several consecutive characters are treated as a single character with regard to their collation sequence.
- There might be certain preferences as to collating letters according to case. For example, the lowercase form of a letter might collate before the uppercase form of the same letter or vice versa.
- There might be preferences as to whether punctuation marks like underscore characters, hyphens, or space characters are considered in the collating sequence or whether they’re ignored.

Given all of these variations with the many languages supported by Unicode, a method is needed to select the specific criteria for determining a collating sequence. This is what the Unicode Collation Algorithm defines.

### Note

Another advantage for using ICU collations (the implementation of the Unicode Collation Algorithm) is for performance. Sorting tasks, including B-tree index creation, can complete in less than half the time it takes with a non-ICU collation. The exact performance gain depends on your operating system version, the language of your text data, and other factors.

### Basic Unicode Collation Algorithm concepts

The official information for the Unicode Collation Algorithm is specified in Unicode Technical Report #10.

The International Components for Unicode (ICU) also provides useful information. You can find an explanation of the collation concepts on the ICU website.

The basic concept behind the Unicode Collation Algorithm is the use of multilevel comparison. This means that a number of levels are defined, which are listed as level 1 through level 5 in the following bullet points. Each level defines a type of comparison. Strings are first compared using the primary level, also called level 1.

If the order can be determined based on the primary level, then the algorithm is done. If the order can’t be determined based on the primary level, then the secondary level, level 2, is applied. If the order can be determined based on the secondary level, then the algorithm is done, otherwise the tertiary level is applied, and so on. There is typically a final, tie-breaking level to determine the order if it can’t be resolved by the prior levels.

- **Level 1 – Primary level for base characters** – The order of basic characters such as letters and digits determines the difference, such as $A < B$.
- **Level 2 – Secondary level for accents** – If there are no primary level differences, then the presence or absence of accents and other such characters determines the order, such as $á < á$.
- **Level 3 – Tertiary level for case** – If there are no primary level or secondary level differences, then a difference in case determines the order, such as $a < A$.
- **Level 4 – Quaternary level for punctuation** – If there are no primary, secondary, or tertiary level differences, then the presence or absence of white-space characters, control characters, and punctuation determine the order, such as $A < ñ$.
- **Level 5 – Identical level for tie breaking** – If there are no primary, secondary, tertiary, or quaternary level differences, then some other difference such as the code point values determines the order.

### International components for Unicode

The Unicode Collation Algorithm is implemented by open source software provided by the International Components for Unicode (ICU). The software is a set of C/C++ and Java libraries.

When you use EDB Postgres Advanced Server to create a collation that invokes the ICU components to produce the collation, the result is referred to as an ICU collation.

### Locale collations

When creating a collation for a locale, a predefined ICU short form name for the given locale is typically provided. An ICU short form is a method of specifying collation attributes, which are the properties of a collation. See Collation attributes for more information.

There are predefined ICU short forms for locales. The ICU short form for a locale incorporates the collation attribute settings typically used for the given locale. The short form simplifies the collation creation process by eliminating the need to specify the entire list of collation attributes for that locale.

The system catalog `pg_catalog.pg_icu_collate_names` contains a list of the names of the ICU short forms for locales. The ICU short form name is listed in column `icu_short_form`.

```
<table>
<thead>
<tr>
<th>icu_short_form</th>
<th>valid_locale</th>
</tr>
</thead>
</table>
```
In this example, the ICU short form named You can specify collation attributes with a locale’s ICU short form name to override those default attribute settings of the locale.

A set of collation attributes and their values is represented by a text string consisting of the collation attribute letter concatenated with the desired attribute value. Each attribute/value pair is joined to form the ICU short form for a given locale by specifying the collation attributes to override that property.

Collation attributes

When creating an ICU collation, you must specify the desired characteristics of the collation. As discussed in Locale collations, you can typically do this with an ICU short form for the desired locale. However, if you need more specific information, you can specify the collation properties using collation attributes.

Collation attributes define the rules of how to compare characters for determining the collation sequence of text strings. As Unicode covers many languages in numerous variations according to country, territory, and culture, these collation attributes are complex.

For the complete and precise meaning and usage of collation attributes, see “Collator Naming Scheme” on the ICU — International Components for Unicode website.

Each collation attribute is represented by an uppercase letter. The possible valid values for each attribute are given by codes shown in the parentheses. Some codes have general meanings for all attributes. X means to set the attribute off. D means to set the attribute to its default value.

Collation attributes

- A – Alternate (N, S, D) — Handles treatment of variable characters such as white spaces, punctuation marks, and symbols. When set to non-ignorable (N), differences in variable characters are treated with the same importance as differences in letters. When set to shifted (S), then differences in variable characters are of minor importance (that is, the variable character is ignored when comparing base characters).
- C – Case first (X, L, U, D) – Controls whether a lowercase letter sorts before the same uppercase letter (L), or the uppercase letter sorts before the same lowercase letter (U). You typically specify off (X) when you want lowercase first (L).
- E – Case level (X, O, D) – Set in combination with the Strength attribute, use the Case Level attribute when you want to ignore accents but not case.
- F – French collation (X, O, D) — When set to on, secondary differences (presence of accents) are sorted from the back of the string as done in the French Canadian locale.
- H – Hiragana quaternary (X, O, D) — Introduces an additional level to distinguish between the Hiragana and Katakana characters for compatibility with the JIS X 4061 collation of Japanese character strings.
- N – Normalization checking (X, O, D) — Controls whether text is thoroughly normalized for comparison. Normalization deals with the issue of canonical equivalence of text whereby different code point sequences represent the same character. This occurrence then presents issues when sorting or comparing such characters. For languages such as Arabic, ancient Greek, Hebrew, Hindi, Thal, or Vietnamese, set normalization checking to on.
- S – Strength (1, 2, 3, 4, I, D) – Maximum collation level used for comparison. Influences whether accents or case are taken into account when collating or comparing strings. Each number represents a level. A setting of 1 represents identical strength (that is, level 5).
- T – Variable top (hexadecimal digits) — Applies only when the Alternate attribute isn’t set to non-ignorable (N). The hexadecimal digits specify the highest character sequence to consider ignorable. For example, if white space is ignorable but visible variable characters aren’t, then set Variable Top to 0020 along with the Alternate attribute set to S and the Strength attribute set to 5. (The space character is hexadecimal 0020. Other nonvisible variable characters, such as backspace, tab, line feed, and carriage return, have values less than 0020. All visible punctuation marks have values greater than 0020.)

A set of collation attributes and their values is represented by a text string consisting of the collation attribute letter concatenated with the desired attribute value. Each attribute/value pair is joined to form the ICU short form for a given locale by specifying the collation attributes to override that property.

AN_CX_EX_FX_HX_ND_S3

You can specify collation attributes with a locale’s ICU short form name to override those default attribute settings of the locale.

In this example, the ICU short form named LRROOT is modified with a number of other collation attribute/value pairs:

AN_CX_EX_LROOT_ND_S3
The Alternate attribute (A) is set to non-ignorable (N). The Case First attribute (C) is set to off (X). The Case Level attribute (E) is set to off (X). The Normalization attribute (N) is set to on (O). The Strength attribute (S) is set to the tertiary level L_ROOT is the ICU short form to which these other attributes are applying modifications.

Using a collation

You can use a newly defined ICU collation anywhere you can use the COLLATION "collation_name" clause in a SQL command. For example, you can use it in the column specifications of the CREATE TABLE command or appended to an expression in the ORDER BY clause of a SELECT command.

The following are some examples of creating and using ICU collations based on the English language in the United States (en_US.UTF8). In these examples, ICU collations are created with the following characteristics:

- Collation `icu_collate_lower` forces the lowercase form of a letter to sort before its uppercase counterpart (CL).
- Collation `icu_collate_upper` forces the uppercase form of a letter to sort before its lowercase counterpart (CU).
- Collation `icu_collate_ignore_punct` causes variable characters (white space and punctuation marks) to be ignored during sorting (AS).
- Collation `icu_collateignore_white_sp` causes white space and other nonvisible variable characters to be ignored during sorting, but visible variable characters (punctuation marks) aren't ignored (AS, T0020).

```sql
CREATE COLLATION icu_collate_lowercase
(
    LOCALE = 'en_US.UTF8',
    ICU_SHORT_FORM = 'AN_CL_EX_NX_LROOT'
);
CREATE COLLATION icu_collate_uppercase
(
    LOCALE = 'en_US.UTF8',
    ICU_SHORT_FORM = 'AN_CU_EX_NX_LROOT'
);
CREATE COLLATION icu_collate_ignore_punct
(
    LOCALE = 'en_US.UTF8',
    ICU_SHORT_FORM = 'AS_CX_EX_NX_LROOT_L3'
);
CREATE COLLATION icu_collate_ignore_white_sp
(
    LOCALE = 'en_US.UTF8',
    ICU_SHORT_FORM = 'AS_CX_EX_NX_LROOT_L3_T0020'
);
```

**Note**

When creating collations, ICU might generate notice and warning messages when attributes are given to modify the L_ROOT collation.

The following `psql` command lists the collations:

```
edb=# 
dO
```

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Collate</th>
<th>Ctype</th>
<th>ICU</th>
</tr>
</thead>
<tbody>
<tr>
<td>enterprisedb</td>
<td>icu_collate_ignore_punct</td>
<td>en_US.UTF8</td>
<td>en_US.UTF8</td>
<td>AS_CX_EX_NX_LROOT_L3</td>
</tr>
<tr>
<td></td>
<td>icu_collate_ignore_white_sp</td>
<td>en_US.UTF8</td>
<td>en_US.UTF8</td>
<td>AS_CX_EX_NX_LROOT_L3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AS_CX_EX_NX_LROOT_L3_T0020</td>
</tr>
<tr>
<td></td>
<td>icu_collate_lowercase</td>
<td>en_US.UTF8</td>
<td>en_US.UTF8</td>
<td>AN_CL_EX_NX_LROOT</td>
</tr>
<tr>
<td></td>
<td>icu_collate_uppercase</td>
<td>en_US.UTF8</td>
<td>en_US.UTF8</td>
<td>AN_CU_EX_NX_LROOT</td>
</tr>
</tbody>
</table>

(4 rows)

The following table is created and populated:

```sql
CREATE TABLE collate_tbl
(
    id INTEGER,
```
Using the default collation

The following query sorts on column \textit{c2} using the default collation. Variable characters (white space and punctuation marks) with \texttt{id} column values of 9, 10, and 11 are ignored and sort with the letter B.

<table>
<thead>
<tr>
<th>id</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>-B</td>
</tr>
<tr>
<td>9</td>
<td>.B</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

(11 rows)

Using collation \texttt{icu_collate_lowercase}

The following query sorts on column \textit{c2} using collation \texttt{icu_collate_lowercase}. This collation forces the lowercase form of a letter to sort before the uppercase form of the same base letter. The \texttt{AN} attribute forces the sort order to include variable characters at the same level when comparing base characters. Thus rows with \texttt{id} values of 9, 10, and 11 appear at the beginning of the sort list before all letters and numbers.

<table>
<thead>
<tr>
<th>id</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>18</td>
<td>-B</td>
</tr>
<tr>
<td>9</td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

(11 rows)

Using collation \texttt{icu_collate_uppercase}

```sql
INSERT INTO collate_tbl VALUES (1, 'A');
INSERT INTO collate_tbl VALUES (2, 'B');
INSERT INTO collate_tbl VALUES (3, 'C');
INSERT INTO collate_tbl VALUES (4, 'a');
INSERT INTO collate_tbl VALUES (5, 'b');
INSERT INTO collate_tbl VALUES (6, 'c');
INSERT INTO collate_tbl VALUES (7, '1');
INSERT INTO collate_tbl VALUES (8, '2');
INSERT INTO collate_tbl VALUES (9, 'B');
INSERT INTO collate_tbl VALUES (10, '-B');
INSERT INTO collate_tbl VALUES (11, 'B');
```

Using the default collation

```
SELECT * FROM collate_tbl ORDER BY c2;
```

<table>
<thead>
<tr>
<th>id</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>-B</td>
</tr>
<tr>
<td>9</td>
<td>.B</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

(11 rows)

Using collation \texttt{icu_collate_lowercase}

```
SELECT * FROM collate_tbl ORDER BY c2 COLLATE "icu_collate_lowercase";
```

<table>
<thead>
<tr>
<th>id</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>18</td>
<td>-B</td>
</tr>
<tr>
<td>9</td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

(11 rows)
The following query sorts on column `c2` using collation `icu_collate_uppercase`. This collation forces the uppercase form of a letter to sort before the lowercase form of the same base letter.

<table>
<thead>
<tr>
<th>id</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>18</td>
<td>-B</td>
</tr>
<tr>
<td>9</td>
<td>.B</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
</tbody>
</table>

(11 rows)

Using collation `icu_collate_ignore_punct`

The following query sorts on column `c2` using collation `icu_collate_ignore_punct`. This collation causes variable characters to be ignored so rows with id values of 9, 18, and 11 sort with the letter B, the character immediately following the ignored variable character.

<table>
<thead>
<tr>
<th>id</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>-B</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

(11 rows)

Using collation `icu_collate_ignore_white_sp`

The following query sorts on column `c2` using collation `icu_collate_ignore_white_sp`. The AS and T0020 attributes of the collation cause variable characters with code points less than or equal to hexadecimal 0020 to be ignored while variable characters with code points greater than hexadecimal 0020 are included in the sort.

The row with id value of 11, which starts with a space character (hexadecimal 0020), sorts with the letter B. The rows with id values of 9 and 18, which start with visible punctuation marks greater than hexadecimal 0020, appear at the beginning of the sort list. These particular variable characters are included in the sort order at the same level when comparing base characters.

<table>
<thead>
<tr>
<th>id</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>-B</td>
</tr>
<tr>
<td>9</td>
<td>.B</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

(11 rows)
13.1.2 ECPGPlus

EDB has enhanced ECPG (the PostgreSQL precompiler) to create ECPGPlus. ECPGPlus allows you to include embedded SQL commands in C applications. When you use ECPGPlus to compile an application that contains embedded SQL commands, the SQL code is syntax checked and translated into C.

ECPGPlus supports Pro*C-compatible syntax in C programs when connected to an EDB Postgres Advanced Server database. ECPGPlus supports:

- Oracle Dynamic SQL – Method 4 (ODS-M4)
- Pro*C-compatible anonymous blocks
- A CALL statement compatible with Oracle databases

As part of ECPGPlus' Pro*C compatibility, you don't need to include the BEGIN DECLARE SECTION and END DECLARE SECTION directives.

See ECPGPlus overview for details about installing and configuring the utility.

13.1.3 libpq C library

libpq is the C application programmer's interface to EDB Postgres Advanced Server. libpq is a set of library functions that allow client programs to pass queries to the EDB Postgres Advanced Server and to receive the results of these queries.

libpq is also the underlying engine for several other EDB application interfaces, including those written for C++, Perl, Python, Tcl, and ECPG. Some aspects of libpq's behavior are important to you if you are using one of those packages.

Prerequisites

Client programs that use libpq must include the header file `libpq-fe.h` and must link with the `libpq` library.

Using libpq with EDB SPL

You can use the EDB SPL language with the libpq interface library, providing support for:

- Procedures, functions, packages
- Prepared statements
- REFCURSORS
- Static cursors
- structs and typedefs
- Arrays
- DML and DDL operations
- IN / OUT / IN OUT parameters

REFCURSOR support

In earlier releases, EDB Postgres Advanced Server provided support for REFCURSORS through the following libpq functions. These functions are now deprecated:

- PQCursorResult()
- PQgetCursorResult()
- PQnCursor()

You can now use PQexec() and PQgetvalue() to retrieve a REFCURSOR returned by an SPL (or PL/pgSQL) function. A REFCURSOR is returned in the form of a null-terminated string indicating the name of the cursor. Once you have the name of the cursor, you can execute one or more FETCH statements to retrieve the values exposed through the cursor.

Note

The examples that follow don't include the error-handling code required in a real-world client application.

Returning a single REFCURSOR

This example shows an SPL function that returns a value of type REFCURSOR:

```c
EDB Postgres Advanced Server (EPAS) - Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
CREATE OR REPLACE FUNCTION getEmployees(p_deptno NUMERIC) 
RETURN REF CURSOR AS 
result 
BEGIN 
OPEN result FOR SELECT * FROM emp WHERE deptno = p_deptno; 
RETURN result; 
END;

This function expects a single parameter, `p_deptno`, and returns a `REF CURSOR` that holds the result set for the `SELECT` query shown in the `OPEN` statement. The `OPEN` statement executes the query and stores the result set in a cursor. The server constructs a name for that cursor and stores the name in a variable named `result`. The function then returns the name of the cursor to the caller.

To call this function from a C client using libpq, you can use `PQexec()` and `PQgetValue()`:

```c
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void fetchAllRows(PGconn *conn, 
const char *cursorName, 
const char *description) {

static void fail(PGconn *conn, const char *msg) {

int main(int argc, char *argv[]) {
    PGconn *conn = PQconnectdb(argv[1]);
    PGresult *result;

    if (PQstatus(conn) != CONNECTION_OK)
        fail(conn, PQerrorMessage(conn));
    result = PQexec(conn, "BEGIN TRANSACTION");

    if (PQresultStatus(result) != PGRES_COMMAND_OK)
        fail(conn, PQerrorMessage(conn));
    PQclear(result);
    result = PQexec(conn, "SELECT * FROM getEmployees(18)");

    if (PQresultStatus(result) != PGRES_TUPLES_OK)
        fail(conn, PQerrorMessage(conn));
   fetchAllRows(conn, PQgetValue(result, 0, 0), "employees");
    PQclear(result);
    PQexec(conn, "COMMIT");
    PQfinish(conn);
    exit(0);
}

static void 
fetchAllRows(PGconn *conn, 
const char *cursorName, 
const char *description) {

size_t commandLength = strlen("FETCH ALL FROM ") + 
                        strlen(cursorName) + 
3;

    char *commandText = malloc(commandLength);
    PGresult *result;
```
The code sample contains a line of code that calls the `getEmployees()` function and returns a result set that contains all of the employees in department 10:

```c
result = PQexec(conn, "SELECT * FROM getEmployees(10)";
```

The `PQexec()` function returns a result set handle to the C program. The result set contains one value: the name of the cursor returned by `getEmployees()`.

Once you have the name of the cursor, you can use the SQL `FETCH` statement to retrieve the rows in that cursor. The function `fetchAllRows()` builds a `FETCH ALL` statement, executes that statement, and then prints the result set of the `FETCH ALL` statement.

The output of this program is:

```
-- employees --
7782,CLARK,MANAGER,7839.00-09-JUN-81 00:00:00,2450.00,10
7839,KING,PRESIDENT,17-NOV-81 00:00:00,5000.00,10
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1380.00,10
```

Returning multiple REFCURSORS

The next example returns two REFCURSORS:

- The first REFCURSOR contains the name of a cursor (`employees`) that contains all employees who work in a department in the range specified by the caller.
- The second REFCURSOR contains the name of a cursor (`departments`) that contains all of the departments in the range specified by the caller.

In this example, instead of returning a single REFCURSOR, the function returns a SETOF REFCURSOR, which means 0 or more REFCURSORS. One other important difference is that the libpq program must not expect a single REFCURSOR in the result set. It must expect two rows, each of which contains a single value. The first row contains the name of the `employees` cursor, and the
second row contains the name of the **departments** cursor.

As in the previous example, you can use **PQexec()** and **PQgetvalue()** to call the SPL function:

```sql
CREATE OR REPLACE FUNCTION getEmpsAndDepts(p_min NUMERIC, p_max NUMERIC)
RETURN SETOF REFCURSOR AS
  employees REFCURSOR;
BEGIN
  OPEN employees FOR
    SELECT * FROM emp WHERE deptno BETWEEN p_min AND p_max;
  RETURN NEXT employees;
  OPEN departments FOR
    SELECT * FROM dept WHERE deptno BETWEEN p_min AND p_max;
  RETURN NEXT departments;
END;
```

As in the previous example, you can use **PQexec()** and **PQgetvalue()** to call the SPL function:

```c
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void fetchAllRows(PGconn *conn,
 const char *cursorName,
 const char *description);
static void fail(PGconn *conn,
 const char *msg);

int main(int argc, char *argv[])
{
  PGconn *conn = PQconnectdb(argv[1]);
  PGresult *result;

  if (PQstatus(conn) != CONNECTION_OK)
    fail(conn, PQerrorMessage(conn));
  result = PQexec(conn, "BEGIN TRANSACTION");

  if (PQresultStatus(result) != PGRES_COMMAND_OK)
    fail(conn, PQerrorMessage(conn));
  PQclear(result);
  result = PQexec(conn, "SELECT * FROM getEmpsAndDepts(20, 30)");

  if (PQresultStatus(result) != PGRES_TUPLES_OK)
    fail(conn, PQerrorMessage(conn));
 fetchAllRows(conn, PQgetvalue(result, 0, 0), "employees");
  fetchAllRows(conn, PQgetvalue(result, 1, 0), "departments");
  PQclear(result);
  PQexec(conn, "COMMIT");
  PQfinish(conn);
  exit(0);
}

static void fetchAllRows(PGconn *conn,
 const char *cursorName,
 const char *description)
{
  size_t commandLength = strlen("FETCH ALL FROM ")
  ```
if you call `getEmpsAndDepts(20, 30)`, the server returns a cursor that contains all employees who work in department 20 or 30 and a second cursor containing the description of departments 20 and 30.

```
-- employees --

7369, SMITH, CLERK, 7902, 17-DEC-80 00:00:00, 806.00, 20
7499, ALLEN, SALESMAN, 7658, 20-FEB-81 00:00:00, 1600.00, 300.00, 30
7521, WARD, SALESMAN, 7566, 22-FEB-81 00:00:00, 1250.00, 1250.00, 30
7566, JONES, MANAGER, 7839, 02-APR-81 00:00:00, 2975.00, 19
7654, MARTIN, SALESMAN, 7698, 28-SEP-81 00:00:00, 1250.00, 1400.00, 30
7698, BLAKE, MANAGER, 7839, 01-MAY-81 00:00:00, 2856.00, 30
7788, SCOTT, ANALYST, 7566, 19-APR-87 00:00:00, 3000.00, 3806.88, 20
7844, TURNER, SALESMAN, 7698, 08-SEP-81 00:00:00, 1500.00, 0.00, 30
7937, ADAMS, CLERK, 7788, 23-MAY-87 00:00:00, 1100.00, 20
7900, JAMES, CLERK, 7688, 03-DEC-81 00:00:00, 950.00, 20
7992, FORD, ANALYST, 7666, 03-DEC-81 00:00:00, 3000.00, 20
-- departments --

- 20, RESEARCH, DALLAS
- 30, SALES, CHICAGO
```
Perform bulk operations with a prepared statement. Use the following function to prepare the statement:

```c
PGresult *PQprepare(PGconn *conn,
    const char *stmtName,
    const char *query,
    int nParams,
    const Oid *paramTypes);
```

You can find details of `PQprepare()` in the prepared statement section.

You can use the following functions to perform bulk operations:

- `PQBulkStart`
- `PQexecBulk`
- `PQBulkFinish`
- `PQexecBulkPrepared`

### PQBulkStart

`PQBulkStart()` initializes bulk operations on the server. You must call this function before sending bulk data to the server. `PQBulkStart()` initializes the prepared statement specified in `stmtName` to receive data in a format specified by `paramFmts`.

API definition

```c
PGresult *PQBulkStart(PGconn *conn,
    const char *Stmt_Name,
    unsigned int nCol,
    const int *paramFmts);
```

### PQexecBulk

Use `PQexecBulk()` to supply data (`paramValues`) for a statement that was previously initialized for bulk operation using `PQBulkStart()`.

You can use this function more than once after `PQBulkStart()` to send multiple blocks of data.

API definition

```c
PGresult *PQexecBulk(PGconn *conn,
    unsigned int nRows,
    const char *const *paramValues,
    const int *paramLengths);
```

### PQBulkFinish

This function completes the current bulk operation. You can use the prepared statement again without preparing it again.

API definition

```c
PGresult *PQBulkFinish(PGconn *conn);
```

### PQexecBulkPrepared

Alternatively, you can use the `PQexecBulkPrepared()` function to perform a bulk operation with a single function call. `PQexecBulkPrepared()` sends a request to execute a prepared statement with the given parameters and waits for the result. This function combines the functionality of `PQbulkStart()`, `PQexecBulk()`, and `PQBulkFinish()`. When using this function, you don’t need to initialize or terminate the bulk operation. This function starts the bulk operation, passes the data to the server, and terminates the bulk operation.

Specify a previously prepared statement in the place of `stmtName`. Commands that are used repeatedly are parsed and planned just once, rather than each time they’re executed.
Using PQBulkStart, PQexecBulk, PQBulkFinish

This example uses PQBulkStart, PQexecBulk, and PQBulkFinish:

```c
void InsertDataUsingBulkStyle(PGconn *conn)
{
    PGresult *res;
    PGresult res;
    Oid paramTypes[2];
    int paramLens[2];
    int paramFmts[2];
    int i;
    char a[5] = { 10, 20, 30, 40, 50 };
    char b[5][10] = { "Test_1", "Test_2", "Test_3", "Test_4", "Test_5" };

    paramTypes[0] = 23;
    paramTypes[1] = 1043;
    res = PQprepare( conn, "stmt_1", "INSERT INTO testtable1 values( $1, $2 )", 2, paramTypes );
    PQclear( res );
    /* Binary format */
    paramFmts[0] = 1;  /* Binary format */
    paramFmts[1] = 0;
    for ( i = 0; i < 5; i++ )
    {
        a[i] = htonl( a[i] );
        paramVals[i][0] = &a[i]);
        paramVals[i][1] = b[i];
        paramLens[i][0] = 4;
        paramLens[i][1] = strlen( b[i] );
    }
    res = PQBulkStart(conn, "stmt_1", 2, paramFmts);  
PQclear( res );
    printf( "< -- PQBulkStart -- >\n" );
    res = PQexecBulk(conn, 5, (const char *const *)paramVals,
                     (const int *)paramLens);
PQclear( res );
    printf( "< -- PQexecBulk -- >\n" );
    res = PQexecBulk(conn, 5, (const char *const *)paramVals,
                     (const int *)paramLens);
PQclear( res );
```
13.2 Connectivity tools

We package and support a number of open source components you can use to connect to as well as manage your Postgres database. In addition, we package and support open-source components to connect your server to external data sources. These EDB components are enhanced versions of the corresponding PostgreSQL components and are intended to support additional EDB Postgres Advanced Server capabilities.

13.2.1 Connecting to your database

```c
void InsertDataUsingBulkStyleCombinedVersion(PGconn *conn)
{
  PGresult *res;
  Oid paramTypes[2];
  char *paramVals[5][2];
  int paramLens[5][2];
  int paramFmts[2];

  int a[5] = { 10, 20, 30, 40, 50 }
  char b[5][10] = { "Test_1", "Test_2", "Test_3", "Test_4", "Test_5" }

  paramTypes[0] = 23;
  paramTypes[1] = 1043;
  res = PQprepare( conn, "stmt_2", "INSERT INTO testtable1
values( $1, $2) ", 2, paramTypes );
  PQclear( res );
  /* Binary format
  paramFmts[0] = 1; /* Binary format
  paramFmts[1] = 8;
  for( i = 0; i < 5; i++
  { }
  a[i] = htonl( a[i] );
  paramVals[1][0] = &a[i];
  paramVals[1][1] = b[i];
  paramLens[1][0] = 4;
  paramLens[1][1] = strlen( b[i] );
  }
  res = PQexecBulkPrepared(conn, "stmt_2", 2, 5, (const char *const *)paramVals, (const int *)paramLens, (const int *)paramFmts);
  PQclear( res );
}
```

Using PQexecBulkPrepared

This example uses `PQexecBulkPrepared`:

```c
printf("< -- PQexecBulk -- >\n");
res = PQBulkFinish(conn);
PQclear( res );
printf("< -- PQBulkFinish -- >\n");
}
```
We package and support a number of components to connect to your EDB Postgres Advanced Server database server. These EDB components are enhanced versions of the corresponding PostgreSQL components and are intended to support additional EDB Postgres Advanced Server capabilities.

The EDB components for connecting to your database include:

- **EDB JDBC Connector** — Provides connectivity between a Java application and a Postgres database server.
- **EDB .NET Connector** — Provides connectivity between a .NET client application and a Postgres database server.
- **EDB OCL Connector** — Provides an API similar to the Oracle Call Interface that you can use to interact with a Postgres database server.
- **EDB ODBC Connector** — Allows an ODBC-compliant client application to connect to a Postgres database server.

You can use the PostgreSQL community components with EDB Postgres Advanced Server. However, if your applications require EDB Postgres Advanced Server features, use the EDB components.

### 13.2.2 Connecting to external data sources

We package and support a number of open source components to connect your Postgres database server to external data sources. These EDB components are enhanced versions of the corresponding PostgreSQL components and are intended to support additional EDB Postgres Advanced Server capabilities.

The EDB components to connect external data sources include:

- **EDB Hadoop Foreign Data Wrapper** — accesses data that resides on a Hadoop file system from a Postgres database server.
- **MongoDB Foreign Data Wrapper** — accesses data that resides on a MongoDB database from a Postgres database server.
- **MySQL Foreign Data Wrapper** — accesses data that resides on a MySQL database from a Postgres database server.

### 13.2.3 Managing your database connections

We package and support a number of open source components to manage your connections to your EDB Postgres Advanced Server database server. These EDB components are enhanced versions of the corresponding PostgreSQL components and are intended to support additional EDB Postgres Advanced Server capabilities.

The EDB components for managing your database connections include:

- **EDB PgBouncer** — a lightweight connection pooling utility for Postgres installations.
- **EDB Pgpool-II** — acts as middleware between client applications and a Postgres database server.

### 13.3 Database administrator tools

EDB Postgres Advanced Server includes features designed to increase database administrator productivity, such as enabling DBAs to prioritize different workloads using the Resource Manager feature.

#### 13.3.1 Tools and utilities overview

Oracle-compatible tools and utility programs allow you to work with EDB Postgres Advanced Server in a familiar environment. The tools supported by EDB Postgres Advanced Server include:

- **EDB*Plus**
- **EDB*Loader**
- **EDB*Wrap**
- The Dynamic Runtime Instrumentation Tools Architecture (DRITA)

#### 13.3.2 EDB clone schema

EDB Clone Schema is an extension module for EDB Postgres Advanced Server that allows you to copy a schema and its database objects from a local or remote database (the source database) to a receiving database (the target database).

The source and target databases can be either:
The same physical database
Different databases in the same database cluster
Separate databases running under different database clusters on separate database server hosts

For details about using the tool, see Copying a database.

13.3.3   EDB*Loader

EDB*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for EDB Postgres Advanced Server. The EDB*Loader command line utility loads data from an input source, typically a file, into one or more tables using a subset of the parameters offered by Oracle SQL*Loader.

For details about using the tool, see Loading bulk data.

13.3.4   EDB Resource Manager

EDB Resource Manager is an EDB Postgres Advanced Server feature that lets you control the use of operating system resources used by EDB Postgres Advanced Server processes.

This capability allows you to protect the system from processes that might uncontrollably overuse and monopolize certain system resources.

For details about using the utility, see Throttling CPU and I/O at the process level.

13.4   Oracle-compatibility tools

These tools and utilities allow a developer that works with Oracle utilities to work with EDB Postgres Advanced Server in a familiar environment.

These compatible tools and utilities that are supported by EDB Postgres Advanced Server:

- EDB*Loader
- EDB*Wrap
- Dynamic Runtime Instrumentation

The EDB*Plus command line client provides a user interface to EDB Postgres Advanced Server that supports SQL*Plus commands; EDB*Plus allows you to:

- Query database objects
- Execute stored procedures
- Format output from SQL commands
- Execute batch scripts
- Execute OS commands
- Record output

See EDB*Plus for detailed installation and usage information about EDB*Plus.

14   Reference

This reference material contains descriptive content needed by application programmers and database administrators. It also includes information that applies to organizations migrating their Oracle applications to use EDB Postgres Advanced Server as well as a SQL reference.

14.1   Application programmer reference

This reference information applies to application programmers.

14.1.1   Table partitioning views reference

Query the catalog views that are compatible with Oracle databases to review detailed information about your partitioned tables.

14.1.1.1   ALL_PART_TABLES
The following table lists the information available in the **ALL_PART_TABLES** view.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>name</td>
<td>The owner of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>name</td>
<td>The schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>partitioning_type</td>
<td>text</td>
<td>RANGE, LIST, or HASH.</td>
</tr>
<tr>
<td>subpartitioning_type</td>
<td>text</td>
<td>RANGE, LIST, HASH, or NONE.</td>
</tr>
<tr>
<td>partition_count</td>
<td>bigint</td>
<td>The number of partitions.</td>
</tr>
<tr>
<td>def_subpartition_count</td>
<td>integer</td>
<td>The default subpartition count. This value is always 0.</td>
</tr>
<tr>
<td>partitioning_key_count</td>
<td>integer</td>
<td>The number of columns listed in the partition by clause.</td>
</tr>
<tr>
<td>subpartitioning_key_count</td>
<td>integer</td>
<td>The number of columns in the subpartition by clause.</td>
</tr>
<tr>
<td>status</td>
<td>character varying(8)</td>
<td>This column is always VALID.</td>
</tr>
<tr>
<td>def_tablespace_name</td>
<td>character varying(30)</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_pct_free</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_pct_used</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_inl_trans</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_max_trans</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_initial_extent</td>
<td>character varying(40)</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_next_extent</td>
<td>character varying(40)</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_min_extents</td>
<td>character varying(40)</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_max_extents</td>
<td>character varying(40)</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_pct_increase</td>
<td>character varying(40)</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_freelists</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_freelist_groups</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>def_logging</td>
<td>character varying(7)</td>
<td>This column is always YES.</td>
</tr>
<tr>
<td>def_compression</td>
<td>character varying(8)</td>
<td>This column is always NONE.</td>
</tr>
<tr>
<td>def_buffer_pool</td>
<td>character varying(7)</td>
<td>This column is always DEFAULT.</td>
</tr>
<tr>
<td>ref_ptn_constraint_name</td>
<td>character varying(30)</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>interval</td>
<td>character varying(1000)</td>
<td>This column is always NULL.</td>
</tr>
</tbody>
</table>

**14.1.1.2 ALL_TAB_PARTITIONS**

The following table lists the information available in the **ALL_TAB_PARTITIONS** view.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_owner</td>
<td>name</td>
<td>The owner of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>name</td>
<td>The schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>composite</td>
<td>text</td>
<td>YES if the table is subpartitioned; NO if it is not subpartitioned.</td>
</tr>
<tr>
<td>partition_name</td>
<td>name</td>
<td>The name of the partition.</td>
</tr>
<tr>
<td>subpartition_count</td>
<td>bigint</td>
<td>The number of subpartitions for this partition.</td>
</tr>
<tr>
<td>high_value</td>
<td>text</td>
<td>The high partitioning value specified in the <strong>CREATE TABLE</strong> statement.</td>
</tr>
<tr>
<td>high_value_length</td>
<td>integer</td>
<td>The length of high partitioning value.</td>
</tr>
<tr>
<td>partition_position</td>
<td>integer</td>
<td>The ordinal position of this partition.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>name</td>
<td>The tablespace in which this partition resides.</td>
</tr>
<tr>
<td>pct_free</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>pct_used</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>init_trans</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>max_trans</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>initial_extent</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>next_extent</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
</tbody>
</table>
### Column Types and Descriptions

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>min_extent</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>max_extent</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>pct_increase</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>freelists</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>freelists_groups</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>logging</td>
<td>character varying(7)</td>
<td>This column is always YES.</td>
</tr>
<tr>
<td>compression</td>
<td>character varying(8)</td>
<td>This column is always NONE.</td>
</tr>
<tr>
<td>num_rows</td>
<td>numeric</td>
<td>The approx. number of rows in this partition.</td>
</tr>
<tr>
<td>blocks</td>
<td>integer</td>
<td>The approx. number of blocks in this partition.</td>
</tr>
<tr>
<td>empty_blocks</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>avg_space</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>chain_cnt</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>avg_row_len</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>sample_size</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>last_analyzed</td>
<td>timestamp without time zone</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>buffer_pool</td>
<td>character varying(7)</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>global_stats</td>
<td>character varying(3)</td>
<td>This column is always YES.</td>
</tr>
<tr>
<td>user_stats</td>
<td>character varying(3)</td>
<td>This column is always NO.</td>
</tr>
<tr>
<td>backing_table</td>
<td>regclass</td>
<td>OID of the backing table for this partition.</td>
</tr>
</tbody>
</table>

#### 14.1.1.3 ALL_TAB_SUBPARTITIONS

The following table lists the information available in the `ALL_TAB_SUBPARTITIONS` view.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_owner</td>
<td>name</td>
<td>The name of the owner of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>name</td>
<td>The name of the schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>partition_name</td>
<td>name</td>
<td>The name of the partition.</td>
</tr>
<tr>
<td>subpartition_name</td>
<td>name</td>
<td>The name of the subpartition.</td>
</tr>
<tr>
<td>high_value</td>
<td>text</td>
<td>The high partitioning value specified in the <code>CREATE TABLE</code> statement.</td>
</tr>
<tr>
<td>high_value_length</td>
<td>integer</td>
<td>The length of high partitioning value.</td>
</tr>
<tr>
<td>subpartition_position</td>
<td>integer</td>
<td>The ordinal position of this subpartition.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>name</td>
<td>The tablespace in which this subpartition resides.</td>
</tr>
<tr>
<td>pct_free</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>pct_used</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>init_trans</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>max_trans</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>initial_extent</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>next_extent</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>min_extent</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>max_extent</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>pct_increase</td>
<td>numeric</td>
<td>This column is always 0.</td>
</tr>
<tr>
<td>freelists</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>freelists_groups</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>logging</td>
<td>character varying(7)</td>
<td>This column is always YES.</td>
</tr>
<tr>
<td>compression</td>
<td>character varying(8)</td>
<td>This column is always NONE.</td>
</tr>
<tr>
<td>num_rows</td>
<td>numeric</td>
<td>The approx. number of rows in this subpartition.</td>
</tr>
<tr>
<td>blocks</td>
<td>integer</td>
<td>The approx. number of blocks in this subpartition.</td>
</tr>
<tr>
<td>empty_blocks</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>avg_space</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>chain_cnt</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
<tr>
<td>avg_row_len</td>
<td>numeric</td>
<td>This column is always NULL.</td>
</tr>
</tbody>
</table>
### 14.1.1.4  ALL_PART_KEY_COLUMNS

The following table lists the information available in the `ALL_PART_KEY_COLUMNS` view.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>name</td>
<td>The name of the table owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>name</td>
<td>The name of the schema on which the table resides.</td>
</tr>
<tr>
<td>name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>object_type</td>
<td>character(5)</td>
<td>This column is always TABLE.</td>
</tr>
<tr>
<td>column_name</td>
<td>name</td>
<td>The name of the partitioning key column.</td>
</tr>
<tr>
<td>column_position</td>
<td>integer</td>
<td>The position of this column in the partitioning key. The first column has a column position of 1, the second column has a column position of 2, and so on.</td>
</tr>
</tbody>
</table>

### 14.1.1.5  ALL_SUBPART_KEY_COLUMNS

The following table lists the information available in the `ALL_SUBPART_KEY_COLUMNS` view.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>name</td>
<td>The name of the table owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>name</td>
<td>The name of the schema on which the table resides.</td>
</tr>
<tr>
<td>name</td>
<td>name</td>
<td>The name of the table.</td>
</tr>
<tr>
<td>object_type</td>
<td>character(5)</td>
<td>This column is always TABLE.</td>
</tr>
<tr>
<td>column_name</td>
<td>name</td>
<td>The name of the partitioning key column.</td>
</tr>
<tr>
<td>column_position</td>
<td>integer</td>
<td>The position of this column within the subpartitioning key (the first column has a column position of 1, the second column has a column position of 2, and so on.)</td>
</tr>
</tbody>
</table>

### 14.1.2  System catalog tables

The following system catalog tables contain definitions of database objects. The layout of the system tables is subject to change. If you're writing an application that depends on information stored in the system tables, use an existing catalog view or create a catalog view to isolate the application from changes to the system table.

**dual**

`dual` is a single-row, single-column table that's provided only for compatibility with Oracle databases.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dummy</td>
<td>VARCHAR2(1)</td>
<td></td>
<td>Provided for compatibility only.</td>
</tr>
</tbody>
</table>

**edb_dir**

The `edb_dir` table contains one row for each alias that points to a directory created with the `CREATE DIRECTORY` command. A directory is an alias for a pathname that allows a user limited access to the host file system.
You can use a directory to fence a user into a specific directory tree in the file system. For example, the `UTL_FILE` package offers functions that allow a user to read and write files and directories in the host file system. However, it allows access only to paths that the database administrator has granted access to by way of a `CREATE DIRECTORY` command.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dirname</td>
<td>&quot;name&quot;</td>
<td>not null</td>
<td>The name of the alias.</td>
</tr>
<tr>
<td>dirowner</td>
<td>oid</td>
<td>not null</td>
<td>The OID of the user that owns the alias.</td>
</tr>
<tr>
<td>dirpath</td>
<td>text</td>
<td></td>
<td>The directory name to which access is granted.</td>
</tr>
<tr>
<td>diracl</td>
<td>aclitem[]</td>
<td></td>
<td>The access control list that determines the users who can access the alias.</td>
</tr>
</tbody>
</table>

**edb_password_history**

The `edb_password_history` table contains one row for each password change. The table is shared across all databases in a cluster.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>passhistroleid</td>
<td>oid</td>
<td>pg_authid.oid</td>
<td>The ID of a role.</td>
</tr>
<tr>
<td>passhistpassword</td>
<td>text</td>
<td></td>
<td>Role password in md5 encrypted form.</td>
</tr>
<tr>
<td>passhistpasswordsetat</td>
<td>timestamptz</td>
<td></td>
<td>The time the password was set.</td>
</tr>
</tbody>
</table>

**edb_policy**

The `edb_policy` table contains one row for each policy.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>policyname</td>
<td>name</td>
<td>not null</td>
<td>The policy name.</td>
</tr>
<tr>
<td>policygroup</td>
<td>oid</td>
<td>not null</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>policyobject</td>
<td>oid</td>
<td>not null</td>
<td>The OID of the table secured by this policy (the <code>object_schema</code> plus the <code>object_name</code>).</td>
</tr>
<tr>
<td>policykind</td>
<td>char</td>
<td>not null</td>
<td>Currently always 'Y'.</td>
</tr>
<tr>
<td>policyproc</td>
<td>oid</td>
<td>not null</td>
<td>The OID of the policy function (<code>function_schema</code> plus <code>policy_function</code>).</td>
</tr>
<tr>
<td>policyinsert</td>
<td>boolean</td>
<td>not null</td>
<td>True if the policy is enforced by <code>INSERT</code> statements.</td>
</tr>
<tr>
<td>policyselect</td>
<td>boolean</td>
<td>not null</td>
<td>True if the policy is enforced by <code>SELECT</code> statements.</td>
</tr>
<tr>
<td>policydelete</td>
<td>boolean</td>
<td>not null</td>
<td>True if the policy is enforced by <code>DELETE</code> statements.</td>
</tr>
<tr>
<td>policyupdate</td>
<td>boolean</td>
<td>not null</td>
<td>True if the policy is enforced by <code>UPDATE</code> statements.</td>
</tr>
<tr>
<td>policyindex</td>
<td>boolean</td>
<td>not null</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>policyenabled</td>
<td>boolean</td>
<td>not null</td>
<td>True if the policy is enabled.</td>
</tr>
<tr>
<td>policyupdatecheck</td>
<td>boolean</td>
<td>not null</td>
<td>True if rows updated by an <code>UPDATE</code> statement must satisfy the policy.</td>
</tr>
<tr>
<td>policystatic</td>
<td>boolean</td>
<td>not null</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>policytype</td>
<td>integer</td>
<td>not null</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>policyopts</td>
<td>integer</td>
<td>not null</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>policyseccols</td>
<td>int2vector</td>
<td>not null</td>
<td>The column numbers for columns listed in <code>sec_relevant_cols</code>.</td>
</tr>
</tbody>
</table>

**edb_profile**

The `edb_profile` table stores information about the available profiles. `edb_profiles` is shared across all databases in a cluster.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>References</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>oid</td>
<td></td>
<td>Row identifier (hidden attribute. Must be explicitly selected).</td>
</tr>
<tr>
<td>prfname</td>
<td>name</td>
<td></td>
<td>The name of the profile.</td>
</tr>
<tr>
<td>prffailedloginattempts</td>
<td>integer</td>
<td></td>
<td>The number of failed login attempts allowed by the profile. -1 indicates to use the value from the default profile. -2 indicates no limit on failed login attempts.</td>
</tr>
<tr>
<td>prfpasswordlocktime</td>
<td>integer</td>
<td></td>
<td>The password lock time associated with the profile, in seconds. -1 indicates to use the value from the default profile. -2 indicates to lock the account permanently.</td>
</tr>
</tbody>
</table>
prfpasswordlifetime integer
The password lifetime associated with the profile, in seconds. -1 indicates to use the value from the default profile. -2 indicates that the password never expires.

prfpasswordgrace lifetime integer
The password grace time associated with the profile, in seconds. -1 indicates to use the value from the default profile. -2 indicates that the password never expires.

prfpasswordreuse time integer
The number of seconds that a user must wait before reusing a password. -1 indicates to use the value from the default profile. -2 indicates that the old passwords can never be reused.

prfpasswordreuse max integer
The number of password changes that have to occur before a password can be reused. -1 indicates to use the value from the default profile. -2 indicates that the old passwords can never be reused.

prfpasswordallow hashed
Specifies whether an encrypted password is allowed for use. The possible values are true/on/yes/1, false/off/no/0, and DEFAULT.

prfpasswordverifyfuncdb oid
The OID of the database in which the password verify function exists.

prfpasswordverifyfunc oid
The OID of the password verify function associated with the profile.

edb_variable
The edb_variable table contains one row for each package level variable (each variable declared in a package).

Column | Type | Modifiers | Description
--- | --- | --- | ---
varname | "name" | not null | The name of the variable.
varpack | oid | not null | The OID of the pg_namespace row that stores the package.
vartype | oid | not null | The OID of the pg_type row that defines the type of the variable.
varaccess | "char" | not null | + if the variable is visible outside of the package. - if the variable is visible only in the package. Note: Public variables are declared in the package header. Private variables are declared in the package body.
varsrc | text | | Contains the source of the variable declaration, including any default value expressions for the variable.
varseq | small int | not null | The order in which the variable was declared in the package.

pg_synonym
The pg_synonym table contains one row for each synonym created with the CREATE SYNONYM command or CREATE PUBLIC SYNONYM command.

Column | Type | Modifiers | Description
--- | --- | --- | ---
synname | "name" | not null | The name of the synonym.
synnamespace | oid | not null | Replaces synowner. Contains the OID of the pg_namespace row where the synonym is stored.
synowner | oid | not null | The OID of the user that owns the synonym.
synobjschemaname | "name" | not null | The schema in which the referenced object is defined.
synobjname | "name" | not null | The name of the referenced object.
synlink | text | | The optional name of the database link in which the referenced object is defined.

product_component_version
The product_component_version table contains information about feature compatibility. An application can query this table during installation or at runtime to verify that features used by the application are available with this deployment.

Column | Type | Description
--- | --- | ---
product | character varying (74) | The name of the product.
version | character varying (74) | The version number of the product.
status | character varying (74) | The status of the release.
14.1.3 Language element reference

ECPGPlus has these language elements.

14.1.3.1 C-preprocessor directives

The ECPGPlus C-preprocessor enforces two behaviors that depend on the mode in which you invoke ECPGPlus:

- **PROC** mode
- **Non-PROC** mode

 Compiling in PROC mode

In **PROC** mode, ECPGPlus allows you to:

- Declare host variables outside of an `EXEC SQL BEGIN/END DECLARE SECTION`.
- Use any C variable as a host variable as long as its data type is compatible with ECPG.

When you invoke ECPGPlus in **PROC** mode (by including the `-C PROC` keywords), the ECPG compiler honors the following C-preprocessor directives:

```c
#include
#if expression
#define symbolName
#undef symbolName
#else
#else
#elif expression
#endif
#define symbolName(expansion)
define symbolName(macro arguments)
define(symbolName)
#endif
```

Preprocessor directives are used to affect or direct the code that’s received by the compiler. For example, consider the following code sample:

```c
#if HAVE_LONG_LONG == 1
#define BALANCE_TYPE long long
#else
#define BALANCE_TYPE double
#endif
... BALANCE_TYPE customerBalance;
```

Suppose you invoke ECPGPlus with the following command-line arguments:

```
ecpg -C PROC -D HAVE_LONG_LONG=1
```

ECPGPlus copies the entire fragment, without change, to the output file. It sends only the following tokens to the ECPG parser:

```
long long customerBalance;
```

On the other hand, suppose you invoke ECPGPlus with the following command-line arguments:

```
ecpg -C PROC -D HAVE_LONG_LONG=0
```

The ECPG parser receives the following tokens:

```
double customerBalance;
```

If your code uses preprocessor directives to filter the code that’s sent to the compiler, the complete code is retained in the original code, while the ECPG parser sees only the processed token stream.

You can also use compatible syntax when executing the following preprocessor directives with an `EXEC` directive:

```
#include
#if expression
#define symbolName
#undef symbolName
#else
#else
#elif expression
#endif
#define symbolName(expansion)
define symbolName(macro arguments)
define(symbolName)
#endif
```
For example, suppose your code includes the following:

```c
EXEC ORACLE
DEFINE ORACLE
DEFINE ORACLE
UNDEF ORACLE
EXEC ORACLE
INCLUDE ORACLE
IFDEF ORACLE
IFDEF ORACLE
ELSE
EXEC ORACLE
ENDIF
EXEC ORACLE
OPTION
```

You invoke ECPGPlus with the following command-line arguments:

```bash
ecp -C PROC DEFINE=HAVE_LONG_LONG=1
```

ECPGPlus sends the following tokens to the output file and the ECPG parser:

```c
long long customerBalance;
```

**Note**

The EXEC ORACLE preprocessor directives work only if you specify `-C PROC` on the ECPG command line.

### Using the SELECT_ERROR precompiler option

When using ECPGPlus in compatible mode, you can use the `SELECT_ERROR` precompiler option to tell your program how to handle result sets that contain more rows than the host variable can accommodate. The syntax is:

```c
SELECT_ERROR={YES|NO}
```

The default value is `YES`. A `SELECT` statement returns an error message if the result set exceeds the capacity of the host variable. Specify `NO` to suppress error messages when a `SELECT` statement returns more rows than a host variable can accommodate.

Use `SELECT_ERROR` with the `EXEC ORACLE OPTION` directive.

### Compiling in non-PROC mode

If you don't include the `-C PROC` command-line option:

- C preprocessor directives are copied to the output file without change.
- You must declare the type and name of each C variable that you intend to use as a host variable in an `EXEC SQL BEGIN/END DECLARE` section.

When invoked in non-PROC mode, ECPG implements the behavior described in the PostgreSQL core documentation.

#### 14.1.3.2 Language element reference

An embedded SQL statement allows your client application to interact with the server. An embedded directive is an instruction to the ECPGPlus compiler.

You can embed any EDB Postgres Advanced Server SQL statement in a C program. Each statement must begin with the keywords `EXEC SQL` and must be terminated with a semi-colon (`;`). In the C program, a SQL statement takes the form:
ALLOCATE DESCRIPTOR

Use the **ALLOCATE DESCRIPTOR** statement to allocate an SQL descriptor area:

```sql
EXEC SQL
ALLOCATE DESCRIPTOR
WHERE:
array_size is a variable that specifies the number of array elements to allocate for the descriptor. array_size can be an INTEGER value or a host variable.
descriptor_name is the host variable that contains the name of the descriptor or the name of the descriptor. This value can take the form of an identifier, a quoted string literal, or of a host variable.
variable_count specifies the maximum number of host variables in the descriptor. The default value of variable_count is 100.
```

The following code fragment allocates a descriptor named `emp_query` that can be processed as an array (`emp_array`):

```sql
EXEC SQL FOR emp_array ALLOCATE DESCRIPTOR
emp_query;
```

CALL

Use the **CALL** statement to invoke a procedure or function on the server. The **CALL** statement works only on EDB Postgres Advanced Server. The **CALL** statement comes in two forms. The first form is used to call a function:

```sql
EXEC SQL CALL <program_name>
'('<actual_arguments>)'
INTO [[:<ret_variable>][:<ret_indicator>]]; 
```

The second form is used to call a procedure:

```sql
EXEC SQL CALL <program_name>
'('<actual_arguments>)'; 
```

Where:

- *program_name* is the name of the stored procedure or function that the **CALL** statement invokes. The program name can be schema qualified, package qualified, or both. If you don’t specify the schema or package in which the program resides, ECPGPlus uses the value of search_path to locate the program.
- *actual_arguments* specifies a comma-separated list of arguments required by the program. Each *actual_argument* corresponds to a formal argument expected by the program. Each formal argument can be an IN parameter, an OUT parameter, or an INOUT parameter.
- *:ret_variable* specifies a host variable that receives the value returned if the program is a function.
- *:ret_indicator* specifies a host variable that receives the indicator value returned if the program is a function.

For example, the following statement invokes the `get_job_desc` function with the value contained in the `:ename` host variable and captures the value returned by that function in the `:job` host variable:

```sql
EXEC SQL CALL get_job_desc(:ename)
INTO :job;
```

CLOSE

Use the **CLOSE** statement to close a cursor and free any resources currently in use by the cursor. A client application cannot fetch rows from a closed cursor. The syntax of the **CLOSE** statement is:

```sql
EXEC SQL CLOSE
[cursor_name];
```
Where \texttt{cursor\_name} is the name of the cursor closed by the statement. The cursor name can take the form of an identifier or of a host variable.

The \texttt{OPEN} statement initializes a cursor. Once initialized, a cursor result set remains unchanged unless the cursor is reopened. You don’t need to \texttt{CLOSE} a cursor before reopening it.

To manually close a cursor named \texttt{emp\_cursor}, use the command:

\begin{verbatim}
EXEC SQL CLOSE
emp_cursor;
\end{verbatim}

A cursor is automatically closed when an application terminates.

**COMMIT**

Use the \texttt{COMMIT} statement to complete the current transaction, making all changes permanent and visible to other users. The syntax is:

\begin{verbatim}
EXEC SQL [AT \texttt{database\_name}] COMMIT
[WORK]
[COMMENT \texttt{<text>}]
[COMMENT \texttt{<text>}]
RELEASE;
\end{verbatim}

Where \texttt{database\_name} is the name of the database or host variable that contains the name of the database in which the work resides. This value can take the form of an unquoted string literal or of a host variable.

For compatibility, ECPGPlus accepts the \texttt{COMMENT} clause without error but doesn’t store any text included with the \texttt{COMMENT} clause.

Include the \texttt{RELEASE} clause to close the current connection after performing the commit.

For example, the following command commits all work performed on the \texttt{dept} database and closes the current connection:

\begin{verbatim}
EXEC SQL AT dept COMMIT
RELEASE;
\end{verbatim}

By default, statements are committed only when a client application performs a \texttt{COMMIT} statement. Include the \texttt{--t} option when invoking ECPGPlus to specify for a client application to invoke \texttt{AUTOCOMMIT} functionality. You can also control \texttt{AUTOCOMMIT} functionality in a client application with the following statements:

\begin{verbatim}
EXEC SQL SET AUTOCOMMIT TO
ON

EXEC SQL SET AUTOCOMMIT TO
OFF
\end{verbatim}

**CONNECT**

Use the \texttt{CONNECT} statement to establish a connection to a database. The \texttt{CONNECT} statement is available in two forms. One form is compatible with Oracle databases, and the other is not.

The first form is compatible with Oracle databases:

\begin{verbatim}
EXEC SQL CONNECT
\{ [:user\_name] IDENTIFIED BY :<password> \} |
]\{:connection\_id\}
[AT \texttt{database\_name}]
[USING \texttt{database\_string}]
[ALTER AUTHORIZATION :new\_password];
\end{verbatim}

Where:

- \texttt{user\_name} is a host variable that contains the role that the client application uses to connect to the server.
- \texttt{password} is a host variable that contains the password associated with that role.
- \texttt{connection\_id} is a host variable that contains a slash-delimited user name and password used to connect to the database.

Include the \texttt{AT} clause to specify the database to which the connection is established. \texttt{database\_name} is the name of the database to which the client is connecting. Specify the value in the form of a variable or as a string literal.

Include the \texttt{USING} clause to specify a host variable that contains a null-terminated string identifying the database to which to establish the connection.
The `ALTER AUTHORIZATION` clause is supported for syntax compatibility only. ECPGPlus parses the `ALTER AUTHORIZATION` clause and reports a warning.

Using the first form of the `CONNECT` statement, a client application might establish a connection with a host variable named `user` that contains the identity of the connecting role and a host variable named `password` that contains the associated password using the following command:

```
EXEC SQL CONNECT :user IDENTIFIED BY :password;
```

A client application can also use the first form of the `CONNECT` statement to establish a connection using a single host variable named `:connection_id`. In the following example, `connection_id` contains the slash-delimited role name and associated password for the user:

```
EXEC SQL CONNECT :connection_id;
```

The syntax of the second form of the `CONNECT` statement is:

```
EXEC SQL CONNECT TO
{<database_name> AS <connection_name>}
{<credentials>};
```

Where `credentials` is one of the following:

- `USER user_name password`
- `USER user_name IDENTIFIED BY password`
- `USER user_name USING password`

In the second form:

- `<database_name>` is the name or identity of the database to which the client is connecting. Specify `database_name` as a variable or as a string literal in one of the following forms:
  - `<database_name>@<hostname>[:<port>]
  - `tcp:postgresql://<hostname>[:<port>][/<database_name>][options]`
  - `unix:postgresql://<hostname>[:<port>][/<database_name>][options]`

Where:

- `hostname` is the name or IP address of the server on which the database resides.
- `port` is the port on which the server listens.

You can also specify a value of `DEFAULT` to establish a connection with the default database, using the default role name. If you specify `DEFAULT` as the target database, don't include a `connection_name` or `credentials`.

- `connection_name` is the name of the connection to the database. `connection_name` takes the form of an identifier (that is, not a string literal or a variable). You can open multiple connections by providing a unique `connection_name` for each connection.

If you don't specify a name for a connection, `ecpglib` assigns a name of `DEFAULT` to the connection. You can refer to the connection by name (`DEFAULT`) in any `EXEC SQL` statement.

- `CURRENT` is the most recently opened or the connection mentioned in the most-recent `SET CONNECTION TO` statement. If you don't refer to a connection by name in an `EXEC SQL` statement, ECPG assumes the name of the connection to be `CURRENT`.

- `user_name` is the role used to establish the connection with the EDB Postgres Advanced Server database. The privileges of the specified role are applied to all commands performed through the connection.

- `password` is the password associated with the specified `user_name`.

The following code fragment uses the second form of the `CONNECT` statement to establish a connection to a database named `edb` using the role `alice` and the password associated with that role, `1safepwd`:

```
EXEC SQL CONNECT TO edb AS acctg_conn
    USER 'alice' IDENTIFIED BY '1safepwd';
```

The name of the connection is `acctg_conn`. You can use the connection name when changing the connection name using the `SET CONNECTION` statement.

**DEALLOCATE DESCRIPTOR**

Use the `DEALLOCATE DESCRIPTOR` statement to free memory in use by an allocated descriptor. The syntax of the statement is:
EXEC SQL DEALLOCATE DESCRIPTOR
<descriptor_name>

Where descriptor_name is the name of the descriptor. This value can take the form of a quoted string literal or of a host variable.

The following example deallocates a descriptor named emp_query:

EXEC SQL DEALLOCATE DESCRIPTOR
emp_query;

DECLARE CURSOR

Use the DECLARE CURSOR statement to define a cursor. The syntax of the statement is:

EXEC SQL [AT <database_name>] DECLARE <cursor_name> CURSOR
FOR
(<select_statement> |
<statement_name>);

Where:

- database_name is the name of the database on which the cursor operates. This value can take the form of an identifier or of a host variable. If you don't specify a database name, the default value of database_name is the default database.
- cursor_name is the name of the cursor.
- select_statement is the text of the SELECT statement that defines the cursor result set. The SELECT statement can't contain an INTO clause.
- statement_name is the name of a SQL statement or block that defines the cursor result set.

The following example declares a cursor named employees:

EXEC SQL DECLARE employees CURSOR
FOR

SELECT
  empno, ename, sal,
  comm
FROM
  emp;

The cursor generates a result set that contains the employee number, employee name, salary, and commission for each employee record that's stored in the emp table.

DECLARE DATABASE

Use the DECLARE DATABASE statement to declare a database identifier for use in subsequent SQL statements (for example, in a CONNECT statement). The syntax is:

EXEC SQL DECLARE <database_name> DATABASE;

Where database_name specifies the name of the database.

The following example shows declaring an identifier for the acctg database:

EXEC SQL DECLARE acctg DATABASE;

After invoking the command declaring acctg as a database identifier, you can reference the acctg database by name when establishing a connection or in AT clauses.

This statement has no effect and is provided for Pro*C compatibility only.

DECLARE STATEMENT

Use the DECLARE STATEMENT directive to declare an identifier for an SQL statement. EDB Postgres Advanced Server supports two versions of the DECLARE STATEMENT directive:

EXEC SQL [<database_name>] DECLARE <statement_name> STATEMENT;

and

EXEC SQL DECLARE STATEMENT
<statement_name>;

EDB Postgres Advanced Server (EPAS) Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Where:

- `statement_name` specifies the identifier associated with the statement.
- `database_name` specifies the name of the database. This value may take the form of an identifier or of a host variable that contains the identifier.

A typical usage sequence that includes the `DECLARE STATEMENT` directive is:

```
EXEC SQL DECLARE give_raise STATEMENT;       // give_raise is now a statement
handle (not prepared)
EXEC SQL PREPARE give_raise FROM :stmtText; // give_raise is now associated with a statement
EXEC SQL EXECUTE
    give_raise;
```

This statement has no effect and is provided for Pro*C compatibility only.

**DELETE**

Use the `DELETE` statement to delete one or more rows from a table. The syntax for the ECPGPlus `DELETE` statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that an expression is allowed. The syntax is:

```
[FOR {exec_count}] DELETE FROM [ONLY] {table} [[AS]
    {alias}] [USING
    <using_list>] [WHERE <condition> | WHERE CURRENT OF
    <cursor_name>] [[RETURNING {RETURN} * | <output_expression>] [[AS]
    <output_name>] [,...] INTO <host_variable_list>
```

- Include the `FOR exec_count` clause to specify the number of times the statement executes. This clause is valid only if the `VALUES` clause references an array or a pointer to an array.
- `table` is the name (optionally schema qualified) of an existing table. Include the `ONLY` clause to limit processing to the specified table. If you don't include the `ONLY` clause, any tables inheriting from the named table are also processed.
- `alias` is a substitute name for the target table.
- `using_list` is a list of table expressions, allowing columns from other tables to appear in the `WHERE` condition.
- Include the `WHERE` clause to specify the rows to delete. If you don't include a `WHERE` clause in the statement, `DELETE` deletes all rows from the table, leaving the table definition intact.
- `condition` is an expression, host variable, or parameter marker that returns a value of type `BOOLEAN`. Those rows for which `condition` returns true are deleted.
- `cursor_name` is the name of the cursor to use in the `WHERE CURRENT OF` clause. The row to be deleted is the one most recently fetched from this cursor. The cursor must be a nongrouping query on the `DELETE` statements target table. You can't specify `WHERE CURRENT OF` in a `DELETE` statement that includes a Boolean condition.

The `RETURN/RETURNING` clause specifies an `output_expression` or `host_variable_list` that's returned by the `DELETE` command after each row is deleted:

- `output_expression` is an expression to be computed and returned by the `DELETE` command after each row is deleted. `output_name` is the name of the returned column. Include * to return all columns.
- `host_variable_list` is a comma-separated list of host variables and optional indicator variables. Each host variable receives a corresponding value from the `RETURNING` clause.

For example, the following statement deletes all rows from the `emp` table, where the `sal` column contains a value greater than the value specified in the host variable, `:max_sal`:

```
DELETE FROM emp WHERE sal > :max_sal;
```

For more information about using the `DELETE` statement, see the PostgreSQL core documentation.

**DESCRIBE**

Use the `DESCRIBE` statement to find the number of input values required by a prepared statement or the number of output values returned by a prepared statement. The `DESCRIBE` statement is used to analyze a SQL statement whose shape is unknown at the time you write your application.

The `DESCRIBE` statement populates an `SQLDA` descriptor. To populate a SQL descriptor, use the `ALLOCATE DESCRIPTOR` and `DESCRIBE...DESCRIPTOR` statements.

```
EXEC SQL DESCRIBE BIND VARIABLES FOR <statement_name> INTO
    <descriptor>;
EXEC SQL DESCRIBE SELECT LIST FOR <statement_name> INTO
    <descriptor>;
```

Where:
statement_name is the identifier associated with a prepared SQL statement or PL/SQL block. descriptor is the name of C variable of type SQLDA*. You must allocate the space for the descriptor by calling sqlald() and initialize the descriptor before executing the DESCRIBE statement.

When you execute the first form of the DESCRIBE statement, ECPG populates the given descriptor with a description of each input variable required by the statement. For example, given two descriptors:

SQLDA *query_values_in;
SQLDA *query_values_out;

You might prepare a query that returns information from the emp table:

EXEC SQL PREPARE get_emp
  FROM
    "SELECT ename, empno, sal FROM emp WHERE empno = ?";

The command requires one input variable for the parameter marker (?).

EXEC SQL DESCRIPTOR SELECT
  FOR get_emp INTO
    query_values_in;

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

EXEC SQL DESCRIPTOR SELECT
  FOR get_emp INTO
    query_values_out;

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

Before executing the statement, you must bind a variable for each input value and a variable for each output value. The variables that you bind for the input values specify the actual values used by the statement. The variables that you bind for the output values tell ECPGPlus where to put the values when you execute the statement.

This is alternative Pro*C-compatible syntax for the DESCRIBE DESCRIPTOR statement.

DESCRIBE DESCRIPTOR

Use the DESCRIBE DESCRIPTOR statement to retrieve information about a SQL statement and store that information in a SQL descriptor. Before using DESCRIBE DESCRIPTOR, you must allocate the descriptor with the ALLOCATE DESCRIPTOR statement. The syntax is:

EXEC SQL [INPUT | OUTPUT]
  <statement_identifier>
  USING [SQL] DESCRIPTOR <descriptor_name>;

Where:

- statement_name is the name of a prepared SQL statement.
- descriptor_name is the name of the descriptor. descriptor_name can be a quoted string value or a host variable that contains the name of the descriptor.

If you include the INPUT clause, ECPGPlus populates the given descriptor with a description of each input variable required by the statement.

For example, given two descriptors:

EXEC SQL ALLOCATE DESCRIPTOR
  query_values_in;
EXEC SQL ALLOCATE DESCRIPTOR
  query_values_out;

You might prepare a query that returns information from the emp table:

EXEC SQL PREPARE get_emp
  FROM
    "SELECT ename, empno, sal FROM emp WHERE empno = ?";

The command requires one input variable for the parameter marker (?).

EXEC SQL DESCRIBE INPUT get_emp USING
  'query_values_in';

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.
If you don’t specify the **INPUT** clause, **DESCRIBE DESCRIPTOR** populates the specified descriptor with the values returned by the statement.

If you include the **OUTPUT** clause, ECPGPlus populates the given descriptor with a description of each value returned by the statement.

For example, the following statement returns three values:

```sql
EXEC SQL DESCRIBE OUTPUT FOR get_emp USING
{query_values_out};
```

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

### DISCONNECT

Use the **DISCONNECT** statement to close the connection to the server. The syntax is:

```sql
EXEC SQL DISCONNECT [<connection_name>][CURRENT][DEFAULT]
[ALL];
```

Where **connection_name** is the connection name specified in the **CONNECT** statement used to establish the connection. If you don’t specify a connection name, the current connection is closed.

Include the **CURRENT** keyword to specify for ECPGPlus to close the connection used most recently.

Include the **DEFAULT** keyword to specify for ECPGPlus to close the connection named **DEFAULT**. If you don’t specify a name when opening a connection, ECPGPlus assigns the name **DEFAULT** to the connection.

Include the **ALL** keyword to close all active connections.

The following example creates a connection named **hr_connection** that connects to the **hr** database and then disconnects from the connection:

```c
/* client.pgc*/
int main()
{
   EXEC SQL CONNECT TO hr AS
   connection_name;
   EXEC SQL DISCONNECT
   connection_name;
   return(0);
}
```

### EXECUTE

Use the **EXECUTE** statement to execute a statement previously prepared using an **EXEC SQL PREPARE** statement. The syntax is:

```sql
EXEC SQL [FOR <array_size>] EXECUTE
<statement_name>
[USING [DESCRIPTOR
SQLDA_descriptor]
| <host_variable> [[INDICATOR
:host_variable_variable]]]
```

Where:

- **array_size** is an integer value or a host variable that contains an integer value that specifies the number of rows to process. If you omit the **FOR** clause, the statement is executed once for each member of the array.
- **statement_name** specifies the name assigned to the statement when the statement was created using the **EXEC SQL PREPARE** statement.

Include the **USING** clause to supply values for parameters in the prepared statement:

- Include the **DESCRIPTOR SQLDA_descriptor** clause to provide an SQLDA descriptor value for a parameter.
- Use a **host_variable** (and an optional **indicator_variable**) to provide a user-specified value for a parameter.

The following example creates a prepared statement that inserts a record into the **emp** table:

```sql
EXEC SQL PREPARE add_emp (numeric, text, text, numeric)
AS
INSERT INTO emp VALUES($1, $2, $3, $4);
```

Each time you invoke the prepared statement, provide fresh parameter values for the statement:
EXEC SQL EXECUTE add_emp USING 8808, 'DAWSON', 'CLERK', 7788;
EXEC SQL EXECUTE add_emp USING 8801, 'EDWARDS', 'ANALYST', 7698;

EXECUTE DESCRIPTOR

Use the `EXECUTE` statement to execute a statement previously prepared by an `EXEC SQL PREPARE` statement, using an SQL descriptor. The syntax is:

```sql
EXEC SQL [FOR <array_size>] EXECUTE <statement_identifier> [USING [SQL] DESCRIPTOR <descriptor_name>] [INTO [SQL] DESCRIPTOR <descriptor_name>];
```

Where:

- `<array_size>` is an integer value or a host variable that contains an integer value that specifies the number of rows to process. If you omit the `FOR` clause, the statement is executed once for each member of the array.
- `<statement_identifier>` specifies the identifier assigned to the statement with the `EXEC SQL PREPARE` statement.
- `<descriptor_name>` specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the `USING` clause to specify values for any input parameters required by the prepared statement.

Include the `INTO` clause to specify a descriptor into which the `EXECUTE` statement writes the results returned by the prepared statement.

The following example executes the prepared statement, `give_raise`, using the values contained in the descriptor `stmtText`:

```sql
EXEC SQL PREPARE give_raise FROM :stmtText;
EXEC SQL EXECUTE give_raise USING DESCRIPTOR :stmtText;
```

EXECUTE...END EXEC

Use the `EXECUTE...END EXEC` statement to embed an anonymous block into a client application. The syntax is:

```sql
EXEC SQL [AT <database_name>] EXECUTE <anonymous_block> END-EXEC;
```

Where:

- `<database_name>` is the database identifier or a host variable that contains the database identifier. If you omit the `AT` clause, the statement executes on the current default database.
- `<anonymous_block>` is an inline sequence of PL/pgSQL or SPL statements and declarations. You can include host variables and optional indicator variables in the block. Each such variable is treated as an `IN/OUT` value.

The following example executes an anonymous block:

```sql
EXEC SQL BEGIN;
IF (current_user $= :admin_user_name) THEN
  DBMS_OUTPUT.PUT_LINE('You are an administrator');
END IF;
END-EXEC;
```

Note

The `EXECUTE...END EXEC` statement is supported only by EDB Postgres Advanced Server.

EXECUTE IMMEDIATE

Use the `EXECUTE IMMEDIATE` statement to execute a string that contains a SQL command. The syntax is:

```sql
EXEC SQL [AT <database_name>] EXECUTE IMMEDIATE <command_text>;
```
Where:

- `database_name` is the database identifier or a host variable that contains the database identifier. If you omit the `AT` clause, the statement executes on the current default database.
- `command_text` is the command executed by the `EXECUTE IMMEDIATE` statement.

This dynamic SQL statement is useful when you don't know the text of an SQL statement when writing a client application. For example, a client application might prompt a trusted user for a statement to execute. After the user provides the text of the statement as a string value, the statement is then executed with an `EXECUTE IMMEDIATE` command.

The statement text can't contain references to host variables. If the statement might contain parameter markers or returns one or more values, use the `PREPARE` and `DESCRIBE` statements.

The following example executes the command contained in the `:command_text` host variable:

```sql
EXEC SQL EXECUTE IMMEDIATE :
command_text;
```

**FETCH**

Use the `FETCH` statement to return rows from a cursor into an SQLDA descriptor or a target list of host variables. Before using a `FETCH` statement to retrieve information from a cursor, you must prepare the cursor using `DECLARE` and `OPEN` statements. The statement syntax is:

```sql
EXEC SQL [FOR <array_size>] FETCH <cursor>
  ([ USING DESCRIPTOR <SQLDA_descriptor> ]) [ INTO <target_list> ];
```

Where:

- `<array_size>` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.
- `<cursor>` is the name of the cursor from which rows are being fetched or a host variable that contains the name of the cursor.

If you include a `USING` clause, the `FETCH` statement populates the specified SQLDA descriptor with the values returned by the server.

If you include an `INTO` clause, the `FETCH` statement populates the host variables (and optional indicator variables) specified in the `target_list`.

The following code fragment declares a cursor named `employees` that retrieves the employee number, name, and salary from the `emp` table:

```sql
EXEC SQL DECLARE employees CURSOR FOR
  SELECT empno, ename, esal FROM emp;
EXEC SQL OPEN employees;
EXEC SQL FETCH employees INTO :emp_no, :emp_name,
  :emp_sal;
```

**FETCH DESCRIPTOR**

Use the `FETCH DESCRIPTOR` statement to retrieve rows from a cursor into an SQL descriptor. The syntax is:

```sql
EXEC SQL [FOR <array_size>] FETCH <cursor>
  INTO [SQL] DESCRIPTOR <descriptor_name>;
```

Where:

- `<array_size>` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.
- `<cursor>` is the name of the cursor from which rows are fetched or a host variable that contains the name of the cursor.
- `<descriptor_name>` specifies the name of a descriptor (as a single-quoted string literal) or a host variable that contains the name of a descriptor. Prior to use, the descriptor must be allocated using an `ALLOCATE DESCRIPTOR` statement.

Include the `INTO` clause to specify a SQL descriptor into which the `EXECUTE` statement writes the results returned by the prepared statement.

The following example allocates a descriptor named `row_desc` that holds the description and the values of a specific row in the result set. It then declares and opens a cursor for a prepared statement (`my_cursor`), before looping through the rows in result set, using a `FETCH` to retrieve the next row from the cursor into the descriptor.
GET DESCRIPTOR

Use the `GET DESCRIPTOR` statement to retrieve information from a descriptor. The `GET DESCRIPTOR` statement comes in two forms. The first form returns the number of values (or columns) in the descriptor.

The second form returns information about a specific value (specified by the `VALUE column_number` clause):

```sql
EXEC SQL GET DESCRIPTOR
<descriptor_name>
:host_variable = COUNT;
```

The second form returns information about a specific value (specified by the `VALUE column_number` clause):

```sql
EXEC SQL [FOR <array_size>] GET DESCRIPTOR <descriptor_name>
VALUE <column_number> [:host_variable] = <descriptor_item> {,…};
```

Where:

- `array_size` is an integer value or a host variable that contains an integer value that specifies the number of rows to process. If you specify an `array_size`, the `host_variable` must be an array of that size. For example, if `array_size` is 10, `:host_variable` must be a 10-member array of `host_variables`. If you omit the `FOR` clause, the statement is executed once for each member of the array.
- `descriptor_name` specifies the name of a descriptor as a single-quoted string literal or a host variable that contains the name of a descriptor.
- `column_number` identifies the position of the variable in the descriptor.
- `host_variable` specifies the name of the host variable that receives the value of the item.
- `descriptor_item` specifies the type of the retrieved descriptor item.

ECPGPlus implements the following `descriptor_item` types:

- `TYPE`
- `LENGTH`
- `OCTET_LENGTH`
- `RETURNED_LENGTH`
- `RETURNED_OCTET_LENGTH`
- `PRECISION`
- `SCALE`
- `NULLABLE`
- `INDICATOR`
- `DATA`
- `NAME`

The following code fragment shows using a `GET DESCRIPTOR` statement to obtain the number of columns entered in a user-provided string:

```sql
EXEC SQL ALLOCATE DESCRIPTOR 'row_desc';
EXEC SQL DECLARE my_cursor CURSOR FOR query;
EXEC SQL OPEN my_cursor;
for ( row = 0; ; row++ )
{
    EXEC SQL BEGIN DECLARE SECTION;
    int col;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL FETCH my_cursor INTO SQL DESCRIPTOR 'row_desc';
}
```

The example allocates an SQL descriptor named `parse_desc` before using a `PREPARE` statement to check the syntax of the string provided by the user `:stmt`. A `DESCRIBE` statement moves the user-provided string into the descriptor, `parse_desc`. The call to `EXEC SQL GET DESCRIPTOR` interrogates the descriptor to discover the number of columns `:col_count` in the result set.
**INSERT**

Use the `INSERT` statement to add one or more rows to a table. The syntax for the ECPGPlus `INSERT` statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables anywhere that a value is allowed. The syntax is:

```
[FOR <exec_count>] INSERT INTO <table> [(<column> [, ...])]
(DEFAULT VALUES |
VALUES ((<expression> | DEFAULT) [, ...]), ...) |
<query>
[RETURNING * | <output_expression> [[ AS ] <output_name>] [, ...]]
```

Include the `FOR <exec_count>` clause to specify the number of times the statement executes. This clause is valid only if the `VALUES` clause references an array or a pointer to an array.

- `table` specifies the (optionally schema-qualified) name of an existing table.
- `column` is the name of a column in the table. The column name can be qualified with a subfield name or array subscript. Specify the `DEFAULT VALUES` clause to use default values for all columns.
- `expression` is the expression, value, host variable, or parameter marker that's assigned to the corresponding column. Specify `DEFAULT` to fill the corresponding column with its default value.
- `query` specifies a `SELECT` statement that supplies the rows to insert.
- `output_expression` is an expression that's computed and returned by the `INSERT` command after each row is inserted. The expression can refer to any column in the table. Specify `*` to return all columns of the inserted rows.
- `output_name` specifies a name to use for a returned column.

The following example adds a row to the `employees` table:

```sql
INSERT INTO emp (empno, ename, job, hiredate)
VALUES ('8400', :ename, 'CLERK', '2011-10-31');
```

**Note**

The `INSERT` statement uses a host variable `:ename` to specify the value of the `ename` column.

For more information about using the `INSERT` statement, see the PostgreSQL core documentation.

**OPEN**

Use the `OPEN` statement to open a cursor. The syntax is:

```
EXEC SQL [FOR <array_size>] OPEN <cursor> [USING <parameters>];
```

`parameters` is one of the following:
- `DESCRIPTOR <SQLDA_descriptor>`
- `<host_variable> [ [ INDICATOR ] <indicator_variable>, .. ]`

Where:
- `array_size` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.
- `cursor` is the name of the cursor being opened.
- `parameters` is either `DESCRIPTOR SQLDA_descriptor` or a comma-separated list of `host variables` and optional `indicator variables` that initialize the cursor. If specifying an `SQLDA_descriptor`, the descriptor must be initialized with a `DESCRIBE` statement.

The `OPEN` statement initializes a cursor using the values provided in `parameters`. Once initialized, the cursor result set remains unchanged unless the cursor is closed and reopened. A cursor is automatically closed when an application terminates.

The following example declares a cursor named `employees` that queries the `emp` table. It returns the `employee number`, `name`, `salary`, and `commission` of an employee whose name matches a user-supplied value stored in the host variable `:emp_name`.

```sql
EXEC SQL DECLARE employees CURSOR
FOR
SELECT empno, ename, sal, comm
FROM
```
After declaring the cursor, the example uses an `OPEN` statement to make the contents of the cursor available to a client application.

### OPEN DESCRIPTOR

Use the `OPEN DESCRIPTOR` statement to open a cursor with a SQL descriptor. The syntax is:

```sql
EXEC SQL [FOR <array_size>] OPEN <cursor>
[USING [SQL] DESCRIPTOR <descriptor_name>]
[INTO [SQL] DESCRIPTOR <descriptor_name>];
```

Where:

- `array_size` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.
- `cursor` is the name of the cursor being opened.
- `descriptor_name` specifies the name of an SQL descriptor in the form of a single-quoted string literal or a host variable that contains the name of an SQL descriptor that contains the query that initializes the cursor.

For example, the following statement opens a cursor named `emp_cursor` using the host variable `:employees`:

```sql
EXEC SQL OPEN emp_cursor USING DESCRIPTOR :employees;
```

### PREPARE

Prepared statements are useful when a client application must perform a task multiple times. The statement is parsed, written, and planned only once rather than each time the statement is executed. This approach saves repetitive processing time.

Use the `PREPARE` statement to prepare a SQL statement or PL/pgSQL block for execution. The statement is available in two forms. The first form is:

```sql
EXEC SQL [AT <database_name>] PREPARE <statement_name> FROM <sql_statement>;
```

The second form is:

```sql
EXEC SQL [AT <database_name>] PREPARE <statement_name> AS <sql_statement>;
```

Where:

- `database_name` is the database identifier or a host variable that contains the database identifier against which the statement executes. If you omit the `AT` clause, the statement executes against the current default database.
- `statement_name` is the identifier associated with a prepared SQL statement or PL/SQL block.
- `sql_statement` can take the form of a `SELECT` statement, a single-quoted string literal, or a host variable that contains the text of an SQL statement.

To include variables in a prepared statement, substitute placeholders (`$1`, `$2`, `$3`, and so on) for statement values that might change when you `PREPARE` the statement. When you `EXECUTE` the statement, provide a value for each parameter. Provide the values in the order in which they replace placeholders.

The following example creates a prepared statement named `add_emp` that inserts a record into the `emp` table:

```sql
EXEC SQL PREPARE add_emp (int, text, text, numeric) AS
    INSERT INTO emp VALUES($1, $2, $3, $4);
```

Each time you invoke the statement, provide fresh parameter values for the statement:

```sql
EXEC SQL EXECUTE add_emp(8003, 'Davis', 'CLERK', 2000.00);
```
A client application must issue a `PREPARE` statement in each session in which a statement executes. Prepared statements persist only for the duration of the current session.

**ROLLBACK**

Use the `ROLLBACK` statement to abort the current transaction and discard any updates made by the transaction. The syntax is:

```sql
EXEC SQL [AT <database_name>] ROLLBACK
[ [ ( TO [SAVEPOINT] <savepoint> ) | RELEASE ]
```

Where `database_name` is the database identifier or a host variable that contains the database identifier against which the statement executes. If you omit the `AT` clause, the statement executes against the current default database.

Include the `TO` clause to abort any commands that executed after the specified `savepoint`. Use the `SAVEPOINT` statement to define the `savepoint`. If you omit the `TO` clause, the `ROLLBACK` statement aborts the transaction, discarding all updates.

Include the `RELEASE` clause to cause the application to execute an `EXEC SQL COMMIT RELEASE` and close the connection.

Use the following statement to roll back a complete transaction:

```sql
EXEC SQL ROLLBACK;
```

Invoking this statement aborts the transaction, undoing all changes, erasing any savepoints, and releasing all transaction locks. Suppose you include a savepoint `my_savepoint` in the following example:

```sql
EXEC SQL ROLLBACK TO SAVEPOINT
my_savepoint;
```

Only the portion of the transaction that occurred after the `my_savepoint` is rolled back. `my_savepoint` is retained, but any savepoints created after `my_savepoint` are erased.

Rolling back to a specified savepoint releases all locks acquired after the savepoint.

**SAVEPOINT**

Use the `SAVEPOINT` statement to define a savepoint. A savepoint is a marker in a transaction. You can use a `ROLLBACK` statement to abort the current transaction, returning the state of the server to its condition prior to the specified savepoint. The syntax of a `SAVEPOINT` statement is:

```sql
EXEC SQL [AT <database_name>] SAVEPOINT
<savepoint_name>
```

Where:

- `database_name` is the database identifier or a host variable that contains the database identifier against which the savepoint resides. If you omit the `AT` clause, the statement executes against the current default database.
- `savepoint_name` is the name of the savepoint. If you reuse a `savepoint_name`, the original savepoint is discarded.

You can establish savepoints only in a transaction block. A transaction block can contain multiple savepoints.

To create a savepoint named `my_savepoint`, include the statement:

```sql
EXEC SQL SAVEPOINT
my_savepoint;
```

**SELECT**

ECPGPlus extends support of the SQL `SELECT` statement by providing the `INTO host_variables` clause. The clause allows you to select specified information from an EDB Postgres Advanced Server database into a host variable. The syntax for the `SELECT` statement is:

```sql
EXEC SQL [AT <database_name>]
```
Where:

- **database_name** is the name of the database or host variable that contains the name of the database in which the table resides. This value can take the form of an unquoted string literal or of a host variable.
- **host_variables** is a list of host variables populated by the `SELECT` statement. If the `SELECT` statement returns more than a single row, `host_variables` must be an array.

ECPGPlus provides support for the additional clauses of the SQL `SELECT` statement as documented in the PostgreSQL core documentation.

To use the `INTO host_variables` clause, include the names of defined host variables when specifying the `SELECT` statement. For example, the following `SELECT` statement populates the `:emp_name` and `:emp_sal` host variables with a list of employee names and salaries:

```sql
EXEC SQL SELECT ename, sal INTO :emp_name, :emp_sal
FROM emp
WHERE empno = 7988;
```

The enhanced `SELECT` statement also allows you to include parameter markers (question marks) in any clause where a value is allowed. For example, the following query contains a parameter marker in the `WHERE` clause:

```sql
SELECT * FROM emp WHERE dept_no = ?;
```

This `SELECT` statement allows you to provide a value at runtime for the `dept_no` parameter marker.

## SET CONNECTION

There are at least three reasons you might need more than one connection in a given client application:

- You might want different privileges for different statements.
- You might need to interact with multiple databases in the same client.
- Multiple threads of execution in a client application can't share a connection concurrently.

The syntax for the `SET CONNECTION` statement is:

```sql
EXEC SQL SET CONNECTION <connection_name>;
```

Where **connection_name** is the name of the connection to the database.

To use the `SET CONNECTION` statement, open the connection to the database using the second form of the `CONNECT` statement. Include the `AS` clause to specify a `connection_name`.

By default, the current thread uses the current connection. Use the `SET CONNECTION` statement to specify a default connection for the current thread to use. The default connection is used only when you execute an `EXEC SQL` statement that doesn't explicitly specify a connection name. For example, the following statement uses the default connection because it doesn't include an `AT connection_name` clause:
This statement doesn't use the default connection because it specifies a connection name using the `AT connection_name` clause:

```
EXEC SQL AT acctg_conn DELETE FROM emp;
```

For example, suppose a client application creates and maintains multiple connections using either of the following approaches:

```
EXEC SQL CONNECT TO edb AS acctg_conn
    USER 'alice' IDENTIFIED BY 'acctpwd';
```
```
EXEC SQL CONNECT TO edb AS hr_conn
    USER 'bob' IDENTIFIED BY 'hrpwd';
```

It can change between the connections with the `SET CONNECTION` statement:

```
SET CONNECTION acctg_conn;
```
```
or
```
SET CONNECTION hr_conn;
```

The server uses the privileges associated with the connection when determining the privileges available to the connecting client. When using the `acctg_conn` connection, the client has the privileges associated with the role `alice`. When connected using `hr_conn`, the client has the privileges associated with `bob`.

**SET DESCRIPTOR**

Use the `SET DESCRIPTOR` statement to assign a value to a descriptor area using information provided by the client application in the form of a host variable or an integer value. The statement comes in two forms. The first form is:

```
EXEC SQL [FOR <array_size>] SET DESCRIPTOR <descriptor_name>
    VALUE <column_number> <descriptor_item> = <host_variable>;
```

The second form is:

```
EXEC SQL [FOR <array_size>] SET DESCRIPTOR <descriptor_name>
    COUNT = integer;
```

Where:

- `array_size` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement executes once for each member of the array.
- `descriptor_name` specifies the name of a descriptor as a single-quoted string literal or a host variable that contains the name of a descriptor.

Include the `VALUE` clause to describe the information stored in the descriptor.

- `column_number` identifies the position of the variable within the descriptor.
- `descriptor_item` specifies the type of the descriptor item.
- `host_variable` specifies the name of the host variable that contains the value of the item.

ECPGPlus implements the following `descriptor_item` types:

- `TYPE`
- `LENGTH`
- `[REF] INDICATOR`
- `[REF] DATA`
- `[REF] RETURNED LENGTH`

For example, a client application might prompt a user for a dynamically created query:

```
query_text = promptUser("Enter a query");
```

To execute a dynamically created query, you must first prepare the query (parsing and validating the syntax of the query) and then describe the input parameters found in the query using the `EXEC SQL DESCRIBE INPUT` statement.
After describing the query, the `query_params` descriptor contains information about each parameter required by the query.

For this example, assume that the user entered:

```sql
SELECT ename FROM emp WHERE sal > ? AND job = ?;
```

In this case, the descriptor describes two parameters, one for `sal > ?` and one for `job = ?`.

To discover the number of parameter markers (question marks) in the query and therefore the number of values you must provide before executing the query, use:

```sql
EXEC SQL GET DESCRIPTOR ... COUNT
```

Then, you can use `EXEC SQL GET DESCRIPTOR` to retrieve the name of each parameter from the descriptor, along with the number of parameters. Or you can supply each value in the form of a character string and ECPG converts that string into the required data type.

The data type of the first parameter is `numeric`. The type of the second parameter is `varchar`. The name of the first parameter is `sal`. The name of the second parameter is `job`.

Next, loop through each parameter, prompting the user for a value, and store those values in host variables. You can use `GET DESCRIPTOR ... COUNT` to find the number of parameters in the query.

Use `GET DESCRIPTOR` to copy the name of the parameter into the `param_name` host variable:

```sql
EXEC SQL GET DESCRIPTOR 'query_params'
  VALUE :param_number :param_name = NAME;
```

To associate a value with each parameter, you use the `EXEC SQL SET DESCRIPTOR` statement. For example:

```sql
EXEC SQL SET DESCRIPTOR 'query_params'
  VALUE :param_number DATA = :reply;
EXEC SQL SET DESCRIPTOR 'query_params'
  VALUE :param_number INDICATOR = :reply_ind;
```

Now, you can use the `EXEC SQL EXECUTE DESCRIPTOR` statement to execute the prepared statement on the server.

**UPDATE**

Use an `UPDATE` statement to modify the data stored in a table. The syntax is:

```sql
EXEC SQL [AT <database_name>][FOR '<exec_count']
  UPDATE [ ONLY ] table [ [ AS ] alias
  ]
  SET column = { expression | DEFAULT }
|
WHERE database_name is the name of the database or host variable that contains the name of the database in which the table resides. This value can take the form of an unquoted string literal or of a host variable.

Include the FOR exec_count clause to specify the number of times the statement executes. This clause is valid only if the SET or WHERE clause contains an array.

ECPGPlus provides support for the additional clauses of the SQL UPDATE statement as documented in the PostgreSQL core documentation.

You can use a host variable in any clause that specifies a value. To use a host variable, substitute a defined variable for any value associated with any of the documented UPDATE clauses.

The following UPDATE statement changes the job description of an employee (identified by the :ename host variable) to the value contained in the :new_job host variable. It increases the employee's salary by multiplying the current salary by the value in the :increase host variable:

```sql
EXEC SQL UPDATE
emp
    SET job = :new_job, sal = sal * :increase
WHERE ename = :ename;
```

The enhanced UPDATE statement also allows you to include parameter markers (question marks) in any clause where an input value is permitted. For example, we can write the same update statement with a parameter marker in the WHERE clause:

```sql
EXEC SQL UPDATE
emp
    SET job = ?, sal = sal * ?
WHERE ename = :ename;
```

This UPDATE statement allows you to prompt the user for a new value for the job column and provide the amount by which the sal column is incremented for the employee specified by :ename.

WHENEVER

Use the WHENEVER statement to specify the action taken by a client application when it encounters an SQL error or warning. The syntax is:

```sql
EXEC SQL WHENEVER <condition> <action>
```

The following table describes the different conditions that might trigger an action.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT FOUND</td>
<td>The server returns a NOT FOUND condition when it encounters a SELECT that returns no rows or when a FETCH reaches the end of a result set.</td>
</tr>
<tr>
<td>SQLERROR</td>
<td>The server returns an SQLERROR condition when it encounters a serious error returned by an SQL statement.</td>
</tr>
<tr>
<td>SQLWARNING</td>
<td>The server returns an SQLWARNING condition when it encounters a nonfatal warning returned by an SQL statement.</td>
</tr>
</tbody>
</table>

The following table describes the actions that result from a client encountering a condition.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL function</td>
<td>Call the named function.</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Proceed to the next statement.</td>
</tr>
<tr>
<td>DO BREAK</td>
<td>Emit a C break statement. A break statement can appear in a loop or a switch statement. If executed, the break statement terminates the loop or the switch statement.</td>
</tr>
<tr>
<td>DO CONTINUE</td>
<td>Emit a C continue statement. A continue statement can exist only in a loop. If executed, it causes the flow of control to return to the top of the loop.</td>
</tr>
<tr>
<td>DO function</td>
<td>Call the named function.</td>
</tr>
<tr>
<td>GOTO label or GO TO label</td>
<td>Proceed to the statement that contains the label.</td>
</tr>
<tr>
<td>SQLPRINT</td>
<td>Print a message to standard error.</td>
</tr>
<tr>
<td>STOP</td>
<td>Stop executing.</td>
</tr>
</tbody>
</table>
The following code fragment prints a message if the client application encounters a warning and aborts the application if it encounters an error:

```sql
EXEC SQL WHENEVER SQLWARNING
  SQLPRINT;
EXEC SQL WHENEVER SQLERROR
  STOP;
```

Include the following code to specify for a client to continue processing after warning a user of a problem:

```sql
EXEC SQL WHENEVER SQLWARNING
  SQLPRINT;
```

Include the following code to call a function if a query returns no rows or when a cursor reaches the end of a result set:

```sql
EXEC SQL WHENEVER NOT FOUND CALL
  error_handler(__LINE__);
```

### 14.1.3.3 The SQLDA structure

Oracle Dynamic SQL method 4 uses the SQLDA data structure to hold the data and metadata for a dynamic SQL statement. A SQLDA structure can describe a set of input parameters corresponding to the parameter markers found in the text of a dynamic statement or the result set of a dynamic statement.

#### Layout

The layout of the SQLDA structure is:

```c
struct SQLDA {
  int N; /* Number of entries */
  char **V; /* Variables */
  int *L; /* Variable lengths */
  short *T; /* Variable types */
  short **I; /* Indicators */
  int *F; /* Count of variables discovered by DESCRIBE */
  char **S; /* Variable names */
  short *M; /* Variable name maximum lengths */
  short *C; /* Variable name actual lengths */
  char **X; /* Indicator names */
  short *Y; /* Indicator name maximum lengths */
  short *Z; /* Indicator name actual lengths */
};
```

#### Parameters

- **N** - maximum number of entries

The **N** structure member contains the maximum number of entries that the SQLDA can describe. This member is populated by the `sqlald()` function when you allocate the SQLDA structure. Before using a descriptor in an `OPEN` or `FETCH` statement, you must set **N** to the actual number of values described.

- **V** - data values

The **V** structure member is a pointer to an array of data values.

- For a `SELECT`-list descriptor, **V** points to an array of values returned by a `FETCH` statement. Each member in the array corresponds to a column in the result set.
- For a bind descriptor, **V** points to an array of parameter values. You must populate the values in this array before opening a cursor that uses the descriptor.

Your application must allocate the space required to hold each value. See `displayResultSet` for an example of how to allocate space for `SELECT`-list values.

- **L** - length of each data value
The S structure member points to an array of lengths. Each member of this array must indicate the amount of memory available in the corresponding member of the V array. For example, if V[5] points to a buffer large enough to hold a 20-byte NULL-terminated string, L[5] must contain the value 21 (20 bytes for the characters in the string plus 1 byte for the NULL-terminator). Your application must set each member of the L array.

T - data types

The T structure member points to an array of data types, one for each column (or parameter) described by the descriptor.

- For a bind descriptor, you must set each member of the T array to tell ECPGPlus the data type of each parameter.
- For a SELECT-list descriptor, the DESCRIBE SELECT LIST statement sets each member of the T array to reflect the type of data found in the corresponding column.

You can change any member of the T array before executing a FETCH statement to force ECPGPlus to convert the corresponding value to a specific data type. For example, if the DESCRIBE SELECT LIST statement indicates that a given column is of type DATE, you can change the corresponding T member to request that the next FETCH statement return that value in the form of a NULL-terminated string. Each member of the T array is a numeric type code (see Type Codes for a list of type codes). The type codes returned by a DESCRIBE SELECT LIST statement differ from those expected by a FETCH statement. After executing a DESCRIBE SELECT LIST statement, each member of T encodes a data type and a flag indicating whether the corresponding column is nullable. You can use the sqlnul() function to extract the type code and nullable flag from a member of the T array. The signature of the sqlnul() function is as follows:

```c
void sqlnul(unsigned short *valType, unsigned short *typeCode, int *isNull)
```

For example, to find the type code and nullable flag for the third column of a descriptor named results, invoke sqlnul() as follows:

```c
sqlnul(&results->T[2], &typeCode, &isNull);
```

I - indicator variables

The I structure member points to an array of indicator variables. This array is allocated for you when your application calls the sqlald() function to allocate the descriptor.

- For a SELECT-list descriptor, each member of the I array indicates whether the corresponding column contains a NULL (non-zero) or non-NULL (zero) value.
- For a bind parameter, your application must set each member of the I array to indicate whether the corresponding parameter value is NULL.

F - number of entries

The F structure member indicates how many values are described by the descriptor. The N structure member indicates the maximum number of values that the descriptor can describe. F indicates the actual number of values. The value of the F member is set by ECPGPlus when you execute a DESCRIBE statement. F can be positive, negative, or zero.

- For a SELECT-list descriptor, F contains a positive value if the number of columns in the result set is equal to or less than the maximum number of values permitted by the descriptor (as determined by the N structure member). It contains 0 if the statement isn’t a SELECT statement. It contains a negative value if the query returns more columns than allowed by the N structure member.
- For a bind descriptor, F contains a positive number if the number of parameters found in the statement is less than or equal to the maximum number of values permitted by the descriptor (as determined by the N structure member). It contains 0 if the statement contains no parameters markers. It contains a negative value if the statement contains more parameter markers than allowed by the N structure member.

If F contains a positive number (after executing a DESCRIBE statement), that number reflects the count of columns in the result set (for a SELECT-list descriptor) or the number of parameter markers found in the statement (for a bind descriptor). If F contains a negative value, you can compute the absolute value of F to discover how many values or parameter markers are required. For example, if F contains −24 after describing a SELECT list, you know that the query returns 24 columns.

S - column/parameter names

The S structure member points to an array of NULL-terminated strings.

- For a SELECT-list descriptor, the DESCRIBE SELECT LIST statement sets each member of this array to the name of the corresponding column in the result set.
- For a bind descriptor, the DESCRIBE BIND VARIABLES statement sets each member of this array to the name of the corresponding bind variable.

In this release, the name of each bind variable is determined by the left-to-right order of the parameter marker within the query. For example, the name of the first parameter is always ?0, the name of the second parameter is always ?1, and so on.

M - maximum column/parameter name length

The M structure member points to an array of lengths. Each member in this array specifies the maximum length of the corresponding member of the S array (that is, M[0] specifies the maximum length of the column/parameter name found at S[0]). This array is populated by the sqlald() function.

C - actual column/parameter name length

The C structure member points to an array of lengths. Each member in this array specifies the actual length of the corresponding member of the S array (that is, C[0] specifies the actual length of the column/parameter name found at S[0]).

This array is populated by the DESCRIBE statement.

X - indicator variable names
The \textit{X} structure member points to an array of NULL-terminated strings. Each string represents the name of a NULL indicator for the corresponding value.

This array isn't used by ECPGPlus but is provided for compatibility with Pro*C applications.

\begin{verbatim}
Y - maximum indicator name length
\end{verbatim}

The \textit{Y} structure member points to an array of lengths. Each member in this array specifies the maximum length of the corresponding member of the \textit{X} array (that is, \textit{Y}[0] specifies the maximum length of the indicator name found at \textit{X}[0]).

This array isn't used by ECPGPlus but is provided for compatibility with Pro*C applications.

\begin{verbatim}
Z - actual indicator name length
\end{verbatim}

The \textit{Z} structure member points to an array of lengths. Each member in this array specifies the actual length of the corresponding member of the \textit{X} array (that is, \textit{Z}[0] specifies the actual length of the indicator name found at \textit{X}[0]).

This array isn't used by ECPGPlus but is provided for compatibility with Pro*C applications.

### 14.1.3.4 Supported C data types

An ECPGPlus application must deal with two sets of data types: SQL data types (such as \texttt{SMALLINT}, \texttt{DOUBLE PRECISION}, and \texttt{CHARACTER VARYING}) and C data types (like \texttt{short}, \texttt{double}, and \texttt{varchar[n]}). When an application fetches data from the server, ECPGPlus maps each SQL data type to the type of the C variable into which the data is returned.

In general, ECPGPlus can convert most SQL server types into similar C types, but not all combinations are valid. For example, ECPGPlus tries to convert a SQL character value into a C integer value, but the conversion might fail at execution time if the SQL character value contains non-numeric characters.

The reverse is also true. When an application sends a value to the server, ECPGPlus tries to convert the C data type into the required SQL type. Again, the conversion might fail at execution time if the C value can't be converted into the required SQL type.

ECPGPlus can convert any SQL type into C character values (\texttt{char[n]} or \texttt{varchar[n]}). Although it's safe to convert any SQL type to or from \texttt{char[n]} or \texttt{varchar[n]}, it's often convenient to use more natural C types such as \texttt{int}, \texttt{double}, or \texttt{float}.

The supported C data types are:

- \texttt{short}
- \texttt{int}
- \texttt{unsigned int}
- \texttt{long long int}
- \texttt{float}
- \texttt{double}
- \texttt{char[n]}
- \texttt{varchar[n]}
- \texttt{bool}
- Any equivalent created by a \texttt{typedef}

In addition to the numeric and character types supported by C, the \texttt{pgtypeslib} runtime library offers custom data types, as well as functions to operate on those types, for dealing with date/time and exact numeric values:

- \texttt{timestamp}
- \texttt{interval}
- \texttt{date}
- \texttt{decimal}
- \texttt{numeric}

To use a data type supplied by \texttt{pgtypeslib}, you must \texttt{include} the proper header file.

### 14.1.3.5 Type codes

#### Type codes for external data types

The following table contains the type codes for external data types. An external data type is used to indicate the type of a C host variable. When an application binds a value to a parameter or binds a buffer to a \texttt{SELECT}-list item, set the type code in the corresponding SQLDA descriptor (\texttt{descriptor->T(column)}) to one of the following values:

<table>
<thead>
<tr>
<th>Type code</th>
<th>Host variable type (C data type)</th>
</tr>
</thead>
</table>

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The following table contains the type codes for internal data types. An internal type code is used to indicate the type of a value as it resides in the database. The `DESCRIBE SELECT LIST` statement populates the data type array `((descriptor->T[column]))` using the following values.

<table>
<thead>
<tr>
<th>Internal type code</th>
<th>Server type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>2</td>
<td>NUMBER</td>
</tr>
<tr>
<td>8</td>
<td>LONG</td>
</tr>
<tr>
<td>11</td>
<td>ROWID</td>
</tr>
<tr>
<td>12</td>
<td>DATE</td>
</tr>
<tr>
<td>23</td>
<td>RAW</td>
</tr>
<tr>
<td>24</td>
<td>LONG RAW</td>
</tr>
<tr>
<td>96</td>
<td>CHAR</td>
</tr>
<tr>
<td>108</td>
<td>BINARY FLOAT</td>
</tr>
<tr>
<td>101</td>
<td>BINARY DOUBLE</td>
</tr>
<tr>
<td>104</td>
<td>UROWID</td>
</tr>
<tr>
<td>187</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>188</td>
<td>TIMESTAMP W/TIMEZONE</td>
</tr>
<tr>
<td>189</td>
<td>INTERVAL YEAR TO MONTH</td>
</tr>
<tr>
<td>198</td>
<td>INTERVAL DAY TO SECOND</td>
</tr>
<tr>
<td>232</td>
<td>TIMESTAMP LOCAL_TZ</td>
</tr>
</tbody>
</table>

14.1.4 Stored procedural language (SPL) reference

EDB Postgres Advanced Server's stored procedural language (SPL) is a highly productive, procedural programming language for writing custom procedures, functions, triggers, and packages for EDB Postgres Advanced Server.

14.1.4.1 Basic SPL elements

The basic programming elements of an SPL program include aspects such as case sensitivity and the available character set.

14.1.4.1.1 Case sensitivity

Keywords and user-defined identifiers that are used in an SPL program are case insensitive. For example, the statement `DBMS_OUTPUT.PUT_LINE('Hello World!');` is interpreted as `dbms_output.put_line('Hello World!');` or `DBMS_OUTPUT.PUT_LINE('Hello World!');` or `DBMS_output.Put_line('Hello World!');`.

Character and string constants, however, are case sensitive. Data retrieved from the EDB Postgres Advanced Server database or from other external sources is also case sensitive.

The statement `DBMS_OUTPUT.PUT_LINE('Hello World!');` produces the following output:
The statement `DBMS_OUTPUT.PUT_LINE('HELLO WORLD!');` produces this output:

```
HELLO WORLD!
```

### 14.1.4.1.2 Identifiers

Identifiers are user-defined names that identify elements of an SPL program including variables, cursors, labels, programs, and parameters. The syntax rules for valid identifiers are the same as for identifiers in the SQL language.

An identifier can’t be the same as an SPL keyword or a keyword of the SQL language. The following are some examples of valid identifiers:

- `x`
- `last__name`
- `a_$_Sign`
- `Many$$$$$$$$signs_____`
- `THIS_IS_AN_EXTREMELY_LONG_NAME`
- `A1`

### 14.1.4.1.3 Qualifiers

A **qualifier** is a name that specifies the owner or context of an entity that’s the object of the qualifier. Specify a qualified object using these elements, in order:

1. The qualifier name
2. A dot with no intervening white space
3. The name of the object being qualified with no intervening white space

This syntax is called **dot notation**.

The following is the syntax of a qualified object:

```
<qualifier>. [ <qualifier>. ]... <object>
```

*qualifier* is the name of the owner of the object. *object* is the name of the entity belonging to *qualifier*. You can have a chain of qualifications in which the preceding qualifier owns the entity identified by the subsequent qualifiers and object.

You can qualify almost any identifier. How you qualify an identifier depends on what the identifier represents and its context.

Some examples of qualification follow:

- Procedure and function names qualified by the schema to which they belong, e.g., `schema_name.procedure_name(...)`
- Trigger names qualified by the schema to which they belong, e.g., `schema_name.trigger_name`
- Column names qualified by the table to which they belong, e.g., `emp.empno`
- Table names qualified by the schema to which they belong, e.g., `public.emp`
- Column names qualified by table and schema, e.g., `public.emp.empno`

As a general rule, where a name appears in the syntax of an SPL statement you can use its qualified name as well. Typically, a qualified name is used only if ambiguity is associated with the name.

Examples of ambiguity include, for example:

- Two procedures with the same name belonging to two different schemas are invoked from a program.
- The same name is used for a table column and SPL variable in the same program.

Avoid using qualified names if possible. We use the following conventions to avoid naming conflicts:

- All variables declared in the declaration section of an SPL program are prefixed by `v_`, e.g., `v_empno`.
- All formal parameters declared in a procedure or function definition are prefixed by `p_`, e.g., `p_empno`.
- Column names and table names don’t have any special prefix conventions, e.g., `empno` in table `emp`.

### 14.1.4.1.4 Constants

Constants or literals are fixed values that you can use in SPL programs to represent values of various types such as numbers, strings, and dates. Constants come in the following types:
14.1.4.1.5  User-defined PL/SQL subtypes

EDB Postgres Advanced Server supports user-defined PL/SQL subtypes and subtype aliases.

About subtypes

A subtype is a data type with an optional set of constraints that restrict the values that can be stored in a column of that type. The rules that apply to the type on which the subtype is based are still enforced, but you can use additional constraints to place limits on the precision or scale of values stored in the type.

You can define a subtype in the declaration of a PL function, procedure, anonymous block, or package. The syntax is:

```
SUBTYPE <subtype_name> IS <type_name>[(<constraint>)] [NOT NULL]
```

Where `constraint` is:

```
{<precision> [, <scale>]} | <length>
```

- `<subtype_name>` specifies the name of the subtype.
- `<type_name>` specifies the name of the original type on which the subtype is based. `<type_name>` can be:
  - The name of any of the types supported by EDB Postgres Advanced Server.
  - The name of any composite type.
  - A column anchored by a `%TYPE` operator.
  - The name of another subtype.

Include the `constraint` clause to define restrictions for types that support precision or scale.

- `<precision>` specifies the total number of digits permitted in a value of the subtype.
- `<scale>` specifies the number of fractional digits permitted in a value of the subtype.
- `<length>` specifies the total length permitted in a value of `CHARACTER`, `VARCHAR`, or `TEXT` base types.

Include the `NOT NULL` clause to specify that you can't store `NULL` values in a column of the specified subtype.

A subtype that is based on a column inherits the column size constraints, but the subtype doesn't inherit `NOT NULL` or `CHECK` constraints.

Unconstrained subtypes

To create an unconstrained subtype, use the `SUBTYPE` command to specify the new subtype name and the name of the type on which the subtype is based. For example, the following command creates a subtype named `address` that has all of the attributes of the type `CHAR`:

```
SUBTYPE address IS CHAR;
```

You can also create a subtype (constrained or unconstrained) that's a subtype of another subtype:

```
SUBTYPE cust_address IS address NOT NULL;
```
This command creates a subtype named `cust_address` that shares all of the attributes of the `address` subtype. Include the `NOT NULL` clause to specify that a value of the `cust_address` can't be `NULL`.

**Constrained subtypes**

Include a `length` value when creating a subtype that's based on a character type to define the maximum length of the subtype. For example:

```sql
SUBTYPE acct_name IS VARCHAR (15);
```

This example creates a subtype named `acct_name` that's based on a `VARCHAR` data type but is limited to 15 characters.

Include values for `precision` to specify the maximum number of digits in a value of the subtype. Optionally, include values for `scale` to specify the number of digits to the right of the decimal point when constraining a numeric base type. For example:

```sql
SUBTYPE acct_balance IS NUMBER (5, 2);
```

This example creates a subtype named `acct_balance` that shares all of the attributes of a `NUMBER` type but that can't exceed three digits to the left of the decimal point and two digits to the right of the decimal.

An argument declaration (in a function or procedure header) is a formal argument. The value passed to a function or procedure is an actual argument. When invoking a function or procedure, the caller provides 0 or more actual arguments. Each actual argument is assigned to a formal argument that holds the value in the body of the function or procedure.

If a formal argument is declared as a constrained subtype, EDB Postgres Advanced Server:

- Enforces subtype constraints when assigning an actual argument to a formal argument when invoking a procedure.
- Doesn't enforce subtype constraints when assigning an actual argument to a formal argument when invoking a function.

**Using the %TYPE operator**

You can use `%TYPE` notation to declare a subtype anchored to a column. For example:

```sql
SUBTYPE emp_type IS emp.empno%TYPE;
```

This command creates a subtype named `emp_type` whose base type matches the type of the `empno` column in the `emp` table. A subtype that's based on a column shares the column size constraints. `NOT NULL` and `CHECK` constraints aren't inherited.

**Subtype conversion**

Unconstrained subtypes are aliases for the type on which they are based. Any variable of type subtype (unconstrained) is interchangeable with a variable of the base type without conversion, and vice versa.

You can interchange a variable of a constrained subtype with a variable of the base type without conversion. However, you can interchange a variable of the base type with a constrained subtype only if it complies with the constraints of the subtype. You can implicitly convert a variable of a constrained subtype to another subtype if it's based on the same subtype and the constraint values are within the values of the subtype to which it is being converted.

### 14.1.4.1.6 Character set

Write identifiers, expressions, statements, control structures, and so on that comprise the SPL language using the following characters:

- Uppercase letters A–Z and lowercase letters a–z
- Digits 0–9
- Symbols `+ - * / < > = ! ~ ^ ; : . ' @ % , " # $ & _ | { } ? [ ]`
- Whitespace characters tab, space, and carriage return

**Note**

The data that can be manipulated by an SPL program is determined by the character set supported by the database encoding.

### 14.1.4.2 Types of programming statements
You can use several programming statements in an SPL program.

### 14.1.4.2.1 Assignment

The assignment statement sets a variable or a formal parameter of mode `OUT` or `IN OUT` specified on the left side of the assignment `:=` to the evaluated expression specified on the right side of the assignment.

```
<variable> := <expression>;
```

*variable* is an identifier for a previously declared variable, `OUT` formal parameter, or `IN OUT` formal parameter.

*expression* is an expression that produces a single value. The value produced by the expression must have a data type compatible with that of *variable*.

This example shows the typical use of assignment statements in the executable section of the procedure:

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt
    (p_deptno NUMBER)
IS
    todays_date DATE;
    rpt_title VARCHAR2(60);
    base_sal INTEGER;
    base_comm_rate NUMBER;
    base_annual NUMBER;
BEGIN
    todays_date := SYSDATE;
    rpt_title := 'Report For Department # ' || p_deptno || ' on ' || todays_date;
    base_sal := 35525;
    base_comm_rate := 1.33333;
    base_annual := ROUND(base_sal * base_comm_rate, 2);
    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

### 14.1.4.2.2 DELETE

You can use the `DELETE` command available in the SQL language in SPL programs.

You can use an expression in the SPL language wherever an expression is allowed in the SQL `DELETE` command. Thus, you can use SPL variables and parameters to supply values to the delete operation.

```
CREATE OR REPLACE PROCEDURE emp_delete
    (p_empno emp.empno%TYPE)
IS
    BEGIN
        DELETE FROM emp WHERE empno = p_empno;
        IF SQL%FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || p_empno);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
        END IF;
    END;
```
The `SQL%FOUND` conditional expression returns `TRUE` if a row is deleted, `FALSE` otherwise. See Obtaining the result status for a discussion of `SQL%FOUND` and other similar expressions.

This example deletes an employee using this procedure:

```sql
EXEC emp_delete(9503);
Deleted Employee # : 9503;
SELECT * FROM emp WHERE empno = 9503;
```

| empno | ename | job | mgr | hiredate | sal | comm | deptno |
|-------+-------+-----+-----+----------+-----+------+--------|
| -------+-------+-----+-----+----------+-----+------+--------|
| (0 rows) |

**Note**

You can include the `DELETE` command in a `FORALL` statement. A `FORALL` statement allows a single `DELETE` command to delete multiple rows from values supplied in one or more collections. See Using the `FORALL` statement for more information.

### 14.1.4.2.3 INSERT

You can use the `INSERT` command available in the SQL language in SPL programs.

You can use an expression in the SPL language wherever an expression is allowed in the SQL `INSERT` command. Thus, you can use SPL variables and parameters to supply values to the insert operation.

This example is a procedure that inserts a new employee using data passed from a calling program:

```sql
CREATE OR REPLACE PROCEDURE emp_insert
( p_empno IN emp.empno%TYPE,
  p_ename IN emp.ename%TYPE,
  p_job IN emp.job%TYPE,
  p_mgr IN emp.mgr%TYPE,
  p_hiredate IN emp.hiredate%TYPE,
  p_sal IN emp.sal%TYPE,
  p_comm IN emp.comm%TYPE,
  p_deptno IN emp.deptno%TYPE
) IS
BEGIN
 INSERT INTO emp VALUES
 ( p_empno,
   p_ename,
   p_job,
   p_mgr,
   p_hiredate,
   p_sal,
   p_comm,
   p_deptno);
   DBMS_OUTPUT.PUT_LINE('Added employee...');
   DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
   DBMS_OUTPUT.PUT_LINE('Name : ' || p_ename);
   DBMS_OUTPUT.PUT_LINE('Job : ' || p_job);
   DBMS_OUTPUT.PUT_LINE('Manager : ' || p_mgr);
   DBMS_OUTPUT.PUT_LINE('Hire Date : ' || p_hiredate);
   DBMS_OUTPUT.PUT_LINE('Salary : ' || p_sal);
   DBMS_OUTPUT.PUT_LINE('Commission : ' || p_comm);
END;
```
If an exception occurs, all database changes made in the procedure are rolled back. In this example, the `EXCEPTION` section with the `WHEN OTHERS` clause catches all exceptions. Two variables are displayed. `SQLCODE` is a number that identifies the specific exception that occurred. `SQLERRM` is a text message explaining the error. See Exception handling for more information.

The following shows the output when this procedure is executed:

```sql
EXEC emp_insert(9503,'PETERSON','ANALYST',7902,'31-MAR-05',5000,NULL,40);
```

Added employee...
Employee #: 9503
Name: PETERSON
Job: ANALYST
Manager: 7902
Hire Date: 31-MAR-05
Salary: 5000
Dept #: 40

----------------------
```
SELECT * FROM emp WHERE empno = 9503;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>9503</td>
<td>PETERSON</td>
<td>ANALYST</td>
<td>7902</td>
<td>31-MAR-05 00:00:00</td>
<td>5000.00</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
END;
```

You can include the `INSERT` command in a `FORALL` statement. A `FORALL` statement allows a single `INSERT` command to insert multiple rows from values supplied in one or more collections. See Using the `FORALL` statement for more information.

### 14.1.4.2.4 NULL

The simplest statement is the `NULL` statement. This statement is an executable statement that does nothing.

```sql
NULL;
```

The following is the simplest possible valid SPL program:

```sql
BEGIN
NULL;
END;
```

The `NULL` statement can act as a placeholder where an executable statement is required such as in a branch of an `IF-THEN-ELSE` statement. For example:

```sql
CREATE OR REPLACE PROCEDURE divide_it
(
  p_numerator IN NUMBER,
  p_denominator IN NUMBER,
  p_result OUT NUMBER
) IS
```
14.1.4.2.5 RETURNING INTO

You can append the INSERT, UPDATE, and DELETE commands with the optional RETURNING INTO clause. This clause allows the SPL program to capture the newly added, modified, or deleted values from the results of an INSERT, UPDATE, or DELETE command, respectively.

Syntax

{ <insert> | <update> | <delete> }

RETURNING { * | <expr_1> | [, <expr_2> ] ...

INTO { <record> | <field_1> | [, <field_2> ] ...

...};

- insert is a valid INSERT command.
- update is a valid UPDATE command.
- delete is a valid DELETE command.
- If you specify *, then the values from the row affected by the INSERT, UPDATE, or DELETE command are made available for assignment to the record or fields to the right of the INTO keyword. (The use of * is an EDB Postgres Advanced Server extension and isn’t compatible with Oracle databases.)
- expr_1, expr_2... are expressions evaluated upon the row affected by the INSERT, UPDATE, or DELETE command. The evaluated results are assigned to the record or fields to the right of the INTO keyword.
- record is the identifier of a record that must contain fields that match in number and order and are data-type compatible with the values in the RETURNING clause.
- field_1, field_2,... are variables that must match in number and order and are data-type compatible with the set of values in the RETURNING clause.

If the INSERT, UPDATE, or DELETE command returns a result set with more than one row, then an exception is thrown with SQLCODE 01422, query returned more than one row. If no rows are in the result set, then the variables following the INTO keyword are set to null.

Note

A variation of RETURNING INTO using the BULK COLLECT clause allows a result set of more than one row that's returned into a collection. See Using the BULK COLLECT clause for more information.

Adding the RETURNING INTO clause

This example modifies the emp_comp_update procedure introduced in UPDATE. It adds the RETURNING INTO clause:

```sql
CREATE OR REPLACE PROCEDURE emp_comp_update
(
  p_empno     IN emp.empno%TYPE,
  p_sal       IN emp.sal%TYPE,
  p_comm      IN emp.comm%TYPE
) IS
  v_empno     emp.empno%TYPE;
  v_ename     emp.ename%TYPE;
  v_job       emp.job%TYPE;
  v_sal       emp.sal%TYPE;
  v_deptno    emp.deptno%TYPE;
BEGIN
  UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
  RETURNING
  empno,
  ename,
  job,
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The following is the output from this procedure, assuming employee 9503 created by the emp_insert procedure still exists in the table:

Updated Employee #: 9503
Name: PETERSON
Job: ANALYST
Department: 40
New Salary: 6540.00
New Commission: 1200.00

Adding the RETURNING INTO clause using record types

This example modifies the emp_delete procedure, adding the RETURNING INTO clause using record types:

```sql
CREATE OR REPLACE PROCEDURE emp_delete
(
p_empno  IN emp.empno%TYPE)
IS
  r_emp emp%ROWTYPE;
BEGIN
  DELETE FROM emp WHERE empno = p_empno
  RETURNING *
  INTO r_emp;
  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
    DBMS_OUTPUT.PUT_LINE('Name : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Manager : ' || r_emp.mgr);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || r_emp.hiredate);
  END IF;
END;
END;
```

EXEC emp_comp_update(9503, 6548, 1200);

Updated Employee # : 9503
Name: PETERSON
Job: ANALYST
Department: 40
New Salary: 6548.00
New Commission: 1200.00
The following is the output from this procedure:

```sql
EXEC emp_delete(9503);
```

Deleted Employee #: 9503
Name : PETERSON
Job : ANALYST
Manager : 7902
Hire Date : 31-MAR-05 00:00:00
Salary : 6540.00
Commission : 1200.00
Department : 40

14.1.4.2.6 SELECT INTO

The `SELECT INTO` statement is an SPL variation of the SQL `SELECT` command. The differences are:

- The `SELECT INTO` assigns the results to variables or records where they can then be used in SPL program statements.
- The accessible result set of `SELECT INTO` is at most one row.

Other than these differences, all of the clauses of the `SELECT` command, such as `WHERE`, `ORDER BY`, `GROUP BY`, and `HAVING`, are valid for `SELECT INTO`.

Syntax

These examples show two variations of `SELECT INTO`:

```
SELECT <select_expressions> INTO <target> FROM ...;
```

- `<target>` is a comma-separated list of simple variables. `<select_expressions>` and the remainder of the statement are the same as for the `SELECT` command. The selected values must exactly match in data type, number, and order the structure of the target or a runtime error occurs.

```
SELECT * INTO <record> FROM <table> ...;
```

- `<record>` is a record variable that was previously declared.

If the query returns zero rows, null values are assigned to the targets. If the query returns multiple rows, the first row is assigned to the targets and the rest are discarded. ("The first row" isn’t well-defined unless you used `ORDER BY`.)

Note

- In either case, where no row is returned or more than one row is returned, SPL throws an exception.
- There is a variation of `SELECT INTO` using the `BULK COLLECT` clause that allows a result set of more than one row that’s returned into a collection. See `SELECT BULK COLLECT` for more information.

Including the WHEN NO_DATA_FOUND clause

You can use the `WHEN NO_DATA_FOUND` clause in an `EXCEPTION` block to determine whether the assignment was successful, that is, at least one row was returned by the query.

This version of the `emp_sal_query` procedure uses the variation of `SELECT INTO` that returns the result set into a record. It also uses the `EXCEPTION` block containing the `WHEN NO_DATA_FOUND` conditional expression.

```sql
CREATE OR REPLACE PROCEDURE emp_sal_query
```
If the query is executed with a nonexistent employee number, the results appear as follows:

Including a TOO_MANY_ROWS exception

Another conditional clause useful in the EXCEPTION section with SELECT INTO is the TOO_MANY_ROWS exception. If more than one row is selected by the SELECT INTO statement, SPL throws an exception.

When the following block is executed, the TOO_MANY_ROWS exception is thrown since there are many employees in the specified department:
You can use the **UPDATE** command available in the SQL language in SPL programs.

You can use an expression in the SPL language wherever an expression is allowed in the SQL **UPDATE** command. Thus, you can use SPL variables and parameters to supply values to the update operation.

The **SQL%FOUND** conditional expression returns **TRUE** if a row is updated, **FALSE** otherwise. See **Obtaining the result status** for a discussion of **SQL%FOUND** and other similar expressions.

This example shows the update on the employee:

```
CREATE OR REPLACE PROCEDURE emp_comp_update
(
    p_empno    IN emp.empno%TYPE,
    p_sal      IN emp.sal%TYPE,
    p_comm     IN emp.comm%TYPE
)
IS
    BEGIN
        UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno;
        IF SQL%FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || p_empno);
            DBMS_OUTPUT.PUT_LINE('New Salary         : ' || p_sal);
            DBMS_OUTPUT.PUT_LINE('New Commission     : ' || p_comm);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found!');
        END IF;
    END;
END;
```

The **SQL%FOUND** conditional expression returns **TRUE** if at least one row was affected by an **INSERT**, **UPDATE** or **DELETE** command or a **SELECT INTO** command retrieved one or more rows.

This anonymous block inserts a row and then displays the fact that the row was inserted:

```
BEGIN
    INSERT INTO emp (empno,ename,job,sal,deptno) VALUES
    (9981, 'JONES', 'CLERK', 850.00, 40);
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Row has been inserted!');
    END IF;
END;
```

You can include the **UPDATE** command in a **FORALL** statement. A **FORALL** statement allows a single **UPDATE** command to update multiple rows from values supplied in one or more collections. See **Using the FORALL statement** for more information.

### 14.1.4.2.8 Obtaining the result status

You can use several attributes to determine the effect of a command. **SQL%FOUND** is a Boolean that returns **TRUE** if at least one row was affected by an **INSERT**, **UPDATE** or **DELETE** command or a **SELECT INTO** command retrieved one or more rows.

This anonymous block inserts a row and then displays the fact that the row was inserted:
SQL%ROWCOUNT provides the number of rows affected by an INSERT, UPDATE, DELETE, or SELECT INTO command. The SQL%ROWCOUNT value is returned as a BIGINT data type. The following example updates the row that was just inserted and displays SQL%ROWCOUNT:

```
BEGIN
  UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9001;
  DBMS_OUTPUT.PUT_LINE('Rows updated: ' || SQL%ROWCOUNT);
END;
```

SQL%NOTFOUND is the opposite of SQL%FOUND. SQL%NOTFOUND returns TRUE if no rows were affected by an INSERT, UPDATE, or DELETE command or a SELECT INTO command retrieved no rows.

```
BEGIN
  UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9008;
  IF SQL%NOTFOUND THEN
    DBMS_OUTPUT.PUT_LINE('No rows were updated');
  END IF;
END;
```

No rows were updated

### 14.1.4.3 Types of control structures

SPL includes programming statements that make it a full procedural complement to SQL.

#### 14.1.4.3.1 IF statement

IF statements let you execute commands based on certain conditions. SPL has four forms of IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE

#### 14.1.4.3.1.1 IF-THEN

**Syntax**

```
IF boolean-expression THEN
  <statements>
END IF;
```

IF-THEN statements are the simplest form of IF. The statements between THEN and END IF are executed if the condition is TRUE. Otherwise, they are skipped.

**Example**

This example uses IF-THEN statement to test and display employees who have a commission:

```
DECLARE
  v_empno empno%TYPE;
```
14.1.4.3.1.2 IF-THEN-ELSE

Syntax

IF boolean-expression THEN
  <statements>
ELSE
  <statements>
END IF;

IF-THEN-ELSE statements add to IF-THEN by letting you specify an alternative set of statements to execute if the condition evaluates to false.

Example

This example shows an IF-THEN-ELSE statement being used to display the text Non-commission if an employee doesn't get a commission:

 DECLARE
  v_empno       emp.empno%TYPE;
  v_comm       emp.comm%TYPE;
 CURSOR emp_cursor IS SELECT empno, comm FROM emp;
 BEGIN
  OPEN emp_cursor;
  DBMS_OUTPUT.PUT_LINE('EMPNO
Comm');
  DBMS_OUTPUT.PUT_LINE('-----    -------
');
  LOOP
    FETCH emp_cursor INTO v_empno, v_comm;
    EXIT WHEN emp_cursor%NOTFOUND;
    -- Test whether or not the employee gets a commission
    --
    IF v_comm IS NOT NULL AND v_comm > 0 THEN
      DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || TO_CHAR(v_comm,'$99999.99'));
    END IF;
  END LOOP;
  CLOSE emp_cursor;
 END;

The following is the output from this program:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>7499</td>
<td>$300.00</td>
</tr>
<tr>
<td>7521</td>
<td>$500.00</td>
</tr>
<tr>
<td>7654</td>
<td>$1400.00</td>
</tr>
</tbody>
</table>
The following is the output from this program:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7499</td>
<td>$300.00</td>
</tr>
<tr>
<td>7521</td>
<td>$500.00</td>
</tr>
<tr>
<td>7566</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7654</td>
<td>$1400.00</td>
</tr>
<tr>
<td>7698</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7782</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7788</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7839</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7844</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7876</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7900</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7902</td>
<td>Non-commission</td>
</tr>
<tr>
<td>7934</td>
<td>Non-commission</td>
</tr>
</tbody>
</table>

14.1.4.3.1.3 IF-THEN-ELSE IF

You can nest \texttt{IF} statements. This allows you to invoke alternative \texttt{IF} statements once it's determined whether the conditional of an outer \texttt{IF} statement is \texttt{TRUE} or \texttt{FALSE}.

In this example, the outer \texttt{IF-THEN-ELSE} statement tests whether an employee has a commission. The inner \texttt{IF-THEN-ELSE} statements then test whether the employee's total compensation exceeds or is less than the company average.

Note

The logic in this program can be simplified by calculating the employee's yearly compensation using the \texttt{NVL} function in the \texttt{SELECT} command of the cursor declaration. However, the purpose of this example is to show the use of \texttt{IF} statements.

DECLARE
\begin{verbatim}
  v_empno  emp.empno%TYPE;
  v_sal    emp.sal%TYPE;
  v_comm   emp.comm%TYPE;
  v_avg    NUMBER(7,2);
  CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
  -- Calculate the average yearly compensation in the company
  -- SELECT AVG((sal + NVL(comm,0)) * 24) INTO v_avg FROM emp;
  DBMS_OUTPUT.PUT_LINE('Average Yearly Compensation: ' ||
                        TO_CHAR(v_avg,'$999,999.99'));
\end{verbatim}
OPEN emp_cursor;
DBMS_OUTPUT.PUT_LINE('EMPNO    YEARLY COMP');
DBMS_OUTPUT.PUT_LINE('-----    -----------');
LOOP
  FETCH emp_cursor INTO v_empno, v_sal, v_comm;
  EXIT WHEN emp_cursor%NOTFOUND;
  -- Test whether or not the employee gets a commission
  -- IF v_comm IS NOT NULL AND v_comm > 0 THEN
  -- Test if the employee's compensation with commission exceeds the average
  -- IF (v_sal + v_comm) * 24 > v_avg THEN
    DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                         TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') || ' Exceeds Average');
  ELSE
    DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                         TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') || ' Below Average');
  END IF;
  ELSE
  -- Test if the employee's compensation without commission exceeds the average
  -- IF v_sal * 24 > v_avg THEN
    DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                         TO_CHAR(v_sal * 24,'$999,999.99') || ' Exceeds Average');
  ELSE
    DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                         TO_CHAR(v_sal * 24,'$999,999.99') || ' Below Average');
  END IF;
END LOOP;
CLOSE emp_cursor;
END;

The following is the output from this program:

__OUTPUT__
Average Yearly Compensation: $53,528.57
EMPNO    YEARLY COMP
-----    ----------
7369    $ 19,208.00 Below Average
7499    $ 45,600.00 Below Average
7521    $ 42,808.00 Below Average
7566    $ 71,400.00 Exceeds Average
7545    $ 63,600.00 Exceeds Average
7698    $ 68,400.00 Exceeds Average
7782    $ 58,808.00 Exceeds Average
7788    $ 72,808.00 Exceeds Average
7839    $128,808.00 Exceeds Average
7844    $ 36,800.00 Below Average
7876    $ 26,400.00 Below Average
7900    $ 22,808.00 Below Average
7982    $ 72,808.00 Exceeds Average
7934    $ 31,208.00 Below Average

When you use this form, you're actually nesting an IF statement inside the ELSE part of an outer IF statement. Thus you need one END IF statement for each nested IF and one for the parent IF-ELSE.

14.1.4.3.14 IF-THEN-ELSIF-ELSE
Syntax

\[
\text{IF boolean-expression THEN}
\]
\[
\quad <\text{statements}>
\]
\[
\text{[ ELSIF boolean-expression THEN}
\]
\[
\quad <\text{statements}>
\]
\[
\text{[ ELSIF boolean-expression THEN}
\]
\[
\quad <\text{statements}>
\]
\[
\ldots]
\]
\[
\text{ELSE}
\]
\[
\quad <\text{statements}>
\]
\]
\[
\text{ENDIF;}
\]

**IF-THEN-ELSIF-ELSE** provides a method of checking many alternatives in one statement. Formally it is equivalent to nested **IF-THEN-ELSE-IF-THEN** commands, but only one **END IF** is needed.

Example

The following example uses an **IF-THEN-ELSIF-ELSE** statement to count the number of employees by compensation ranges of $25,000:

```sql
DECLARE
  v_empno    emp.empno%TYPE;
  v_comp    NUMBER(8,2);
  v_lt_25K    SMALLINT := 0;
  v_25K_50K    SMALLINT := 0;
  v_50K_75K    SMALLINT := 0;
  v_75K_100K    SMALLINT := 0;
END;

CURSOR emp_cursor IS
  SELECT empno, (sal + NVL(comm,0)) * 24 FROM emp;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_empno, v_comp;
    EXIT WHEN emp_cursor%NOTFOUND;
    IF v_comp < 25000
      THEN
        v_lt_25K := v_lt_25K + 1;
      END IF;
    ELSIF v_comp < 50000
      THEN
        v_25K_50K := v_25K_50K + 1;
      END IF;
    ELSIF v_comp < 75000
      THEN
        v_50K_75K := v_50K_75K + 1;
      END IF;
    ELSIF v_comp < 100000
      THEN
        v_75K_100K := v_75K_100K + 1;
      END IF;
    ELSE
      v_ge_100K := v_ge_100K + 1;
    END IF;
    END LOOP;
  CLOSE emp_cursor;
  DBMS_OUTPUT.PUT_LINE('Number of employees by yearly compensation');
  DBMS_OUTPUT.PUT_LINE('Less than 25,000 : ' || v_lt_25K);
  DBMS_OUTPUT.PUT_LINE('25,000 - 49,999 : ' || v_25K_50K);
  DBMS_OUTPUT.PUT_LINE('50,000 - 74,999 : ' || v_50K_75K);
  DBMS_OUTPUT.PUT_LINE('75,000 - 99,999 : ' || v_75K_100K);
  DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;
```

The following is the output from this program:

```
Number of employees by yearly compensation
Less than 25,000 : 2
```
14.1.4.3.2 RETURN statement

The RETURN statement terminates the current function, procedure, or anonymous block and returns control to the caller.

Syntax

The RETURN statement has two forms. The first form of the RETURN statement terminates a procedure or function that returns void. The syntax of this form is:

```
RETURN;
```

The second form of RETURN returns a value to the caller. The syntax of this form is:

```
RETURN <expression>;
```

`expression` must evaluate to the same data type as the return type of the function.

Example

This example uses the RETURN statement and returns a value to the caller:

```sql
CREATE OR REPLACE FUNCTION emp_comp
(
    p_sal NUMBER,
    p_comm NUMBER
) RETURN NUMBER
IS
    BEGIN
        RETURN (p_sal + NVL(p_comm, 0)) * 24;
    END emp_comp;
```

14.1.4.3.3 GOTO statement

The GOTO statement causes the point of execution to jump to the statement with the specified label.

Syntax

The syntax of a GOTO statement is:

```
GOTO <label>
```

`label` is a name assigned to an executable statement. `label` must be unique in the scope of the function, procedure, or anonymous block.

To label a statement, use this syntax:

```
<<label>> <statement>
```

`statement` is the point of execution that the program jumps to.
Statements eligible for labeling

You can label assignment statements, any SQL statement (like `INSERT`, `UPDATE`, and `CREATE`), and selected procedural language statements. The procedural language statements that can be labeled are:

- `IF`
- `EXIT`
- `RETURN`
- `RAISE`
- `EXECUTE`
- `PERFORM`
- `GET DIAGNOSTICS`
- `OPEN`
- `FETCH`
- `CLOSE`
- `NULL`
- `COMMIT`
- `ROLLBACK`
- `GOTO`
- `CASE`
- `LOOP`
- `WHILE`
- `FOR`

`EXIT` is considered a keyword and you can’t use it as the name of a label.

Restrictions

`GOTO` statements can’t transfer control into a conditional block or sub-block. However, they can transfer control from a conditional block or sub-block.

`GOTO` statements have the following restrictions:

- A `GOTO` statement can’t jump to a declaration.
- A `GOTO` statement can’t transfer control to another function, or procedure.
- Don’t place a `label` at the end of a block, function, or procedure.

Example

This example verifies that an employee record contains a name, job description, and employee hire date. If any piece of information is missing, a `GOTO` statement transfers the point of execution to a statement that prints a message that the employee isn’t valid.

```sql
CREATE OR REPLACE PROCEDURE verify_emp
    (p_empno NUMBER)
IS
    v_ename emp.ename%TYPE;
    v_job emp.job%TYPE;
    v_hiredate emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, hiredate
    INTO v_ename, v_job, v_hiredate FROM emp
    WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        GOTO invalid_emp;
        END IF;
    IF v_job IS NULL THEN
        GOTO invalid_emp;
    END IF;
    IF v_hiredate IS NULL THEN
        GOTO invalid_emp;
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' validated without errors.');
RETURN;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
14.1.4.3.4  CASE expression

The CASE expression returns a value that's substituted where the CASE expression is located in an expression.

The two formats of the CASE expression are called a searched CASE and one that uses a selector.

14.1.4.3.4.1  Selector CASE expression

The selector CASE expression attempts to match an expression, called the selector, to the expression specified in one or more WHEN clauses. result is an expression that is type-compatible in the context where the CASE expression is used. If a match is found, the value given in the corresponding THEN clause is returned by the CASE expression. If there are no matches, the value following ELSE is returned. If ELSE is omitted, the CASE expression returns null.

Syntax

```sql
CASE <selector-expression>
WHEN <match-expression> THEN
  <result>
  [ WHEN <match-expression> THEN
    <result>
    [ WHEN <match-expression> THEN
      <result> ]
    ...]
  [ELSE]
  <result>
END
```

- match-expression is evaluated in the order in which it appears in the CASE expression.
- result is an expression that is type-compatible in the context where the CASE expression is used.
- When the first match-expression is encountered that equals selector-expression, result in the corresponding THEN clause is returned as the value of the CASE expression.
- If none of match-expression equals selector-expression, then result following ELSE is returned.
- If no ELSE is specified, the CASE expression returns null.

Example

This example uses a selector CASE expression to assign the department name to a variable based on the department number:

```sql
DECLARE
  v_empno         emp.empno%TYPE;
  v_ename         emp.ename%TYPE;
  v_deptno        emp.deptno%TYPE;
  v_dname         dept.dname%TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
  OPEN emp_cursor;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------');
  LOOP
    FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
    EXIT WHEN emp_cursor%NOTFOUND;
    v_dname := CASE
      WHEN 10 THEN 'Accounting'
      WHEN 20 THEN 'Research'
      ELSE 'Invalid'
    END;
    DBMS_OUTPUT.PUT_LINE(v_empno || ' is not a valid employee.
END;
```
The following is the output from this program:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>10</td>
<td>Accounting</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>10</td>
<td>Accounting</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>20</td>
<td>Sales</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>10</td>
<td>Accounting</td>
</tr>
</tbody>
</table>

14.1.4.3.4.2 Searched CASE expression

A searched CASE expression uses one or more Boolean expressions to determine the resulting value to return.

Syntax

```
CASE WHEN <boolean-expression> THEN <result>
[ WHEN <boolean-expression> THEN <result> ]
[ WHEN <boolean-expression> THEN <result> ]
...
[ ELSE <result> ]
END;
```

- **boolean-expression** is evaluated in the order in which it appears in the CASE expression.
- **result** is an expression that is type-compatible in the context where the CASE expression is used.
- When the first **boolean-expression** is encountered that evaluates to TRUE, **result** in the corresponding THEN clause is returned as the value of the CASE expression.
- If none of **boolean-expression** evaluates to true, then **result** following ELSE is returned.
- If no ELSE is specified, the CASE expression returns null.

Example

This example uses a searched CASE expression to assign the department name to a variable based on the department number:
The following is the output from this program:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>Smith</td>
<td>20</td>
<td>Research</td>
</tr>
<tr>
<td>7499</td>
<td>Allen</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7521</td>
<td>Ward</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7566</td>
<td>Jones</td>
<td>20</td>
<td>Sales</td>
</tr>
<tr>
<td>7654</td>
<td>Martin</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7698</td>
<td>Blake</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7782</td>
<td>Clark</td>
<td>10</td>
<td>Accounting</td>
</tr>
<tr>
<td>7788</td>
<td>Scott</td>
<td>20</td>
<td>Sales</td>
</tr>
<tr>
<td>7839</td>
<td>King</td>
<td>10</td>
<td>Accounting</td>
</tr>
<tr>
<td>7844</td>
<td>Turner</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7876</td>
<td>Adams</td>
<td>20</td>
<td>Sales</td>
</tr>
<tr>
<td>7900</td>
<td>James</td>
<td>30</td>
<td>Sales</td>
</tr>
<tr>
<td>7982</td>
<td>Ford</td>
<td>20</td>
<td>Sales</td>
</tr>
<tr>
<td>7934</td>
<td>Miller</td>
<td>10</td>
<td>Accounting</td>
</tr>
</tbody>
</table>

14.1.4.3.5 CASE statement

The CASE statement executes a set of one or more statements when a specified search condition is TRUE. The CASE statement is a standalone statement while the CASE expression must appear as part of an expression.

The two formats of the CASE statement are a searched CASE and one that uses a selector.

14.1.4.3.5.1 Selector CASE statement
The selector `CASE` statement attempts to match an expression called the selector to the expression specified in one or more `WHEN` clauses. When a match is found, one or more corresponding statements are executed.

**Syntax**

```
CASE <selector-expression>
   WHEN <match-expression> THEN
      <statements>
   [ WHEN <match-expression> THEN
      <statements> ]
   [...]
   [ ELSE
      <statements> ]
END CASE;
```

- `<selector-expression>` returns a value type-compatible with each `<match-expression>`.
- `<match-expression>` is evaluated in the order in which it appears in the `CASE` statement.
- `<statements>` are one or more SPL statements, each terminated by a semi-colon.
- When the value of `<selector-expression>` equals the first `<match-expression>`, the statements in the corresponding `THEN` clause are executed, and control continues following the `END CASE` keywords.
- If there are no matches, the statements following `ELSE` are executed.
- If there are no matches and there is no `ELSE` clause, an exception is thrown.

**Example**

This example uses a selector `CASE` statement to assign a department name and location to a variable based on the department number:

```sql
DECLARE
  v_empno     emp.empno%TYPE;
  v_ename     emp.ename%TYPE;
  v_deptno    emp.deptno%TYPE;
  v_dname     dept.dname%TYPE;
  v_loc       dept.loc%TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
  OPEN emp_cursor;
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME
                      ----    -------   ------    ----------
                      1        2        3        4');
  LOOP
    FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
    EXIT WHEN emp_cursor%NOTFOUND;
    CASE v_deptno
      WHEN 1 THEN v_dname := 'Accounting';
      WHEN 2 THEN v_dname := 'Research';
      WHEN 3 THEN v_dname := 'Sales';
      WHEN 4 THEN v_dname := 'Operations';
      ELSE v_dname := 'unknown';
    END CASE;
    DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || RPAD(v_ename, 10) || ' ' || v_deptno || ' ' || RPAD(v_dname, 14) || ' ' || v_loc);
  END LOOP;
  CLOSE emp_cursor;
END;
```
The following is the output from this program:

```
<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>DEPTNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7722</td>
<td>CLARK</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7982</td>
<td>FORD</td>
<td>20</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
<tr>
<td>7902</td>
<td>BLAKE</td>
<td>30</td>
<td>Sales</td>
<td>Chicago</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>10</td>
<td>Accounting</td>
<td>New York</td>
</tr>
</tbody>
</table>
```

### 14.1.4.3.5.2 Searched CASE statement

A searched **CASE** statement uses one or more Boolean expressions to determine the resulting set of statements to execute.

#### Syntax

```
CASE WHEN <boolean-expression> THEN
  <statements>
[ WHEN <boolean-expression> THEN
  <statements> ]
[ WHEN <boolean-expression> THEN
  <statements> ]
  ...
[ ELSE <statements> ]
END CASE;
```

- **boolean-expression** is evaluated in the order in which it appears in the **CASE** statement.
- When the first **boolean-expression** is encountered that evaluates to **TRUE**, the statements in the corresponding **THEN** clause are executed and control continues following the **END CASE** keywords.
- If none of **boolean-expression** evaluates to **TRUE**, the statements following **ELSE** are executed.
- If none of **boolean-expression** evaluates to **TRUE** and there is no **ELSE** clause, an exception is thrown.

#### Example

This example uses a searched **CASE** statement to assign a department name and location to a variable based on the department number:

```sql
DECLARE
  v_empno     emp.empno%TYPE;
  v_ename     emp.ename%TYPE;
  v_deptno    emp.deptno%TYPE;
  v_dname     dept.dname%TYPE;
  v_loc       dept.loc%TYPE;
CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
  OPEN emp_cursor;
```
14.1.4.3.6 Loops

When you use the LOOP, EXIT, CONTINUE, WHILE, and FOR statements, your SPL program can repeat a series of commands.

14.1.4.3.6.1 LOOP

LOOP

$stmtements$

END LOOP;
**14.1.4.3.6.2**  **EXIT**

**Syntax**

```plaintext
EXIT [WHEN <expression>]
```

The innermost loop is terminated, and the statement following `END LOOP` is executed next.

If `WHEN` is present, loop exit occurs only if the specified condition is `TRUE`. Otherwise control passes to the statement after `EXIT`.

You can use `EXIT` to cause early exit from all types of loops, not just unconditional loops.

**Example**

This example shows a loop that iterates 10 times and then uses the `EXIT` statement to terminate:

```sql
DECLARE
  v_counter NUMBER(2);
BEGIN
  v_counter := 1;
  LOOP
    EXIT WHEN v_counter > 10;
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
    v_counter := v_counter + 1;
  END LOOP;
END;
```

The following is the output from this example:

```
__OUTPUT__
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

**14.1.4.3.6.3**  **CONTINUE**

The `CONTINUE` statement provides a way to proceed with the next iteration of a loop while skipping intervening statements.

When the `CONTINUE` statement is encountered, the next iteration of the innermost loop begins, skipping all statements following the `CONTINUE` statement until the end of the loop. That is, control is passed back to the loop control expression, if any, and the body of the loop is reevaluated.

If you use the `WHEN` clause, then the next iteration of the loop begins only if the specified expression in the `WHEN` clause evaluates to `TRUE`. Otherwise, control is passed to the statement following the `CONTINUE` statement.

You can use the `CONTINUE` statement only inside a loop.
This example uses the `CONTINUE` statement to skip the display of the odd numbers:

```sql
DECLARE
  v_counter NUMBER(2);
BEGIN
  v_counter := 0;
  LOOP
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
    CONTINUE WHEN MOD(v_counter, 2) = 1;
    DBMS_OUTPUT.PUT_LINE('Iteration #' || v_counter);
  END LOOP;
END;
```

The following is the output from this example:

```
Iteration # 2
Iteration # 4
Iteration # 6
Iteration # 8
Iteration # 10
```

### 14.1.4.3.6.4 WHILE

#### Syntax

```sql
WHILE <expression> LOOP
  <statements>
END LOOP;
```

The `WHILE` statement repeats a sequence of statements when the condition expression evaluates to `TRUE`. The condition is checked just before each entry to the loop body.

#### Example

This example uses the `WHILE` statement instead of the `EXIT` statement to determine when to exit the loop:

```sql
DECLARE
  v_counter NUMBER(2);
BEGIN
  v_counter := 1;
  WHILE v_counter <= 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration #' || v_counter);
    v_counter := v_counter + 1;
  END LOOP;
END;
```

The following is the output from this example:

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
```

Note

The conditional expression used to determine when to exit the loop differs when using `WHILE` versus `EXIT`. The `EXIT` statement terminates the loop when its conditional expression is true. The `WHILE` statement terminates the loop or never begins it when its conditional expression is false.
14.1.4.3.6.5 FOR (integer variant)

Syntax

```
FOR <name> [REVERSE] IN <expression> .. <expression>
LOOP
  <statements>
END LOOP;
```

This form of `FOR` creates a loop that iterates over a range of integer values. The variable `<name>` is of type `INTEGER` and exists only inside the loop. The two expressions giving the loop range are evaluated once when entering the loop. The iteration step is +1.

- `<name>` begins with the value of `<expression>` to the left of `..` and terminates when `<name>` exceeds the value of `<expression>` to the right of `..`. Thus the two expressions take on the roles `start-value .. end-value`.
- The optional `REVERSE` clause specifies for the loop to iterate in reverse order. The first time through the loop, `<name>` is set to the value of the right-most `<expression>`. The loop terminates when the `<name>` is less than the left-most `<expression>`.

Example

This example uses a `FOR` loop that iterates from 1 to 10:

```
BEGIN
  FOR i IN 1 .. 10
  LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

The following is the output after using the `FOR` statement:

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

If the start value is greater than the end value, the loop body doesn't execute. No error occurs, as shown by the following example:

```
BEGIN
  FOR i IN 10 .. 1
  LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```
This example has no output as the loop body never executes.

Note

SPL also supports cursor FOR loops. See Cursor FOR loop.

14.1.4.3.7 Exception handling

By default, any error occurring in an SPL program stops the program from executing. You can trap errors and recover from them by using a BEGIN block with an EXCEPTION section.

Syntax

The syntax is an extension of the normal syntax for a BEGIN block:

```
<declarations>

BEGIN
<statements>
EXCEPTION
WHEN <condition> [ OR <condition> ]...
THEN <handler_statements>
[ WHEN <condition> [ OR <condition> ]...
THEN <handler_statements> ]...
END;
```

Error handling process

- If no error occurs, this form of block executes all the statements, and then control passes to the next statement after END.
- If an error occurs in the statements, further processing of the statements is abandoned, and control passes to the EXCEPTION list. The list is searched for the first condition matching the error.
- If a match is found, the corresponding handler_statements are executed, and then control passes to the next statement after END.
- If no match is found, the error propagates out as though the EXCEPTION clause wasn’t there. You can catch the error by an enclosing block with EXCEPTION. If there is no enclosing block, it aborts the subprogram.

The special condition named OTHERS matches every error type. Condition names aren’t case sensitive.

If a new error occurs in the selected handler_statements, this EXCEPTION clause can’t catch it, but it is propagated out. A surrounding EXCEPTION clause might catch it.

Performance implications

Heavy use of the EXCEPTION clause can have visible performance consequences. An EXCEPTION clause establishes a sub-transaction internally in the database server. If the code in the EXCEPTION clause completes without an error, the implicit sub-transaction commits. If an error occurs, it rolls back.

This process has some overhead. If the code protected by the EXCEPTION clause modifies the database, the transaction acquires another transaction ID (XID), increasing the overhead considerably. XID consumption is one factor determining how frequently each table in the database must be subject to VACUUM, so a very high rate of XID consumption leads to more VACUUM activity. Also, when any individual session has more than 64 XIDs assigned simultaneously, some tuple visibility checks incur more overhead.

List of condition names to use

<table>
<thead>
<tr>
<th>Condition name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE_NOT_FOUND</td>
<td>The application encountered a situation where none of the cases in CASE statement evaluates to TRUE and there is no ELSE condition.</td>
</tr>
<tr>
<td>COLLECTION_IS_NULL</td>
<td>The application attempted to invoke a collection method on a null collection, such as an uninitialized nested table.</td>
</tr>
<tr>
<td>CURSOR_ALREADY_OPEN</td>
<td>The application attempted to open a cursor that’s already open.</td>
</tr>
</tbody>
</table>
Condition name | Description |
--- | --- |
DUP_VAL_ON_INDEX | The application attempted to store a duplicate value that currently exists in a constrained column. |
INVALID_CURSOR | The application attempted to access an unopened cursor. |
INVALID_NUMBER | The application encountered a data exception equivalent to SQLSTATE class code 22. INVALID_NUMBER is an alias for VALUE_ERROR. |
NO_DATA_FOUND | No rows satisfy the selection criteria. |
OTHERS | The application encountered an exception that wasn’t caught by a prior condition in the exception section. |
SUBSCRIPT_BEYOND_COUNT | The application attempted to reference a subscript of a nested table or varray beyond its initialized or extended size. |
SUBSCRIPT_OUTSIDE_LIMIT | The application attempted to reference a subscript or extend a varray beyond its maximum size limit. |
TOO_MANY_ROWS | The application encountered more than one row that satisfies the selection criteria where only one row is allowed to be returned. |
VALUE_ERROR | The application encountered a data exception equivalent to SQLSTATE class code 22. VALUE_ERROR is an alias for INVALID_NUMBER. |
ZERO_DIVIDE | The application tried to divide by zero. |
User-defined Exception | See User-defined exceptions. |

Note

Condition names INVALID_NUMBER and VALUE_ERROR aren't compatible with Oracle databases for which these condition names are used for exceptions that result only from a failed conversion of a string to a numeric literal. In addition, for Oracle databases, an INVALID_NUMBER exception applies only to SQL statements, while a VALUE_ERROR exception applies only to procedural statements.

14.1.4.3 User-defined exceptions

Any number of errors (referred to in PL/SQL as exceptions) can occur while a program executes. When an exception is thrown, normal execution of the program stops, and control of the program transfers to the error-handling portion of the program. An exception can be a predefined error that’s generated by the server, or it can be a logical error that raises a user-defined exception.

The server never raises user-defined exceptions. They are raised explicitly by a RAISE statement. A user-defined exception is raised when a developer-defined logical rule is broken. A common example of a logical rule being broken occurs when a check is presented against an account with insufficient funds. An attempt to cash a check against an account with insufficient funds causes a user-defined exception.

You can define exceptions in functions, procedures, packages, or anonymous blocks. While you can't declare the same exception twice in the same block, you can declare the same exception in two different blocks.

Syntax

Before implementing a user-defined exception, you must declare the exception in the declaration section of a function, procedure, package, or anonymous block. You can then raise the exception using the RAISE statement:

```sql
DECLARE
  <exception_name> EXCEPTION;
BEGIN
  ...
  RAISE <exception_name>;
  ...
END;
```

Unhandled exceptions propagate back through the call stack. If the exception remains unhandled, the exception is eventually reported to the client application.

User-defined exceptions declared in a block are considered to be local to that block and global to any blocks nested in the block. To reference an exception that resides in an outer block, you must assign a label to the outer block. Then, preface the name of the exception with block_name.exception_name.

Outer blocks can't reference exceptions declared in nested blocks.

The scope of a declaration is limited to the block in which it’s declared unless it’s created in a package and, when referenced, qualified by the package name. For example, to raise an exception named out_of_stock that resides in a package named inventory_control, a program must raise an error named inventory_control.out_of_stock.

Example

This example declares a user-defined exception in a package. The user-defined exception doesn't require a package qualifier when raised in check_balance, since it resides in the same package as the exception.
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER)
  IS
    BEGIN
      IF (p_amount > p_balance) THEN
        RAISE overdrawn;
      END IF;
  END;
END;

Here, the procedure purchase calls the check_balance procedure. If p_amount is greater than p_balance, check_balance raises an exception. purchase catches the ar.overdrawn exception. purchase must refer to the exception with a package-qualified name (ar.overdrawn) because purchase isn't defined in the ar package.

CREATE или REPLACE СОБЫТИЕ ar AS
  overdrawn EXCEPTION;
END

CREATE или REPLACE СОБЫТИЕ BODY ar AS
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER)
  IS
    BEGIN
      IF (p_amount > p_balance) THEN
        RAISE overdrawn;
      END IF;
  END;
END;

CREATE PROCEDURE purchase(customerID INT, amount NUMERIC) AS
BEGIN
  ar.check_balance(getcustomerbalance(customerid), amount);
  record_purchase(customerid, amount);
  EXCEPTION
  WHEN ar.overdrawn THEN
    raise_credit_limit(customerid, amount*1.5);
END;

EXCEPTION
  WHEN ar.overdrawn THEN
    raise_credit_limit(customerid, amount*1.5);

When ar.check_balance raises an exception, execution jumps to the exception handler defined in purchase.

The exception handler raises the customer's credit limit and ends. When the exception handler ends, execution resumes with the statement that follows ar.check_balance.

### 14.1.4.3.9 PRAGMA EXCEPTION_INIT

 задает пользовательский код ошибки для исключения. Вы можете включить PRAGMA EXCEPTION_INIT в любом блоке, под-блоке, или пакете. Вы можете назначить пользовательский код ошибки исключению, используя PRAGMA EXCEPTION_INIT после объявления исключения.

**Синтаксис**

Формат PRAGMA EXCEPTION_INIT:

```
PRAGMA EXCEPTION_INIT(exception_name, INTEGER exception_number);
```

где:

- `exception_name` - имя связанного исключения.
- `exception_number` - пользовательский код ошибки, связанной с прологом. Если вы указываете несоответствующий `exception_number`, сервер возвращает предупреждение.
- `exception_code` - имя предопределенного исключения. Для полного списка возможных исключений, см. документацию Postgres core.

**Использование PRAGMA EXCEPTION_INIT**

Этот пример использует PRAGMA EXCEPTION_INIT:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
END;
```
The following procedure calls the `check_balance` procedure. If `p_amount` is greater than `p_balance`, `check_balance` raises an exception. The `purchase` procedure catches the `ar.overdrawn` exception.

```sql
CREATE OR REPLACE PACKAGE BODY ar AS
  PROCEDURE check_balance(p_balance NUMBER, p_amount NUMBER)
  IS
    BEGIN
      IF (p_amount > p_balance) THEN
        RAISE overdrawn;
      END IF;
    END;
END;
END;
```

The following procedure calls the `check_balance` procedure. If `p_amount` is greater than `p_balance`, `check_balance` raises an exception. The `purchase` procedure catches the `ar.overdrawn` exception.

```sql
CREATE PROCEDURE purchase(customerID int, amount NUMERIC) AS
  BEGIN
    ar.check_balance(getcustomerbalance(customerid), amount);
    record_purchase(customerid, amount);
  EXCEPTION
    WHEN ar.overdrawn THEN
      DBMS_OUTPUT.PUT_LINE ('This account is overdrawn.');
      DBMS_OUTPUT.PUT_LINE ('SQLCode : ' || SQLCODE || ' ' || SQLERRM);
  END;
END;
```

When `ar.check_balance` raises an exception, execution jumps to the exception handler defined in `purchase`.

```sql
EXCEPTION
  WHEN ar.overdrawn THEN
    DBMS_OUTPUT.PUT_LINE ('This account is overdrawn.');
    DBMS_OUTPUT.PUT_LINE ('SQLCode : ' || SQLCODE || ' ' || SQLERRM);
END;
```

The exception handler returns an error message, followed by `SQL_CODE` information:

```
This account is overdrawn.
SQLCODE: -20100 User-Defined Exception
```

Using a predefined exception

This example uses a predefined exception. The code creates a more meaningful name for the `no_data_found` exception. If the given customer doesn’t exist, the code catches the exception, calls `DBMS_OUTPUT.PUT_LINE` to report the error, and then raises the original exception again:

```sql
CREATE OR REPLACE PACKAGE ar AS
  unknown_customer
  EXCEPTION;
  PRAGMA EXCEPTION_INIT (unknown_customer, -20180);
  PROCEDURE check_balance(p_customer_id NUMBER);
END;
```

```sql
CREATE OR REPLACE PACKAGE BODY ar AS
  PROCEDURE check_balance(p_customer_id NUMBER) IS
    DECLARE
      v_balance NUMBER;
    BEGIN
      SELECT balance INTO v_balance FROM customer
      WHERE cust_id = p_customer_id;
      EXCEPTION WHEN unknown_customer THEN
        DBMS_OUTPUT.PUT_LINE('invalid customer id');
      RAISE;
    END;
END;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
14.1.4.3.10 RAISE_APPLICATION_ERROR

The procedure RAISE_APPLICATION_ERROR allows you to abort processing in an SPL program by causing an exception. The exception is handled in the same manner as described in Exception handling. In addition, the RAISE_APPLICATION_ERROR procedure makes a user-defined code and error message available to the program, which you can then use to identify the exception.

Syntax

RAISE_APPLICATION_ERROR(<error_number>, <message>);

Where:

error_number is an integer value or expression returned in a variable named SQLCODE when the procedure is executed. The value is between -20000 and -20999.

message is a string literal or expression returned in a variable named SQLERRM.

For more information on the SQLCODE and SQLERRM variables, see Errors and messages.

Example

This example uses the RAISE_APPLICATION_ERROR procedure to display a different code and message depending on the information missing from an employee:

```sql
CREATE OR REPLACE PROCEDURE verify_emp
  ( p_empno NUMBER )
IS
  v_ename emp.ename%TYPE;
  v_job emp.job%TYPE;
  v_mngr emp.mgr%TYPE;
  v_hiredate emp.hiredate%TYPE;
BEGIN
  SELECT ename, job, mgr, hiredate INTO v_ename, v_job, v_mngr, v_hiredate
  FROM emp
  WHERE empno = p_empno;
  IF v_ename IS NULL THEN
    RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);
  END IF;
  IF v_job IS NULL THEN
    RAISE_APPLICATION_ERROR(-20020, 'No job for ' || p_empno);
  END IF;
  IF v_mngr IS NULL THEN
    RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);
  END IF;
  IF v_hiredate IS NULL THEN
    RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);
  END IF;
  DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno
|| ' validated without errors');
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
    DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
END;
```

The following shows the output in a case where the manager number is missing from an employee record:

```
Employee 100 validated without errors
```

END;
END;
14.1.4.4  Collection methods

Collection methods are functions and procedures that provide useful information about a collection that can aid in the processing of data in the collection.

14.1.4.4.1  COUNT

COUNT is a method that returns the number of elements in a collection.

Syntax

The syntax for using COUNT is:

```
<collection>.COUNT
```

Where `collection` is the name of a collection.

For a varray, COUNT always equals LAST.

Example

This example shows that an associative array can be sparsely populated, with gaps in the sequence of assigned elements. COUNT includes only the elements that were assigned a value.

```
DECLARE
  TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  sparse_arr sparse_arr_typ;
BEGIN
  sparse_arr(-100) := -100;
  sparse_arr(-10) := -10;
  sparse_arr(0) := 0;
  sparse_arr(10) := 10;
  sparse_arr(100) := 100;
  DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
END;
```

The following output shows that there are five populated elements included in COUNT:

```
__OUTPUT__
COUNT: 5
```

14.1.4.4.2  DELETE

The DELETE method deletes entries from a collection. You can call the DELETE method in three different ways.

Removing all entries from a collection

Use this form of the DELETE method to remove all entries from a collection:
Removing a specified entry from a collection

Use this form of the `DELETE` method to remove the specified entry from a collection:

```sql
<collection>.DELETE(<subscript>)
```

Removing all entries in a range

Use this form of the `DELETE` method to remove the entries that fall in the range specified by `first_subscript` and `last_subscript` (including the entries for the `first_subscript` and the `last_subscript`) from a collection.

```sql
<collection>.DELETE(<first_subscript>, <last_subscript>)
```

If `first_subscript` and `last_subscript` refer to elements that don't exist, elements that are in the range between the specified subscripts are deleted. If `first_subscript` is greater than `last_subscript`, or if you specify a value of `NULL` for one of the arguments, `DELETE` has no effect.

When you delete an entry, the subscript remains in the collection. You can reuse the subscript with an alternate entry. If you specify a subscript that doesn't exist in the call to the `DELETE` method, `DELETE` doesn't raise an exception.

Example

This example uses the `DELETE` method to remove the element with subscript 0 from the collection:

```sql
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    sparse_arr sparse_arr_typ;
    v_results VARCHAR2(50);
    v_sub NUMBER;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10) := -10;
    sparse_arr(0) := 0;
    sparse_arr(10) := 10;
    sparse_arr(100) := 100;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.DELETE(0);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    v_sub := sparse_arr.FIRST;
    WHILE v_sub IS NOT NULL LOOP
        IF sparse_arr(v_sub) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(v_sub) || ' ';
        END IF;
        v_sub := sparse_arr.NEXT(v_sub);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
```

COUNT: 5
COUNT: 4
Results: -100 -10 10 100

`COUNT` indicates that before the `DELETE` method, there were five elements in the collection. After the `DELETE` method is invoked, the collection contains four elements.
14.1.4.3 EXISTS

The EXISTS method verifies that a subscript exists in a collection. EXISTS returns TRUE if the subscript exists. If the subscript doesn't exist, EXISTS returns FALSE.

Syntax

The method takes a single argument: the subscript that you are testing for. The syntax is:

<collection>.EXISTS(<subscript>)

Where:

collection is the name of the collection.

subscript is the value that you are testing for. If you specify a value of NULL, EXISTS returns false.

Example

This example verifies that subscript number 10 exists in the associative array:

```sql
DECLARE
  TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  sparse_arr sparse_arr_typ;
BEGIN
  sparse_arr(-100) := -100;
  sparse_arr(-10)  := -10;
  sparse_arr(0)    :=  0;
  sparse_arr(10)   := 10;
  sparse_arr(100)  := 100;
  DBMS_OUTPUT.PUT_LINE('The index exists: ' ||
    CASE WHEN sparse_arr.exists(10) = TRUE THEN 'true' ELSE 'false' END);
END;
```

The index exists: true

Some collection methods raise an exception if you call them with a subscript that doesn't exist in the specified collection. Rather than raising an error, the EXISTS method returns a value of FALSE.

14.1.4.4 EXTEND

The EXTEND method increases the size of a collection. The EXTEND method has three variations.

Variation 1: Appending a single element

The first variation appends a single NULL element to a collection. The syntax for this variation is:

<collection>.EXTEND

Where collection is the name of a collection.

This example uses the EXTEND method to append a single, null element to a collection:

```sql
DECLARE
  TYPE sparse_arr_typ IS TABLE OF NUMBER;
  sparse_arr sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
```
**Variation 2: Appending a specified number of elements**

This variation of the `EXTEND` method appends a specified number of elements to the end of a collection:

```sql
<collection>.EXTEND(<count>)
```

Where:

- `<collection>` is the name of a collection.
- `<count>` is the number of null elements added to the end of the collection.

This example uses the `EXTEND` method to append multiple null elements to a collection:

```sql
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST
    LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
COUNT: 5
COUNT: 8
Results: -100 -10 0 10 100 NULL NULL NULL

**COUNT** indicates that before the `EXTEND` method, there were five elements in the collection. After the `EXTEND` method is invoked, the collection contains eight elements.

**Variation 3: Appending copies of an element**

```sql
v_results VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST .. sparse_arr.LAST
    LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL ';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
COUNT: 5
COUNT: 6
Results: -100 -10 0 10 100 NULL NULL

**COUNT** indicates that before the `EXTEND` method, there were five elements in the collection. After the `EXTEND` method is invoked, the collection contains six elements.

```
This variation of the `EXTEND` method appends a specified number of copies of a particular element to the end of a collection:

```
<collection>.EXTEND(<count>, <index_number>)
```

Where:

- `collection` is the name of a collection.
- `count` is the number of elements added to the end of the collection.
- `index_number` is the subscript of the element that's being copied to the collection.

This example uses the `EXTEND` method to append multiple copies of the second element to the collection:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER;
    sparse_arr   sparse_arr_typ := sparse_arr_typ(-100,-10,0,10,100);
    v_results    VARCHAR2(50);
BEGIN
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    sparse_arr.EXTEND(3, 2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || sparse_arr.COUNT);
    FOR i IN sparse_arr.FIRST..sparse_arr.LAST LOOP
        IF sparse_arr(i) IS NULL THEN
            v_results := v_results || 'NULL';
        ELSE
            v_results := v_results || sparse_arr(i) || ' ';
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;
```

```
COUNT: 5
COUNT: 8
Results: -100 -100 -100 -100 -100 -10 -10
```

`COUNT` indicates that before the `EXTEND` method, there were five elements in the collection. After the `EXTEND` method is invoked, the collection contains eight elements.

**Note**

You can't use the `EXTEND` method on a null or empty collection.

**14.1.4.4.5 FIRST**

`FIRST` is a method that returns the subscript of the first element in a collection.

**Syntax**

The syntax for using `FIRST` is as follows:

```
<collection>.FIRST
```

Where `collection` is the name of a collection.

**Example**

This example displays the first element of the associative array:

```
DECLARE
```

```
```
14.1.4.4.6 LAST

LAST is a method that returns the subscript of the last element in a collection.

Syntax

The syntax for using LAST is as follows:

```
<collection>.LAST
```

Where `collection` is the name of a collection.

Example

This example displays the last element of the associative array:

```
DECLARE
    TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY
        BINARY_INTEGER;
    sparse_arr sparse_arr_typ;
BEGIN
    sparse_arr(-100) := -100;
    sparse_arr(-10)  := -10;
    sparse_arr(0)    := 0;
    sparse_arr(10)   := 10;
    sparse_arr(100)  := 100;
    DBMS_OUTPUT.PUT_LINE('FIRST element: ' || sparse_arr(sparse_arr.FIRST));
END;
FIRST element: -100
```

14.1.4.4.7 LIMIT

LIMIT is a method that returns the maximum number of elements permitted in a collection. LIMIT applies only to varrays. The syntax for using LIMIT is:

```
<collection>.LIMIT
```

Where `collection` is the name of a collection.

For an initialized varray, LIMIT returns the maximum size limit determined by the varray type definition. If the varray is uninitialized (that is, it's a null varray), an exception is thrown.
For an associative array or an initialized nested table, the `LIMIT` returns `NULL`. If the nested table is uninitialized (that is, it’s a null nested table), an exception is thrown.

### 14.1.4.4.8 NEXT

`NEXT` is a method that returns the subscript that follows a specified subscript.

**Syntax**

The method takes a single argument: the `subscript` that you are testing for. The syntax is:

```sql
<collection>.NEXT(<subscript>)
```

Where `collection` is the name of the collection.

If the specified subscript is less than the first subscript in the collection, the function returns the first subscript. If the subscript doesn’t have a successor, `NEXT` returns `NULL`. If you specify a `NULL` subscript, `PRIOR` doesn’t return a value.

**Example**

This example uses `NEXT` to return the subscript that follows subscript 10 in the associative array, `sparse_arr`:

```sql
DECLARE
  TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  sparse_arr      sparse_arr_typ;
BEGIN
  sparse_arr(-100) := -100;
  sparse_arr(-10)  := -10;
  sparse_arr(0)    := 0;
  sparse_arr(10)   := 10;
  sparse_arr(100)  := 100;
  DBMS_OUTPUT.PUT_LINE('NEXT element: ' || sparse_arr.next(10));
END;
```

**NEXT element:** 100

### 14.1.4.4.9 PRIOR

The `PRIOR` method returns the subscript that precedes a specified subscript in a collection.

**Syntax**

The method takes a single argument: the `subscript` that you are testing for. The syntax is:

```sql
<collection>.PRIOR(<subscript>)
```

Where `collection` is the name of the collection.

If the subscript specified doesn’t have a predecessor, `PRIOR` returns `NULL`. If the specified subscript is greater than the last subscript in the collection, the method returns the last subscript. If you specify a `NULL` subscript, `PRIOR` doesn’t return a value.

**Example**

This example uses `PRIOR` to return the subscript that precedes subscript 10 in the associative array, `sparse_arr`:

```sql
DECLARE
  TYPE sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  sparse_arr      sparse_arr_typ;
BEGIN
  sparse_arr(-100) := -100;
  sparse_arr(-10)  := -10;
  sparse_arr(0)    := 0;
  sparse_arr(10)   := 10;
  sparse_arr(100)  := 100;
  DBMS_OUTPUT.PUT_LINE('PRIOR element: ' || sparse_arr.prior(10));
END;
```

**PRIOR element:** 10
This example returns the subscript that precedes subscript 100 in the associative array `sparse_arr`:

```
DECLARE
type sparse_arr_typ IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  sparse_arr sparse_arr_typ;
BEGIN
  sparse_arr(-100) := -100;
  sparse_arr(-10)  := -10;
  sparse_arr(0)   :=  0;
  sparse_arr(10)  := 10;
  sparse_arr(100) := 100;
  dbms_output.put_line('PRIOR element: ' || sparse_arr.prior(100));
END;
```

**14.1.4.10 TRIM**

The `TRIM` method removes one or more elements from the end of a collection.

### Syntax

The syntax for the `TRIM` method is:

```
<collection>.TRIM([<count>])
```

Where:

- `<collection>` is the name of a collection.
- `<count>` is the number of elements removed from the end of the collection. EDB Postgres Advanced Server returns an error if `<count>` is less than 0 or greater than the number of elements in the collection.

### Example

This example uses the `TRIM` method to remove an element from the end of a collection:

```
DECLARE
  type sparse_arr_typ IS TABLE OF NUMBER;
  sparse_arr sparse_arr_typ := sparse_arr_typ(-106,-10,0,10,100);
BEGIN
  dbms_output.put_line('COUNT: ' || sparse_arr.COUNT);
  sparse_arr.TRIM;
  dbms_output.put_line('COUNT: ' || sparse_arr.COUNT);
END;
```

`COUNT` indicates that before the `TRIM` method, there were five elements in the collection. After the `TRIM` method is invoked, the collection contains four elements.

You can also specify the number of elements to remove from the end of the collection using the `TRIM` method:

```
DECLARE
  type sparse_arr_typ IS TABLE OF NUMBER;
```
COUNT indicates that before the TRIM method, there were five elements in the collection. After the TRIM method is invoked, the collection contains three elements.

14.1.4.5  EDB Postgres Advanced Server exceptions

List of server exceptions

The following table lists the predefined exceptions, the SQLSTATE values, associated redwood error code, and descriptions of the exceptions.

<table>
<thead>
<tr>
<th>Exception name</th>
<th>SQLSTATE</th>
<th>Redwood error code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value_error</td>
<td>22000</td>
<td>-6502</td>
<td>The exception occurs when the conversion of a character string to a number fails.</td>
</tr>
<tr>
<td>invalid_number</td>
<td>22000</td>
<td>-6502</td>
<td>The exception is raised in PL statement when the conversion of a character string to number fails.</td>
</tr>
<tr>
<td>datetime_value_out_of_range</td>
<td>22008</td>
<td>-1863</td>
<td>The exception occurs while writing a field in date format that is outside the valid range.</td>
</tr>
<tr>
<td>divide_by_zero</td>
<td>22012</td>
<td>-1876</td>
<td>The exception occurs when a program attempts to divide a number by zero.</td>
</tr>
<tr>
<td>zero_divide</td>
<td>22012</td>
<td>-1876</td>
<td>The exception occurs when a program attempts to divide a number by zero.</td>
</tr>
<tr>
<td>dup_val_on_index</td>
<td>23505</td>
<td>-1</td>
<td>The exception occurs when a program attempts to store duplicate values in a column that's constrained by a unique index.</td>
</tr>
<tr>
<td>invalid_cursor</td>
<td>34000</td>
<td>-1001</td>
<td>The exception occurs when a program attempts a cursor operation on an invalid cursor, such as closing an unopened cursor.</td>
</tr>
<tr>
<td>cursor_already_open</td>
<td>42P03</td>
<td>-6511</td>
<td>The exception occurs when a program attempts to open an already open cursor.</td>
</tr>
<tr>
<td>collection_is_null</td>
<td>P1403</td>
<td>-6531</td>
<td>The exception occurs when a program attempts to assign values to the elements of nested table or varray that are uninitialized.</td>
</tr>
<tr>
<td>subscript_beyond_count</td>
<td>P1404</td>
<td>-6533</td>
<td>The exception occurs when a program attempts to reference a nested table or varray using an index number larger than the number of elements in the collection.</td>
</tr>
<tr>
<td>subscript_outside_limit</td>
<td>P1405</td>
<td>-6532</td>
<td>The exception occurs when a program attempts to reference a nested table or varray element using an index number that's outside the range.</td>
</tr>
</tbody>
</table>

DBMS_CRYPTO package

| ciphersuiteinvalid       | 00009    | -28827             | The exception occurs when the cipher suite isn't defined. |
| ciphersuitenull           | 00009    | -28829             | The exception occurs when no value is specified for the cipher suite or it contains a NULL value. |
| keybadsize               | 00009    | 0                  | The exception occurs when the specified key size is bad. |
| keynull                  | 00009    | -28239             | The exception occurs when the key isn't specified. |

UTL_FILE package

| invalid_filehandle       | P0001 00009 | -29282     | The exception occurs when file handle is invalid. |
| invalid_maxlinesize      | P0001 00009 | -29287     | The exception occurs when the max line size is invalid or the max line size value isn't within the range. |
| invalid_mode             | P0001 00009 | -29281     | The exception occurs when the open_mode parameter in FOPEN is invalid. |
| invalid_operation        | P0001 00009 | -29283     | The exception occurs when the file can't be opened or used upon request. |
| invalid_path             | P0001 00009 | -29280     | The exception occurs when the file location or file name is invalid. |
The exception occurs when the operating system error occurred during the read operation.

The exception occurs when the operating system error occurred during the write operation.

The exception occurs when the end of HTTP response body is reached.

The exception occurs when the fixed-width multibyte character isn’t allowed as a URL character set.

The exception occurs when the URL includes badly formed escape-code sequences.

EDB Postgres Advanced Server keywords

A keyword is a word that’s recognized by the EDB Postgres Advanced Server parser as having a special meaning or association. You can use the `pg_get_keywords()` function to retrieve an up-to-date list of the EDB Postgres Advanced Server keywords:

```
acctg=#
acctg> SELECT * FROM pg_get_keywords();

<table>
<thead>
<tr>
<th>word</th>
<th>catcode</th>
<th>catdesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>U</td>
<td>unreserved</td>
</tr>
<tr>
<td>absolute</td>
<td>U</td>
<td>unreserved</td>
</tr>
<tr>
<td>access</td>
<td>U</td>
<td>unreserved</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`pg_get_keywords()` returns a table containing the keywords recognized by EDB Postgres Advanced Server:

- The `word` column displays the keyword.
- The `catcode` column displays a category code.
- The `catdesc` column displays a brief description of the category to which the keyword belongs.

You can use any character in an identifier if the name is enclosed in double quotes. You can selectively query the `pg_get_keywords()` function to retrieve an up-to-date list of the EDB Postgres Advanced Server keywords that belong to a specific category:

```
SELECT * FROM pg_get_keywords() WHERE catcode = 'code';
```

Where `code` is:

- **R** – The word is reserved. You can never use reserved keywords as an identifier. They are reserved for use by the server.
- **U** – The word is unreserved. Unreserved words are used internally in some contexts, but you can use them as a name for a database object.
- **T** – The word is used internally but can be used as a name for a function or type.
- **C** – The word is used internally and can’t be used as a name for a function or type.

For more information about EDB Postgres Advanced Server identifiers and keywords, see the PostgreSQL core documentation.

14.2 Database administrator reference

This reference information applies to database administrators.

14.2.1 Audit logging configuration parameters

Use the following configuration parameters to control database auditing. See Summary of configuration parameters to determine if a change to the configuration parameter:

- Takes effect immediately
- Requires reloading the configuration
- Requires restarting the database server

```
edb_audit
```
Enables or disables database auditing. The values xml or csv enable database auditing. These values represent the file format in which to capture auditing information. none disables database auditing and is the default.

Note
Set the logging_collector parameter to on to enable the edb_audit parameter.

db_audit_archiver
Enables or disables database audit archiving.

db_audit_archiver_timeout
Enforces a timeout in seconds when a database attempts to archive a log file. The valid range is 30 seconds to one day.

db_audit_archiver_filename_prefix
Specifies the file name of an audit log file that needs to be archived. The file name must align with the edb_audit_filename parameter. The default value for edb_audit_archiver_filename_prefix is audit-. The audit files with edb_audit_archiver_filename_prefix in the edb_audit_directory are eligible for compression and expiration.

db_audit_archiver_compress_time_limit
Specifies the time in seconds after which audit logs are eligible for compression. The possible values are:

* 0 — Compression starts as soon as the log file isn't a current file.
* -1 — Compression of the log file on a timely basis doesn't occur.

db_audit_archiver_compress_size_limit
Specifies a file size threshold, in megabytes, after which audit logs are eligible for compression. If you have multiple audit logs whose combined size exceeds the threshold, the individual files are eligible for compression even though the size of each individual audit log may not meet the threshold.

If the parameter is set to -1, compression of the log file on a size basis doesn't occur.

db_audit_archiver_compress_command
Specifies the command to execute compression of the audit log files. The default value for edb_audit_archiver_compress_command is gzip %p. The gzip provides a standard method of compressing files. The %p in the string is replaced with the path name of the file to archive.

db_audit_archiver_compress_suffix
Specifies the file name of an already compressed log file. The file name must align with edb_audit_archiver_compress_command. The default file name is .gz.

db_audit_archiver_expire_time_limit
Specifies the time, in seconds, after which audit logs are eligible to expire. The possible values to set this parameter are:

* 0 — Expiration starts as soon as the log file isn't a current file.
* -1 — Expiration of the log file on a timely basis doesn't occur.
**edb_audit_archiver_expire_size_limit**

Specifies a file size threshold in megabytes, after which audit logs are eligible to expire. If you have multiple audit logs whose combined size exceeds the threshold, the individual files are eligible for expiration even though the size of each individual audit log may not meet the threshold.

If the parameter is set to `-1`, expiration of a log file on the size basis doesn't occur.

**edb_audit_archiver_expire_command**

Specifies the command to execute on an expired audit log file before removal.

**edb_audit_archiver_sort_file**

Identifies the oldest log file to sort alphabetically or based on `mtime`. `mtime` indicates sorting of files based on file modification time. `alphabetic` indicates sorting of files alphabetically based on the file name.

**edb_audit_directory**

Specifies the directory where the log files are created. The path of the directory can be relative to the data folder or absolute. The default is the `PGDATA/edb_audit` directory.

**edb_audit_filename**

Specifies the file name of the audit file where the auditing information are stored. The default file name is `audit-%Y%m%d_%H%M%S`. The escape sequences `%Y`, `%m`, and so on are replaced by the appropriate current values according to the system date and time.

**edb_audit_rotation_day**

Specifies the day of the week on which to rotate the audit files. Valid values are `sun`, `mon`, `tue`, `wed`, `thu`, `fri`, `sat`, `every`, and `none`. To disable rotation, set the value to `none`. To rotate the file every day, set the `edb_audit_rotation_day` value to `every`. To rotate the file on a specific day of the week, set the value to the desired day of the week. `every` is the default value.

**edb_audit_rotation_size**

Specifies a file size threshold, in megabytes, when file rotation is forced to occur. The default value is 0 MB. If the parameter is commented out or set to 0, rotation of the file on a size basis doesn't occur.

**edb_audit_rotation_seconds**

Specifies the rotation time in seconds to create a log file. To disable this feature, set this parameter to 0, which is the default.

**edb_audit_connect**

Enables auditing of database connection attempts by users. To disable auditing of all connection attempts, set `edb_audit_connect` to `none`. To audit all connection attempts, set the value to `all`. To audit all failed connection attempts, set the value to `failed`, which is the default.

**edb_audit_disconnect**

Enables auditing of database disconnections by connected users. To enable auditing of disconnections, set the value to `all`. To disable, set the value to `none`, which is the default.
**edb_audit_statement**

Specifies auditing of different categories of SQL statements. You can specify various combinations of the following values: none, dml, insert, update, delete, truncate, select, error, rollback, ddl, create, drop, alter, grant, revoke, set, all, and (select | update | delete | insert)@groupname. The default is ddl and error. See Selecting SQL statements to audit for information about setting this parameter.

**edb_audit_tag**

Specifies a string value to include in the audit log file for each entry as a tracking tag.

**edb_log_every_bulk_value**

Bulk processing logs the resulting statements into both the EDB Postgres Advanced Server log file and the EDB Audit log file. However, logging every statement in bulk processing is costly. You can control this with the edb_log_every_bulk_value configuration parameter. When set to true, every statement in bulk processing is logged. During bulk execution, when edb_log_every_bulk_value is set to false, a log message is recorded once per bulk processing along with the number of rows processed. In addition, the duration is emitted once per bulk processing. Default is false.

**edb_audit_destination**

Specifies whether to record the audit log information in the directory as given by the edb_audit_directory parameter or to the directory and file managed by the syslog process. Set to file to use the directory specified by edb_audit_directory, which is the default setting. Set to syslog to use the syslog process and its location as configured in the /etc/syslog.conf file. The syslog setting is valid for EDB Postgres Advanced Server running on a Linux host and isn't supported on Windows systems.

!!! Note In recent Linux versions, syslog was replaced with rsyslog, and the configuration file is in /etc/rsyslog.conf.

Note

EDB Postgres Advanced Server allows administrative users associated with administrative privileges to audit statements by any user, group, or role. By auditing specific users, you can minimize the number of audit records generated. For information, see the examples under Selecting SQL statements to audit.

### 14.2.2 Index Advisor components

The Index Advisor shared library interacts with the query planner to make indexing recommendations.

On Windows, the EDB Postgres Advanced Server installer creates the shared library in the libdir subdirectory of your EDB Postgres Advanced Server home directory. The shared library is:

**index_advisor.dll**

For Linux, install the edb-as<xx>-server-indexadvisor RPM package, where <xx> is the EDB Postgres Advanced Server version number. The shared library is:

**index_advisor.so**

Only a superuser can load libraries in the libdir directory. A database administrator can allow a non-superuser to use Index Advisor by manually copying the Index Advisor file from the libdir directory into the libdir/plugins directory under your EDB Postgres Advanced Server home directory. Only allow a trusted non-superuser to have access to the plugin. This is an unsafe practice in a production environment.

The installer also creates the Index Advisor utility program and setup script:

**pg_advise_index**

A utility program that reads a user-supplied input file containing SQL queries and produces a text file containing CREATE INDEX statements. You can use these statements to create the indexes recommended by the Index Advisor. The pg_advise_index program is located in the bin subdirectory of the EDB Postgres Advanced Server home directory.
pg_advise_index asks the backend process to load the index_advisor plugin first from $libdir/plugins. If not found, then it writes the error in the server log file and attempts to load from $libdir.

index_advisor_log

Index Advisor logs indexing recommendations in the index_advisor_log table. If Index Advisor doesn’t find the index_advisor_log table in the user’s search path, it stores any indexing recommendations in a temporary table of the same name. The temporary table exists only for rest of the current session.

show_index_recommendations()

A PL/pgSQL function that interprets and displays the recommendations made during a specific Index Advisor session, as identified by its backend process ID.

index_recommendations

Index Advisor creates the index_recommendations view based on information stored in the index_advisor_log table during a query analysis. The view produces output in the same format as the show_index_recommendations() function but contains Index Advisor recommendations for all stored sessions. The result set returned by the show_index_recommendations() function is limited to a specified session.

14.2.3 Configuration parameters compatible with Oracle databases

EDB Postgres Advanced Server supports developing and running applications compatible with PostgreSQL and Oracle. You can alter some system behaviors to act in a more PostgreSQL- or in a more Oracle-compliant manner. You control these behaviors by using configuration parameters.

- edb_redwood_date — Controls whether or not a time component is stored in DATE columns. For behavior compatible with Oracle databases, set edb_redwood_date to TRUE. See edb_redwood_date.
- edb_redwood_raw_names — Controls whether database object names appear in uppercase or lowercase letters when viewed from Oracle system catalogs. For behavior compatible with Oracle databases, edb_redwood_raw_names is set to its default value of FALSE. To view database object names as they are actually stored in the PostgreSQL system catalogs, set edb_redwood_raw_names to TRUE. See edb_redwood_raw_names.
- edb_redwood_strings — Equates NULL to an empty string for purposes of string concatenation operations. For behavior compatible with Oracle databases, set edb_redwood_strings to TRUE. See edb_redwood_strings.
- edb_stmt_level_tx — Isolates automatic rollback of an aborted SQL command to statement level rollback only – the entire, current transaction is not automatically rolled back as is the case for default PostgreSQL behavior. For behavior compatible with Oracle databases, set edb_stmt_level_tx to TRUE; however, use only when absolutely necessary. See edb_stmt_level_tx.
- oracle_home — Point EDB Postgres Advanced Server to the correct Oracle installation directory. See oracle_home.

14.2.3.1 edb_redwood_date

DATE can appear as the data type of a column in the commands. To translate it to TIMESTAMP when the table definition is stored in the database, set the configuration parameter edb_redwood_date to TRUE. In this case, a time component is stored in the column with the date. This is consistent with Oracle’s DATE data type.

If edb_redwood_date is set to FALSE, the column’s data type in a CREATE TABLE or ALTER TABLE command remains as a native PostgreSQL DATE data type. It's stored as such in the database. The PostgreSQL DATE data type stores only the date without a time component in the column.

Regardless of the setting of edb_redwood_date, when DATE appears as a data type in any other context, it’s always internally translated to a TIMESTAMP, and thus can handle a time component if present. These contexts include the data type of a variable in an SPL declaration section or the data type of a formal parameter in an SPL procedure or SPL function or the return type of an SPL function.

14.2.3.2 edb_redwood_raw_names

About database object name display

When edb_redwood_raw_names is set to its default value of FALSE, database object names such as table names, column names, trigger names, program names, and user names appear in uppercase letters when viewed from Oracle catalogs. For a complete list of supported catalog views, see Catalog views. In addition, quotation marks enclose names that were created with enclosing quotation marks.
When `<edb_redwood_raw_names>` is set to `TRUE`, the database object names are displayed exactly as stored in the PostgreSQL system catalogs when viewed from the Oracle catalogs. Thus, names created without quotation marks appear in lowercase as expected in PostgreSQL. Names created in quotation marks appear exactly as they were created but without the quotation marks.

**Example**

For example, the following user name is created, and then a session is started with that user:

```plaintext
CREATE USER reduser IDENTIFIED BY password;
edb=\c reduser
Password for user reduser:
You are now connected to database "edb" as user "reduser".
```

When connected to the database as `reduser`, the following tables are created:

```
CREATE TABLE all_lower (col INTEGER);
CREATE TABLE all_upper (col INTEGER);
CREATE TABLE "Mixed_Case" ("col" INTEGER);
```

When viewed from the Oracle catalog `USER_TABLES` with `<edb_redwood_raw_names>` set to the default value `FALSE`, the names appear in upper case. The exception is the `Mixed_Case` name, which appears as created and also in quotation marks.

<table>
<thead>
<tr>
<th>schema_name</th>
<th>table_name</th>
<th>tablespace_name</th>
<th>status</th>
<th>temporary</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDUSER</td>
<td>ALL_LOWER</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
<tr>
<td>REDUSER</td>
<td>ALL_UPPER</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
<tr>
<td>REDUSER</td>
<td>&quot;Mixed_Case&quot;</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
</tbody>
</table>
(3 rows)

When viewed with `<edb_redwood_raw_names>` set to `TRUE`, the names appear in lower case except for the `Mixed_Case` name. That name appears as created but without the quotation marks.

```
edb=> SELECT * FROM USER_TABLES;
```

<table>
<thead>
<tr>
<th>schema_name</th>
<th>table_name</th>
<th>tablespace_name</th>
<th>status</th>
<th>temporary</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduser</td>
<td>all_lower</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
<tr>
<td>reduser</td>
<td>all_upper</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
<tr>
<td>reduser</td>
<td>Mixed_Case</td>
<td></td>
<td>VALID</td>
<td>N</td>
</tr>
</tbody>
</table>
(3 rows)

These names now match the case when viewed from the PostgreSQL `pg_tables` catalog.

```
edb=> SELECT schemaname, tablename, tableowner FROM pg_tables WHERE tableowner = 'reduser';
```

<table>
<thead>
<tr>
<th>schemaname</th>
<th>tablename</th>
<th>tableowner</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduser</td>
<td>all_lower</td>
<td>reduser</td>
</tr>
<tr>
<td>reduser</td>
<td>all_upper</td>
<td>reduser</td>
</tr>
<tr>
<td>reduser</td>
<td>Mixed_Case</td>
<td>reduser</td>
</tr>
</tbody>
</table>
(3 rows)

### 14.2.3.3 `<edb_redwood_strings>`

**About string display after concatenation**

In Oracle, when a string is concatenated with a null variable or null column, the result is the original string. However, in PostgreSQL, concatenation of a string with a null variable or null column gives a null result.

If the `<edb_redwood_strings>` parameter is set to `TRUE`, the concatenation operation results in the original string as done by Oracle. If `<edb_redwood_strings>` is set to `FALSE`, the native PostgreSQL behavior is maintained.
Example

The sample application contains a table of employees. This table has a column named `comm` that's null for most employees. The following query is run with `edb_redwood_string` set to `FALSE`. The concatenation of a null column with non-empty strings produces a final result of null, so only employees that have a commission appear in the query result. The output line for all other employees is `null`.

```
SET edb_redwood_strings TO off;
SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
      TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;

EMPLOYEE COMPENSATION
-----------------------
ALLEN     1,600.00     300.00
WARD      1,250.00     500.00
MARTIN    1,250.00    1,400.00
TURNER    1,500.00     .00

(14 rows)
```

The following is the same query executed when `edb_redwood_strings` is set to `TRUE`. Here, the value of a null column is treated as an empty string. The concatenation of an empty string with a non-empty string produces the non-empty string. This result is consistent with the results produced by Oracle for the same query.

```
SET edb_redwood_strings TO on;
SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
      TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;

EMPLOYEE COMPENSATION
-----------------------
SMITH      800.00
ALLEN     1,600.00     300.00
WARD      1,250.00     500.00
JONES      2,075.00
MARTIN    1,250.00    1,400.00
BLAKE      2,450.00
CLARK      2,450.00
SCOTT      3,080.00
KING       5,080.00
TURNER    1,500.00     .00
ADAMS      1,180.00
JAMES      950.00
FORD       3,080.00
MILLER     1,380.00

(14 rows)
```

14.2.3.4 edb_stmt_level_tx

About statement level transaction isolation

In Oracle, when a runtime error occurs in a SQL command, all the updates on the database caused by that single command are rolled back. This is called statement level transaction isolation. For example, if a single `UPDATE` command successfully updates five rows but an attempt to update a sixth row results in an exception, the updates to all six rows made by this `UPDATE` command are rolled back. The effects of prior SQL commands that weren't yet committed or rolled back are pending until a `COMMIT` or `ROLLBACK` command is executed.

In PostgreSQL, if an exception occurs while executing a SQL command, all the updates on the database since the start of the transaction are rolled back. In addition, the transaction is left in an aborted state, and either a `COMMIT` or `ROLLBACK` command must be issued before another transaction can start.

If `edb_stmt_level_tx` is set to `TRUE`, then an exception doesn't automatically roll back prior uncommitted database updates, emulating the Oracle behavior. If `edb_stmt_level_tx` is set to `FALSE`, then an exception rolls back uncommitted database updates.
Note

Use `edb_stmt_level_tx` set to `TRUE` only when necessary, as this setting can have a negative performance impact.

Example

The following example run in PSQL shows that when `edb_stmt_level_tx` is `FALSE`, the abort of the second `INSERT` command also rolls back the first `INSERT` command. In PSQL, the command `\set AUTOCOMMIT off` must be issued, otherwise every statement commits automatically, which defeats the purpose of showing the effect of `edb_stmt_level_tx`.

```sql
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO off;

INSERT INTO emp (empno, ename, deptno) VALUES (9801, 'JONES', 40);
INSERT INTO emp (empno, ename, deptno) VALUES (9802, 'JONES', 40);
ERROR: insert or update on table "emp" violates foreign key constraint "emp_ref_dept_fk"
DETAIL: Key (deptno)=<0> is not present in table "dept".

COMMIT;
SELECT empno, ename, deptno FROM emp WHERE empno > 9800;
empno | ename | deptno
-------+-------+--------
(0 rows)

In the following example, with `edb_stmt_level_tx` set to `TRUE`, the first `INSERT` command wasn’t rolled back after the error on the second `INSERT` command. At this point, the first `INSERT` command can either be committed or rolled back.

```sql
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

INSERT INTO emp (empno, ename, deptno) VALUES (9801, 'JONES', 40);
INSERT INTO emp (empno, ename, deptno) VALUES (9802, 'JONES', 40);
ERROR: insert or update on table "emp" violates foreign key constraint "emp_ref_dept_fk"
DETAIL: Key (deptno)=<0> is not present in table "dept".

SELECT empno, ename, deptno FROM emp WHERE empno > 9800;
empno | ename | deptno
-------+-------+--------
9001   | JONES | 40
(1 row)
COMMIT;
```

You can issue a `ROLLBACK` command instead of the `COMMIT` command. In that case, the insertion of employee number `9001` is rolled back as well.

14.2.3.5 oracle_home

Before creating a link to an Oracle server, you must direct EDB Postgres Advanced Server to the correct Oracle home directory. Set the `LD_LIBRARY_PATH` environment variable on Linux (or `PATH` on Windows) to the `lib` directory of the Oracle client installation directory.

Alternatively, you can also set the value of the `oracle_home` configuration parameter in the `postgresql.conf` file. The value specified in the `oracle_home` configuration parameter overrides the `LD_LIBRARY_PATH` environment variable in Linux and `PATH` environment variable in Windows.

Note

The `oracle_home` configuration parameter must provide the correct path to the Oracle client, i.e., OCI library.
Setting the configuration parameter

To set the `oracle_home` configuration parameter in the `postgresql.conf` file, add the following line:

```plaintext
oracle_home = 'lib_directory'
```

Substitute the name of the `oracle_home` path to the Oracle client installation directory that contains `libclntsh.so` in Linux and `oci.dll` in Windows for `lib_directory`.

Restarting the server

After setting the `oracle_home` configuration parameter, you must restart the server for the changes to take effect. Restart the server:

- On Linux, using the `systemctl` command or `pg_ctl` services.
- On Windows, from the Windows Services console.

14.2.4 Summary of configuration parameters

This summary table list all EDB Postgres Advanced Server configuration parameters along with a number of key attributes of the parameters.

These attributes are described by the following columns of the summary table:

- **Parameter** — Configuration parameter name and a brief description of the configuration parameter. Parameters for use only with EDB Postgres Advanced Server (EPAS) are denoted by (EPAS only).

- **Scope** — Scope of effect of the configuration parameter setting:
  - **Cluster** — Setting affects the entire database cluster, that is, all databases managed by the database server instance.
  - **Database** — Setting can vary by database and is established when the database is created. Applies to a small number of parameters related to locale settings.
  - **Session** — Setting can vary down to the granularity of individual sessions.

  In other words, different settings can be made for the following entities in which the latter settings in this list override prior ones:

  - The entire database cluster
  - Specific databases in the database cluster
  - Specific roles
  - Specific roles when connected to specific databases
  - A specific session

- **In effect** — When a changed parameter setting takes effect:
  - **Preset** — Established when the EDB Postgres Advanced Server product is built or a particular database is created. This is a read-only parameter and can’t be changed.
  - **Restart** — You must restart the database server.
  - **Reload** — You must reload configuration file or restart the database server.
  - **Immediately** — Immediately effective in a session if the `PGOPTIONS` environment variable or the `SET` command is used to change the setting in the current session. Effective in new sessions if `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` commands are used to change the setting.

- **Used by** — Type of operating system account or database role that must be used to put the parameter setting into effect.
  - **EPAS account** — The EDB Postgres Advanced Server service account (`enterprisedb` for an installation compatible with Oracle databases, `postgres` for a PostgreSQL compatible mode installation).
  - **Superuser** — Database role with superuser privileges.
  - **User** — Any database role with permissions on the affected database object, that is, the database or role to alter with the `ALTER` command.
  - **n/a** — No user can change the parameter setting.

**Note**

Some parameters can never be altered. These are designated as *Note: For internal use only* in the description.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Scope</th>
<th>In effect</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>application_name</td>
<td>Session</td>
<td>Immediately</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>User</td>
</tr>
<tr>
<td>archive_command</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>archive_mode</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>archive_timeout</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>array_nulls</td>
<td>Session</td>
<td>Immediately</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>authentication_timeout</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>autovacuum</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>autovacuum_analyze_scale_factor</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>autovacuum_analyze_threshold</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>autovacuum_freeze_max_age</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>autovacuum_max_workers</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>autovacuum_multixact_freeze_max_age</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>autovacuum_naptime</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>autovacuum_vacuum_cost_delay</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-------------</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td><code>autovacuum_vacuum_cost_limit</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>autovacuum_vacuum_scale_factor</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>autovacuum_vacuum_threshold</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>autovacuum_work_mem</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>backslash_quote</code></td>
<td>Session</td>
<td>Immediately Used EPAS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bgwriter_delay</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bgwriter_lru_maxpages</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bgwriter_lru_multiplier</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>block_size</code></td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bonjour</code></td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bonjour_name</code></td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bytea_output</code></td>
<td>Session</td>
<td>Immediately Used EPAS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>check_function_bodies</code></td>
<td>Session</td>
<td>Immediately Used EPAS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>checkpoint_completion_target</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>checkpoint_segments</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td>(Deprecated)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>checkpoint_timeout</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------</td>
<td>-----------</td>
<td>----------------------------</td>
</tr>
<tr>
<td><strong>checkpoint_warning</strong></td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>client_encoding</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><strong>client_min_messages</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><strong>commit_delay</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td><strong>commit_siblings</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><strong>config_file</strong></td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td><strong>constraint_exclusion</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><strong>cpu_index_tuple_cost</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><strong>cpu_operator_cost</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><strong>cpu_tuple_cost</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><strong>cursor_tuple_fraction</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><strong>custom_variable_classes</strong></td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td><strong>data_checksums</strong></td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td><strong>data_directory</strong></td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>datestyle</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><strong>db_dialect</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>dbms_alert.max_alerts</strong></td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>dbms_pipe.total_message_buffer</strong></td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td><strong>db_user_namespace</strong></td>
<td>Cluster</td>
<td>Reload</td>
<td>Super user</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>deadlock_timeout</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------</td>
<td>------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>debug_assertions</td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>debug_pretty_print</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>debug_print_parse</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>debug_print_plan</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>debug_print Rewritten</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>default_heap_fillfactor (EPAS only)</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>default_statistics_target</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>default_tablespace</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>default_text_search_config</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>default_transaction_deferrable</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>default_transaction_isolation</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>default_transaction_read_only</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>default_with_oids</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>default_with_rowids (EPAS only)</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>dynamic_library_path</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>dynamic_shared_memory_type</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>edb_audit (EPAS only)</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>edb_audit_archiver (EPAS only)</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>edb_audit_archiver_timeout</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>edb_audit_archiver_filename_prefix</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>edb_audit_archiver_compress_type</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>edb_audit_archiver_compress_size_limit</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Scope</td>
<td>In effect</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td><code>edb_audit_archiver_compress_command</code></td>
<td>(EPAS only) — Compresses the audit log files.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_archiver_compress_suffix</code></td>
<td>(EPAS only) — Determines suffix for an already compressed log file. The parameter must align with <code>edb_audit_archiver_compress_command</code>.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_archiver_expire_time_limit</code></td>
<td>(EPAS only) — Shows time in seconds after which the audit logs are eligible for expiration.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_archiver_expire_size_limit</code></td>
<td>(EPAS only) — Shows total size in megabytes after which the audit logs are eligible for expiration.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_archiver_expire_command</code></td>
<td>(EPAS only) — Execute on an expired audit log before removing it.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_archiver_sort_file</code></td>
<td>(EPAS only) — Identifies the oldest log file and sorts them alphabetically or based on mtime.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_connect</code></td>
<td>(EPAS only) — Audits each successful connection.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_destination</code></td>
<td>(EPAS only) — Sets <code>edb_audit_directory</code> or syslog as the destination directory for audit files. The syslog setting is only valid for a Linux system.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_directory</code></td>
<td>(EPAS only) — Sets the destination directory for audit files.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_disconnect</code></td>
<td>(EPAS only) — Audits end of a session.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_filename</code></td>
<td>(EPAS only) — Sets the file name pattern for audit files.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_rotation_day</code></td>
<td>(EPAS only) — Automatic rotation of log files based on day of week.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td><code>edb_audit_rotation_seconds</code></td>
<td>(EPAS only) — Automatic log file rotation occurs after N seconds.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------</td>
<td>---------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td><strong>edb_audit_rotation_size</strong></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>(EPAS only) — Automatic log file rotation occurs after N Megabytes.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbAuditStatement</strong></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>(EPAS only) — Sets the type of statements to audit.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbAuditTag</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>(EPAS only) — Specifies a tag to include in the audit log.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbConnectByOrder</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>(EPAS only) — Sorts results of CONNECT BY queries with no ORDER BY to depth-first order. Note: For internal use only.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbDataRedaction</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>(EPAS only) — Enables data redaction.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbDynatune</strong></td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>(EPAS only) — Sets the edb utilization percentage.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbDynatuneProfile</strong></td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>(EPAS only) — Sets the workload profile for dynatune.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbEnablePruning</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>(EPAS only) — Enables the planner to early-prune partitioned tables.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbLogEveryBulkValue</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>Superuser</td>
</tr>
<tr>
<td>(EPAS only) — Sets the statements logged for bulk processing.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbMaxResourceGroups</strong></td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>(EPAS only) — Specifies the maximum number of resource groups for simultaneous use.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbMaxSpinsPerDelay</strong></td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>(EPAS only) — Specifies the number of times a session spins while waiting for a lock.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbRedwoodDate</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>(EPAS only) — Determines whether DATE should behave like a TIMESTAMP or not.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbRedwoodGreatestLeast</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>(EPAS only) — Determines how GREATEST and LEAST functions should handle NULL parameters.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbRedwoodRawNames</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>(EPAS only) — Returns the unmodified name stored in the PostgreSQL system catalogs from Redwood Interfaces.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbRedwoodStrings</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>(EPAS only) — Treats NULL as an empty string when concatenated with a text value.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbResourceGroup</strong></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>(EPAS only) — Specifies the resource group to be used by the current process.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbSqlProtect.Enabled</strong></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>(EPAS only) — Defines whether SQL/Protect should track queries or not.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbSqlProtect.Level</strong></td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>(EPAS only) — Defines the behavior of SQL/Protect when an event is found.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>edbSqlProtect.MaxProtectedRelations</strong></td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>(EPAS only) — Sets the maximum number of relations protected by SQL/Protect per role.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
<td>--------------------------</td>
</tr>
<tr>
<td><code>edb_sql_protect.max_protected_roles</code> (EPAS only) — Sets the maximum number of roles protected by SQL/Protect.</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td><code>edb_sql_protect.max_queries_to_save</code> (EPAS only) — Sets the maximum number of offending queries to save by SQL/Protect.</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td><code>edb_stat_level_tx</code> (EPAS only) — Allows continuing on errors instead of requiring a transaction abort.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>edb_wait_states.directory</code> (EPAS only) — Stores the EDB wait states logs in this directory.</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td><code>edb_wait_states.retainment_period</code> (EPAS only) — Deletes EDB wait state log files after retention period.</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td><code>edb_wait_states.sampling_interval</code> (EPAS only) — The interval between two EDB wait state sampling cycles.</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td><code>edbldr.empty_csv_field</code> (EPAS only) — Specifies how EDB*Loader handles empty strings.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>effective_cache_size</code> — Sets the planner's assumption about the size of the disk cache.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>effective_io_concurrency</code> — Number of simultaneous requests that can be handled efficiently by the disk subsystem.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_bitmapscan</code> — Enables the planner's use of bitmap-scan plans.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_hashagg</code> — Enables the planner's use of hashed aggregation plans.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_hashjoin</code> — Enables the planner's use of hash join plans.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_hints</code> (EPAS only) — Enables optimizer hints in SQL statements.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_indexonlyscan</code> — Enables the planner's use of index-only-scan plans.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_indexscan</code> — Enables the planner's use of index-scan plans.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_material</code> — Enables the planner's use of materialization.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_mergejoin</code> — Enables the planner's use of merge join plans.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_nestloop</code> — Enables the planner's use of nested-loop join plans.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_seqscan</code> — Enables the planner's use of sequential-scan plans.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_sort</code> — Enables the planner's use of explicit sort steps.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>enable_tidscan</code> — Enables the planner's use of TID scan plans.</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>escape_string_warning</code> — Warns about backslash escapes in ordinary string literals.</td>
<td>Session</td>
<td>Immediately</td>
<td>EPAS account</td>
</tr>
<tr>
<td><code>event_source</code> — Sets the application name used to identify PostgreSQL messages in the event log.</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>exit_on_error</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>external_pid_file</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>extra_float_digits</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>from_collapse_limit</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>fsync</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>full_page_writes</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>geqo</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>geqo_effort</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>geqo_generations</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>geqo_pool_size</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>geqo_seed</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>geqo_selection_bias</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>geqo_threshold</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>gin_fuzzy_search_limit</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>hba_file</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>hot_standby</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>hot_standby_feedback</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>huge_pages</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>icu_short_form</td>
<td>Database</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>ident_file</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>---------</td>
<td>-----------</td>
<td>--------------------------</td>
</tr>
<tr>
<td><code>ignore_checksum_failure</code></td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td><code>ignore_system_indexes</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>index_advisor.enabled</code></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>integer_datetimes</code></td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td><code>intervalStyle</code></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>join_collapse_limit</code></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>krb_caseins_users</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>krb_server_keyfile</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>lc_collate</code></td>
<td>Database</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td><code>lc_ctype</code></td>
<td>Database</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td><code>lc_messages</code></td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td><code>lc_monetary</code></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>lc_numeric</code></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>lc_time</code></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>listen_addresses</code></td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>local_preload_libraries</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>lock_timeout</code></td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td><code>lo_compat_privileges</code></td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>log_autovacuum_min_duration</code></td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Scope</td>
<td>In effect</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>log_checkpoints</td>
<td>Logs each checkpoint.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td>log_connections</td>
<td>Logs each successful connection. (Can also be set with PGOPTIONS at session start.)</td>
<td>Cluster</td>
<td>Session Reload Immediately</td>
</tr>
<tr>
<td>log_destination</td>
<td>Sets the destination for server log output.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td>log_directory</td>
<td>Sets the destination directory for log files.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td>log_disconnections</td>
<td>Logs end of a session, including duration. (Can also be set with PGOPTIONS at session start.)</td>
<td>Cluster</td>
<td>Session Reload Immediately</td>
</tr>
<tr>
<td>log_duration</td>
<td>Logs the duration of each completed SQL statement.</td>
<td>Session</td>
<td>Immediately</td>
</tr>
<tr>
<td>log_error_verbosity</td>
<td>Sets the verbosity of logged messages.</td>
<td>Session</td>
<td>Immediately</td>
</tr>
<tr>
<td>log_executor_stats</td>
<td>Writes executor performance statistics to the server log.</td>
<td>Session</td>
<td>Immediately</td>
</tr>
<tr>
<td>log_file_mode</td>
<td>Sets the file permissions for log files.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td>log_filename</td>
<td>Sets the file name pattern for log files.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td>log_hostname</td>
<td>Logs the host name in the connection logs.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td>log_line_prefix</td>
<td>Controls information prefixed to each log line.</td>
<td>Cluster</td>
<td>Reload</td>
</tr>
<tr>
<td>log_lock_waits</td>
<td>Logs long lock waits.</td>
<td>Session</td>
<td>Immediately</td>
</tr>
<tr>
<td>log_min_duration_statement</td>
<td>Sets the minimum execution time above which statements are logged.</td>
<td>Session</td>
<td>Immediately</td>
</tr>
<tr>
<td>log_min_error_statement</td>
<td>Causes all statements generating error at or above this level to be logged.</td>
<td>Session</td>
<td>Immediately</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>log_min_messages</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>log_parser_stats</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>log_planner_stats</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>log_rotation_age</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>log_rotation_size</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>log_statement</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>log_statement_stats</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>log_temp_files</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>log_timezone</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>log_truncate_on_rotation</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>logging_collector</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>maintenance_work_mem</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>max_connections</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_files_per_process</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_function_args</td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>max_identifier_length</td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------</td>
<td>-------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>max_index_keys</td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>max_locks_per_transaction</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_pred_locks_per_transaction</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_prepared_transactions</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_replication_slots</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_stack_depth</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>max_standby_archive_delay</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_standby_streaming_delay</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_wal_senders</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_wal_size</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>max_worker_processes</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>min_wal_size</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>nls_length_semantics</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>odbc_lib_path</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-----------</td>
<td>--------</td>
</tr>
<tr>
<td>optimizer_mode (EPAS only)</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>oracle_home (EPAS only)</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>password_encryption</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>pg_prewarm.autoprewarm (EPAS only)</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>pg_prewarm.autoprewarm_interval (EPAS only)</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>port</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>post_auth_delay</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>pre_auth_delay</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>qreplace_function (EPAS only)</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>query_rewrite_enabled (EPAS only)</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>query_rewrite_integrity (EPAS only)</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>quote_all_identifiers</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>random_page_cost</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>restart_after_crash</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>search_path</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>segment_size</td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>seq_page_cost</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>server_encoding</td>
<td>Database</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>server_version</td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>server_version_num</td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Usage</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------</td>
<td>----------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>session_preload_libraries</td>
<td>Session</td>
<td>Immediately but only at connection start</td>
<td>Super user</td>
</tr>
<tr>
<td>session_replication_role</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>shared_buffers</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>shared_preload_libraries</td>
<td>Cluster</td>
<td>Restart</td>
<td>EP AS account</td>
</tr>
<tr>
<td>sql_inheritance</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>ssl</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>ssl_ca_file</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>ssl_cert_file</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>ssl_ciphers</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>ssl_crl_file</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>ssl_ecdh_curve</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>ssl_key_file</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>ssl_prefer_server_ciphers</td>
<td>Cluster</td>
<td>Reload</td>
<td>EP AS account</td>
</tr>
<tr>
<td>ssl_renegotiation_limit</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>standard_conforming_strings</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>statement_timeout</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------</td>
<td>--------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>stats_temp_directory</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>superuser_reserved_connections</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>synchronize_seqscans</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>synchronous_commit</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>synchronous_standby_names</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>syslog_facility</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>syslog_ident</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>tcp_keepalives_count</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>tcp_keepalives_idle</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>tcp_keepalives_interval</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>temp_buffers</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>temp_file_limit</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>temp_tablespaces</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>timed_statistics</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>timezone</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>timezone_abbreviations</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>trace_hints</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>trace_notify</td>
<td>Session</td>
<td>Immediately</td>
<td>EPAS account</td>
</tr>
<tr>
<td>trace_recovery_messages</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>trace_sort</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>track_activities</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
</tbody>
</table>

stats_temp_directory — Writes temporary statistics files to the specified directory.

superuser_reserved_connections — Sets the number of connection slots reserved for superusers.

synchronize_seqscans — Enables synchronized sequential scans.

synchronous_commit — Sets immediate fsync at commit.

synchronous_standby_names — Lists names of potential synchronous standbys.

syslog_facility — Sets the syslog "facility" to be used when syslog enabled.

syslog_ident — Sets the program name used to identify PostgreSQL messages in syslog.

tcp_keepalives_count — Maximum number of TCP keepalive retransmits.

tcp_keepalives_idle — Time between issuing TCP keepalives.

tcp_keepalives_interval — Time between TCP keepalive retransmits.

temp_buffers — Sets the maximum number of temporary buffers used by each session.

temp_file_limit — Limits the total size of all temporary files used by each session.

temp_tablespaces — Sets the tablespace(s) to use for temporary tables and sort files.

timed_statistics (EPAS only) — Enables the recording of timed statistics.
	timezone — Sets the time zone for displaying and interpreting time stamps.

timezone_abbreviations — Selects a file of time zone abbreviations.

trace_hints (EPAS only) — Emits debug info about hints being honored.

trace_notify — Generates debugging output for LISTEN and NOTIFY.

trace_recovery_messages — Enables logging of recovery-related debugging information.

trace_sort — Emits information about resource usage in sorting.

track_activities — Collects information about executing commands.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Scope</th>
<th>In effect</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>track_activity_query_size</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>track_counts</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>track_functions</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>track_io_timing</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>transaction_deferrable</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>transaction_isolation</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>transaction_read_only</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>transform_null_equals</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>unix_socket_directories</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>unix_socket_group</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>unix_socket_permissions</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS account</td>
</tr>
<tr>
<td>update_process_title</td>
<td>Session</td>
<td>Immediately</td>
<td>Super user</td>
</tr>
<tr>
<td>utl_encode.uudecode_redwood</td>
<td>Session</td>
<td>Immediately</td>
<td>User (EPAS only)</td>
</tr>
<tr>
<td>utl_file.umask</td>
<td>Session</td>
<td>Immediately</td>
<td>User (EPAS only)</td>
</tr>
<tr>
<td>vacuum_cost_delay</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>vacuum_cost_limit</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>vacuum_cost_page_dirty</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>vacuum_cost_page_hit</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>vacuum_cost_page_miss</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>vacuum_defer_cleanup_age</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS account</td>
</tr>
<tr>
<td>vacuum_freeze_min_age</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>vacuum_freeze_table_age</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>Parameter</td>
<td>Scope</td>
<td>In effect</td>
<td>Used by</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>----------------</td>
<td>-------------</td>
<td>------------------</td>
</tr>
<tr>
<td>vacuum_multixact_freeze_min_age</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>vacuum_multixact_freeze_table_age</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>wal_block_size</td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>wal_buffers</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS</td>
</tr>
<tr>
<td>wal_keep_segments</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td>wal_level</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS</td>
</tr>
<tr>
<td>wal_log_hints</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS</td>
</tr>
<tr>
<td>wal_receiver_status_interval</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td>wal_receiver_timeout</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td>wal_segment_size</td>
<td>Cluster</td>
<td>Preset</td>
<td>n/a</td>
</tr>
<tr>
<td>wal_sender_timeout</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td>wal_sync_method</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td>wal_writer_delay</td>
<td>Cluster</td>
<td>Reload</td>
<td>EPAS</td>
</tr>
<tr>
<td>work_mem</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>xloginsert_locks</td>
<td>Cluster</td>
<td>Restart</td>
<td>EPAS</td>
</tr>
<tr>
<td>xmlbinary</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
<tr>
<td>xmloption</td>
<td>Session</td>
<td>Immediately</td>
<td>User</td>
</tr>
</tbody>
</table>
### 14.2.5 Audit log files

You can generate the audit log file in CSV or XML format. The format is determined by the `edb_audit` configuration parameter.

The information in the audit log is based on the logging performed by PostgreSQL, as described in "Using CSV-Format Log Output" under "Error Reporting and Logging" in the PostgreSQL core documentation.

#### Overview of the CSV audit log format

The following table lists the fields in the order they appear in the CSV audit log format. The table contains the following information:

- **Field** — Name of the field as shown in the sample table definition in the PostgreSQL documentation.
- **XML element/attribute** — For the XML format, name of the XML element and its attribute (if used), referencing the value.
- **Data type** — Data type of the field as given by the PostgreSQL sample table definition.
- **Description** — Description of the field.

The fields that don’t have any values for logging appear as consecutive commas (,,) in the CSV format.

<table>
<thead>
<tr>
<th>Field</th>
<th>XML element/attribute</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_time</td>
<td>event/log_time</td>
<td>timestamp with time zone</td>
<td>Log date/time of the statement.</td>
</tr>
<tr>
<td>user_name</td>
<td>event/user</td>
<td>text</td>
<td>Database user who executed the statement.</td>
</tr>
<tr>
<td>database_name</td>
<td>event/database</td>
<td>text</td>
<td>Database in which the statement was executed.</td>
</tr>
<tr>
<td>process_id</td>
<td>event/process_id</td>
<td>integer</td>
<td>Operating system process ID in which the statement was executed.</td>
</tr>
<tr>
<td>connection_from</td>
<td>event/remote_host</td>
<td>text</td>
<td>Host and port location from where the statement was executed.</td>
</tr>
<tr>
<td>session_id</td>
<td>event/session_id</td>
<td>text</td>
<td>Session ID in which the statement was executed.</td>
</tr>
<tr>
<td>session_line_num</td>
<td>event/session_line_num</td>
<td>bigint</td>
<td>Order of the statement within the session.</td>
</tr>
<tr>
<td>process_status</td>
<td>event/process_status</td>
<td>text</td>
<td>Processing status.</td>
</tr>
<tr>
<td>session_start_time</td>
<td>event/session_start_time</td>
<td>timestamp with time zone</td>
<td>Date/time when the session was started.</td>
</tr>
<tr>
<td>virtual_transaction_id</td>
<td>event/virtual_transaction_id</td>
<td>text</td>
<td>Virtual transaction ID of the statement.</td>
</tr>
<tr>
<td>transaction_id</td>
<td>event/transaction_id</td>
<td>bigint</td>
<td>Regular transaction ID of the statement.</td>
</tr>
<tr>
<td>error_severity</td>
<td>error_severity</td>
<td>text</td>
<td>Statement severity. Values are <code>AUDIT</code> for audited statements and <code>ERROR</code> for any resulting error messages.</td>
</tr>
<tr>
<td>sql_state_code</td>
<td>event/sql_state_code</td>
<td>text</td>
<td>SQL state code returned for the statement. The <code>sql_state_code</code> isn’t logged when its value is 00000 for XML log format.</td>
</tr>
<tr>
<td>message</td>
<td>message</td>
<td>text</td>
<td>The SQL statement that was attempted for execution.</td>
</tr>
<tr>
<td>detail</td>
<td>detail</td>
<td>text</td>
<td>Error message detail.</td>
</tr>
<tr>
<td>hint</td>
<td>hint</td>
<td>text</td>
<td>Hint for error.</td>
</tr>
<tr>
<td>internal_query</td>
<td>internal_query</td>
<td>text</td>
<td>Internal query that led to the error, if any.</td>
</tr>
<tr>
<td>internal_query_pos</td>
<td>internal_query_pos</td>
<td>integer</td>
<td>Character count of the error position therein.</td>
</tr>
<tr>
<td>context</td>
<td>context</td>
<td>text</td>
<td>Error context.</td>
</tr>
<tr>
<td>query</td>
<td>query</td>
<td>text</td>
<td>User query that led to the error. For errors only.</td>
</tr>
<tr>
<td>query_pos</td>
<td>query_pos</td>
<td>integer</td>
<td>Character count of the error position therein. For errors only.</td>
</tr>
<tr>
<td>location</td>
<td>location</td>
<td>text</td>
<td>Location of the error in the source code. The location field is populated if <code>log_error_verbosity</code> is set to verbose.</td>
</tr>
<tr>
<td>application_name</td>
<td>event/application_name</td>
<td>text</td>
<td>Name of the application from which the statement was executed, for example, <code>psql.bin</code>.</td>
</tr>
<tr>
<td>backend_type</td>
<td>event/backend_type</td>
<td>text</td>
<td>The <code>backend_type</code> corresponds to what <code>pg_stat_activity.backend_type</code> shows and is added as a column to the CSV log.</td>
</tr>
<tr>
<td>leader_pid</td>
<td>event/leader_pid</td>
<td>integer</td>
<td>Process ID of leader for active parallel workers.</td>
</tr>
</tbody>
</table>
The following examples are generated in the CSV and XML formats.

The non-default audit settings in the `postgresql.conf` file are as follows:

```
logging_collector = 'on'
edb_audit = 'csv'
edb_audit_connect = 'all'
edb_audit_disconnect = 'all'
edb_audit_statement = 'ddl, dml, select, error'
edb_audit_tag = 'edbaudit'
```

The `edb_audit` parameter is changed to `xml` when generating the XML format.

The following is the audited session:

```
$ psql edb
enterprisedb
Password
Password
for
for
user
enterprisedb:
psql.bin
(14.0.0)
Type
Type
"help" for help.
edb=# CREATE SCHEMA edb;
CREATE SCHEMA edb
edb=# SET search_path TO edb;
SET
edb=# CREATE TABLE dept

edb(#     deptno 
     NUMBER 
     (2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
edb(#     dname 
     VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
edb(#     loc 
     VARCHAR2(13))
edb(#);
CREATE TABLE edb=# INSERT INTO dept VALUES (10, 'ACCOUNTING', 'NEW YORK');
INSERT 0 1
edb=# UPDATE department SET loc = 'BOSTON' WHERE deptno = 10;
ERROR:  relation "department" does not exist
LINE 1: UPDATE department SET loc = 'BOSTON' WHERE deptno = 10;
^
edb=# UPDATE dept SET loc = 'BOSTON' WHERE deptno = 10;
UPDATE 1
edb=# SELECT * FROM dept;

deptno | dname | loc
--------|-------|--------
10 | ACCOUNTING | BOSTON
(1 row)
edb=# \q
```

CSV audit log file

The following is the CSV format of the audit log file. (Each audit log entry was split and displays across multiple lines. A blank line was inserted between the audit log entries for visual clarity.)

```
2022-12-14 12:19:01.035 UTC,"enterprisedb","edb",9298,"{local}"
6399b235.244a.1,"authentication",2022-12-14 12:19:01 UTC,4/19,0,AUDIT,00000,"connection authorized: user=enterprisedb database=edb",
```

Field | XML element/attribute | Data type | Description
--- | --- | --- | ---
query_id | event/query_id | long | Identifier of this backend's most recent query.
command_tag | event/command_tag | text | SQL command of the statement.
audit_tag | event/audit_tag | text | Value specified by the audit_tag parameter in the configuration file.
type | event/type | text | Determines the audit event_type to identify messages in the log.
XML audit log file

The following is the XML format of the audit log file. (The output was formatted for visual clarity.)

```xml
<event user="amul" database="edb" process_id="110405">
    <remote_host>[local]" session_id="63e1f4cf.1af45" session_line_num="1" process_status="idle" session_start_time="2023-02-07 12:20:56.920 IST" log_time="2023-02-07 12:20:56.920 IST" virtual_transaction_id="4/22" type="create" command_tag="CREATE ROLE" application_name="psql" backend_type="client backend" query_id="0">
        <error severity="AUDIT"/>
    </event>
```
**Message:** statement: create user edb superuser;

**Event:**
- user="edb" database="postgres" process_id="118424"
  remote_host=[local] session_id="63e1f4d4.1af58"
  session_line_num="1" process_status="idle"
  session_start_time="2023-02-07 12:21:00.561 IST"
  log_time="2023-02-07 12:21:00 IST" virtual_transaction_id="4/25"
  type="set" command_tag="SET" application_name="pg_regress"
  backend_type="client backend"
  query_id="0"

**Error Severity:** AUDIT</error_severity>

**Message:** statement: SET client_min_messages = warning</message>

**Event:**
- user="edb" database="postgres" process_id="118426"
  remote_host=[local] session_id="63e1f4d6.1af5a"
  session_line_num="1" process_status="idle"
  session_start_time="2023-02-07 12:21:00.568 IST"
  log_time="2023-02-07 12:21:00 IST" virtual_transaction_id="4/26"
  type="drop" command_tag="DROP DATABASE" application_name="pg_regress"
  backend_type="client backend"
  query_id="0"

**Error Severity:** AUDIT</error_severity>

**Message:** statement: DROP DATABASE IF EXISTS &quot;regression&quot;;</message>

**Event:**
- user="edb" database="postgres" process_id="118426"
  remote_host=[local] session_id="63e1f4d6.1af5a"
  session_line_num="1" process_status="idle"
  session_start_time="2023-02-07 12:21:00.983 IST"
  log_time="2023-02-07 12:21:00 IST" virtual_transaction_id="4/30"
  type="alter" command_tag="ALTER DATABASE" application_name="pg_regress"
  backend_type="client backend"
  query_id="0"

**Error Severity:** AUDIT</error_severity>

**Message:** statement: CREATE DATABASE &quot;regression&quot;

**Event:**
- user="edb" database="postgres" process_id="118426"
  remote_host=[local] session_id="63e1f4d6.1af5a"
  session_line_num="1" process_status="idle"
  session_start_time="2023-02-07 12:21:01.152 IST"
  log_time="2023-02-07 12:21:01 IST" virtual_transaction_id="4/33"
  type="set" command_tag="SET" application_name="pg_regress/test_setup"
  backend_type="client backend"
  query_id="0"

**Error Severity:** AUDIT</error_severity>

**Message:** statement: SET synchronous_commit = on;

**Event:**
- user="enterprise$db" database="edb" process_id="5942"
  remote_host="[local]"
  session_id="Sec7ac4d.1735" session_line_num="8"
  process_status="idle"
  session_start_time="2023-02-07 16:11:17 IST" log_time="2023-02-07 16:12:45 IST" virtual_transaction_id="4/27"
  type="select"
  command_tag="SELECT audit_tags" application_name="psql"
  backend_type="parallel worker" leader_pid="5940"
  query_id="0"

**Error Severity:** AUDIT</error_severity>

**Message:** statement: SELECT * FROM dept;

**Event:**
14.2.6 EDB*Loader control file parameters

Use the following parameters when building the EDB*Loader control file.

OPTIONS param=value

Use the OPTIONS clause to specify param=value pairs that represent an EDB*Loader directive. If you specify a parameter in the OPTIONS clause and on the command line when edbldr is invoked, the command line setting takes effect.

Specify one or more of the following parameter/value pairs:

- **DIRECT** (FALSE | TRUE)
  
  With DIRECT set to TRUE, EDB*Loader performs a direct path load instead of a conventional path load. The default value of DIRECT is FALSE.
  
  Don't set DIRECT=true when loading the data into a replicated table. If you're using EDB*Loader to load data into a replicated table and set DIRECT=true, indexes might omit rows that are in a table or can potentially contain references to rows that were deleted. Direct inserts to load data into replicated tables aren't supported.

  For information on direct path loads see, [Direct path load](#).

- **ERRORS=error_count**
  
  error_count specifies the number of errors permitted before aborting the EDB*Loader session. The default is 50.

- **FREEZE** (FALSE | TRUE)
  
  Set FREEZE to TRUE to copy the data with the rows frozen. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wraparound. For more information about frozen tuples, see the PostgreSQL core documentation.

  You must specify a data-loading type of TRUNCATE in the control file when using the FREEZE option. FREEZE isn't supported for direct loading.

  By default, FREEZE is FALSE.

- **PARALLEL** (FALSE | TRUE)
  
  Set PARALLEL to TRUE to indicate that this EDB*Loader session is one of a number of concurrent EDB*Loader sessions participating in a parallel direct path load. The default value of
When PARALLEL is TRUE, you must also set the DIRECT parameter to TRUE. For more information about parallel direct path loads, see Parallel direct path load.

- **ROWS=n**
  
  n specifies the number of rows that EDB*Loader commits before loading the next set of n rows.
  
  If EDB*Loader encounters an invalid row during a load in which the ROWS parameter is specified, those rows committed prior to encountering the error remain in the destination table.

- **SKIP=skip_count**
  
  skip_count specifies the number of records at the beginning of the input data file to skip before loading begins. The default is 0.

- **SKIP_INDEX_MAINTENANCE={ FALSE | TRUE }**
  
  If SKIP_INDEX_MAINTENANCE is TRUE, index maintenance isn’t performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of SKIP_INDEX_MAINTENANCE is FALSE.

Note

During a parallel direct path load, target table indexes aren’t updated. They are marked as invalid after the load is complete.

You can use the REINDEX command to rebuild an index. For more information about the REINDEX command, see the PostgreSQL core documentation.

**charset**

Use the CHARACTERSET clause to identify the character set encoding of data_file, where charset is the character set name. This clause is required if the data file encoding differs from the control file encoding. The control file encoding must always be in the encoding of the client where edbldr is invoked.

Examples of charset settings are UTF8, SQL_ASCII, and SJIS.

For more information about client-to-database character-set conversion, see the PostgreSQL core documentation.

**data_file**

File containing the data to load into target_table. Each record in the data file corresponds to a row to insert into target_table.

If you don't include an extension in the file name, EDB*Loader assumes the file has an extension of .dat, for example, mydatafile.dat.

!! Note If you specify the DATA parameter on the command line when invoking edbldr, the file given with the command line DATA parameter is used instead.

If you omit the INFILE clause and the command line DATA parameter, then the data file name is assumed to be the same as the control file name but with the extension .dat.

**stdin**

Specify stdin (all lowercase letters) if you want to use standard input to pipe the data to load directly to EDB*Loader. This technique is useful for data sources generating a large number of records to load.

**bad_file**

A file that receives data_file records that can't load due to errors. The bad file is generated for collecting rejected or bad records.

For EDB Postgres Advanced Server version 12 and later, a bad file is generated only if there are any bad or rejected records. However, if an existing bad file has the same name and location, and no bad records are generated after invoking a new version of edbldr, the existing bad file remains intact.

If you don't include an extension in the file name, EDB*Loader assumes the file has an extension of .bad, for example, mybadfile.bad.

!! Note If you specify the BAD parameter on the command line when invoking edbldr, the file given with the command line BAD parameter is used instead.
discard_file

File that receives input data records that aren't loaded into any table. This input data records are discarded because none of the selection criteria are met for tables with the WHEN clause and there are no tables without a WHEN clause. All records meet the selection criteria of a table without a WHEN clause.

If you don't include an extension with the file name, EDB*Loader assumes the file has an extension of .dsc, for example, mydiscardfile.dsc.

!!! Note If you specify the DISCARD parameter on the command line when invoking edbldr, the file given with the command line DISCARD parameter is used instead.

max_discard_recs

Maximum number of discarded records that the input data records can encounter before terminating the EDB*Loader session. You can use either keyword DISCARDMAX or DISCARDS preceding the integer value specified by max_discard_recs.

For example, if max_discard_recs is 0, then the EDB*Loader session is terminated when a first discarded record is encountered. If max_discard_recs is 1, then the EDB*Loader session is terminated when a second discarded record is encountered.

When the EDB*Loader session is terminated due to exceeding max_discard_recs, prior input data records that were loaded into the database are retained. They aren't rolled back.

INSERT, APPEND, REPLACE, TRUNCATE

Specifies how to load data into the target tables. Specifying one of INSERT, APPEND, REPLACE, or TRUNCATE establishes the default action for all tables, overriding the default of INSERT.

- **INSERT**
  
  Data is loaded into an empty table. EDB*Loader throws an exception and doesn't load any data if the table isn't initially empty.

  **Note**

  If the table contains rows, you must use the TRUNCATE command to empty the table before invoking EDB*Loader. EDB*Loader throws an exception if you use the DELETE command to empty the table instead of the TRUNCATE command. Oracle SQL*Loader allows you to empty the table by using either the DELETE or TRUNCATE command.

- **APPEND**
  
  Data is added to any existing rows in the table. The table also can be empty initially.

- **REPLACE**
  
  The REPLACE keyword and TRUNCATE keywords are functionally identical. The table is truncated by EDB*Loader before loading the new data.

  **Note**

  Delete triggers on the table aren't fired as a result of the REPLACE operation.

- **TRUNCATE**
  
  The table is truncated by EDB*Loader before loading the new data. Delete triggers on the table aren't fired as a result of the TRUNCATE operation.

PRESERVE BLANKS

The PRESERVE BLANKS option works only with the OPTIONALLY ENCLOSED BY clause. It retains leading and trailing whitespaces for both delimited and predetermined size fields.

In case of NO PRESERVE BLANKS, if the fields are delimited, then only leading whitespaces are omitted. If any trailing whitespaces are present, they are left intact. In the case of predetermined-sized fields with NO PRESERVE BLANKS, the trailing whitespaces are omitted. Any leading whitespaces are left intact.

!!! Note If you don't explicitly provide PRESERVE BLANKS or NO PRESERVE BLANKS, then the behavior defaults to NO PRESERVE BLANKS. This option doesn't work for ideographic whitespaces.

target_table
Name of the table into which to load data. The table name can be schema-qualified (for example, `enterprisedb.emp`). The specified target can't be a view.

**field_condition**

Conditional clause taking the following form:

```
[ ( { (start:end) | column_name } { = | != | <> } 'val' [ ) ] ]
```

This conditional clause is used for the **WHEN** clause, which is part of the **INTO TABLE** `target_table` clause. It's also used for the **NULLIF** clause, which is part of the field definition denoted as `field_def` in the syntax diagram.

`start` and `end` are positive integers specifying the column positions in `data_file` that mark the beginning and end of a field to compare with the constant `val`. The first character in each record begins with a `start` value of 1.

`column_name` specifies the name assigned to a field definition of the data file as defined by `field_def` in the syntax diagram.

Using `{start:end}` or `column_name` defines the portion of the record in `data_file` to compare with the value specified by `val` to evaluate as either true or false.

All characters used in the `field_condition` text (particularly in the `val` string) must be valid in the database encoding. For performing data conversion, EDB*Loader first converts the characters in `val` string to the database encoding and then to the data file encoding.

In the **WHEN** `field_condition [ AND field_condition ]` clause, if all such conditions evaluate to **TRUE** for a given record, then EDB*Loader attempts to insert that record into `target_table`. If the insert operation fails, the record is written to `bad_file`.

Suppose, for a given record, that none of the **WHEN** `field_condition` clauses evaluate to **TRUE** for all **INTO TABLE** clauses. The record is written to `discard_file` if a discard file was specified for the EDB*Loader session.

See the description of the **NULLIF** clause in this parameters list for the effect of `field_condition` on this clause.

**termstring**

String of one or more characters that separates each field in `data_file`. The characters can be single byte or multibyte. However, they must be valid in the database encoding. Two consecutive appearances of `termstring` with no intervening character results in the corresponding column being set to null.

**enclstring**

String of one or more characters used to enclose a field value in `data_file`. The characters can be single byte or multibyte. However, they must be valid in the database encoding. Use `enclstring` on fields where `termstring` appears as part of the data.

**delimstring**

String of one or more characters that separates each record in `data_file`. The characters can be single byte or multibyte. However, they must be valid in the database encoding. Two consecutive appearances of `delimstring` with no intervening character results in no corresponding row being loaded into the table. You must also terminate the last record (that is, the end of the data file) with the `delimstring` characters. Otherwise, the final record isn't loaded into the table.

!!! Note The **RECORDS DELIMITED BY 'delimstring'** clause isn't compatible with Oracle databases.

**TRAILING NULLCOLS**

If you specify **TRAILING NULLCOLS**, then the columns in the column list for which there's no data in `data_file` for a given record are set to null when the row is inserted. This option applies only to one or more consecutive columns at the end of the column list.

If fields are omitted at the end of a record and you don't specify **TRAILING NULLCOLS**, EDB*Loader assumes the record contains formatting errors and writes it to the bad file.

**column_name**
Name of a column in target_table into which to insert a field value defined by field_def. If the field definition includes the FILLER or BOUNDFILLER clause, then column_name isn't required as the name of a column in the table. It can be any identifier name since the FILLER and BOUNDFILLER clauses prevent loading the field data into a table column.

CONSTANT val

Specifies a constant that’s type-compatible with the column data type to which it’s assigned in a field definition. You can use single or double quotes around val, if val contains white space, then you must use quotation marks around it.

The use of the CONSTANT clause determines the value to assign to a column in each inserted row. No other clause can appear in the same field definition.

If you use the TERMINATED BY clause to delimit the fields in data_file, there must be no delimited field in data_file corresponding to any field definition with a CONSTANT clause. In other words, EDB*Loader assumes there’s no field in data_file for any field definition with a CONSTANT clause.

FILLER

Specifies not to load the data in the field defined by the field definition into the associated column if the identifier of the field definition is an actual column name in the table. In this case, the column is set to null. Use of the FILLER or BOUNDFILLER clause is the only circumstance in which you don’t have to identify the field definition with an actual column name.

Unlike the BOUNDFILLER clause, you can’t reference an identifier defined with the FILLER clause in a SQL expression. See the discussion of the expr parameter.

BOUNDFILLER

Specifies not to load the data in the field defined by the field definition into the associated column if the identifier of the field definition is an actual column name in the table. In this case, the column is set to null. Use of the FILLER or BOUNDFILLER clause is the only circumstance in which you don’t have to identify the field definition with an actual column name.

Unlike the FILLER clause, a SQL expression can reference an identifier defined with the BOUNDFILLER clause. See the discussion of the expr parameter.

POSITION (start:end)

Defines the location of the field in a record in a fixed-width field data file. start and end are positive integers. The first character in the record has a start value of 1.

Field type that describes the format of the data field in data_file.

!! Note. Specifying a field type is optional for descriptive purposes and has no effect on whether EDB*Loader successfully inserts the data in the field into the table column. Successful loading depends on the compatibility of the column data type and the field value. For example, a column with data type NUMBER(7,2) successfully accepts a field containing 2008. However, if the field contains a value such as 26XX, the insertion fails, and the record is written to bad_file.

ZONED data is not human readable. ZONED data is stored in an internal format where each digit is encoded in a separate nibble/nybble/4-bit field. In each ZONED value, the last byte contains a single digit in the high-order 4 bits and the sign in the low-order 4 bits.

length

Specifies the length of the value to load into the associated column.

If you specify the POSITION (start:end) clause with a fieldtype(length) clause, then the ending position of the field is overridden by the specified length value. That is, the length of the value to load into the column is determined by the length value beginning at the start position and not by the end position of the POSITION (start:end) clause. Thus, the value to load into the column might be shorter than the field defined by POSITION (start:end). Or, it might go beyond the end position, depending on the specified length size.

If you specify the FIELDS TERMINATED BY ’termstring’ clause as part of the INTO TABLE clause, and a field definition contains the fieldtype(length) clause, then a record is accepted. However, the specified length values must be greater than or equal to the field lengths as determined by the termstring characters enclosing all such fields of the record. If the specified length value is less than a field length as determined by the enclosing termstring characters for any such field, then the record is rejected.

If you don’t specify the FIELDS TERMINATED BY ’termstring’ clause, and you don’t include the POSITION (start:end) clause with a field containing the fieldtype(length)
clause, then the starting position of this field begins with the next character following the ending position of the preceding field. The ending position of the preceding field is either:

- The end of its length value if the preceding field contains the fieldtype(length) clause
- Its end parameter if the field contains the POSITION (start; end) clause without the fieldtype(length) clause

precision

Use precision to specify the length of the ZONED value.

If the precision value specified for ZONED conflicts with the length calculated by the server based on information provided with the POSITION clause, EDB*Loader uses the value specified for precision.

cscale

Specifies the number of digits to the right of the decimal point in a ZONED value.

datemask

Specifies the ordering and abbreviation of the day, month, and year components of a date field.

!!! Note If you specify the DATE or TIMESTAMP field type with a SQL expression for the column, then you must specify datemask after DATE or TIMESTAMP and before the SQL expression. See the discussion of the expr parameter.

When using the TIMESTAMP field datatype, if you specify time_stamp timestamp "yyyymmddhh24miss". The datemask is converted to the SQL expression. However, in the case of time_stamp timestamp "select to_timestamp(:time_stamp, 'yyyymmddhh24miss')", EDB*Loader can't differentiate between datemask and the SQL expression. It treats the third field (SQL expression in the example) as datemask and prepares the SQL expression, which isn't valid. Where:

- first field specifies the column name.
- second field specifies the datatype.
- third field specifies the datemask.

If you want to provide an SQL expression, then a workaround is to specify the datemask and SQL expression using the TO_CHAR function as:

time_stamp timestamp "yyyymmddhh24miss" "to_char(to_timestamp(:time_stamp, 'yyyymmddhh24miss'), 'yyyymmddhh24miss')"

NULLIF field_condition [ AND field_condition ] ...

See the description of field_condition in this parameter list for the syntax of field_condition.

If all field conditions evaluate to TRUE, then the column identified by column_name in the field definition is set to null. If any field condition evaluates to FALSE, then the column is set to the appropriate value as normally occurs according to the field definition.

PRESERVE BLANKS

The PRESERVE BLANKS option works only with the OPTIONALLY ENCLOSED BY clause and retains leading and trailing whitespaces for both delimited and predetermined size fields.

In case of NO PRESERVE BLANKS, if the fields are delimited, then only leading whitespaces are omitted. If any trailing whitespaces are present, they are left intact. In the case of predetermined-sized fields with NO PRESERVE BLANKS, the trailing whitespaces are omitted, and any leading whitespaces are left intact.

!!! Note If you don't provide PRESERVE BLANKS or NO PRESERVE BLANKS, then the behavior defaults to NO PRESERVE BLANKS. This option doesn't work for ideographic whitespaces.

expr

A SQL expression returning a scalar value that's type-compatible with the column data type to which it's assigned in a field definition. Use double quotes around expr. expr can contain a reference to any column in the field list except for fields with the FILLER clause. Prefix the column name using a colon ( : ).

expr can also consist of a SQL SELECT statement. If you use a SELECT statement:

- Enclose the SELECT statement in parentheses, that is, (SELECT ...).
The select list must consist of one expression following the `SELECT` keyword.

The result set must not return more than one row. If no rows are returned, then the returned value of the resulting expression is null.

The following is the syntax for a `SELECT` statement:

```
(SELECT <expr> [ FROM <table_list> [ WHERE <condition> ]])
```

**Note**

Omitting the `FROM table_list` clause isn't compatible with Oracle databases. If you don't need to specify any tables, using the `FROM DUAL` clause is compatible with Oracle databases.

### 14.2.7 System catalogs

System catalogs store the resource group information used by EDB Resource Manager.

#### `edb_all_resource_groups`

The following table lists the information available in the `edb_all_resource_groups` catalog.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>group_name</code></td>
<td><code>name</code></td>
<td>The name of the resource group.</td>
</tr>
<tr>
<td><code>active_processes</code></td>
<td><code>integer</code></td>
<td>Number of currently active processes in the resource group.</td>
</tr>
<tr>
<td><code>cpu_rate_limit</code></td>
<td><code>float8</code></td>
<td>Maximum CPU rate limit for the resource group.</td>
</tr>
<tr>
<td><code>per_process_cpu_rate_limit</code></td>
<td><code>float8</code></td>
<td>Maximum CPU rate limit per currently active process in the resource group.</td>
</tr>
<tr>
<td><code>dirty_rate_limit</code></td>
<td><code>float8</code></td>
<td>Maximum dirty rate limit for a resource group.</td>
</tr>
<tr>
<td><code>per_process_dirty_rate_limit</code></td>
<td><code>float8</code></td>
<td>Maximum dirty rate limit per currently active process in the resource group.</td>
</tr>
</tbody>
</table>

#### `edb_resource_group`

The following table lists the information available in the `edb_resource_group` catalog.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rgrpname</code></td>
<td><code>name</code></td>
<td>The name of the resource group.</td>
</tr>
<tr>
<td><code>rgroupcpratelimit</code></td>
<td><code>float8</code></td>
<td>Maximum CPU rate limit for a resource group.</td>
</tr>
<tr>
<td><code>rgroupdirtyratelimit</code></td>
<td><code>float8</code></td>
<td>Maximum dirty rate limit for a resource group.</td>
</tr>
</tbody>
</table>

### 14.3 Oracle compatibility reference

This reference information applies to organizations migrating their Oracle applications to use EDB Postgres Advanced Server.

#### 14.3.1 `dblink_ora` functions and procedures

The `dblink_ora` utility supports functions and procedures that enable you to issue arbitrary queries to a remote Oracle server.

##### 14.3.1.1 `dblink_ora_connect()`

The `dblink_ora_connect()` function establishes a connection to an Oracle database with user-specified connection information. The function comes in two forms.
The signature of the first form is:

```sql
dblink_ora_connect(<conn_name>, <server_name>, <service_name>, <user_name>, <password>, <port>, <asDBA>)
```

Where:
- **conn_name** specifies the name of the link.
- **server_name** specifies the name of the host.
- **service_name** specifies the name of the service.
- **user_name** specifies the name used to connect to the server.
- **password** specifies the password associated with the user name.
- **port** specifies the port number.
- **asDBA** is True if you want to request SYSDBA privileges on the Oracle server. This parameter is optional. If omitted, the value is FALSE.

The first form of `dblink_ora_connect()` returns a TEXT value.

The signature of the second form of the `dblink_ora_connect()` function is:

```sql
dblink_ora_connect(<foreign_server_name>, <asDBA>)
```

Where:
- **foreign_server_name** specifies the name of a foreign server.
- **asDBA** is True if you want to request SYSDBA privileges on the Oracle server. This parameter is optional. If omitted, the value is FALSE.

The second form of the `dblink_ora_connect()` function allows you to use the connection properties of a predefined foreign server when connecting to the server.

Before invoking the second form of the `dblink_ora_connect()` function, use the `CREATE SERVER` command to store the connection properties for the link to a system table. When you call the `dblink_ora_connect()` function, substitute the server name specified in the `CREATE SERVER` command for the name of the link.

The second form of `dblink_ora_connect()` returns a TEXT value.

### 14.3.1.2 dblink_ora_status()

The `dblink_ora_status()` function returns the database connection status. The signature is:

```sql
dblink_ora_status(<conn_name>)
```

Where:
- **conn_name** specifies the name of the link.

If the specified connection is active, the function returns a TEXT value of OK.

### 14.3.1.3 dblink_ora_disconnect()

The `dblink_ora_disconnect()` function closes a database connection. The signature is:

```sql
dblink_ora_disconnect(<conn_name>)
```

Where:
- **conn_name** specifies the name of the link.

The function returns a TEXT value.
14.3.1.4  dblink_ora_record()

The `dblink_ora_record()` function retrieves information from a database. The signature is:

```
dblink_ora_record(<conn_name>, <query_text>)
```

Where:

- `conn_name` specifies the name of the link.
- `query_text` specifies the text of the SQL `SELECT` statement invoked on the Oracle server.

The function returns a `SETOF record`.

14.3.1.5  dblink_ora_call()

The `dblink_ora_call()` function executes a non-`SELECT` statement on an Oracle database and returns a result set. The signature is:

```
dblink_ora_call(<conn_name>, <command>, <iterations>)
```

Where:

- `conn_name` specifies the name of the link.
- `command` specifies the text of the SQL statement invoked on the Oracle server.
- `iterations` specifies the number of times the statement is executed.

The function returns a `SETOF record`.

14.3.1.6  dblink_ora_exec()

The `dblink_ora_exec()` procedure executes a DML or DDL statement in the remote database. The signature is:

```
dblink_ora_exec(<conn_name>, <command>)
```

Where:

- `conn_name` specifies the name of the link.
- `command` specifies the text of the `INSERT`, `UPDATE`, or `DELETE` SQL statement invoked on the Oracle server.

The function returns a `VOID`.

14.3.1.7  dblink_ora_copy()

The `dblink_ora_copy()` function copies an Oracle table to an EDB table. The `dblink_ora_copy()` function returns a `BIGINT` value that represents the number of rows copied. The signature is:

```
dblink_ora_copy(<conn_name>, <command>, <schema_name>, <table_name>, <truncate>, <count>)
```

Where:

- `conn_name` specifies the name of the link.
- `command` specifies the text of the SQL `SELECT` statement invoked on the Oracle server.
- `schema_name` specifies the name of the target schema.
- `table_name` specifies the name of the target table.
14.3.2 Partitioning commands compatible with Oracle Databases

EDB Postgres Advanced Server supports using table partitioning syntaxes that are compatible with Oracle databases.

### 14.3.2.1 CREATE TABLE...PARTITION BY

Use the `PARTITION BY` clause of the `CREATE TABLE` command to create a partitioned table with data distributed among one or more partitions and subpartitions. The syntax can take several forms.

#### List partitioning syntax

Use this form to create a list-partitioned table:

```sql
CREATE TABLE [ schema. ]<table_name>
    <table_definition>
    PARTITION BY LIST(<column>) [ AUTOMATIC ]
    [ SUBPARTITION BY (|RANGE|LIST|HASH) (<column>[, <column>]...)
      [ |SUBPARTITION TEMPLATE |
        <(subpartition_template_description)>]]
    [ (list_partition_definition|...)
      [ (list_partition_definition)|...]
    [ ENABLE ROW MOVEMENT];
```

Where `list_partition_definition` is:

```sql
PARTITION
  [<(partition_name)>]
  VALUES (value[, value]...)
  [TABLESPACE <tablespace_name>]
  [[(subpartition[, ...])]
```

Where `subpartition_template_description` is:

```sql
  SUBPARTITION [<(subpartition_name)>]
  VALUES (value[, value]...)
  [TABLESPACE <tablespace_name>],
  ...
```

#### Range partitioning syntax

Use this form to create a range-partitioned table:

```sql
CREATE TABLE [ schema. ]<table_name>
    <table_definition>
    PARTITION BY RANGE(<column>[, <column> ]...)
      [ INTERVAL (<constant>| <expression>)]
      [ SUBPARTITION BY (|RANGE|LIST|HASH) (<column>[, <column>]...)
        [ |SUBPARTITION TEMPLATE |
          <(range_partition_definition)>]]
      [ (range_partition_definition|...)
        [ |ENABLE ROW MOVEMENT|]
```

Where `range_partition_definition` is:

```sql
```
PARTITION
[<partition_name>]
VALUES LESS THAN (<value>[, 
<value>...])
[TABLESPACE <tablespace_name>]
[[<subpartition>, ...]]

Where ‘subpartition_template_description’ is:

SUBPARTITION [<subpartition_name>]
VALUES LESS THAN (<value>[, 
<value>...])
[TABLESPACE <tablespace_name>], ...

Hash partitioning syntax

Use this form to create a hash-partitioned table:

```
CREATE TABLE [ schema. ]<table_name>
  <table_definition>
  PARTITION BY HASH(<column>[, <column> ]...)
  [SUBPARTITION BY (RANGE|LIST|HASH) (<column>[, <column> ]...)]
  [SUBPARTITION TEMPLATE (<subpartition_template_description>)]
  [SUBPARTITIONS <num> [STORE IN ( <tablespace_name> [, <tablespace_name>]... )]]
  [PARTITIONS <num> [STORE IN ( <tablespace_name> [, <tablespace_name>]... )]]
  [<hash_partition_definition>[: <hash_partition_definition>]]
  [ENABLE ROW MOVEMENT];
```

Where ‘hash_partition_definition’ is:

```
PARTITION [<partition_name>]
[TABLESPACE <tablespace_name>]
[[<subpartition>, ...]]
```

Where ‘subpartition_template_description’ is:

```
SUBPARTITION [<subpartition_name>]
[TABLESPACE <tablespace_name>], ...
```

Subpartitioning syntax

```
subpartition can be one of the following:

[<list_subpartition> | <range_subpartition> | 
[hash_subpartition>]
```

Where ‘list_subpartition’ is:

```
SUBPARTITION [<subpartition_name>]
VALUES (<value>[, 
<value>...])
[TABLESPACE <tablespace_name>]
```

Where ‘range_subpartition’ is:

```
SUBPARTITION [<subpartition_name>]
VALUES LESS THAN (<value>[, 
<value>...])
[TABLESPACE <tablespace_name>]
```
Description

The `CREATE TABLE... PARTITION BY` command creates a table with one or more partitions. Each partition can have one or more subpartitions. There's no upper limit to the number of defined partitions. However, if you include the `PARTITION BY` clause, you must specify at least one partitioning rule. The resulting table is owned by the user that creates it.

**PARTITION BY LIST clause**

Use the `PARTITION BY LIST` clause to divide a table into partitions based on the values entered in a specified column. Each partitioning rule must specify at least one literal value. However, there's no upper limit placed on the number of values you can specify. Include a rule that specifies a matching value of `DEFAULT` to direct any unqualified rows to the given partition.

For more information about using the `DEFAULT` keyword, see Handling stray values in a LIST or RANGE partitioned table.

**AUTOMATIC clause**

Use the `AUTOMATIC` clause to specify for the table to use automatic list partitioning. By specifying the `AUTOMATIC` clause, the database creates partitions when new data is inserted into a table that doesn't correspond to any value declared for an existing partition.

For more information about `AUTOMATIC LIST PARTITIONING`, see Automatic list partitioning.

**PARTITION BY RANGE clause**

Use the `PARTITION BY RANGE` clause to specify boundary rules by which to create partitions. Each partitioning rule must contain at least one column of a data type that has two operators (that is, a greater-than or equal-to operator, and a less-than operator). Range boundaries are evaluated against a `LESS THAN` clause and are non-inclusive. For example, a date boundary of January 1, 2013 includes only date values that fall on or before December 31, 2012.

Specify range partition rules in ascending order. `INSERT` commands that store rows with values that exceed the top boundary of a range-partitioned table fail unless the partitioning rules include a boundary rule that specifies a value of `MAXVALUE`. If you don't include a `MAXVALUE` partitioning rule, any row that exceeds the maximum limit specified by the boundary rules results in an error.

For more information about using the `MAXVALUE` keyword, see Handling stray values in a LIST or RANGE partitioned table.

**INTERVAL clause**

Use the `INTERVAL` clause to specify an interval range-partitioned table. With an `INTERVAL` clause, the database extends the range partitioning to create partitions of a specified interval when data is inserted into a table that exceeds an existing range partition.

For more information about `INTERVAL RANGE PARTITION`, see Interval range partitioning.

**PARTITION BY HASH clause**

Use the `PARTITION BY HASH` clause to create a hash-partitioned table. In a `HASH` partitioned table, data is divided among equal-sized partitions based on the hash value of the column specified in the partitioning syntax. When specifying a `HASH` partition, choose a column or combination of columns that's as close to unique as possible. This technique helps to ensure that data is evenly distributed among the partitions. When selecting a partitioning column (or combination of columns), for best performance, select columns that you frequently search for exact matches.

**STORE IN clause**

Use the `STORE IN` clause to specify the tablespace list across which the autogenerated partitions or subpartitions are stored.

**Note**

If you're upgrading EDB Postgres Advanced Server, you must rebuild the hash-partitioned table on the upgraded version server.
**TABLESPACE** keyword

Use the **TABLESPACE** keyword to specify the name of a tablespace on which a partition or subpartition resides. If you don't specify a tablespace, the partition or subpartition resides in the default tablespace.

**SUBPARTITION BY** clause

If a table definition includes the **SUBPARTITION BY** clause, each partition in that table must have at least one subpartition. You can explicitly define each subpartition, or it can be system defined.

If the subpartition is system defined, the server-generated subpartition resides in the default tablespace, and the name of the subpartition is assigned by the server. The server creates:

- A **DEFAULT** subpartition if the **SUBPARTITION BY** clause specifies **LIST**.
- A **MAXVALUE** subpartition if the **SUBPARTITION BY** clause specifies **RANGE**.

The server generates a subpartition name that's a combination of the partition table name and a unique identifier. You can query the **ALL_TAB_SUBPARTITIONS** table to review a complete list of subpartition names.

**SUBPARTITION TEMPLATE**

Use **ENABLE ROW MOVEMENT** with a list, range, or hash partition table to support migrating tables with similar syntax from Oracle databases. However, it doesn't enable the actual row movement. This syntax isn't supported with the **ALTER TABLE** command.

**Parameters**

**table_name**

The name (optionally schema-qualified) of the table to create.

**table_definition**

The column names, data types, and constraint information as described in the PostgreSQL core documentation for the **CREATE TABLE** statement.

**partition_name**

The name of the partition to create. Partition names must be unique among all partitions and subpartitions and must follow the naming conventions for object identifiers.

**subpartition_name**

The name of the subpartition to create. Subpartition names must be unique among all partitions and subpartitions and must follow the naming conventions for object identifiers.

**num**

You can use the **PARTITIONS num** clause to specify the number of **HASH** partitions to create. Alternatively, you can use the partition definition to specify individual partitions and their tablespaces.

**Note** You can use either **PARTITIONS** or **PARTITION DEFINITION** when creating a table.

The **SUBPARTITIONS num** clause is supported only for **HASH** partitions. You can use it to specify the number of hash subpartitions to create. Alternatively, you can use the subpartition definition to specify individual subpartitions and their tablespaces. If you don't specify **SUBPARTITIONS** or **SUBPARTITION DEFINITION**, then the partition creates a subpartition based on the **SUBPARTITION TEMPLATE**.

**subpartition_template_description**

The list of subpartition details with subpartition name, subpartition bound, and tablespace name.

**column**

The name of a column on which the partitioning rules are based. Each row is stored in a partition that corresponds to the **value** of the specified columns.

**constant | expression**

The **constant** and **expression** specifies a **NUMERIC**, **DATE**, or **TIME** value.

**(value[, value]...)**
Use `value` to specify a quoted literal value or comma-delimited list of literal values by which table entries are grouped into partitions. Each partitioning rule must specify at least one value. However, there is no limit on the number of values specified in a rule. `value` can be `NULL`, `DEFAULT` if specifying a `LIST` partition, or `MAXVALUE` if specifying a `RANGE` partition.

When specifying rules for a list-partitioned table, include the `DEFAULT` keyword in the last partition rule to direct any unmatched rows to the given partition. If you don’t include a rule that includes a value of `DEFAULT`, any `INSERT` statement that attempts to add a row that doesn’t match the specified rules of at least one partition fails and returns an error.

When specifying rules for a list-partitioned table, include the `MAXVALUE` keyword in the last partition rule to direct any uncategorized rows to the given partition. If you don’t include a `MAXVALUE` partition, any `INSERT` statement that attempts to add a row where the partitioning key is greater than the highest value specified fails and returns an error.

### Example: PARTITION BY LIST

This example creates a partitioned table `sales` using the `PARTITION BY LIST` clause. The `sales` table stores information in three partitions: `europe`, `asia`, and `americas`.

```sql
CREATE TABLE sales (
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
) PARTITION BY LIST (country) (
  PARTITION europe VALUES ('FRANCE', 'ITALY'),
  PARTITION asia VALUES ('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES ('US', 'CANADA')
);
```

The resulting table is partitioned by the value specified in the `country` column:

```
edb# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
partition_name | high_value
---------------|-----------------------
EUROPE         | 'FRANCE', 'ITALY'
ASIA           | 'INDIA', 'PAKISTAN'
AMERICAS       | 'US', 'CANADA'
(3 rows)
```

- Rows with a value of `FRANCE` or `ITALY` in the `country` column are stored in the `europe` partition.
- Rows with a value of `INDIA` or `PAKISTAN` in the `country` column are stored in the `asia` partition.
- Rows with a value of `US` or `CANADA` in the `country` column are stored in the `americas` partition.

The server evaluates the following statement against the partitioning rules and stores the row in the `europe` partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', 650000);
```

### Example: AUTOMATIC LIST PARTITION

This example shows a `sales` table that uses an `AUTOMATIC` clause to create an automatic list partitioned table on the `sales_state` column. The database creates a partition and adds data to a table.

```sql
CREATE TABLE sales (
  dept_no number,
  part_no varchar2,
  sales_state varchar2(20),
  date date,
  amount number
) PARTITION BY LIST (sales_state);
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
PARTITION BY LIST(sales_state) AUTOMATIC
  
  PARTITION P_CAL VALUES('CALIFORNIA'),
  PARTITION P_FLO VALUES('FLORIDA')
);

Query the ALL_TAB_PARTITIONS view to see an existing partition that's successfully created:

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>P_CAL</td>
<td>'CALIFORNIA'</td>
</tr>
<tr>
<td>SALES</td>
<td>P_FLO</td>
<td>'FLORIDA'</td>
</tr>
</tbody>
</table>

(2 rows)

Insert data into a sales table that can't fit into an existing partition. For the regular list partitioned table, you get an error. However, automatic list partitioning creates and inserts the data into a new partition.

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>P_CAL</td>
<td>'INDIANA'</td>
</tr>
<tr>
<td>SALES</td>
<td>P_FLO</td>
<td>'OHIO'</td>
</tr>
</tbody>
</table>

(4 rows)

14.3.2.1.3 Example: PARTITION BY RANGE

This example creates a partitioned table sales using the PARTITION BY RANGE clause. The sales table stores information in four partitions: q1_2012, q2_2012, q3_2012 and q4_2012.

```sql
CREATE TABLE sales
  
  (  
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date      date,
    amount    number
  )
PARTITION BY RANGE(date)
  
  PARTITION q1_2012 VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012 VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012 VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012 VALUES LESS THAN('2013-Jan-01')
);
```

The resulting table is partitioned by the value specified in the date column:

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_2012</td>
<td>'01-APR-12 00:00:00'</td>
</tr>
</tbody>
</table>
The server evaluates the following statement against the partitioning rules and stores the row in the q3_2012 partition:

```sql
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');
```

### 14.3.2.1.4 Example: INTERVAL RANGE PARTITION

This example shows a `sales` table that's partitioned by interval on the `sold_month` column. The range partition is created to establish a transition point, and new partitions are created beyond that transition point. The database creates a new interval range partition and adds data to a table.

```sql
CREATE TABLE sales
(
  prod_id int,
  prod_quantity int,
  sold_month date
)
PARTITION BY RANGE (sold_month)
INTERVAL (NUMTOMININTERVAL(1, 'MONTH'))
(
  PARTITION p1
    VALUES LESS THAN ('15-JAN-2019'),
  PARTITION p2
    VALUES LESS THAN ('15-FEB-2019')
);
```

Query the `ALL_TAB_PARTITIONS` view before the database creates an interval range partition:

```sql
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
```

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>'15-JAN-19 00:00:00'</td>
</tr>
<tr>
<td>P2</td>
<td>'15-FEB-19 00:00:00'</td>
</tr>
</tbody>
</table>

(2 rows)

Insert data into a `sales` table that exceeds the high value of a range partition:

```sql
edb=# INSERT INTO sales VALUES (1,208,'18-MAY-2019');
```

Query the `ALL_TAB_PARTITIONS` view again after the insert. The data is successfully inserted, and a system-generated name of the interval range partition is created. The name varies for each session.

```sql
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
```

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>'15-JAN-19 00:00:00'</td>
</tr>
<tr>
<td>P2</td>
<td>'15-FEB-19 00:00:00'</td>
</tr>
<tr>
<td>SYS916348103</td>
<td>'15-MAY-19 00:00:00'</td>
</tr>
</tbody>
</table>

(3 rows)

### 14.3.2.1.5 Example: PARTITION BY HASH

This example creates a partitioned table `sales` using the `PARTITION BY HASH` clause. The `sales` table stores information in three partitions: p1, p2, and p3.

```sql
CREATE TABLE sales
```
The table returns an empty string for the hash partition value specified in the `part_no` column:

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

(3 rows)

Use the following command to view the hash value of the `part_no` column:

```sql
eddb=# SELECT partition_name, high_value from ALL_TAB_PARTITIONS;

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

(3 rows)
```

The server evaluates the hash value of the `part_no` column and distributes the rows into approximately equal partitions.

14.3.2.1.6 Example: PARTITION BY HASH...PARTITIONS num...

This example creates a hash-partitioned table `sales` using the `PARTITION BY HASH` clause. The partitioning column is `part_no`. The example specifies the number of partitions to create.

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
) PARTITION BY HASH (part_no) PARTITIONS 8;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The eight partitions are created and assigned system-generated names. The partitions are stored in the default tablespace of the table.

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0102</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0103</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0105</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0106</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0107</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0108</td>
</tr>
</tbody>
</table>

(8 rows)

Example: PARTITION BY HASH...PARTITIONS num...STORE IN

This example creates a hash-partitioned table named `sales`. The example specifies the number of partitions to create and the tablespaces in which the partition resides.

```
CREATE TABLE sales
(
  dept_no   number,
  part_no   varchar2,
  country   varchar2(20),
  date      date,
  amount    number
)
PARTITION BY HASH (part_no)
PARTITIONS 5
STORE IN (ts1, ts2, ts3);
```

The `STORE IN` clause evenly distributes the partitions across the tablespaces `ts1`, `ts2`, and `ts3`.

```
edb=# SELECT table_name, partition_name, tablespace_name
FROM ALL_TAB_PARTITIONS
WHERE table_name = 'SALES' ORDER BY 1,2;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>SYS0101</td>
<td>TS1</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0102</td>
<td>TS2</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0103</td>
<td>TS3</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0104</td>
<td>TS1</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0105</td>
<td>TS2</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0106</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0107</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0108</td>
<td></td>
</tr>
</tbody>
</table>

(5 rows)

Example: HASH/RANGE PARTITIONS num...

The `HASH` partition clause allows you to define a partitioning strategy. You can extend the `PARTITION BY HASH` clause to include `SUBPARTITION BY` either `[ RANGE | LIST | HASH ]` to create subpartitions in a `HASH` partitioned table.

This example creates a table `sales` that’s hash partitioned by `part_no` and subpartitioned using a range by `dept_no`. The example specifies the number of partitions when creating the table `sales`.

```
CREATE TABLE sales
(
  dept_no   number,
  part_no   varchar2,
  country   varchar2(20),
  date      date,
  amount    number
)
PARTITION BY HASH (part_no)
SUBPARTITION BY RANGE (dept_no)
PARTITIONS 5;
```

The five partitions are created with default subpartitions and assigned system-generated names:

```
edb=# SELECT table_name, partition_name, subpartition_name
FROM
```
### Example: LIST/HASH Subpartitions num...

This example shows the table `sales` that’s list-partitioned by `country`. It is subpartitioned using hash partitioning by the `dept_no` column. This example specifies the number of subpartitions when creating the table.

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
) PARTITION BY LIST (country) SUBPARTITION BY HASH (dept_no) SUBPARTITIONS 3
(
    PARTITION p1 VALUES('FRANCE', 'ITALY'),
    PARTITION p2 VALUES('INDIA', 'PAKISTAN'),
    PARTITION p3 VALUES('US', 'CANADA')
);
```

The three partitions `p1`, `p2`, and `p3` each contain three subpartitions with system-generated names:

```sql
EDB=# SELECT table_name, partition_name, subpartition_name
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

### Example: HASH/HASH Partitions num... Subpartitions num...

This example creates the `sales` table, hash partitioned by `part_no` and hash subpartitioned by `dept_no`:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
) PARTITION BY HASH (part_no) SUBPARTITION BY HASH (dept_no) SUBPARTITIONS 3
PARTITIONS 2;
```

The two partitions are created. Each partition includes three subpartitions with the system-generated name assigned to them.

```sql
EDB=# SELECT table_name, partition_name, subpartition_name
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```
Example: HASH/HASH SUBPARTITIONS num... STORE IN

This example creates a hash-partitioned table `sales`. This example specifies the number of partitions and subpartitions to create when creating a hash-partitioned table. It also specifies the tablespaces in which the subpartitions reside when creating a hash-partitioned table.

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY HASH (part_no) SUBPARTITION BY HASH (dept_no) SUBPARTITIONS 3 PARTITIONS 2 STORE IN (ts1, ts2);
```

The two partitions are created and assigned system-generated names. The partitions are stored in the default tablespace. Subpartitions are stored in tablespaces `ts1` and `ts2`.

```sql
edb=# SELECT table_name, partition_name, tablespace_name
FROM ALL_TAB_PARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

```
<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>SYS0101</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0105</td>
<td></td>
</tr>
</tbody>
</table>
(2 rows)
```

The `STORE IN` clause assigns the hash subpartitions to the tablespaces and stores them in the two named tablespaces `ts1` and `ts2`:

```sql
edb=# SELECT table_name, partition_name, subpartition_name, tablespace_name
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

```
<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>SYS0101</td>
<td>SYS0102</td>
<td>TS1</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0101</td>
<td>SYS0103</td>
<td>TS1</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0101</td>
<td>SYS0104</td>
<td>TS1</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0105</td>
<td>SYS0106</td>
<td>TS1</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0105</td>
<td>SYS0107</td>
<td>TS2</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS0105</td>
<td>SYS0108</td>
<td>TS1</td>
</tr>
</tbody>
</table>
(6 rows)
```

Example: HASH/HASH PARTITIONS num ... STORE IN SUBPARTITIONS num... STORE IN

This example creates the hash-partitioned table `sales`. It specifies the number of partitions and subpartitions to create and the tablespaces in which the partitions and subpartitions reside.

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY HASH (part_no) SUBPARTITION BY HASH (dept_no) SUBPARTITIONS 3 STORE IN (ts3) PARTITIONS 2 STORE IN (ts1, ts2);
```
The two partitions are created with system-generated names and stored in the default tablespace:

```
SALES    | SYS0101        |
SALES    | SYS0105        |
```

(2 rows)

Each partition includes three subpartitions. The **STORE IN** clause stores the subpartitions in tablespaces `ts1` and `ts2`:

```
SALES    | SYS0101        | SYS0102           | TS1
SALES    | SYS0101        | SYS0103           | TS2
SALES    | SYS0101        | SYS0104           | TS1
SALES    | SYS0105        | SYS0106           | TS1
SALES    | SYS0105        | SYS0107           | TS2
SALES    | SYS0105        | SYS0108           | TS1
```

(6 rows)

**Note**

If you specify the **STORE IN** clause for partitions and subpartitions, then the subpartitions are stored in the tablespaces defined in the **PARTITIONS...STORE IN** clause. The **SUBPARTITIONS...STORE IN** clause is ignored.

### Example: RANGE/HASH SUBPARTITIONS num...

This example creates a range-partitioned table `sales`, which is first partitioned by the transaction date. Two range partitions are created and then hash subpartitioned using the value of the `country` column.

```
CREATE TABLE sales
(
  dept_no   number,
  part_no   varchar2,
  country   varchar2(20),
  date      date,
  amount    number
)
PARTITION BY RANGE (date) SUBPARTITION BY HASH (country) SUBPARTITIONS 2
(
  PARTITION p1 VALUES LESS THAN ('2012-Apr-01') (SUBPARTITION q1_europe),
  PARTITION p2 VALUES LESS THAN ('2012-Jul-01')
);
```

This statement creates a table with two partitions. The subpartition explicitly named `q1_europe` is created for partition `p1`. Because subpartitions aren't named for partition `p2`, the subpartitions are created based on the subpartition number and are assigned a system-generated name.

```
SALES    | P1           | Q1_EUROPE
SALES    | P2           | SYS0101
SALES    | P2           | SYS0102
```

(3 rows)

### Example: RANGE/HASH SUBPARTITIONS num... IN PARTITION DESCRIPTION
This example creates a range-partitioned table `sales`. The table is first partitioned by the transaction date. Two range partitions are created and then hash subpartitioned using the value of the `country` column.

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY RANGE (date) SUBPARTITION BY HASH (country) SUBPARTITIONS 2
(
    PARTITION p1 VALUES LESS THAN ('2012-Apr-01') SUBPARTITIONS 3,
    PARTITION p2 VALUES LESS THAN ('2012-Jul-01')
);
```

The partition `p1` explicitly defines the subpartition count in the partition description. By default, two subpartitions are created for partition `p2`. Since you don't name subpartitions, system-generated names are assigned.

```
edb=# SELECT table_name, partition_name, subpartition_name
    FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>P1</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>P1</td>
<td>SYS0102</td>
</tr>
<tr>
<td>SALES</td>
<td>P1</td>
<td>SYS0103</td>
</tr>
<tr>
<td>SALES</td>
<td>P2</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>P2</td>
<td>SYS0105</td>
</tr>
<tr>
<td>(5 rows)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: LIST/HASH SUBPARTITIONS num STORE IN... IN PARTITION DESCRIPTION

This example creates a list-partitioned table `sales` with two list partitions. Partition `p1` consists of three subpartitions, and partition `p2` consists of two subpartitions. Since you don't name subpartitions, system-generated names are assigned.

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST (country) SUBPARTITION BY HASH (part_no) SUBPARTITIONS 3
STORE IN (ts1)
(
    PARTITION p1 VALUES ('FRANCE', 'ITALY'),
    PARTITION p2 VALUES ('INDIA', 'PAKISTAN') SUBPARTITIONS 2 STORE IN (ts2)
);
```

The partition `p2` explicitly defines the subpartition count in the partition description. Based on the definition, two subpartitions are created and stored in the tablespace named `ts2`. The subpartitions for partition `p1` are stored in the tablespace named `ts1`.

```
edb=# SELECT table_name, partition_name, subpartition_name, tablespace_name
    FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>P1</td>
<td>SYS0101</td>
<td>TS1</td>
</tr>
<tr>
<td>SALES</td>
<td>P1</td>
<td>SYS0102</td>
<td>TS1</td>
</tr>
<tr>
<td>SALES</td>
<td>P1</td>
<td>SYS0103</td>
<td>TS1</td>
</tr>
<tr>
<td>SALES</td>
<td>P2</td>
<td>SYS0104</td>
<td>TS2</td>
</tr>
<tr>
<td>SALES</td>
<td>P2</td>
<td>SYS0105</td>
<td>TS2</td>
</tr>
<tr>
<td>(5 rows)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: LIST/HASH STORE IN...TABLESPACES

This example creates a list-partitioned table `sales`. Partition `p1` consists of three subpartitions stored explicitly in the tablespace `ts2`.

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
) PARTITION BY LIST (country) SUBPARTITION BY HASH (part_no) SUBPARTITIONS 3
STORE IN (ts1)
(
    PARTITION p1 VALUES ('FRANCE', 'ITALY') TABLESPACE ts2
);
```

The `SELECT` statement shows partition `p1`, consisting of three subpartitions stored in the tablespace `ts2`:

```sql
edb=# SELECT table_name, partition_name, subpartition_name, tablespace_name
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

```
table_name | partition_name | subpartition_name | tablespace_name
------------+----------------+-------------------+-----------------
SALES      | P1             | SYS0101           | TS2
SALES      | P1             | SYS0102           | TS2
SALES      | P1             | SYS0103           | TS2
(3 rows)
```

This command adds a partition `p2` to the `sales` table. Five subpartitions are created and distributed across the tablespace listed by the `STORE IN` clause.

```sql
ALTER TABLE sales ADD PARTITION p2 VALUES ('US', 'CANADA') SUBPARTITIONS 5
STORE IN (ts1);
```

A query of the `ALL_TAB_PARTITIONS` view shows the `sales` table with a partition named `p2`. The partition has five subpartitions. The `STORE IN` clause distributes the subpartitions across a tablespace named `ts1`.

```sql
edb=# SELECT table_name, partition_name, subpartition_name, tablespace_name
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' and partition_name = 'P2' ORDER BY 1,2;
```

```
table_name | partition_name | subpartition_name | tablespace_name
------------+----------------+-------------------+-----------------
SALES      | P2             | SYS0105           | TS1
SALES      | P2             | SYS0106           | TS1
SALES      | P2             | SYS0107           | TS1
SALES      | P2             | SYS0108           | TS1
SALES      | P2             | SYS0109           | TS1
(5 rows)
```

14.3.2.1.7 Example: PARTITION BY RANGE, SUBPARTITION BY LIST

This example creates a partitioned table `sales` that's first partitioned by the transaction date. The range partitions `q1_2012`, `q2_2012`, `q3_2012` and `q4_2012` are then list subpartitioned using the value of the `country` column.

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
) PARTITION BY RANGE(date)
```
This statement creates a table with four partitions. Each partition has three subpartitions.

```sql
SUBPARTITION BY LIST(country)
{
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01')
    
    SUBPARTITION q1_europe VALUES ('FRANCE', 'ITALY'),
    SUBPARTITION q1_asia VALUES ('INDIA', 'PAKISTAN'),
    SUBPARTITION q1_americas VALUES ('US', 'CANADA'),
  }
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01')
    
    SUBPARTITION q2_europe VALUES ('FRANCE', 'ITALY'),
    SUBPARTITION q2_asia VALUES ('INDIA', 'PAKISTAN'),
    SUBPARTITION q2_americas VALUES ('US', 'CANADA'),
  }
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01')
    
    SUBPARTITION q3_europe VALUES ('FRANCE', 'ITALY'),
    SUBPARTITION q3_asia VALUES ('INDIA', 'PAKISTAN'),
    SUBPARTITION q3_americas VALUES ('US', 'CANADA'),
  }
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
    
    SUBPARTITION q4_europe VALUES ('FRANCE', 'ITALY'),
    SUBPARTITION q4_asia VALUES ('INDIA', 'PAKISTAN'),
    SUBPARTITION q4_americas VALUES ('US', 'CANADA'),
}

(12 rows)
```

When a row is added to this table, the value in the `date` column is compared to the values specified in the range-partitioning rules. The server selects the partition for the row to reside in. The value in the `country` column is then compared to the values specified in the list subpartitioning rules. When the server locates a match for the value, the row is stored in the corresponding subpartition.

Any row added to the table is stored in a subpartition, so the partitions contain no data.

The server evaluates the following statement against the partitioning and subpartitioning rules. It stores the row in the `q3_europe` partition.

```sql
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650800');
```

14.3.2.1.8 Example: CREATING UNIQUE INDEX on PARTITION TABLE
For unique-index partitioned tables, you can use the range, list, or hash-partitioning method. To create a unique ROWID for the partitioned table, you must set the parameter `default_with_rowids = true`.

This example creates a partitioned table `sales` using the `PARTITION BY LIST` clause. The `sales` table stores information in three partitions: `europe`, `asia`, and `americas`.

```sql
CREATE TABLE sales (
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY LIST (country) {
  PARTITION europe VALUES ('FRANCE', 'ITALY'),
  PARTITION asia VALUES ('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES ('US', 'CANADA')
} WITH (ROWIDS = TRUE);
```

The table is partitioned by the value specified in the `country` column:

```sql
edb# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
```

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
<tr>
<td>AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
</tbody>
</table>

(3 rows)

Insert values into the `sales` table:

```sql
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '19-Aug-2012', '650000');
INSERT INTO sales VALUES (10, '9519b', 'ITALY', '20-Aug-2012', '700000');
```

This query shows that a unique index is created for the `sales` table:

```sql
edb# SELECT oid, relname, relhasindex FROM pg_class WHERE relname='sales';
```

```
oid | relname | relhasindex
-----|---------|-------------
16557 | sales   | f
```

(1 row)

Querying the contents of the `sales_europe` confirms that the unique index is created for the table:

```sql
edb# SELECT oid, relname, relhasindex FROM pg_class WHERE relname='sales_europe';
```

```
oid | relname | relhasindex
-----|---------|-------------
16561 | sales_europe | t
```

(1 row)

```sql
edb# SELECT * FROM pg_index WHERE indrelid='16561';
```

```
indexrelid | indrelid | inndatts | indnkeyatts | indisunique | indisprimary | indisexclusion | indimmediate | indisclustered | indisvalid | indcheckxmin | indisready | indislive | indisreplident | indkey | indcollation | indclass | indoption | indexprs | indpred
----------|----------|----------|-------------|-------------|--------------|----------------|--------------|----------------|------------|-------------|-----------|----------|----------------|--------|--------------|----------|----------|----------|----------
16565 | 16561 | 1 | 1 | t | f | f | t | f | t | 1 |
1 | t | f | f | 1 | 0 | 3124 | 0 | |
```

(1 row)
14.3.2.1.9 Example: CREATING SUBPARTITION TEMPLATE

These examples show how to create subpartitions in a partitioned table using a subpartition template.

This example creates a table `sales` list partitioned by `country` and subpartitioned using list by `date`. The `sales` table uses a subpartition template and displays the subpartition and tablespace name.

```
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST (country)
SUBPARTITION BY LIST (date)
SUBPARTITION TEMPLATE
(
    SUBPARTITION europe VALUES('2021-Jan-01') TABLESPACE tsl,
    SUBPARTITION asia VALUES('2021-Apr-01') TABLESPACE ts2,
    SUBPARTITION americas VALUES('2021-Jul-01') TABLESPACE ts3
)
)
PARTITION q1_2021 VALUES('2021-Jul-01')
);
```

The `SELECT` statement shows partition `q1_2021` consisting of three subpartitions stored in tablespaces `ts1`, `ts2`, and `ts3`:

```
EDB Postgres Advanced Server (EPAS)
EDB Postgres Advanced Server (EPAS)
Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
708
```

```
SELECT partition_name, subpartition_name, high_value, tablespace_name
FROM DBA_TAB_SUBPARTITIONS
WHERE table_name = 'SALES'
ORDER BY 1, 2;
```

```
partition_name | subpartition_name | high_value          | tablespace_name
----------------+-------------------+---------------------+------------------
Q1_2021         | Q1_2021_AMERICAS  | '01-JUL-21 00:00:00' | TS3              
Q1_2021         | Q1_2021_ASIA      | '01-APR-21 00:00:00' | TS2              
Q1_2021         | Q1_2021_EUROPE    | '01-JAN-21 00:00:00' | TS1              
(3 rows)
```

Example: Creating a subpartition template for LIST/RANGE partitioned table

This example creates a table `sales` list partitioned by `country` and subpartitioned using range partitioning by the `date` column:

```
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST (country)
SUBPARTITION BY RANGE (date)
SUBPARTITION TEMPLATE
(
    SUBPARTITION europe VALUES LESS THAN('2021-Jan-01') TABLESPACE tsl,
    SUBPARTITION asia VALUES LESS THAN('2021-Apr-01') TABLESPACE ts2,
    SUBPARTITION americas VALUES LESS THAN('2021-Jul-01') TABLESPACE ts3
)
)
PARTITION q1_2021 VALUES ('2021-Jul-01')
);
```

The `sales` table creates a partition named `q1_2021` that includes three subpartitions stored in tablespaces `ts1`, `ts2`, and `ts3`. 
Example: Creating a subpartition template for LIST/HASH partitioned table

This example creates a list-partitioned table `sales` that’s first partitioned by `country` and then hash subpartitioned using the value of the `dept_no` column:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST (country)
   SUBPARTITION BY HASH (dept_no)
SUBPARTITION TEMPLATE
(
    SUBPARTITION europe TABLESPACE ts1,
    SUBPARTITION asia TABLESPACE ts2,
    SUBPARTITION americas TABLESPACE ts3
) )

PARTITION q1_2021 VALUES ('2021-Jul-01')
);
```

The `sales` table creates a `q1_2021` partition that includes three subpartitions stored in tablespaces `ts1`, `ts2`, and `ts3`.

Example: Creating a subpartition template for INTERVAL/HASH partitioned table

This example creates a `sales` table, interval partitioned using monthly intervals on the `date` column and hash subpartitioned using the value of the `dept_no` column:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY RANGE (date) INTERVAL (NUMTOYMINTERVAL (1, 'MONTH'))
   SUBPARTITION BY HASH (dept_no)
SUBPARTITION TEMPLATE
(
    SUBPARTITION europe TABLESPACE ts1,
```

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The `sales` table creates a partition `q2_2021` consisting of three subpartitions stored in tablespaces `ts1`, `ts2`, and `ts3`:

<table>
<thead>
<tr>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_AMERICAS</td>
<td></td>
<td>TS3</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_ASIA</td>
<td></td>
<td>TS2</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_EUROPE</td>
<td></td>
<td>TS1</td>
</tr>
</tbody>
</table>

(3 rows)

Insert values into the `sales` table:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '05-Jul-2021', '650000');
```

The `SELECT` statement shows a system-generated name of partitions and subpartitions stored in tablespaces `ts1`, `ts2`, and `ts3`:

```
<table>
<thead>
<tr>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_AMERICAS</td>
<td></td>
<td>TS3</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_ASIA</td>
<td></td>
<td>TS2</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_EUROPE</td>
<td></td>
<td>TS1</td>
</tr>
<tr>
<td>SYS368340105</td>
<td>SYS368340106</td>
<td></td>
<td>TS1</td>
</tr>
<tr>
<td>SYS368340105</td>
<td>SYS368340107</td>
<td></td>
<td>TS2</td>
</tr>
<tr>
<td>SYS368340105</td>
<td>SYS368340108</td>
<td></td>
<td>TS3</td>
</tr>
</tbody>
</table>
```

(6 rows)

14.3.2.2 ALTER TABLE...ADD PARTITION

Use the `ALTER TABLE...ADD PARTITION` command to add a partition to an existing partitioned table. The syntax is:

```
ALTER TABLE <table_name> ADD PARTITION <partition_definition>;
```

Where `partition_definition` is:

```
{(list_partition | range_partition)}
```

and `list_partition` is:

```
PARTITION 
<partition_name>
VALUES ((value[, 
(value) ...])
[TABLESPACE 
<tablespace_name>]
[{{subpartition}, ...}]
```

and `range_partition` is:

```
PARTITION 
<partition_name>
VALUES LESS THAN ((value[, 
(value) ...])
[TABLESPACE 
<tablespace_name>]
[{{subpartition}, ...}]
```

Where `subpartition` is:
The ALTER TABLE... ADD PARTITION command adds a partition to an existing partitioned table. There's no upper limit to the number of defined partitions in a partitioned table.

New partitions must be of the same type (LIST, RANGE, or HASH) as existing partitions. The new partition rules must reference the same column specified in the partitioning rules that define the existing partitions.

You can use the ALTER TABLE... ADD PARTITION statement to add a partition to a table with a DEFAULT rule. However, there can't be conflicting values between existing rows in the table and the values of the partition to add.

You can't use the ALTER TABLE... ADD PARTITION statement to add a partition to a table with a MAXVALUE rule.

You can alternatively use the ALTER TABLE... SPLIT PARTITION statement to split an existing partition, effectively increasing the number of partitions in a table.

Specify RANGE partitions in ascending order. You can't add a new partition that precedes existing partitions in a RANGE partitioned table.

Include the TABLESPACE clause to specify the tablespace in which the new partition resides. If you don't specify a tablespace, the partition resides in the default tablespace.

Use the STORE IN clause to specify the tablespace list across which the autogenerated subpartitions are stored.

If the table is indexed, the index is created on the new partition.

To use the ALTER TABLE... ADD PARTITION command, you must be the table owner or have superuser or administrative privileges.

Parameters

```sql
<table_name>

The name (optionally schema-qualified) of the partitioned table.

```partition_name

The name of the partition to create. Partition names must be unique among all partitions and subpartitions and must follow the naming conventions for object identifiers.

```subpartition_name

The name of the subpartition to create. Subpartition names must be unique among all partitions and subpartitions and must follow the naming conventions for object identifiers.

```(value[, value]...}

Description

The ALTER TABLE... ADD PARTITION command adds a partition to an existing partitioned table. There's no upper limit to the number of defined partitions in a partitioned table.

New partitions must be of the same type (LIST, RANGE, or HASH) as existing partitions. The new partition rules must reference the same column specified in the partitioning rules that define the existing partitions.

You can use the ALTER TABLE... ADD PARTITION statement to add a partition to a table with a DEFAULT rule. However, there can't be conflicting values between existing rows in the table and the values of the partition to add.

You can't use the ALTER TABLE... ADD PARTITION statement to add a partition to a table with a MAXVALUE rule.

You can alternatively use the ALTER TABLE... SPLIT PARTITION statement to split an existing partition, effectively increasing the number of partitions in a table.

Specify RANGE partitions in ascending order. You can't add a new partition that precedes existing partitions in a RANGE partitioned table.

Include the TABLESPACE clause to specify the tablespace in which the new partition resides. If you don't specify a tablespace, the partition resides in the default tablespace.

Use the STORE IN clause to specify the tablespace list across which the autogenerated subpartitions are stored.

If the table is indexed, the index is created on the new partition.

To use the ALTER TABLE... ADD PARTITION command, you must be the table owner or have superuser or administrative privileges.
Use `value` to specify a quoted literal value or comma-delimited list of literal values by which rows are distributed into partitions. Each partitioning rule must specify at least one `value`. There's no limit on the number of values specified in a rule. `value` can also be `NULL`, `DEFAULT` if specifying a `LIST` partition, or `MAXVALUE` if specifying a `RANGE` partition.

For information about creating a `DEFAULT` or `MAXVALUE` partition, see Handling stray values in a `LIST` or `RANGE` partitioned table.

The `SUBPARTITIONS num` clause is supported only for `HASH` and can be used to specify the number of hash subpartitions. Alternatively, you can use the subpartition definition to specify individual subpartitions and their tablespaces. If you don't specify `SUBPARTITIONS` or `SUBPARTITION DEFINITION`, then the partition creates a subpartition based on the `SUBPARTITION TEMPLATE`.

The name of the tablespace in which a partition or subpartition resides.

### 14.3.2.2.1 Example: Adding a partition to a `LIST` partitioned table

This example adds a partition to the list-partitioned `sales` table. The table was created using the command:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
) PARTITION BY LIST(country)
(
    PARTITION europe VALUES('FRANCE', 'ITALY'),
    PARTITION asia VALUES('INDIA', 'PAKISTAN'),
    PARTITION americas VALUES('US', 'CANADA')
);
```

The table contains the three partitions `americas`, `asia`, and `europe`:

```
<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
<tr>
<td>AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
</tbody>
</table>
(3 rows)
```

This command adds a partition named `east_asia` to the `sales` table:

```sql
ALTER TABLE sales ADD PARTITION east_asia VALUES ('CHINA', 'KOREA');
```

After the command is invoked, the table includes the `east_asia` partition:

```
<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
<tr>
<td>AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
<tr>
<td>EAST_ASIA</td>
<td>'CHINA', 'KOREA'</td>
</tr>
</tbody>
</table>
(4 rows)
```

### 14.3.2.2.2 Example - Adding a partition to a `RANGE` partitioned table

This example adds a partition to a range-partitioned table named `sales`:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
) PARTITION BY RANGE(country)
(
    PARTITION europe VALUES LESS THAN ('US'),
    PARTITION asia VALUES LESS THAN ('INDIA'),
    PARTITION americas VALUES LESS THAN ('US'),
    PARTITION usa VALUES LESS THAN ('US'),
    PARTITION world VALUES LESS THAN ('US')
);
```

The tablespace name

The name of the tablespace in which a partition or subpartition resides.

```
<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
<tr>
<td>AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
<tr>
<td>EAST_ASIA</td>
<td>'CHINA', 'KOREA'</td>
</tr>
</tbody>
</table>
(4 rows)
```

```
<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
<tr>
<td>AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
<tr>
<td>EAST_ASIA</td>
<td>'CHINA', 'KOREA'</td>
</tr>
</tbody>
</table>
(4 rows)
```
CREATE TABLE sales
(
  dept_no   number,
  part_no   varchar2,
  country   varchar2(20),
  date      date,
  amount    number
)
PARTITION BY RANGE (date)
(
  PARTITION q1_2012
    VALUES LESS THAN ('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN ('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN ('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN ('2013-Jan-01')
);

The table contains the four partitions q1_2012, q2_2012, q3_2012, and q4_2012:

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1_2012</td>
<td>'01-APR-12 00:00:00'</td>
</tr>
<tr>
<td>q2_2012</td>
<td>'01-JUL-12 00:00:00'</td>
</tr>
<tr>
<td>q3_2012</td>
<td>'01-OCT-12 00:00:00'</td>
</tr>
<tr>
<td>q4_2012</td>
<td>'01-JAN-13 00:00:00'</td>
</tr>
</tbody>
</table>

This command adds a partition named q1_2013 to the sales table:

ALTER TABLE sales ADD PARTITION q1_2013
  VALUES LESS THAN ('01-APR-2013');

After the command is invoked, the table includes the q1_2013 partition:

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1_2012</td>
<td>'01-APR-12 00:00:00'</td>
</tr>
<tr>
<td>q2_2012</td>
<td>'01-JUL-12 00:00:00'</td>
</tr>
<tr>
<td>q3_2012</td>
<td>'01-OCT-12 00:00:00'</td>
</tr>
<tr>
<td>q4_2012</td>
<td>'01-JAN-13 00:00:00'</td>
</tr>
<tr>
<td>q1_2013</td>
<td>'01-APR-13 00:00:00'</td>
</tr>
</tbody>
</table>

14.3.2.2.3 Example: Adding a partition with SUBPARTITIONS num...IN PARTITION DESCRIPTION

This example adds a partition to a list-partitioned sales table. You can specify a SUBPARTITIONS clause to add a specified number of subpartitions. The sales table was created with the command:

CREATE TABLE sales
(
  dept_no   number,
  part_no   varchar2,
  country   varchar2(20),
  date      date,
  amount    number
)
PARTITION BY LIST (country) SUBPARTITION BY HASH (part_no) SUBPARTITIONS 2
(
  PARTITION europe VALUES ('FRANCE', 'ITALY'),
  PARTITION asia VALUES ('INDIA', 'PAKISTAN')
);
The table contains two partitions: europe and asia. Each contains two subpartitions. Because the subpartitions aren't named, system-generated names are assigned to them.

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0103</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0102</td>
</tr>
</tbody>
</table>

(4 rows)

This command adds a partition americas to the sales table and creates a number of subpartitions as specified in the partition description:

```sql
ALTER TABLE sales ADD PARTITION americas VALUES ('US', 'CANADA');
```

After invoking the command, the table includes the partition americas and two newly added subpartitions:

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0107</td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0108</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0103</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0102</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
</tr>
</tbody>
</table>

(6 rows)

Example - Adding a Partition with SUBPARTITIONS num...

This example adds a partition a list-partitioned table sales consisting of three subpartitions. The sales table was created with the command:

```sql
CREATE TABLE sales (
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
) PARTITION BY LIST (country) SUBPARTITION BY HASH (part_no) SUBPARTITIONS 3 (
    PARTITION europe VALUES ('FRANCE', 'ITALY'),
    PARTITION asia VALUES ('INDIA', 'PAKISTAN')
);
```

The table contains partitions europe and asia, each containing three subpartitions:

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0105</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0106</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0102</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0103</td>
</tr>
</tbody>
</table>

(6 rows)
This command adds a partition `americas` and five subpartitions, as specified in the `ADD PARTITION` clause:

```sql
ALTER TABLE sales ADD PARTITION americas
  VALUES ('US', 'CANADA') SUBPARTITIONS 5;
```

After the command is invoked, the `sales` table includes the partition `americas` and five newly added subpartitions:

```
table_name | partition_name | subpartition_name
------------+----------------+-------------------
SALES      | AMERICAS       | SYS0109
SALES      | AMERICAS       | SYS0110
SALES      | AMERICAS       | SYS0111
SALES      | AMERICAS       | SYS0112
SALES      | AMERICAS       | SYS0113
(5 rows)
```

Example: Adding a partition with `SUBPARTITIONS num... STORE IN`

This example adds a partition to a list-partitioned table `sales` consisting of three subpartitions. The table was created using the command:

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY LIST (country)
  SUBPARTITION BY HASH (part_no)
  SUBPARTITIONS 3
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA'));
```

The table contains the three partitions `americas`, `asia`, and `europe`. Each contains three subpartitions with system-generated names:

```
EDB# SELECT table_name, partition_name, subpartition_name
FROM ALL_TAB_SUBPARTITIONS
WHERE table_name = 'SALES' ORDER BY 1, 2;
```

```
<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0109</td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0107</td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0108</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0105</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0106</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0103</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0102</td>
</tr>
</tbody>
</table>
(9 rows)
```

This command adds a partition `east_asia` with five subpartitions as specified in the `ADD PARTITION` clause. It stores them in the tablespace named `ts1`:

```sql
ALTER TABLE sales ADD PARTITION east_asia
  VALUES ('CHINA', 'KOREA') SUBPARTITIONS 5 STORE IN (ts1);
```

After the command is invoked, the table includes the partition `east_asia` and five newly added subpartitions stored in tablespace `ts1`:

```
EDB# SELECT table_name, partition_name, subpartition_name, tablespace_name
FROM ALL_TAB_SUBPARTITIONS
WHERE table_name = 'SALES' ORDER BY 1, 2;
```

```
<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0109</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0110</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0111</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0112</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0113</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0105</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0104</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0106</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0103</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0102</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>EAST ASIA</td>
<td>SYS0109</td>
<td>ts1</td>
</tr>
<tr>
<td>SALES</td>
<td>EAST ASIA</td>
<td>SYS0110</td>
<td>ts1</td>
</tr>
<tr>
<td>SALES</td>
<td>EAST ASIA</td>
<td>SYS0111</td>
<td>ts1</td>
</tr>
<tr>
<td>SALES</td>
<td>EAST ASIA</td>
<td>SYS0112</td>
<td>ts1</td>
</tr>
<tr>
<td>SALES</td>
<td>EAST ASIA</td>
<td>SYS0113</td>
<td>ts1</td>
</tr>
<tr>
<td>SALES</td>
<td>CHINA</td>
<td>SYS0109</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>KOREA</td>
<td>SYS0110</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>CHINA</td>
<td>SYS0111</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>KOREA</td>
<td>SYS0112</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>CHINA</td>
<td>SYS0113</td>
<td></td>
</tr>
</tbody>
</table>
(11 rows)
```
### 14.3.2.3 ALTER TABLE...ADD SUBPARTITION

The **ALTER TABLE... ADD SUBPARTITION** command adds a subpartition to an existing subpartitioned partition. The syntax is:

```
ALTER TABLE <table_name> MODIFY PARTITION <partition_name>
    ADD SUBPARTITION <subpartition_definition>;
```

Where **subpartition_definition** is:

- `{<list_subpartition> | <range_subpartition>}`

**And list_subpartition is:**

```
SUBPARTITION [
<subpartition_name>
VALUES <value>[, 
<value>]...]
[TABLESPACE <tablespace_name>]
```

**And range_subpartition is:**

```
SUBPARTITION [
<subpartition_name>
VALUES LESS THAN <value>[, 
<value>]...]
[TABLESPACE <tablespace_name>]
```

### Description

The **ALTER TABLE... ADD SUBPARTITION** command adds a subpartition to an existing partition. The partition must already be subpartitioned. There's no upper limit to the number of defined subpartitions.

New subpartitions must be of the same type (**LIST, RANGE** or **HASH**) as existing subpartitions. The new subpartition rules must reference the same column specified in the subpartitioning rules that define the existing subpartitions.

You can use the **ALTER TABLE... ADD SUBPARTITION** statement to add a subpartition to a table with a **DEFAULT** rule. However, there must not be conflicting values between existing rows in the table and the values of the subpartition to add.

You can't use the **ALTER TABLE... ADD SUBPARTITION** statement to add a subpartition to a table with a **MAXVALUE** rule.

You can split an existing subpartition with the **ALTER TABLE... SPLIT SUBPARTITION** statement, effectively adding a subpartition to a table.

You can add a subpartition that precedes existing subpartitions in a range-subpartitioned table. You must specify range subpartitions in ascending order.

Include the **TABLESPACE** clause to specify the tablespace in which the subpartition resides. If you don't specify a tablespace, the subpartition is created in the default tablespace.

If the table is indexed, the index is created on the new subpartition.

To use the **ALTER TABLE... ADD SUBPARTITION** command, you must be the table owner or have superuser or administrative privileges.

### Parameters
**table_name**

The name (optionally schema-qualified) of the partitioned table in which the subpartition resides.

**partition_name**

The name of the partition in which the new subpartition resides.

**subpartition_name**

The name of the subpartition to create. Subpartition names must be unique among all partitions and subpartitions and must follow the naming conventions for object identifiers.

**{value[, value]...}**

Use `value` to specify a quoted literal value or comma-delimited list of literal values by which table entries are grouped into partitions. Each partitioning rule must specify at least one value. However, there's no limit on the number of values specified in a rule. `value` can also be `NULL`, `DEFAULT` if specifying a `LIST` partition, or `MAXVALUE` if specifying a `RANGE` partition.

For information about creating a `DEFAULT` or `MAXVALUE` partition, see Handling stray values in a LIST or RANGE partitioned table.

**tablespace_name**

The name of the tablespace in which the subpartition resides.

### 14.3.2.3.1 Example: Adding a subpartition to a LIST/RANGE partitioned table

This example adds a `RANGE` subpartition to the list-partitioned `sales` table. The `sales` table was created with the command:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST(country)
    SUBPARTITION BY RANGE(date)
    (        PARTITION europe VALUES('FRANCE', 'ITALY')
        (            SUBPARTITION europe_2011 VALUES LESS THAN('2012-Jan-01'),
            SUBPARTITION europe_2012 VALUES LESS THAN('2013-Jan-01')
        ),
        PARTITION asia VALUES('INDIA', 'PAKISTAN')
        (            SUBPARTITION asia_2011 VALUES LESS THAN('2012-Jan-01'),
            SUBPARTITION asia_2012 VALUES LESS THAN('2013-Jan-01')
        ),
        PARTITION americas VALUES('US', 'CANADA')
        (            SUBPARTITION americas_2011 VALUES LESS THAN('2012-Jan-01'),
                SUBPARTITION americas_2012 VALUES LESS THAN('2013-Jan-01')
        )
    );
```

The `sales` table has three partitions named `europe`, `asia`, and `americas`. Each partition has two range-defined subpartitions.

```sql
edb# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
```
This command adds a subpartition named `europe_2013`:

```
ALTER TABLE sales MODIFY PARTITION europe
ADD SUBPARTITION europe_2013 VALUES LESS THAN ('2015-Jan-01');
```

After the command is invoked, the table includes a subpartition named `europe_2013`:

```
partition_name | subpartition_name | high_value
----------------+-------------------+----------------------
EUROPE         | EUROPE_2011       | '01-JAN-12 00:00:00'  
EUROPE         | EUROPE_2012       | '01-JAN-13 00:00:00'  
EUROPE         | EUROPE_2013       | '01-JAN-15 00:00:00'  
ASIA           | ASIA_2011         | '01-JAN-12 00:00:00'  
ASIA           | ASIA_2012         | '01-JAN-13 00:00:00'  
AMERICAS       | AMERICAS_2011     | '01-JAN-12 00:00:00'  
AMERICAS       | AMERICAS_2012     | '01-JAN-13 00:00:00'  
-------------------- (7 rows)---------------------
```

**Note**

When adding a range subpartition, the subpartitioning rules must specify a range that falls after any existing subpartitions.

### 14.3.2.3.2 Example: Adding a subpartition to a RANGE/LIST partitioned table

This example adds a `LIST` subpartition to the `RANGE` partitioned `sales` table. The `sales` table was created with the command:

```
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
) PARTITION BY RANGE(date)
    SUBPARTITION BY LIST (country)
    (
        PARTITION first_half_2012 VALUES LESS THAN ('01-JUL-2012')
        (
            SUBPARTITION europe VALUES ('ITALY', 'FRANCE'),
            SUBPARTITION americas VALUES ('US', 'CANADA')
        ),
        PARTITION second_half_2012 VALUES LESS THAN ('01-JAN-2013')
        ( 'asia VALUES ('INDIA', 'PAKISTAN')
        )
    );
```

After the command is invoked, the `sales` table has two partitions, named `first_half_2012` and `second_half_2012`. The `first_half_2012` partition has two subpartitions, named `europe` and `americas`. The `second_half_2012` partition has one partition, named `asia`.

```
edb=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
```
This command adds a subpartition named `east_asia` to the `second_half_2012` partition:

```
ALTER TABLE sales MODIFY PARTITION second_half_2012 ADD SUBPARTITION east_asia VALUES ('CHINA');
```

After the command is invoked, the table includes a subpartition named `east_asia`:

```
partition_name  | subpartition_name |     high_value
------------------+-------------------+---------------------
SECOND_HALF_2012 | ASIA              | 'INDIA', 'PAKISTAN'
SECOND_HALF_2012 | EAST_ASIA         | 'CHINA'
FIRST_HALF_2012  | AMERICAS          | 'US', 'CANADA'
FIRST_HALF_2012  | EUROPE            | 'ITALY', 'FRANCE'
(4 rows)
```

**14.3.2.4 ALTER TABLE...SPLIT PARTITION**

Use the `ALTER TABLE...SPLIT PARTITION` command to divide a single partition into two partitions. This command maintains the partitioning of the original table in the newly created partitions and redistributes the partition's contents between the new partitions. The command syntax comes in two forms.

The first form splits a `RANGE` partition into two partitions:

```
ALTER TABLE <table_name> SPLIT PARTITION <partition_name> 
AT (<range_part_value>) 
INTO 

{ 
  PARTITION <new_part1> 
  [TABLESPACE <tablespace_name>] 
  [SUBPARTITIONS <num>] [STORE IN ( <tablespace_name> [, <tablespace_name>]...) ] 
}, 
  PARTITION <new_part2> 
  [TABLESPACE <tablespace_name>] 
  [SUBPARTITIONS <num>] [STORE IN ( <tablespace_name> [, <tablespace_name>]...) ]
};
```

The second form splits a `LIST` partition into two partitions:

```
ALTER TABLE <table_name> SPLIT PARTITION <partition_name> 
VALUES <value>[, <value>...] 
INTO 

{ 
  PARTITION <new_part1> 
  [TABLESPACE <tablespace_name>] 
  [SUBPARTITIONS <num>] [STORE IN ( <tablespace_name> [, <tablespace_name>]...) ] 
}, 
  PARTITION <new_part2> 
  [TABLESPACE <tablespace_name>] 
  [SUBPARTITIONS <num>] [STORE IN ( <tablespace_name> [, <tablespace_name>]...) ]
};
```
The **ALTER TABLE...SPLIT PARTITION** command adds a partition to an existing **LIST** or **RANGE** partitioned table. The **ALTER TABLE...SPLIT PARTITION** command can't add a partition to a **HASH** partitioned table. There's no upper limit to the number of partitions.

When you execute an **ALTER TABLE...SPLIT PARTITION** command, EDB Postgres Advanced Server:

- Creates two new partitions
- Maintains the partitioning of the original table in the newly created partitions
- Redistributes the content of the old partition between them, as constrained by the partitioning rules

Include the **TABLESPACE** clause to specify the tablespace in which a partition resides. If you don't specify a tablespace, the partition resides in the tablespace of the original partitioned table.

Use the **STORE IN** clause to specify the tablespace list across which the autogenerated subpartitions are stored.

If the table is indexed, the index is created on the new partition.

To use the **ALTER TABLE...SPLIT PARTITION** command, you must be the table owner or have superuser or administrative privileges.

### Parameters

- **table_name**
  - The name (optionally schema-qualified) of the partitioned table.

- **partition_name**
  - The name of the partition that's being split.

- **new_part1**
  - The name of the first partition to create. Partition names must be unique among all partitions and subpartitions and must follow the naming conventions for object identifiers. **new_part1** receives the rows that meet the partitioning constraints specified in the **ALTER TABLE...SPLIT PARTITION** command.

- **new_part2**
  - The name of the second partition to create. Partition names must be unique among all partitions and subpartitions and must follow the naming conventions for object identifiers. **new_part2** receives the rows that aren't directed to **new_part1** by the partitioning constraints specified in the **ALTER TABLE...SPLIT PARTITION** command.

- **range_part_value**
  - Use **range_part_value** to specify the boundary rules by which to create the new partition. The partitioning rule must contain at least one column of a data type that has two operators, that is, a greater-than-or-equal-to operator and a less-than operator. Range boundaries are evaluated against a **LESS THAN** clause and are non-inclusive. A date boundary of January 1, 2010 includes only those date values that fall on or before December 31, 2009.

  - `(value[, value]...)`

- **value**
  - Use **value** to specify a quoted literal value or comma-delimited list of literal values by which to distribute rows into partitions. Each partitioning rule must specify at least one value. There's no limit on the number of values specified in a rule.

For information about creating a **DEFAULT** or **MAXVALUE** partition, see Handling stray values in a **LIST** or **RANGE** partitioned table.

- **num**
  - The **SUBPARTITIONS** **num** clause is supported only for **HASH** subpartitions. Use the clause to specify the number of hash subpartitions. Alternatively, you can use the subpartition definition to specify individual subpartitions and their tablespaces. If you don't specify **SUBPARTITIONS** or **SUBPARTITION DEFINITION**, then the partition creates a subpartition based on the **SUBPARTITION TEMPLATE**.

- **tablespace_name**
  - The name of the tablespace in which the partition or subpartition resides.

### Example: Splitting a LIST partition

This example divides one of the partitions in the list-partitioned **sales** table into two new partitions and redistributes the contents of the partition between them. The **sales** table is created with the statement:
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);

The table definition creates three partitions: europe, asia, and americas. This command adds rows to each partition:

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This command splits the americas partition into partitions named us and canada:

The rows are distributed among the partitions:

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This command splits the americas partition into partitions named us and canada:
A SELECT statement confirms that the rows were redistributed:

```
14.3.2.4.2 Example: Splitting a RANGE partition

This example divides the q4_2012 partition of the range-partitioned sales table into two partitions and redistributes the partition's contents. Use this command to create the sales table:

CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
) PARTITION BY RANGE (date)
(
  PARTITION q1_2012
VALUES LESS THAN ('2012-Apr-01'),
PARTITION q2_2012
VALUES LESS THAN ('2012-Jul-01'),
PARTITION q3_2012
VALUES LESS THAN ('2012-Oct-01'),
PARTITION q4_2012
VALUES LESS THAN ('2013-Jan-01')
);

The table definition creates four partitions: q1_2012, q2_2012, q3_2012, and q4_2012. This command adds rows to each partition:

INSERT INTO sales VALUES
((10, '4519b', 'FRANCE', '17-Jan-2012',
'45000'),
(20, '3788a', 'INDIA', '01-Mar-2012',
'75000'),
(40, '9519b', 'US', '12-Apr-2012',
'145000'),
(28, '3788a', 'PAKISTAN', '04-Jun-2012',
'37500'),
(48, '4577b', 'US', '11-Nov-2012',
'25000'),
(30, '7588b', 'CANADA', '14-Dec-2012',
'50000'),
(38, '9519b', 'CANADA', '01-Feb-2012',
'75000'),
(38, '4519b', 'CANADA', '08-Apr-2012',
'120000'),
...)
```
A `SELECT` statement confirms that the rows are distributed among the partitions:

```
<table>
<thead>
<tr>
<th>tableoid:regclass</th>
<th>*</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tableoid</td>
<td>dept_no</td>
<td>part_no</td>
</tr>
<tr>
<td>sales_q1_2012</td>
<td>10</td>
<td>4519b</td>
</tr>
<tr>
<td>sales_q1_2012</td>
<td>28</td>
<td>3788a</td>
</tr>
<tr>
<td>sales_q1_2012</td>
<td>38</td>
<td>9519b</td>
</tr>
<tr>
<td>sales_q2_2012</td>
<td>48</td>
<td>9519b</td>
</tr>
<tr>
<td>sales_q2_2012</td>
<td>38</td>
<td>4519b</td>
</tr>
<tr>
<td>sales_q3_2012</td>
<td>18</td>
<td>9519b</td>
</tr>
<tr>
<td>sales_q3_2012</td>
<td>10</td>
<td>9519a</td>
</tr>
<tr>
<td>sales_q3_2012</td>
<td>10</td>
<td>9519b</td>
</tr>
<tr>
<td>sales_q3_2012</td>
<td>28</td>
<td>3788b</td>
</tr>
<tr>
<td>sales_q4_2012_p1</td>
<td>40</td>
<td>4788a</td>
</tr>
<tr>
<td>sales_q4_2012_p1</td>
<td>40</td>
<td>4577b</td>
</tr>
<tr>
<td>sales_q4_2012_p1</td>
<td>20</td>
<td>4519a</td>
</tr>
<tr>
<td>sales_q4_2012_p2</td>
<td>30</td>
<td>7588b</td>
</tr>
</tbody>
</table>
```

This command splits the `q4_2012` partition into two partitions named `q4_2012_p1` and `q4_2012_p2`:

```
ALTER TABLE sales SPLIT PARTITION q4_2012 
    AT ('15-Nov-2012')
    INTO
    (PARTITION q4_2012_p1,
     PARTITION q4_2012_p2);
```

A `SELECT` statement confirms that the rows were redistributed across the new partitions:

```
<table>
<thead>
<tr>
<th>tableoid:regclass</th>
<th>*</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tableoid</td>
<td>dept_no</td>
<td>part_no</td>
</tr>
<tr>
<td>sales_q1_2012</td>
<td>10</td>
<td>4519b</td>
</tr>
<tr>
<td>sales_q1_2012</td>
<td>20</td>
<td>3788a</td>
</tr>
<tr>
<td>sales_q1_2012</td>
<td>30</td>
<td>9519b</td>
</tr>
<tr>
<td>sales_q2_2012</td>
<td>40</td>
<td>9519b</td>
</tr>
<tr>
<td>sales_q2_2012</td>
<td>30</td>
<td>4519b</td>
</tr>
<tr>
<td>sales_q3_2012</td>
<td>18</td>
<td>9519b</td>
</tr>
<tr>
<td>sales_q3_2012</td>
<td>10</td>
<td>9519a</td>
</tr>
<tr>
<td>sales_q3_2012</td>
<td>10</td>
<td>9519b</td>
</tr>
<tr>
<td>sales_q3_2012</td>
<td>28</td>
<td>3788b</td>
</tr>
<tr>
<td>sales_q4_2012_p1</td>
<td>40</td>
<td>4788a</td>
</tr>
<tr>
<td>sales_q4_2012_p1</td>
<td>40</td>
<td>4577b</td>
</tr>
<tr>
<td>sales_q4_2012_p1</td>
<td>20</td>
<td>4519a</td>
</tr>
<tr>
<td>sales_q4_2012_p2</td>
<td>30</td>
<td>7588b</td>
</tr>
</tbody>
</table>
```

### 14.3.2.4.3 Example: Splitting a RANGE/LIST partition

This example divides one of the partitions in the range-partitioned `sales` table into new partitions. This approach maintains the partitioning of the original table in the newly created partitions and redistributes the contents of the partition between them.

```sql
CREATE TABLE sales (  
department number,  
part_number varchar2,  
country varchar2(20),  
date date,  
amount number  )  
PARTITION BY RANGE(date)  
SUBPARTITION BY LIST (country)  
(  
PARTITION q1_2012 VALUES LESS THAN ('2012-Apr-01')  
(  
    SUBPARTITION europe VALUES ('FRANCE', 'ITALY'),  
    SUBPARTITION americas VALUES ('US', 'CANADA'),  
    SUBPARTITION asia VALUES ('INDIA', 'PAKISTAN')  
)  
);  

The sales table contains partition q1_2012 and the three subpartitions europe, americas, and asia.

```sql
edb# SELECT table_name, partition_name, subpartition_name, high_value  
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;  
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>Q1_2012</td>
<td>EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2012</td>
<td>AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2012</td>
<td>ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
</tbody>
</table>
(3 rows)
```

This command splits the q1_2012 partition into partitions named q1_2012_p1 and q1_2012_p2:

```sql
ALTER TABLE sales SPLIT PARTITION q1_2012  
AT ('01-Mar-2012')  
INTO  
(  
    PARTITION q1_2012_p1,  
    PARTITION q1_2012_p2  
)  
;
```

A SELECT statement confirms that the same number of subpartitions is created in the newly created partitions q1_2012_p1 and q1_2012_p2 with system-generated names:

```sql
edb# SELECT table_name, partition_name, subpartition_name, high_value  
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2,3;  
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>Q1_2012_p1</td>
<td>SYS0105</td>
<td>'US', 'CANADA'</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2012_p1</td>
<td>SYS0106</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2012_p1</td>
<td>SYS0107</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2012_p2</td>
<td>SYS0108</td>
<td>'US', 'CANADA'</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2012_p2</td>
<td>SYS0109</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2012_p2</td>
<td>SYS0110</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
</tbody>
</table>
(6 rows)
Example: Splitting a partition with SUBPARTITIONS num...

This example divides one of the partitions in the list-partitioned sales table into new partitions. It maintains the partitioning of the original table in the newly created partitions and redistributes the contents of the partition between them. The SUBPARTITIONS clause lets you add a specified number of subpartitions. Without the SUBPARTITIONS clause, the new partitions inherit the default number of subpartitions.

This statement creates the sales table:

```
CREATE TABLE sales
(
dept_no number,
part_no varchar2,
country varchar2(20),
date date,
amount number)
PARTITION BY LIST(country) SUBPARTITION BY HASH (part_no) SUBPARTITIONS 2
(
  PARTITION europe VALUES('FRANCE',
                        'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA'))
);
```

The table definition creates the three partitions europe, asia, and americas. Each contains two subpartitions.

```
<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0105</td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0106</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0103</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0102</td>
</tr>
</tbody>
</table>
(6 rows)
```

This command splits the americas partition into partitions named partition_us and partition_canada:

```
ALTER TABLE sales SPLIT PARTITION americas VALUES ('US') INTO
  (PARTITION partition_us SUBPARTITIONS 5, PARTITION partition_canada);
```

A SELECT statement confirms that the americas partition is split into partition_us and partition_canada. The partition_us contains five subpartitions. partition_canada contains two default subpartitions with system-generated names.

```
<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0103</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0102</td>
</tr>
<tr>
<td>SALES</td>
<td>PARTITION_CANADA</td>
<td>SYS0115</td>
</tr>
<tr>
<td>SALES</td>
<td>PARTITION_CANADA</td>
<td>SYS0116</td>
</tr>
<tr>
<td>SALES</td>
<td>PARTITION_US</td>
<td>SYS0110</td>
</tr>
<tr>
<td>SALES</td>
<td>PARTITION_US</td>
<td>SYS0111</td>
</tr>
<tr>
<td>SALES</td>
<td>PARTITION_US</td>
<td>SYS0112</td>
</tr>
<tr>
<td>SALES</td>
<td>PARTITION_US</td>
<td>SYS0113</td>
</tr>
<tr>
<td>SALES</td>
<td>PARTITION_US</td>
<td>SYS0114</td>
</tr>
</tbody>
</table>
(11 rows)
```

Example: Splitting a partition with SUBPARTITIONS num...STORE IN

This example divides the europe partition of the list-partitioned sales table into two partitions. It maintains the partitioning of the original table in the newly created partitions and redistributes the partition’s contents. The SUBPARTITIONS clause lets you add a specified number of subpartitions.
Use the following command to create the `sales` table:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
) PARTITION BY LIST(country) SUBPARTITION BY HASH (part_no) SUBPARTITIONS 4
(
    PARTITION europe VALUES('FRANCE', 'ITALY'),
    PARTITION asia VALUES('INDIA', 'PAKISTAN'),
    PARTITION americas VALUES('US', 'CANADA')
);
```

The `sales` table contains the partitions `europe`, `asia`, and `americas`. Each contains four subpartitions with system-generated names.

```sql
edb=# SELECT table_name, partition_name, subpartition_name
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0109</td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0110</td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0111</td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0112</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0105</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0106</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0107</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0108</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0103</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0102</td>
</tr>
<tr>
<td>(12 rows)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This command splits the `europe` partition into the partitions `france` and `italy`:

```sql
ALTER TABLE sales SPLIT PARTITION europe VALUES ('FRANCE') INTO
PARTITION france SUBPARTITIONS 10 STORE IN (ts1), PARTITION
italy;
```

A `SELECT` statement confirms that the `europe` partition is split into two partitions. The partition `france` contains 10 subpartitions that are stored in the tablespace `ts1`. Partition `italy` contains four subpartitions as in the original partition `europe`.

```sql
edb=# SELECT table_name, partition_name, subpartition_name, tablespace_name
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2,3;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0109</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0110</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0111</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0112</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0105</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0106</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0107</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0108</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0116</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0117</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0118</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0119</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0120</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0121</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0122</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0123</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0124</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>FRANCE</td>
<td>SYS0125</td>
<td>TSI</td>
</tr>
<tr>
<td>SALES</td>
<td>ITALY</td>
<td>SYS0126</td>
<td></td>
</tr>
<tr>
<td>SALES</td>
<td>ITALY</td>
<td>SYS0127</td>
<td></td>
</tr>
<tr>
<td>(12 rows)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
14.3.2.5 ALTER TABLE...SPLIT SUBPARTITION

Use the ALTER TABLE...SPLIT SUBPARTITION command to divide a single subpartition into two subpartitions and redistribute the subpartition’s contents. The command comes in two variations.

The first variation splits a range subpartition into two subpartitions:

```
ALTER TABLE <table_name> SPLIT SUBPARTITION <subpartition_name>
  AT (range_part_value)
  INTO
  {  
    SUBPARTITION <new_subpart1>
    TABLESPACE <tablespace_name>,
    SUBPARTITION <new_subpart2>
    TABLESPACE <tablespace_name>
  };
```

The second variation splits a list subpartition into two subpartitions:

```
ALTER TABLE <table_name> SPLIT SUBPARTITION <subpartition_name>
  VALUES {<value>,,<value>..}
  INTO
  {  
    SUBPARTITION <new_subpart1>
    TABLESPACE <tablespace_name>,
    SUBPARTITION <new_subpart2>
    TABLESPACE <tablespace_name>
  };
```

Description

The ALTER TABLE...SPLIT SUBPARTITION command adds a subpartition to an existing subpartitioned table. There’s no upper limit to the number of defined subpartitions. When you execute an ALTER TABLE...SPLIT SUBPARTITION command, EDB Postgres Advanced Server creates two subpartitions. It moves any rows that contain values that are constrained by the specified subpartition rules into new_subpart1 and any remaining rows into new_subpart2.

The new subpartition rules must reference the column specified in the rules that define the existing subpartitions.

Include the TABLESPACE clause to specify a tablespace in which a new subpartition resides. If you don’t specify a tablespace, the subpartition is created in the default tablespace.

If the table is indexed, the index is created on the new subpartition.

To use the ALTER TABLE... SPLIT SUBPARTITION command, you must be the table owner or have superuser or administrative privileges.

Parameters

- **table_name**
  The name (optionally schema-qualified) of the partitioned table.

- **subpartition_name**
  The name of the subpartition that’s being split.

- **new_subpart1**
  The name of the first subpartition to create. Subpartition names must be unique among all partitions and subpartitions and must follow the naming conventions for object identifiers.
new_subpart1 receives the rows that meet the subpartitioning constraints specified in the `ALTER TABLE... SPLIT SUBPARTITION` command.

new_subpart2 receives the rows that aren't directed to new_subpart1 by the subpartitioning constraints specified in the `ALTER TABLE... SPLIT SUBPARTITION` command.

The name of the second subpartition to create. Subpartition names must be unique among all partitions and subpartitions and must follow the naming conventions for object identifiers.

Use `value` to specify a quoted literal value or comma-delimited list of literal values by which table entries are grouped into partitions. Each partitioning rule must specify at least one value. There's no limit on the number of values specified in a rule. `value` can also be `NULL`, `DEFAULT` if specifying a `LIST` subpartition, or `MAXVALUE` if specifying a `RANGE` subpartition.

For information about creating a `DEFAULT` or `MAXVALUE` partition, see Handling stray values in a LIST or RANGE partitioned table.

tablespace_name

The name of the tablespace in which the partition or subpartition resides.

### 14.3.2.5.1 Example: Splitting a LIST subpartition

This example splits a list subpartition, redistributing the subpartition's contents between two new subpartitions.

The sample table `sales` was created with the command:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY RANGE(date)
    SUBPARTITION BY LIST (country)
    {
        PARTITION first_half_2012 VALUES LESS THAN ('01-JUL-2012')
        {
            SUBPARTITION p1_europe VALUES ('ITALY', 'FRANCE'),
            SUBPARTITION p1_americas VALUES ('US', 'CANADA')
        },
        PARTITION second_half_2012 VALUES LESS THAN ('01-JAN-2013')
        {
            SUBPARTITION p2_europe VALUES ('ITALY', 'FRANCE'),
            SUBPARTITION p2_americas VALUES ('US', 'CANADA')
        }
    );
```

The `sales` table has partitions named `first_half_2012` and `second_half_2012`. Each partition has two range-defined subpartitions that distribute the partition's contents into subpartitions based on the value of the `country` column.

```sql
edb# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
```

<table>
<thead>
<tr>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECOND_HALF_2012</td>
<td>P2_AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
<tr>
<td>SECOND_HALF_2012</td>
<td>P2_EUROPE</td>
<td>'ITALY', 'FRANCE'</td>
</tr>
<tr>
<td>FIRST_HALF_2012</td>
<td>P1_AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
<tr>
<td>FIRST_HALF_2012</td>
<td>P1_EUROPE</td>
<td>'ITALY', 'FRANCE'</td>
</tr>
</tbody>
</table>

This command adds rows to each subpartition:

```sql
INSERT INTO sales VALUES
```
A `SELECT` statement confirms that the rows are correctly distributed among the subpartitions:

```
SELECT tableoid, dept_no, part_no, country, date, amount
FROM sales;
```

```
Tableoid | Dept_no | Part_no | Country | Date            | Amount
--------+---------+---------+---------+-----------------+--------
sales_p1_americas | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000
sales_p1_americas | 30 | 9519b | CANADA | 01-FEB-12 00:00:00 | 75000
sales_p1_americas | 30 | 4519b | CANADA | 08-APR-12 00:00:00 | 120000
sales_p1_americas | 40 | 3788a | US | 12-MAY-12 00:00:00 | 4950
sales_p2_americas | 10 | 9519b | ITALY | 07-JUL-12 00:00:00 | 15000
sales_p2_americas | 10 | 9519a | FRANCE | 18-AUG-12 00:00:00 | 650000
sales_p2_americas | 10 | 9519b | FRANCE | 18-AUG-12 00:00:00 | 650000
sales_p2_europe | 10 | 4788a | 'US', 23-SEP-2012, 4950
sales_p2_europe | 10 | 4788b | 'US' | 09-OCT-12 00:00:00 | 15000
sales_p2_europe | 10 | 9519b | 'US' | 12-MAY-12 00:00:00 | 4950
sales_p2_europe | 10 | 9519a | 'US', 09-OCT-12 00:00:00 | 650000
sales_p2_europe | 10 | 9519b | 'US', 09-OCT-12 00:00:00 | 650000
(12 rows)
```

This command splits the `p2_americas` subpartition into two new subpartitions and redistributes the contents:

```
ALTER TABLE sales SPLIT SUBPARTITION p2_americas VALUES ('US') INTO

  SUBPARTITION p2_us,
  SUBPARTITION p2_canada;
```

After the command is invoked, the `p2_americas` subpartition is deleted. In its place, the server creates the subpartitions `p2_us` and `p2_canada`:

```
SELECT partition_name, subpartition_name, high_value
FROM ALL_TAB_SUBPARTITIONS;
```

```
Partition_name | Subpartition_name | High_value
---------------+------------------+-----------------
FIRST_HALF_2012 | P1_EUROPE | 'ITALY', 'FRANCE'
FIRST_HALF_2012 | P1_AMERICAS | 'US', 'CANADA'
SECOND_HALF_2012 | P2_EUROPE | 'ITALY', 'FRANCE'
SECOND_HALF_2012 | P2_US | 'US'
SECOND_HALF_2012 | P2_CANADA | 'CANADA'
(5 rows)
```

Queroting the `sales` table shows that the content of the `p2_americas` subpartition was redistributed:

```
SELECT tableoid, dept_no, part_no, country, date, amount
FROM sales;
```

```
Tableoid | Dept_no | Part_no | Country | Date            | Amount
--------+---------+---------+---------+-----------------+--------
sales_p1_americas | 40 | 9519b | US | 12-APR-12 00:00:00 | 145000
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
14.3.2.5.2 Example: Splitting a RANGE subpartition

This example splits a range subpartition, redistributing the subpartition's contents between two new subpartitions.

The sample table `sales` was created with the command:

```sql
CREATE TABLE sales
(
  dept_no_number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY LIST(country)
  SUBPARTITION BY RANGE(date)
  PARTITION europe VALUES('FRANCE', 'ITALY')
  PARTITION europe_2011 VALUES LESS THAN('2012-Jan-01'),
  PARTITION europe_2012 VALUES LESS THAN('2013-Jan-01')
  PARTITION asia VALUES('INDIA', 'PAKISTAN')
  PARTITION asia_2011 VALUES LESS THAN('2012-Jan-01'),
  PARTITION asia_2012 VALUES LESS THAN('2013-Jan-01')
  PARTITION americas VALUES('US', 'CANADA')
  PARTITION americas_2011 VALUES LESS THAN('2012-Jan-01'),
  PARTITION americas_2012 VALUES LESS THAN('2013-Jan-01')
);
```

The `sales` table has the partitions `europe`, `asia`, and `americas`. Each partition has two range-defined subpartitions that sort the partitions contents into subpartitions by the value of the `date` column:

```sql
edb=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
```

<table>
<thead>
<tr>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE</td>
<td>EUROPE_2011</td>
<td>'01-JAN-12 00:00:00'</td>
</tr>
<tr>
<td>EUROPE</td>
<td>EUROPE_2012</td>
<td>'01-JAN-13 00:00:00'</td>
</tr>
<tr>
<td>ASIA</td>
<td>ASIA_2011</td>
<td>'01-JAN-12 00:00:00'</td>
</tr>
<tr>
<td>ASIA</td>
<td>ASIA_2012</td>
<td>'01-JAN-13 00:00:00'</td>
</tr>
<tr>
<td>AMERICAS</td>
<td>AMERICAS_2011</td>
<td>'01-JAN-12 00:00:00'</td>
</tr>
<tr>
<td>AMERICAS</td>
<td>AMERICAS_2012</td>
<td>'01-JAN-13 00:00:00'</td>
</tr>
</tbody>
</table>

(6 rows)
This command adds rows to each subpartition:

```
INSERT INTO sales VALUES
(18, '4519b', 'FRANCE', '17-Jan-2012', 145000),
(28, '3788a', 'INDIA', '01-Mar-2012', 75000),
(40, '9519b', 'US', '12-Apr-2012', 145000),
(28, '3788a', 'PAKISTAN', '04-Jun-2012', 37500),
(40, '4577b', 'US', '11-Nov-2012', 25000),
(38, '7588b', 'CANADA', '14-Dec-2012', 50000),
(38, '9519b', 'CANADA', '01-Feb-2012', 75000),
(38, '4519b', 'CANADA', '08-Apr-2012', 120000),
(40, '3788a', 'US', '12-May-2012', 4950),
(10, '9519b', 'ITALY', '07-Jul-2012', 15000),
(18, '9519a', 'FRANCE', '18-Aug-2012', 650000),
(18, '9519b', 'FRANCE', '18-Aug-2012', 650000),
(28, '3788b', 'INDIA', '21-Sept-2012', 5090),
(40, '4788a', 'US', '23-Sept-2012', 4950),
(40, '4788b', 'US', '09-Oct-2012', 15000),
(28, '4519a', 'INDIA', '18-Oct-2012', 5090),
(28, '4519b', 'INDIA', '02-Dec-2012', 5090);
```

A `SELECT` statement confirms that the rows are distributed among the subpartitions:

```
edb# SELECT tableoid::regclass, * FROM sales;
```

```
tableoid | dept_no | part_no | country | date            | amount
-----------------------+---------+---------+---------+-----------------+-------
sales_americas_2012   | 40      | 9519b   | US      | 12-APR-12 00:00:00 | 145000
sales_americas_2012   | 40      | 4577b   | US      | 11-NOV-12 00:00:00 | 25000
sales_americas_2012   | 30      | 7588b   | CANADA  | 14-DEC-12 00:00:00 | 50000
sales_americas_2012   | 30      | 9519b   | CANADA  | 01-FEB-12 00:00:00 | 75000
sales_americas_2012   | 30      | 4519b   | CANADA  | 08-APR-12 00:00:00 | 120000
sales_americas_2012   | 40      | 3788a   | US      | 12-MAY-12 00:00:00 | 4950
sales_americas_2012   | 40      | 4788a   | US      | 23-SEP-12 00:00:00 | 4950
sales_americas_2012   | 40      | 4788b   | US      | 09-OCT-12 00:00:00 | 15000
sales_europe_2012     | 10      | 4519b   | FRANCE  | 17-JAN-12 00:00:00 | 45000
sales_europe_2012     | 10      | 9519b   | ITALY   | 07-JUL-12 00:00:00 | 15000
sales_europe_2012     | 10      | 9519a   | FRANCE  | 18-AUG-12 00:00:00 | 650000
sales_europe_2012     | 10      | 9519b   | FRANCE  | 18-AUG-12 00:00:00 | 650000
sales_asia_2012       | 20      | 3788a   | INDIA   | 01-MAR-12 00:00:00 | 75000
sales_asia_2012       | 20      | 3788a   | PAKISTAN| 04-JUN-12 00:00:00 | 37500
sales_asia_2012       | 20      | 3788a   | INDIA   | 21-SEP-12 00:00:00 | 5090
sales_asia_2012       | 20      | 4519a   | INDIA   | 18-OCT-12 00:00:00 | 650000
sales_asia_2012       | 20      | 4519b   | INDIA   | 02-DEC-12 00:00:00 | 5090
(17 rows)
```

This command splits the `americas_2012` subpartition into two new subpartitions and redistributes the contents:

```
ALTER TABLE sales
SPLIT SUBPARTITION americas_2012
AT ('2012-Jun-01')
INTO
(
    SUBPARTITION americas_p1_2012,
    SUBPARTITION americas_p2_2012
);
```

After the command is invoked, the `americas_2012` subpartition is deleted. In its place, the server creates the subpartitions `americas_p1_2012` and `americas_p2_2012`:

```
edb# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
```

```
partition_name | subpartition_name | high_value
-----------------------+------------------+---------
```

731
Querying the sales table shows that the subpartition’s contents are redistributed:

```
edb# SELECT tableoid::regclass, * FROM sales;
```

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_americas_p1_2012</td>
<td>40</td>
<td>9519b</td>
<td>US</td>
<td>12-APR-12 00:00:00</td>
<td>145000</td>
</tr>
<tr>
<td>sales_americas_p1_2012</td>
<td>30</td>
<td>9519b</td>
<td>CANADA</td>
<td>01-FEB-12 00:00:00</td>
<td>75000</td>
</tr>
<tr>
<td>sales_americas_p1_2012</td>
<td>30</td>
<td>4519b</td>
<td>CANADA</td>
<td>08-APR-12 00:00:00</td>
<td>120000</td>
</tr>
<tr>
<td>sales_americas_p1_2012</td>
<td>40</td>
<td>3788a</td>
<td>US</td>
<td>12-MAY-12 00:00:00</td>
<td>4950</td>
</tr>
<tr>
<td>sales_americas_p2_2012</td>
<td>40</td>
<td>4577b</td>
<td>US</td>
<td>11-NOV-12 00:00:00</td>
<td>25000</td>
</tr>
<tr>
<td>sales_americas_p2_2012</td>
<td>30</td>
<td>7588b</td>
<td>CANADA</td>
<td>14-DEC-12 00:00:00</td>
<td>50000</td>
</tr>
<tr>
<td>sales_americas_p2_2012</td>
<td>40</td>
<td>4788a</td>
<td>US</td>
<td>23-SEP-12 00:00:00</td>
<td>4950</td>
</tr>
<tr>
<td>sales_americas_p2_2012</td>
<td>40</td>
<td>4788b</td>
<td>US</td>
<td>09-OCT-12 00:00:00</td>
<td>15000</td>
</tr>
<tr>
<td>sales_europe_2012</td>
<td>10</td>
<td>4519b</td>
<td>FRANCE</td>
<td>17-JAN-12 00:00:00</td>
<td>45000</td>
</tr>
<tr>
<td>sales_europe_2012</td>
<td>10</td>
<td>9519b</td>
<td>ITALY</td>
<td>07-JUL-12 00:00:00</td>
<td>15000</td>
</tr>
<tr>
<td>sales_europe_2012</td>
<td>10</td>
<td>9519a</td>
<td>FRANCE</td>
<td>18-AUG-12 00:00:00</td>
<td>650000</td>
</tr>
<tr>
<td>sales_europe_2012</td>
<td>10</td>
<td>9519b</td>
<td>FRANCE</td>
<td>18-AUG-12 00:00:00</td>
<td>650000</td>
</tr>
<tr>
<td>sales_asia_2012</td>
<td>20</td>
<td>3788a</td>
<td>INDIA</td>
<td>01-MAR-12 00:00:00</td>
<td>75000</td>
</tr>
<tr>
<td>sales_asia_2012</td>
<td>20</td>
<td>3788b</td>
<td>PAKISTAN</td>
<td>04-JUN-12 00:00:00</td>
<td>37500</td>
</tr>
<tr>
<td>sales_asia_2012</td>
<td>20</td>
<td>3788a</td>
<td>INDIA</td>
<td>18-OCT-12 00:00:00</td>
<td>5090</td>
</tr>
<tr>
<td>sales_asia_2012</td>
<td>20</td>
<td>4519a</td>
<td>INDIA</td>
<td>21-SEP-12 00:00:00</td>
<td>5090</td>
</tr>
<tr>
<td>sales_asia_2012</td>
<td>20</td>
<td>4519b</td>
<td>INDIA</td>
<td>02-DEC-12 00:00:00</td>
<td>5090</td>
</tr>
</tbody>
</table>

(17 rows)

14.3.2.6 ALTER TABLE...EXCHANGE PARTITION

The ALTER TABLE...EXCHANGE PARTITION command swaps an existing table with a partition. If you plan to add a large quantity of data to a partitioned table, you can use the ALTER TABLE...EXCHANGE PARTITION command to implement a bulk load. You can also use the ALTER TABLE...EXCHANGE PARTITION command to remove old or unneeded data for storage.

The command syntax is available in two forms. The first form swaps a table for a partition:

```
ALTER TABLE <target_table>
    EXCHANGE PARTITION <target_partition>
    WITH TABLE <source_table>
    [INCLUDING | EXCLUDING] INDEXES
    [WITH | WITHOUT] VALIDATION;
```

The second form swaps a table for a subpartition:

```
ALTER TABLE <target_table>
    EXCHANGE SUBPARTITION <target_subpartition>
    WITH TABLE <source_table>
    [INCLUDING | EXCLUDING] INDEXES
    [WITH | WITHOUT] VALIDATION;
```

Description

When the ALTER TABLE...EXCHANGE PARTITION command completes, the data originally located in the target_partition is located in the source_table. The data originally located in the source_table is located in the target_partition.

The ALTER TABLE...EXCHANGE PARTITION command can exchange partitions in a LIST, RANGE or HASH partitioned table. The structure of the source_table must match the structure of the target_table in that both tables must have matching columns and data types. The data in the table must adhere to the partitioning constraints.

If the INCLUDING INDEXES clause is specified with EXCHANGE PARTITION, then matching indexes in the target_partition and source_table are swapped. Indexes in the target_partition with no match in the source_table are rebuilt and vice versa. That is, indexes in the source_table with no match in the target_partition are also rebuilt.
If the **EXCLUDING INDEXES** clause is specified with **EXCHANGE PARTITION**, then matching indexes in the **target_partition** and **source_table** are swapped. However the **target_partition** indexes with no match in the **source_table** are marked as invalid and vice versa. That is, indexes in the **source_table** with no match in the **target_partition** are also marked as invalid.

The term matching index refers to indexes that have the same attributes. Attribute examples include the collation order, ascending or descending direction, ordering of nulls first or nulls last, and so forth as determined by the **CREATE INDEX** command.

If you omit both **INCLUDING INDEXES** and **EXCLUDING INDEXES**, then the default action is the **EXCLUDING INDEXES** behavior.

The same behavior described applies for the **target_subpartition** used with the **EXCHANGE SUBPARTITION** clause.

You must own a table to invoke **ALTER TABLE... EXCHANGE PARTITION** or **ALTER TABLE... EXCHANGE SUBPARTITION** against that table.

**Parameters**

**target_table**

The name (optionally schema-qualified) of the table in which the partition or subpartition resides.

**target_partition**

The name of the partition to replace.

**target_subpartition**

The name of the subpartition to replace.

**source_table**

The name of the table that replaces the **target_partition** or **target_subpartition**.

**14.3.2.6.1  Example: Exchanging a table for a partition**

This example swaps a table for the partition **americas** of the **sales** table. You can create the **sales** table with this command:

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Use this command to add sample data to the **sales** table:

```sql
INSERT INTO sales VALUES
(48, '9519b', 'US', '12-Apr-2012', 145000),
(18, '4519b', 'FRANCE', '17-Jan-2012', 45000),
(20, '3788a', 'INDIA', '01-Mar-2012', 75000),
(18, '3788a', 'PAKISTAN', '04-Jun-2012', 37500),
(18, '9519b', 'ITALY', '07-Jul-2012', 15000),
(18, '9519a', 'FRANCE', '18-Aug-2012', 650000),
(18, '9519b', 'FRANCE', '18-Aug-2012', 650000)
```
Querying the `sales` table shows that only one row resides in the `americas` partition:

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>9519b</td>
<td>US</td>
<td>12-APR-12 00:00:00</td>
<td>145000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>4519b</td>
<td>FRANCE</td>
<td>17-JAN-12 00:00:00</td>
<td>45000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>9519b</td>
<td>ITALY</td>
<td>07-JUL-12 00:00:00</td>
<td>15000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>9519b</td>
<td>FRANCE</td>
<td>18-AUG-12 00:00:00</td>
<td>650000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>9519b</td>
<td>FRANCE</td>
<td>18-AUG-12 00:00:00</td>
<td>650000</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788a</td>
<td>INDIA</td>
<td>01-MAR-12 00:00:00</td>
<td>75000</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788a</td>
<td>PAKISTAN</td>
<td>04-JUN-12 00:00:00</td>
<td>37500</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>3788b</td>
<td>INDIA</td>
<td>21-SEP-12 00:00:00</td>
<td>5090</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>4519a</td>
<td>INDIA</td>
<td>18-OCT-12 00:00:00</td>
<td>650000</td>
</tr>
<tr>
<td>sales_asia</td>
<td>20</td>
<td>4519b</td>
<td>INDIA</td>
<td>02-DEC-12 00:00:00</td>
<td>5090</td>
</tr>
</tbody>
</table>

This command creates a table `n_america` that matches the definition of the `sales` table:

```sql
CREATE TABLE n_america
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
);
```

This command adds data to the `n_america` table. The data conforms to the partitioning rules of the `americas` partition.

```sql
INSERT INTO n_america VALUES
(40, '9519b', 'US', '12-Apr-2012', '145000'),
(40, '4577b', 'US', '11-Nov-2012', '250000'),
(30, '7588b', 'CANADA', '14-Dec-2012', '75000'),
(30, '9519b', 'CANADA', '01-Feb-2012', '120000'),
(40, '3788a', 'US', '12-May-2012', '4950'),
(40, '4788a', 'US', '23-Sept-2012', '4950'),
(40, '4788b', 'US', '09-Oct-2012', '150000');
```

This command swaps the table into the partitioned table:

```sql
ALTER TABLE sales
EXCHANGE PARTITION americas
WITH TABLE n_america;
```

Querying the `sales` table shows that the contents of the `n_america` table were exchanged for the contents of the `americas` partition:

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>9519b</td>
<td>US</td>
<td>12-APR-12 00:00:00</td>
<td>145000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>4577b</td>
<td>US</td>
<td>11-NOV-12 00:00:00</td>
<td>250000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>30</td>
<td>7588b</td>
<td>CANADA</td>
<td>14-DEC-12 00:00:00</td>
<td>500000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>30</td>
<td>9519b</td>
<td>CANADA</td>
<td>01-FEB-12 00:00:00</td>
<td>750000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>30</td>
<td>4519b</td>
<td>CANADA</td>
<td>08-APR-12 00:00:00</td>
<td>120000</td>
</tr>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>3788a</td>
<td>US</td>
<td>12-MAY-12 00:00:00</td>
<td>4950</td>
</tr>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>4788a</td>
<td>US</td>
<td>23-SEP-12 00:00:00</td>
<td>4950</td>
</tr>
<tr>
<td>sales_americas</td>
<td>40</td>
<td>4788b</td>
<td>US</td>
<td>09-OCT-12 00:00:00</td>
<td>150000</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>4519b</td>
<td>FRANCE</td>
<td>08-APR-12 00:00:00</td>
<td>4950</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>9519b</td>
<td>FRANCE</td>
<td>09-OCT-12 00:00:00</td>
<td>4950</td>
</tr>
<tr>
<td>sales_europe</td>
<td>10</td>
<td>4519b</td>
<td>FRANCE</td>
<td>17-JAN-12 00:00:00</td>
<td>45000</td>
</tr>
</tbody>
</table>
Querying the `n_america` table shows that the row that was previously stored in the `americas` partition was moved to the `n_america` table:

```
<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_america</td>
<td>48</td>
<td>9519b</td>
<td>US</td>
<td>12-APR-12 08:00:00</td>
<td>145000</td>
</tr>
</tbody>
</table>
```

(1 row)

### 14.3.2.7 ALTER TABLE...MOVE PARTITION

Use the `ALTER TABLE... MOVE PARTITION` command to move a partition to a different tablespace. The command takes two forms.

The first form moves a partition to a new tablespace:

```
ALTER TABLE <table_name>
MOVE PARTITION <partition_name>
TABLESPACE <tablespace_name>;
```

The second form moves a subpartition to a new tablespace:

```
ALTER TABLE <table_name>
MOVE SUBPARTITION <subpartition_name>
TABLESPACE <tablespace_name>;
```

### Description

The `ALTER TABLE.. MOVE PARTITION` command moves a partition from its current tablespace to a different tablespace. The `ALTER TABLE.. MOVE PARTITION` command can move partitions of a `LIST`, `RANGE` or `HASH` partitioned table.

The same behavior applies for the `subpartition_name` used with the `MOVE SUBPARTITION` clause.

You must own the table to invoke `ALTER TABLE.. MOVE PARTITION` or `ALTER TABLE.. MOVE SUBPARTITION`.

### Parameters

- **table_name**: The name (optionally schema-qualified) of the table in which the partition or subpartition resides.
- **partition_name**: The name of the partition to move.
- **subpartition_name**: The name of the subpartition to move.
- **tablespace_name**: The name of the tablespace to which the partition or subpartition is moved.
14.3.2.7.1 Example: Moving a partition to a different tablespace

This following example moves a partition of the sales table from one tablespace to another.

Create the sales table with the command:

```
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY RANGE (date)
(
  PARTITION q1_2012 VALUES LESS THAN ('2012-Apr-01'),
  PARTITION q2_2012 VALUES LESS THAN ('2012-Jul-01'),
  PARTITION q3_2012 VALUES LESS THAN ('2012-Oct-01'),
  PARTITION q4_2012 VALUES LESS THAN ('2013-Jan-01') TABLESPACE ts_1,
  PARTITION q1_2013 VALUES LESS THAN ('2013-Mar-01') TABLESPACE ts_2
);
```

Querying the ALL_TAB_PARTITIONS view confirms that the partitions reside on the expected servers and tablespaces:

<table>
<thead>
<tr>
<th>partition_name</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_2012</td>
<td></td>
</tr>
<tr>
<td>Q2_2012</td>
<td></td>
</tr>
<tr>
<td>Q3_2012</td>
<td></td>
</tr>
<tr>
<td>Q4_2012</td>
<td>TS_1</td>
</tr>
<tr>
<td>Q1_2013</td>
<td>TS_2</td>
</tr>
</tbody>
</table>
(5 rows)

After preparing the target tablespace, invoke the ALTER TABLE...MOVE PARTITION command to move the q1_2013 partition from a tablespace named ts_2 to a tablespace named ts_3:

```
ALTER TABLE sales MOVE PARTITION q1_2013 TABLESPACE ts_3;
```

Querying the ALL_TAB_PARTITIONS view shows that the move was successful:

<table>
<thead>
<tr>
<th>partition_name</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_2012</td>
<td></td>
</tr>
<tr>
<td>Q2_2012</td>
<td></td>
</tr>
<tr>
<td>Q3_2012</td>
<td></td>
</tr>
<tr>
<td>Q4_2012</td>
<td>TS_1</td>
</tr>
<tr>
<td>Q1_2013</td>
<td>TS_3</td>
</tr>
</tbody>
</table>
(5 rows)

14.3.2.8 ALTER TABLE...RENAME PARTITION

Use the ALTER TABLE...RENAME PARTITION command to rename a table partition. The syntax takes two forms.

The first form renames a partition:

```
ALTER TABLE <table_name>
RENAME PARTITION <partition_name>
TO <new_name>;
```

The second form renames a subpartition:  

```
ALTER TABLE <table_name>
RENAME SUBPARTITION <subpartition_name>
TO <new_name>;

Description

The \texttt{ALTER TABLE... RENAME PARTITION} command renames a partition.

The behavior for \texttt{subpartition_name} is the same as for the same variable used with the \texttt{RENAME SUBPARTITION} clause.

You must own the specified table to invoke \texttt{ALTER TABLE... RENAME PARTITION} or \texttt{ALTER TABLE... RENAME SUBPARTITION}.

Parameters

- **table_name**
  The name (optionally schema-qualified) of the table in which the partition or subpartition resides.

- **partition_name**
  The name of the partition to rename.

- **subpartition_name**
  The name of the subpartition to rename.

- **new_name**
  The new name of the partition or subpartition.

14.3.2.8.1 Example: Renaming a partition

This command creates a list-partitioned table named \texttt{sales}:

```sql
CREATE TABLE sales
(
  dept_no  number,
  part_no  varchar2,
  country  varchar2(20),
  date     date,
  amount   number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Query the \texttt{ALL_TAB_PARTITIONS} view to display the partition names:

```sql
edbc=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
```

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
<tr>
<td>AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
</tbody>
</table>

(3 rows)

This command renames the \texttt{americas} partition to \texttt{n_america}:
Query the `ALL_TAB_PARTITIONS` view to show that the partition was successfully renamed:

```
SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
```

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
<tr>
<td>N_AMERICA</td>
<td>'US', 'CANADA'</td>
</tr>
</tbody>
</table>

(3 rows)

14.3.2.9 ALTER TABLE...SET INTERVAL

Use the `ALTER TABLE...SET INTERVAL` command to convert an existing range-partitioned table to an interval range-partitioned table. The database creates a partition of a specified range or interval for the partitioned table when you set `INTERVAL`. The syntax is:

```
ALTER TABLE <table_name> SET INTERVAL (<constant> | <expression>);
```

To disable an interval range partitioned table and convert it to a range-partitioned table, the syntax is:

```
ALTER TABLE <table_name> SET INTERVAL ();
```

Parameters

- `table_name`
  
  The name (optionally schema-qualified) of the range-partitioned table.

- `constant | expression`

  Specifies a `NUMERIC`, `DATE`, or `TIME` value.

Description

You can use the `ALTER TABLE...SET INTERVAL` command to convert the range-partitioned table to use interval range partitioning. The command creates a new partition of a specified interval. You can then insert data into the new partition.

You can use the `SET INTERVAL ()` command to disable interval range partitioning. The database converts an interval range-partitioned table to range partitioned and sets the boundaries of the interval range partitions to the boundaries for the range partitions.

14.3.2.9.1 Example: Setting an interval range partition

This example sets an interval range partition of the `sales` table. It converts the table from range partitioning to start using monthly interval range partitioning.

This command creates the `sales` table:

```
CREATE TABLE sales
(
  prod_id int,
  prod_quantity int,
  sold_month date
)
PARTITION BY RANGE (sold_month)
(
  PARTITION p1 VALUES LESS THAN ('15-JAN-2019'),
```
Set the interval range partitioning from the sales table:

```
ALTER TABLE sales SET INTERVAL (NUMTOYMINTERVAL(1, 'MONTH'));
```

Query the ALL_TAB_PARTITIONS view before a database creates an interval range partition:

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
partition_name | high_value
---------------+----------------------
P1             | '15-JAN-19 00:00:00'
P2             | '15-FEB-19 00:00:00'
(2 rows)
```

Add data to the sales table that exceeds the high value of a range partition:

```
edb=# INSERT INTO sales VALUES (1,100,'05-APR-2019');
```

Query the ALL_TAB_PARTITIONS view again after the INSERT statement. The interval range partition is successfully created and data is inserted. A system-generated name of the interval range partition is created that varies for each session.

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
partition_name | high_value
---------------+----------------------
P1             | '15-JAN-19 00:00:00'
P2             | '15-FEB-19 00:00:00'
SYS916348103   | '15-APR-19 00:00:00'
(3 rows)
```

### 14.3.2.10 ALTER TABLE...SET [ PARTITIONING ] AUTOMATIC

Use the `ALTER TABLE...SET [ PARTITIONING ] AUTOMATIC` command to convert an existing list-partitioned table to automatic list partitioning. The database automatically creates a partition based on a new value inserted into a table when `AUTOMATIC` is set. The syntax is:

```
ALTER TABLE <table_name> SET [ PARTITIONING ]
AUTOMATIC;
```

To disable `AUTOMATIC LIST PARTITIONING` and convert to regular `LIST` partition, the syntax is:

```
ALTER TABLE <table_name> SET [ PARTITIONING ]
MANUAL;
```

#### Parameters

- **table_name**

  The name (optionally schema-qualified) of the list-partitioned table.

#### Description

You can use the `ALTER TABLE...SET [ PARTITIONING ] AUTOMATIC` command to convert the regular list-partitioned table to use automatic partitioning. A partition is created, and you can insert data into the new partition.

You can use the `ALTER TABLE...SET [ PARTITIONING ] MANUAL` command to disable the automatic list partitioning. The database converts an automatic list-partitioned table to a regular list-partitioned table.
### 14.3.2.10.1 Example: Setting an AUTOMATIC list partition

This example modifies a table `sales` to use automatic list partitioning instead of regular list partitioning.

This command creates a `sales` table:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    sales_state varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST(sales_state)
(
    PARTITION P_KAN VALUES('KANSAS'),
    PARTITION P_TEX VALUES('TEXAS')
);
```

Implement automatic list partitioning on the `sales` table:

```sql
ALTER TABLE sales SET AUTOMATIC;
```

Query the `ALL_TAB_PARTITIONS` view to show that an existing partition is successfully created:

```sql
edb=# SELECT table_name, partition_name, high_value 
FROM ALL_TAB_PARTITIONS;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>P_KAN</td>
<td>'KANSAS'</td>
</tr>
<tr>
<td>SALES</td>
<td>P_TEX</td>
<td>'TEXAS'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(2 rows)

Insert data into the `sales` table to create a partition and add the value:

```sql
edb=# INSERT INTO sales VALUES (1, 'VIR', 'VIRGINIA');
```

Query the `ALL_TAB_PARTITIONS` view again after the insert. The automatic list partition is successfully created and data is inserted. A system-generated name of the partition is created that varies for each session.

```sql
edb=# SELECT table_name, partition_name, high_value 
FROM ALL_TAB_PARTITIONS;
```

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>P_KAN</td>
<td>'KANSAS'</td>
</tr>
<tr>
<td>SALES</td>
<td>P_TEX</td>
<td>'TEXAS'</td>
</tr>
<tr>
<td>SALES</td>
<td>SYS106900103</td>
<td>'VIRGINIA'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(3 rows)

### 14.3.2.11 ALTER TABLE...SET SUBPARTITION TEMPLATE

Use the `ALTER TABLE...SET SUBPARTITION TEMPLATE` command to update the subpartition template for a table. The command syntax comes in several forms.

To set the subpartition number for `HASH` subpartitions only:

```sql
ALTER TABLE <table_name> SET SUBPARTITION TEMPLATE <num>;
```

To set a subpartition description for an existing partitioned table:

```sql
ALTER TABLE <table_name> 
[SET SUBPARTITION TEMPLATE 
[<subpartition_template_description>]]
```

To reset the subpartitions number to default using the subpartition template:

```sql
ALTER TABLE <table_name> 
[RESET SUBPARTITION TEMPLATE]
```
### Parameters

- **table_name**
  The name (optionally schema-qualified) of the partitioned table.

- **num**
  The number of subpartitions to add for a partition.

- **subpartition_template_description**
  The list of subpartition details with subpartition name, subpartition bound, and tablespace name.

### Description

You can use the `ALTER TABLE... SET SUBPARTITION TEMPLATE` command to update the subpartition template for range, list, or hash-partitioned table. If you're specifying a subpartition descriptor for a partition, use a subpartition descriptor instead of a subpartition template. You can use the subpartition template whenever a subpartition descriptor isn't specified for a partition. If you don't specify either the subpartition descriptor or subpartition template, then by default a single subpartition is created.

For more information about creating a subpartition template, see [CREATE TABLE...PARTITION BY](#).

### Note

The partitions added to a table after invoking `ALTER TABLE... SET SUBPARTITION TEMPLATE` command use the new `SUBPARTITION TEMPLATE`.

You can use the `ALTER TABLE... SET SUBPARTITION TEMPLATE ()` command to reset the subpartitions number to default 1.

### 14.3.2.11 Example: Setting a SUBPARTITION TEMPLATE

This example creates a table `sales` that's range partitioned by `date` and hash-subpartitioned by `country`.

This command creates the `sales` table:

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY RANGE (date) SUBPARTITION BY HASH (country) SUBPARTITIONS 2
(
  PARTITION q1_2012 VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012 VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012 VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012 VALUES LESS THAN('2013-Jan-01')
);
```

The table definition creates four partitions: `q1_2012`, `q2_2012`, `q3_2012`, and `q4_2012`. Each partition consists of two subpartitions with system-generated names.

```sql
edb=#
SELECT table_name, partition_name, subpartition_name
FROM ALL_TAB_SUBPARTITIONS
WHERE table_name = 'SALES' ORDER BY 1,2;
```

| table_name | partition_name | subpartition_name |
Set the subpartition template on the `sales` table:

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE 8;
```

The `sales` table is modified, with the subpartition template set to eight. If you try to add a partition `q1_2013`, a new partition is created and consists of eight subpartitions.

```
ALTER TABLE sales ADD PARTITION q1_2013 VALUES LESS THAN ('2013-Apr-01');
```

Query the `ALL_TAB_PARTITIONS` view. The `q1_2013` partition is successfully added. It has eight subpartitions that have system-generated names.

<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>Q1_2013</td>
<td>SYS0113</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2013</td>
<td>SYS0114</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2013</td>
<td>SYS0115</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2013</td>
<td>SYS0116</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2013</td>
<td>SYS0117</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2013</td>
<td>SYS0118</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2013</td>
<td>SYS0119</td>
</tr>
<tr>
<td>SALES</td>
<td>Q1_2013</td>
<td>SYS0120</td>
</tr>
</tbody>
</table>

(8 rows)

Example: Adding a subpartition template for LIST/LIST partitioned table

This example creates a table `sales` that's list-partitioned by country. It is subpartitioned using the list by the `date` column.

```
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST (country)
    SUBPARTITION BY LIST (date)
    SUBPARTITION TEMPLATE
    {
        SUBPARTITION europe VALUES('2021-Jan-01') TABLESPACE ts1,
        SUBPARTITION asia VALUES('2021-Apr-01') TABLESPACE ts2,
        SUBPARTITION americas VALUES('2021-Jul-01') TABLESPACE ts3
    }
    (PARTITION q1_2021 VALUES('2021-Jul-01'))
);
```

The `SELECT` statement shows partition `q1_2021` consisting of three subpartitions stored in tablespaces `ts1`, `ts2`, and `ts3`.

```
edb# SELECT partition_name, subpartition_name, high_value, tablespace_name FROM DBA_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

(8 rows)
This command adds a partition named **q2_2021** to the **sales** table:

```sql
ALTER TABLE sales ADD PARTITION q2_2021 VALUES('US', 'CANADA');
```

This command shows that the **sales** table includes the **q2_2021** partition:

```sql
edb=# SELECT partition_name, subpartition_name, high_value, tablespace_name FROM DBA_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

Example: Adding a subpartition template for LIST/RANGE partitioned table

This example creates a table **sales** list-partitioned by country and subpartitioned using range partitioning by the **date** column:

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
) PARTITION BY LIST(country)
  SUBPARTITION BY RANGE(date)
  SUBPARTITION TEMPLATE
  (
    SUBPARTITION europe VALUES LESS THAN('2021-Jan-01') TABLESPACE ts1,
    SUBPARTITION asia VALUES LESS THAN('2021-Apr-01') TABLESPACE ts2,
    SUBPARTITION americas VALUES LESS THAN('2021-Jul-01') TABLESPACE ts3
  )
  (
    PARTITION q1_2021 VALUES ('2021-Jul-01')
  );
```

The **sales** table creates a partition named **q1_2021** that includes three subpartitions stored in tablespaces **ts1**, **ts2**, and **ts3**.

```sql
edb=# SELECT partition_name, subpartition_name, high_value, tablespace_name FROM DBA_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

This command adds a partition named **q2_2021** to the **sales** table:

```sql
ALTER TABLE sales ADD PARTITION q2_2021 VALUES('INDIA', 'PAKISTAN');
```

This command shows that the **sales** table includes the **q2_2021** partition:
<table>
<thead>
<tr>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_AMERICAS</td>
<td>'01-JUL-21 00:00:00'</td>
<td>TS3</td>
</tr>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_ASIA</td>
<td>'01-APR-21 00:00:00'</td>
<td>TS2</td>
</tr>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_EUROPE</td>
<td>'01-JAN-21 00:00:00'</td>
<td>TS1</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_AMERICAS</td>
<td>'01-JUL-21 00:00:00'</td>
<td>TS3</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_ASIA</td>
<td>'01-APR-21 00:00:00'</td>
<td>TS2</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_EUROPE</td>
<td>'01-JAN-21 00:00:00'</td>
<td>TS1</td>
</tr>
</tbody>
</table>

Example: Adding a subpartition template for LIST/HASH partitioned table

This example creates a list-partitioned table `sales` that's first partitioned by country and then hash-subpartitioned using the value of the `dept_no` column:

```
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST(country)
    SUBPARTITION BY HASH (dept_no)
    SUBPARTITION TEMPLATE
    {
        SUBPARTITION europe TABLESPACE ts1,
        SUBPARTITION asia TABLESPACE ts2,
        SUBPARTITION americas TABLESPACE ts3
    }
    {
        PARTITION q1_2021 VALUES ('2021-Jul-01')
    };
```

The `sales` table creates a `q1_2021` partition that includes three subpartitions stored in tablespaces `ts1`, `ts2`, and `ts3`:

```
<table>
<thead>
<tr>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_AMERICAS</td>
<td></td>
<td>TS3</td>
</tr>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_ASIA</td>
<td></td>
<td>TS2</td>
</tr>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_EUROPE</td>
<td></td>
<td>TS1</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_AMERICAS</td>
<td></td>
<td>TS3</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_ASIA</td>
<td></td>
<td>TS2</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_EUROPE</td>
<td></td>
<td>TS1</td>
</tr>
</tbody>
</table>
```

This command adds a partition named `q2_2021` to the `sales` table:

```
ALTER TABLE sales ADD PARTITION q2_2021 VALUES('FRANCE', 'ITALY');
```

This command shows that the `sales` table includes the `q2_2021` partition:

```
<table>
<thead>
<tr>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_AMERICAS</td>
<td></td>
<td>TS3</td>
</tr>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_ASIA</td>
<td></td>
<td>TS2</td>
</tr>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_EUROPE</td>
<td></td>
<td>TS1</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_AMERICAS</td>
<td></td>
<td>TS3</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_ASIA</td>
<td></td>
<td>TS2</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_EUROPE</td>
<td></td>
<td>TS1</td>
</tr>
</tbody>
</table>
(6 rows)
```
Examples: Resetting a SUBPARTITION TEMPLATE

This example creates a list-partitioned table `sales` that's list partitioned by `country` and hash subpartitioned by `part_no`.

This command creates the `sales` table:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST (country) SUBPARTITION BY HASH (part_no) SUBPARTITIONS 3
(
    PARTITION europe VALUES('FRANCE',
                                'ITALY'),
    PARTITION asia VALUES('INDIA', 'PAKISTAN'),
    PARTITION americas VALUES('US', 'CANADA')
);
```

The table contains three partitions: `americas`, `asia`, and `europe`. Each partition consists of three subpartitions with system-generated names.

```sql
edb# SELECT table_name, partition_name, subpartition_name
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

```
+----------+----------------+-------------------+
<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0109</td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0107</td>
</tr>
<tr>
<td>SALES</td>
<td>AMERICAS</td>
<td>SYS0108</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0105</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0104</td>
</tr>
<tr>
<td>SALES</td>
<td>ASIA</td>
<td>SYS0106</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0101</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0103</td>
</tr>
<tr>
<td>SALES</td>
<td>EUROPE</td>
<td>SYS0102</td>
</tr>
</tbody>
</table>
```

(9 rows)

This command resets the subpartition template on the `sales` table:

```sql
ALTER TABLE sales SET SUBPARTITION TEMPLATE ();
```

The `sales` table is modified with the subpartition template reset to default 1. Try to add a new partition `east_asia` using this command:

```sql
ALTER TABLE sales ADD PARTITION east_asia VALUES ('CHINA',
                                                       'KOREA');
```

Query the `ALL_TAB_PARTITIONS` view. A new partition `east_asia` is created consisting of one subpartition with a system-generated name.

```sql
edb# SELECT table_name, partition_name, subpartition_name
FROM ALL_TAB_SUBPARTITIONS WHERE table_name = 'SALES' and partition_name = 'EAST_ASIA' ORDER BY 1,2;
```

```
+----------+------------+---------------+
<table>
<thead>
<tr>
<th>table_name</th>
<th>partition_name</th>
<th>subpartition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES</td>
<td>EAST_ASIA</td>
<td>SYS0113</td>
</tr>
</tbody>
</table>
```

(1 row)

This example creates a table `sales` list-partitioned by country and subpartitioned using range partitioning by the `date` column:

```sql
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY RANGE (date) INTERVAL ('1 DAY') SUBPARTITIONS 3
(
    PARTITION europe VALUES ('US',
                               'CANADA'),
    PARTITION asia VALUES ('INDIA', 'PAKISTAN'),
    PARTITION americas VALUES ('FRANCE',
                                'ITALY')
);
```

This command resets the subpartition template on the `sales` table:

```sql
ALTER TABLE sales SET SUBPARTITION TEMPLATE ();
```

The `sales` table is modified with the subpartition template reset to default 1. Try to add a new partition `east_asia` using this command:

```sql
ALTER TABLE sales ADD PARTITION east_asia VALUES ('CHINA',
                                                       'KOREA');
```

Query the `ALL_TAB_PARTITIONS` view. A new partition `east_asia` is created consisting of one subpartition with a system-generated name.
PARTITION BY LIST (country)
  SUBPARTITION BY RANGE (date)

SUBPARTITION TEMPLATE
{
  SUBPARTITION europe VALUES LESS THAN ('2021-Jan-01') TABLESPACE ts1,
  SUBPARTITION asia VALUES LESS THAN ('2021-Apr-01') TABLESPACE ts2,
  SUBPARTITION americas VALUES LESS THAN ('2021-Jul-01') TABLESPACE ts3
}

PARTITION q3_2021 VALUES ('2021-Jul-01')
);

The `sales` table contains a partition named q1_2021 that includes three subpartitions stored in tablespaces ts1, ts2, and ts3.

```
edb# SELECT partition_name, subpartition_name, high_value, tablespace_name
FROM dba_tab_subpartitions WHERE table_name = 'SALES' ORDER BY 1, 2;

 partition_name | subpartition_name | high_value  | tablespace_name
----------------+-------------------+-------------+------------------
 Q1_2021        | Q1_2021_AMERICAS  | '01-JUL-21 00:00:00' | TS3
 Q1_2021        | Q1_2021_ASIA      | '01-APR-21 00:00:00' | TS2
 Q1_2021        | Q1_2021_EUROPE    | '01-JAN-21 00:00:00' | TS1
(3 rows)
```

This command adds a partition named q2_2021 to the `sales` table:

```
ALTER TABLE sales ADD PARTITION q2_2021 VALUES ('INDIA', 'PAKISTAN');
```

This command shows that the `sales` table includes the q2_2021 partition:

```
edb# SELECT partition_name, subpartition_name, high_value, tablespace_name
FROM dba_tab_subpartitions WHERE table_name = 'SALES' ORDER BY 1, 2;

 partition_name | subpartition_name | high_value  | tablespace_name
----------------+-------------------+-------------+------------------
 Q1_2021        | Q1_2021_AMERICAS  | '01-JUL-21 00:00:00' | TS3
 Q1_2021        | Q1_2021_ASIA      | '01-APR-21 00:00:00' | TS2
 Q1_2021        | Q1_2021_EUROPE    | '01-JAN-21 00:00:00' | TS1
 Q2_2021        | Q2_2021_AMERICAS  | '01-JUL-21 00:00:00' | TS3
 Q2_2021        | Q2_2021_ASIA      | '01-APR-21 00:00:00' | TS2
 Q2_2021        | Q2_2021_EUROPE    | '01-JAN-21 00:00:00' | TS1
(6 rows)
```

Use the `ALTER TABLE... SET SUBPARTITION TEMPLATE` command to specify a new subpartition template:

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE
{
  SUBPARTITION europe VALUES LESS THAN ('2021-Jan-01'),
  SUBPARTITION asia VALUES LESS THAN ('2021-Apr-01') TABLESPACE ts2
}
```

This command adds a partition named q3_2021 to the `sales` table:

```
ALTER TABLE sales ADD PARTITION q3_2021 VALUES ('US', 'CANADA');
```

This command shows that the `sales` table includes the q3_2021 partition:

```
edb# SELECT partition_name, subpartition_name, high_value, tablespace_name
FROM dba_tab_subpartitions WHERE table_name = 'SALES' ORDER BY 1, 2;

 partition_name | subpartition_name | high_value  | tablespace_name
----------------+-------------------+-------------+------------------
 Q1_2021        | Q1_2021_AMERICAS  | '01-JUL-21 00:00:00' | TS3
 Q1_2021        | Q1_2021_ASIA      | '01-APR-21 00:00:00' | TS2
 Q1_2021        | Q1_2021_EUROPE    | '01-JAN-21 00:00:00' | TS1
 Q2_2021        | Q2_2021_AMERICAS  | '01-JUL-21 00:00:00' | TS3
 Q2_2021        | Q2_2021_ASIA      | '01-APR-21 00:00:00' | TS2
 Q2_2021        | Q2_2021_EUROPE    | '01-JAN-21 00:00:00' | TS1
 Q3_2021        | Q3_2021_ASIA      | '01-APR-21 00:00:00' | TS2
 Q3_2021        | Q3_2021_EUROPE    | '01-JAN-21 00:00:00' | PG_DEFAULT
(8 rows)
```
This command resets or drops the subpartition template on the `sales` table:

```
ALTER TABLE sales SET SUBPARTITION TEMPLATE ();
```

This command adds a partition named `q4_2021` to the `sales` table:

```
ALTER TABLE sales ADD PARTITION q4_2021 VALUES('FRANCE', 'ITALY');
```

The `SELECT` statement shows partition `q4_2021` consists of a subpartition with a system-generated name:

```
SELECT partition_name, subpartition_name, high_value, tablespace_name FROM DBA_TAB_SUBPARTITIONS WHERE table_name = 'SALES' ORDER BY 1,2;
```

```
<table>
<thead>
<tr>
<th>partition_name</th>
<th>subpartition_name</th>
<th>high_value</th>
<th>tablespace_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_AMERICAS</td>
<td>'01-JUL-21 00:00:00'</td>
<td>TS3</td>
</tr>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_ASIA</td>
<td>'01-APR-21 00:00:00'</td>
<td>TS2</td>
</tr>
<tr>
<td>Q1_2021</td>
<td>Q1_2021_EUROPE</td>
<td>'01-JAN-21 00:00:00'</td>
<td>TS1</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_AMERICAS</td>
<td>'01-JUL-21 00:00:00'</td>
<td>TS3</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_ASIA</td>
<td>'01-APR-21 00:00:00'</td>
<td>TS2</td>
</tr>
<tr>
<td>Q2_2021</td>
<td>Q2_2021_EUROPE</td>
<td>'01-JAN-21 00:00:00'</td>
<td>TS1</td>
</tr>
<tr>
<td>Q3_2021</td>
<td>Q3_2021_ASIA</td>
<td>'01-APR-21 00:00:00'</td>
<td>TS2</td>
</tr>
<tr>
<td>Q3_2021</td>
<td>Q3_2021_EUROPE</td>
<td>'01-JAN-21 00:00:00'</td>
<td>PG_DEFAULT</td>
</tr>
<tr>
<td>Q4_2021</td>
<td>SYS0112</td>
<td>MAXVALUE</td>
<td>PG_DEFAULT</td>
</tr>
</tbody>
</table>
```

(9 rows)

### 14.3.2.12 DROP TABLE

Use the PostgreSQL `DROP TABLE` command to remove a partitioned table definition, its partitions and subpartitions, and the table contents. The syntax is:

```
DROP TABLE <table_name>
```

**Parameters**

- **`table_name`**

  The name (optionally schema-qualified) of the partitioned table.

**Description**

The `DROP TABLE` command removes an entire table and the data that resides in that table. Deleting a table deletes any partitions or subpartitions of that table.

To use the `DROP TABLE` command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

**Example**

To delete a table, connect to the controller node (the host of the partitioning root), and invoke the `DROP TABLE` command. For example, delete the `sales` table:

```
DROP TABLE sales;
```

The server confirms that the table was dropped:

```
edb# drop table sales;
DROP TABLE
edb#
```

For more information about the `DROP TABLE` command, see the PostgreSQL core documentation.
14.3.2.13 ALTER TABLE...DROP PARTITION

Use the `ALTER TABLE... DROP PARTITION` command to delete a partition definition and the data stored in that partition. The syntax is:

```
ALTER TABLE <table_name> DROP PARTITION <partition_name>;
```

Parameters

- **table_name**
  
The name (optionally schema-qualified) of the partitioned table.

- **partition_name**
  
The name of the partition to delete.

Description

The `ALTER TABLE... DROP PARTITION` command deletes a partition and any data stored on that partition. The `ALTER TABLE... DROP PARTITION` command can drop partitions of a `LIST` or `RANGE` partitioned table. This command doesn't work on a `HASH` partitioned table. Deleting a partition deletes its subpartitions.

To use the `DROP PARTITION` clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have database superuser or administrative privileges.

14.3.2.13.1 Example: Deleting a partition

This example deletes a partition of the `sales` table.

Create the `sales` table:

```sql
CREATE TABLE sales
(
    dept_no  number,
    part_no  varchar2,
    country  varchar2(20),
    date     date,
    amount   number
)
PARTITION BY LIST(country)
(
    PARTITION europe VALUES('FRANCE', 'ITALY'),
    PARTITION asia VALUES('INDIA', 'PAKISTAN'),
    PARTITION americas VALUES('US', 'CANADA')
);
```

Query the `ALL_TAB_PARTITIONS` view to display the partition names:

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;

<table>
<thead>
<tr>
<th>partition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
</tr>
<tr>
<td>ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
</tr>
<tr>
<td>AMERICAS</td>
<td>'US', 'CANADA'</td>
</tr>
</tbody>
</table>
```

(3 rows)

Delete the `americas` partition from the `sales` table:

```
ALTER TABLE sales DROP PARTITION americas;
```

Query the `ALL_TAB_PARTITIONS` view to show that the partition was successfully deleted:
14.3.2.14  ALTER TABLE...DROP SUBPARTITION

Use the `ALTER TABLE... DROP SUBPARTITION` command to drop a subpartition definition and the data stored in that subpartition. The syntax is:

```
ALTER TABLE <table_name> DROP SUBPARTITION <subpartition_name>;
```

**Parameters**

- **table_name**
  
The name (optionally schema-qualified) of the partitioned table.

- **subpartition_name**
  
The name of the subpartition to delete.

**Description**

The `ALTER TABLE... DROP SUBPARTITION` command deletes a subpartition and the data stored in that subpartition. To use the `DROP SUBPARTITION` clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have superuser or administrative privileges.

14.3.2.14.1  Example: Deleting a subpartition

This example deletes a subpartition of the `sales` table.

Create the `sales` table:

```sql
CREATE TABLE sales
(
  dept_no  number,
  part_no  varchar2,
  country  varchar2(20),
  date     date,
  amount   number
) PARTITION BY RANGE(date)
  SUBPARTITION BY LIST (country)
  (PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012'))
  (PARTITION europe VALUES ('ITALY', 'FRANCE'),
   SUBPARTITION americas VALUES ('CANADA', 'US'),
   SUBPARTITION asia VALUES ('PAKISTAN', 'INDIA')
  )
  (PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013'))
);
```

Query the `ALL_TAB_SUBPARTITIONS` view to display the subpartition names:

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
---------------+---------------------
 EUROPE         | 'FRANCE', 'ITALY'
 ASIA           | 'INDIA', 'PAKISTAN'
(2 rows)
```
Delete the americas subpartition from the sales table:

```
ALTER TABLE sales DROP SUBPARTITION americas;
```

Query the `ALL_TAB_SUBPARTITIONS` view to show that the subpartition was successfully deleted:

```
SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
```

### 14.3.2.15 TRUNCATE TABLE

Use the `TRUNCATE TABLE` command to remove the contents of a table while preserving the table definition. Truncating a table also truncates the table’s partitions or subpartitions. The syntax is:

```
TRUNCATE TABLE <table_name>
```

**Parameters**

- **table_name**
  
The name (optionally schema-qualified) of the partitioned table.

**Description**

The `TRUNCATE TABLE` command removes an entire table and the data that resides in that table. Deleting a table deletes any partitions or subpartitions of that table.

To use the `TRUNCATE TABLE` command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

### 14.3.2.15.1 Example: Emptying a table

This example removes the data from the `sales` table.

Create the `sales` table:

```
CREATE TABLE sales
(
    dept_no number,
    part_no varchar2,
    country varchar2(20),
    date date,
    amount number
)
PARTITION BY LIST(country)
(
    PARTITION europe VALUES('FRANCE', 'ITALY'),
    PARTITION us VALUES('CANADA', 'US'),
    PARTITION europe VALUES('FRANCE', 'ITALY'),
    PARTITION us VALUES('CANADA', 'US'),
    PARTITION india VALUES('PAKISTAN', 'INDIA'),
    PARTITION default
);
```
Populate the `sales` table:

```sql
INSERT INTO sales VALUES 
(18, '4519b', 'FRANCE', '17-Jan-2012', 145000),
(28, '3788a', 'INDIA', '01-Mar-2012', 75000),
(40, '9519b', 'US', '12-Apr-2012', 145000),
(28, '3788a', 'PAKISTAN', '04-Jun-2012', 137500),
(40, '4788b', 'US', '11-Nov-2012', 25000),
(38, '7588b', 'CANADA', '14-Dec-2012', 50000),
(38, '9519b', 'CANADA', '01-Feb-2012', 75000),
(38, '4519b', 'CANADA', '08-Apr-2012', 120000),
(48, '3788a', 'US', '12-May-2012', 4950),
(18, '9519b', 'ITALY', '07-Jul-2012', 5090),
(18, '9519b', 'FRANCE', '18-Aug-2012', 650000),
(28, '3788b', 'INDIA', '21-Sept-2012', 5090),
(48, '4788a', 'US', '23-Sept-2012', 4950),
(48, '4788b', 'US', '09-Oct-2012', 5090),
(20, '3788a', 'PAKISTAN', '18-Oct-2012', 650000),
(20, '4519b', 'INDIA', '2-Dec-2012', 5090);
```

Query the `sales` table to show that the partitions have data:

```sql
SELECT tableoid:regclass, * FROM sales;
```

Delete the contents of the `sales` table:

```sql
TRUNCATE TABLE sales;
```

Query the `sales` table to show that the data was removed but the structure is intact:

```sql
edb# SELECT tableoid::regclass, * FROM sales;
```

(0 rows)
For more information about the `TRUNCATE TABLE` command, see the PostgreSQL documentation.

### 14.3.2.16 ALTER TABLE...TRUNCATE PARTITION

Use the `ALTER TABLE...TRUNCATE PARTITION` command to remove the data from the specified partition and leave the partition structure intact. The syntax is:

```
ALTER TABLE <table_name> TRUNCATE PARTITION <partition_name>
[[DROP|REUSE] STORAGE]
```

**Parameters**

- **table_name**
  - The name (optionally schema-qualified) of the partitioned table.

- **partition_name**
  - The name of the partition to delete.

**Description**

Use the `ALTER TABLE...TRUNCATE PARTITION` command to remove the data from the specified partition and leave the partition structure intact. Truncating a partition also truncates its subpartitions. `ALTER TABLE...TRUNCATE PARTITION` doesn't cause `ON DELETE` triggers for the table to fire, but it fires `ON TRUNCATE` triggers. If an `ON TRUNCATE` trigger is defined for the partition, all `BEFORE TRUNCATE` triggers are fired before any truncation happens. All `AFTER TRUNCATE` triggers are fired after the last truncation occurs.

You must have the `TRUNCATE` privilege on a table to invoke `ALTER TABLE...TRUNCATE PARTITION`. `DROP STORAGE` and `REUSE STORAGE` are included only for compatibility. The clauses are parsed and ignored.

#### 14.3.2.16.1 Example: Emptying a partition

This example removes the data from a partition of the `sales` table.

Create the `sales` table:

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no varchar2,
  country varchar2(20),
  date date,
  amount number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Populate the `sales` table:

```sql
INSERT INTO sales VALUES
(18, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
(20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
(28, '3788a', 'INDIA', '01-Mar-2012', '75000'),
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Query the `sales` table to show that the partitions are populated with data:

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>9519b</td>
<td>US</td>
<td>12-Apr-2012 00:00:00</td>
<td>145000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4577b</td>
<td>US</td>
<td>11-Nov-2012</td>
<td>25000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7588b</td>
<td>CANADA</td>
<td>14-Dec-2012</td>
<td>50000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9519b</td>
<td>CANADA</td>
<td>01-Feb-2012</td>
<td>75000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4519b</td>
<td>CANADA</td>
<td>08-Apr-2012</td>
<td>120000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3788a</td>
<td>US</td>
<td>12-May-2012</td>
<td>4950</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4788a</td>
<td>US</td>
<td>23-Sep-2012</td>
<td>4950</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4788b</td>
<td>US</td>
<td>09-Oct-2012</td>
<td>15000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9519b</td>
<td>ITALY</td>
<td>07-Jul-2012</td>
<td>15000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9519a</td>
<td>FRANCE</td>
<td>18-Aug-2012</td>
<td>650000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9519b</td>
<td>FRANCE</td>
<td>18-Aug-2012</td>
<td>650000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3788a</td>
<td>INDIA</td>
<td>01-Mar-2012</td>
<td>75000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3788a</td>
<td>PAKISTAN</td>
<td>04-Jun-2012</td>
<td>37500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3788b</td>
<td>INDIA</td>
<td>21-Sep-2012</td>
<td>5090</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4519a</td>
<td>INDIA</td>
<td>18-Oct-2012</td>
<td>650000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4519b</td>
<td>INDIA</td>
<td>02-Dec-2012</td>
<td>5090</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(17 rows)</td>
</tr>
</tbody>
</table>

Delete the contents of the `americas` partition:

```
ALTER TABLE sales TRUNCATE PARTITION americas;
```

Query the `sales` table to show that the contents of the `americas` partition were removed:

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4519b</td>
<td>FRANCE</td>
<td>17-Jan-2012</td>
<td>45000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9519b</td>
<td>ITALY</td>
<td>07-Jul-2012</td>
<td>15000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9519a</td>
<td>FRANCE</td>
<td>18-Aug-2012</td>
<td>650000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9519b</td>
<td>FRANCE</td>
<td>18-Aug-2012</td>
<td>650000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3788a</td>
<td>INDIA</td>
<td>01-Mar-2012</td>
<td>75000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3788a</td>
<td>PAKISTAN</td>
<td>04-Jun-2012</td>
<td>37500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3788b</td>
<td>INDIA</td>
<td>21-Sep-2012</td>
<td>5090</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4519a</td>
<td>INDIA</td>
<td>18-Oct-2012</td>
<td>650000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4519b</td>
<td>INDIA</td>
<td>02-Dec-2012</td>
<td>5090</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(9 rows)</td>
</tr>
</tbody>
</table>

The rows were removed, but the structure of the `americas` partition is intact.
**14.3.2.17 ALTER TABLE...TRUNCATE SUBPARTITION**

Use the `ALTER TABLE...TRUNCATE SUBPARTITION` command to remove all of the data from the specified subpartition and leave the subpartition structure intact. The syntax is:

```
ALTER TABLE <table_name>
    TRUNCATE SUBPARTITION <subpartition_name>
    [[DROP|REUSE] STORAGE]
```

**Parameters**

- **table_name**
  The name (optionally schema-qualified) of the partitioned table.

- **subpartition_name**
  The name of the subpartition to truncate.

**Description**

The `ALTER TABLE...TRUNCATE SUBPARTITION` command removes all data from a specified subpartition, leaving the subpartition structure intact. The command doesn't cause `ON DELETE` triggers for the table to fire, but it fires `ON TRUNCATE` triggers. If an `ON TRUNCATE` trigger is defined for the subpartition, all `BEFORE TRUNCATE` triggers are fired before any truncation happens. All `AFTER TRUNCATE` triggers are fired after the last truncation occurs.

You must have the `TRUNCATE` privilege on a table to invoke `ALTER TABLE...TRUNCATE SUBPARTITION`. The `DROP STORAGE` and `REUSE STORAGE` clauses are included only for compatibility. The clauses are parsed and ignored.

**14.3.2.17.1 Example: Emptying a subpartition**

This example removes the data from a subpartition of the `sales` table.

Create the `sales` table:

```sql
CREATE TABLE sales (    dept_no number,    part_no varchar2,    country varchar2(20),    date date,    amount number )    PARTITION BY RANGE(date) SUBPARTITION BY LIST (country) (        PARTITION "2011" VALUES LESS THAN ('01-JAN-2012') (            SUBPARTITION europe_2011 VALUES ('ITALY', 'FRANCE'),            SUBPARTITION asia_2011 VALUES ('PAKISTAN', 'INDIA')        )    )
```
Populate the `sales` table:

```
INSERT INTO sales VALUES
(18, '4519b', 'FRANCE', '17-Jan-2011', '45000'),
(28, '3788a', 'INDIA', '01-Mar-2012', '75000'),
(48, '9519b', 'US', '12-Apr-2012', '145000'),
(28, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
(48, '4577b', 'US', '11-Nov-2012', '25000'),
(38, '7588b', 'CANADA', '14-Dec-2011', '58000'),
(38, '4519b', 'CANADA', '08-Apr-2012', '120000'),
(48, '3788a', 'US', '12-May-2011', '4950'),
(28, '3788a', 'US', '04-Apr-2012', '37500'),
(48, '4577b', 'INDIA', '11-Jun-2011', '25000'),
(18, '9519b', 'ITALY', '07-Jul-2012', '15000'),
(28, '4519b', 'INDIA', '2-Dec-2012', '58000');
```

Query the `sales` table to show that the rows were distributed among the subpartitions:

```
SELECT tableoid, * FROM sales;
```

Delete the contents of the `2012_americas` partition:

```
ALTER TABLE sales TRUNCATE SUBPARTITION "americas_2012";
```

Query the `sales` table to show that the contents of the `americas_2012` partition were removed:

```
SELECT tableoid, * FROM sales;
```
<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_americas_2011</td>
<td>30</td>
<td>7588b</td>
<td>CANADA</td>
<td>14-DEC-11 00:00:00</td>
<td>50000</td>
</tr>
<tr>
<td>sales_americas_2011</td>
<td>40</td>
<td>3788a</td>
<td>US</td>
<td>12-MAY-11 00:00:00</td>
<td>4950</td>
</tr>
<tr>
<td>sales_europe_2011</td>
<td>10</td>
<td>4519b</td>
<td>FRANCE</td>
<td>17-JAN-11 00:00:00</td>
<td>45000</td>
</tr>
<tr>
<td>sales_asia_2011</td>
<td>40</td>
<td>4577b</td>
<td>INDIA</td>
<td>11-JUN-11 00:00:00</td>
<td>25000</td>
</tr>
<tr>
<td>sales_europe_2012</td>
<td>10</td>
<td>9519b</td>
<td>ITALY</td>
<td>07-JUL-12 00:00:00</td>
<td>15000</td>
</tr>
<tr>
<td>sales_asia_2012</td>
<td>20</td>
<td>3788a</td>
<td>INDIA</td>
<td>01-MAR-12 00:00:00</td>
<td>75000</td>
</tr>
<tr>
<td>sales_asia_2012</td>
<td>20</td>
<td>3788a</td>
<td>PAKISTAN</td>
<td>04-JUN-12 00:00:00</td>
<td>37500</td>
</tr>
<tr>
<td>sales_asia_2012</td>
<td>20</td>
<td>4519b</td>
<td>INDIA</td>
<td>02-DEC-12 00:00:00</td>
<td>5090</td>
</tr>
</tbody>
</table>

The rows were removed, but the structure of the `2012_americas` partition is intact:

```sql
edb# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
```

<table>
<thead>
<tr>
<th>subpartition_name</th>
<th>high_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EUROPE_2011</td>
<td>'ITALY', 'FRANCE'</td>
</tr>
<tr>
<td>ASIA_2011</td>
<td>'PAKISTAN', 'INDIA'</td>
</tr>
<tr>
<td>AMERICAS_2011</td>
<td>'US', 'CANADA'</td>
</tr>
<tr>
<td>EUROPE_2012</td>
<td>'ITALY', 'FRANCE'</td>
</tr>
<tr>
<td>ASIA_2012</td>
<td>'PAKISTAN', 'INDIA'</td>
</tr>
<tr>
<td>AMERICAS_2012</td>
<td>'US', 'CANADA'</td>
</tr>
</tbody>
</table>

14.3.2.18 Accessing a PARTITION or SUBPARTITION

For a partitioned table, you can access the partition or subpartition using `PARTITION part_name` or `SUBPARTITION subpart_name`. This example creates a partitioned table `sales` that's range partitioned by `date` and subpartitioned using list partitioning by the `country` column:

```sql
CREATE TABLE sales
(
  dept_no number,
  part_no  varchar2,
  country  varchar2(20),
  date     date,
  amount   number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST(country)
  PARTITION q1_2020
    VALUES LESS THAN('2020-Apr-01')
    (SUBPARTITION q1_europe VALUES ('FRANCE', 'ITALY'),
     SUBPARTITION q1_asia VALUES ('INDIA', 'PAKISTAN'),
     SUBPARTITION q1_americas VALUES ('US', 'CANADA'))
  PARTITION q2_2020
    VALUES LESS THAN('2020-Jul-01')
    (SUBPARTITION q2_europe VALUES ('FRANCE', 'ITALY'),
     SUBPARTITION q2_asia VALUES ('INDIA', 'PAKISTAN'),
     SUBPARTITION q2_americas VALUES ('US', 'CANADA'));
)
```

The `SELECT` statement shows two partitions. Each partition has three subpartitions.

```sql
edb# SELECT subpartition_name, high_value, partition_name FROM ALL_TAB_SUBPARTITIONS;
```
This INSERT statement inserts rows into the sales table using specific PARTITION part_name or SUBPARTITION subpart_name values:

```sql
INSERT INTO sales PARTITION (q1_2020) VALUES (10, 'q1_2020', 'FRANCE', '2020-Feb-01', '500000');
INSERT INTO sales PARTITION (q2_2020) VALUES (10, 'q2_2020', 'ITALY', '2020-Apr-01', '550000');
INSERT INTO sales SUBPARTITION (q1_europe) VALUES (10, 'q1_europe', 'FRANCE', '2020-Feb-01', '600000');
INSERT INTO sales SUBPARTITION (q2_asia) VALUES (10, 'q2_asia', 'INDIA', '2028-Apr-01', '650000');
```

edb# SELECT tableoid::regclass, * FROM sales ORDER BY date;

```sql
<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_q1_europe</td>
<td>10</td>
<td>q1_2020</td>
<td>FRANCE</td>
<td>01-FEB-20 00:00:00</td>
<td>500000</td>
</tr>
<tr>
<td>sales_q1_europe</td>
<td>10</td>
<td>q1_europe</td>
<td>FRANCE</td>
<td>01-FEB-20 00:00:08</td>
<td>600000</td>
</tr>
<tr>
<td>sales_q2_europe</td>
<td>10</td>
<td>q2_2020</td>
<td>ITALY</td>
<td>01-APR-20 00:00:00</td>
<td>550000</td>
</tr>
<tr>
<td>sales_q2_europe</td>
<td>10</td>
<td>q2_asia</td>
<td>INDIA</td>
<td>01-APR-20 00:00:00</td>
<td>650000</td>
</tr>
</tbody>
</table>
```

(4 rows)

Use this query to fetch the values from a specific partition q1_2020 or subpartition q1_europe:

```sql
edb# SELECT tableoid::regclass, country FROM sales PARTITION (q1_2020) ORDER BY date;
```

```sql
<table>
<thead>
<tr>
<th>tableoid</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_q1_europe</td>
<td>FRANCE</td>
</tr>
<tr>
<td>sales_q1_europe</td>
<td>FRANCE</td>
</tr>
</tbody>
</table>
```

(2 rows)

This SELECT statement selects rows from a specific partition or subpartition of a partitioned table by specifying the keyword PARTITION or SUBPARTITION:

```sql
edb# SELECT tableoid::regclass, * FROM sales PARTITION (q2_2020) ORDER BY country;
```

```sql
<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_q2_europe</td>
<td>10</td>
<td>q2_asia</td>
<td>INDIA</td>
<td>01-APR-20 00:00:00</td>
<td>650000</td>
</tr>
<tr>
<td>sales_q2_europe</td>
<td>10</td>
<td>q2_2020</td>
<td>ITALY</td>
<td>01-APR-20 00:00:00</td>
<td>550000</td>
</tr>
</tbody>
</table>
```

(2 rows)

```sql
edb# SELECT tableoid::regclass, date FROM sales SUBPARTITION (q2_asia) ORDER BY country;
```

```sql
<table>
<thead>
<tr>
<th>tableoid</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_q2_asia</td>
<td>01-APR-20 00:00:00</td>
</tr>
</tbody>
</table>
```

(1 row)

This UPDATE statement updates values in a partition or subpartition of the sales table:

```sql
edb# UPDATE sales PARTITION (q1_2020) SET amount = 500000 WHERE amount = 500000 RETURNING tableoid::regclass, *;
```

```sql
<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
</table>
```

(1 row)
UPDATE 1

This DELETE statement removes rows from the partition q2_2020 or subpartition q2_asia of the sales table:

```sql
edb# DELETE FROM sales PARTITION (q2_2020) WHERE amount = 550000 RETURNING tableoid::regclass, *;

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_q2_europe</td>
<td>10</td>
<td>q2_2020</td>
<td>ITALY</td>
<td>01-APR-20 00:00:00</td>
<td>550000</td>
</tr>
</tbody>
</table>
(1 row)
DELETE 1

edb# DELETE FROM sales SUBPARTITION (q2_asia) RETURNING tableoid::regclass, *

<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_q2_asia</td>
<td>10</td>
<td>q2_asia</td>
<td>INDIA</td>
<td>01-APR-20 00:00:00</td>
<td>650000</td>
</tr>
</tbody>
</table>
(1 row)
DELETE 1

Using alias for accessing PARTITION or SUBPARTITION

You can use aliases with SELECT, INSERT, UPDATE, or DELETE statements to access a partition or subpartition. This example creates a partitioned table sales that's range partitioned by date and subpartitioned using list partitioning by the country column:

```sql
CREATE TABLE sales
(
  dept_no  number,
  part_no  varchar2,
  country  varchar2(20),
  date     date,
  amount   number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST(country)
  (PARTITION q1_2020
    VALUES LESS THAN('2020-Apr-01')
    (SUBPARTITION q1_europe VALUES ('FRANCE', 'ITALY'),
     SUBPARTITION q1_asia VALUES ('INDIA', 'PAKISTAN'),
     SUBPARTITION q1_americas VALUES ('US', 'CANADA'))
  , PARTITION q2_2020
    VALUES LESS THAN('2020-Jul-01')
    (SUBPARTITION q2_europe VALUES ('FRANCE', 'ITALY'),
     SUBPARTITION q2_asia VALUES ('INDIA', 'PAKISTAN'),
     SUBPARTITION q2_americas VALUES ('US', 'CANADA'))
)
The \texttt{SELECT} statement shows two partitions. Each partition has three subpartitions.

\begin{verbatim}
<table>
<thead>
<tr>
<th>subpartition_name</th>
<th>high_value</th>
<th>partition_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
<td>Q1_2020</td>
</tr>
<tr>
<td>Q1_ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
<td>Q1_2020</td>
</tr>
<tr>
<td>Q1_AMERICAS</td>
<td>'US', 'CANADA'</td>
<td>Q1_2020</td>
</tr>
<tr>
<td>Q2_EUROPE</td>
<td>'FRANCE', 'ITALY'</td>
<td>Q2_2020</td>
</tr>
<tr>
<td>Q2_ASIA</td>
<td>'INDIA', 'PAKISTAN'</td>
<td>Q2_2020</td>
</tr>
<tr>
<td>Q2_AMERICAS</td>
<td>'US', 'CANADA'</td>
<td>Q2_2020</td>
</tr>
</tbody>
</table>
\end{verbatim}

This \texttt{INSERT} statement creates an alias of the \texttt{sales} table and inserts rows into partition \texttt{q1_2020} and \texttt{q2_2020} or subpartition \texttt{q1_europe} and \texttt{q1_asia}:

\begin{verbatim}
INSERT INTO sales PARTITION (q1_2020) AS q1_sales VALUES (10, 'q1_2020', 'FRANCE', '2020-Feb-01', '500000');
INSERT INTO sales PARTITION (q2_2020) AS q2_sales (q2_sales.dept_no, q2_sales.part_no, q2_sales.country, q2_sales.date, q2_sales.amount) VALUES (20, 'q2_2020', 'ITALY', '2020-Apr-01', '500000');
INSERT INTO sales PARTITION (q1_europe) AS sales_q1_europe VALUES (10, 'q1_europe', 'FRANCE', '2020-Feb-01', '600000');
INSERT INTO sales PARTITION (q1_asia) Sales_q1_asia (sales_q1_asia.dept_no, sales_q1_asia.part_no, sales_q1_asia.country, sales_q1_asia.date, sales_q1_asia.amount) VALUES (20, 'q1_asia', 'INDIA', '2020-Mar-01', '650000');
\end{verbatim}

This \texttt{SELECT} statement selects rows from a specific partition or subpartition of a \texttt{sales} table:

\begin{verbatim}
<table>
<thead>
<tr>
<th>tableoid</th>
<th>dept_no</th>
<th>part_no</th>
<th>country</th>
<th>date</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>sales_q1_europe</td>
<td>10</td>
<td>q1_2020</td>
<td>FRANCE</td>
<td>01-FEB-20 00:00:00</td>
<td>500000</td>
</tr>
<tr>
<td>sales_q1_europe</td>
<td>10</td>
<td>q1_europe</td>
<td>FRANCE</td>
<td>01-FEB-20 00:00:00</td>
<td>600000</td>
</tr>
<tr>
<td>sales_q1_asia</td>
<td>20</td>
<td>q1_asia</td>
<td>INDIA</td>
<td>01-MAR-20 00:00:00</td>
<td>650000</td>
</tr>
</tbody>
</table>
\end{verbatim}

This \texttt{UPDATE} statement updates values in a partition or subpartition of the \texttt{sales} table:

\begin{verbatim}
UPDATE sales PARTITION (q1_2020) AS q1_sales SET q1_sales.amount = 10000 WHERE q1_sales.amount = 500000;
UPDATE 1
\end{verbatim}

\begin{verbatim}
UPDATE sales PARTITION (q1_europe) AS sales_q1_europe SET sales_q1_europe.amount = 5000 WHERE sales_q1_europe.amount = 600000;
UPDATE 1
\end{verbatim}

This \texttt{DELETE} statement removes rows from the partition \texttt{q1_2020} or subpartition \texttt{q1_europe} of the \texttt{sales} table:

\begin{verbatim}
DELETE FROM sales PARTITION (q1_2020) q1_sales WHERE q1_sales.amount = 10000;
DELETE 1
\end{verbatim}

\begin{verbatim}
DELETE FROM sales PARTITION (q1_europe) AS sales_q1_europe WHERE sales_q1_europe.amount = 5000;
DELETE 1
\end{verbatim}

### 14.3.3 Built-in packages

Database compatibility for Oracle means that an application runs in an Oracle environment as well as in the EDB Postgres Advanced Server environment with minimal or no changes to the application code. EDB Postgres Advanced Server provides packages that support this compatibility.
14.3.3.1 Built-in packages

Built-in packages are provided with EDB Postgres Advanced Server. For certain packages, non-superusers must be explicitly granted the `EXECUTE` privilege on the package before using any of the package's functions or procedures. For most of the built-in packages, `EXECUTE` privilege is granted to `PUBLIC` by default.

For information about using the `GRANT` command to provide access to a package, see SQL reference.

All built-in packages are owned by the special `sys` user that must be specified when granting or revoking privileges on built-in packages:

```sql
GRANT EXECUTE ON PACKAGE SYS.UTL_FILE TO john;
```

14.3.3.1.1 DBMS_ALERT

The `DBMS_ALERT` package lets you register for, send, and receive alerts. The following table lists the supported procedures:

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGISTER(name)</td>
<td>n/a</td>
<td>Register to be able to receive alerts named <code>name</code>.</td>
</tr>
<tr>
<td>REMOVE(name)</td>
<td>n/a</td>
<td>Remove registration for the alert named <code>name</code>.</td>
</tr>
<tr>
<td>REMOVEALL</td>
<td>n/a</td>
<td>Remove registration for all alerts.</td>
</tr>
<tr>
<td>SIGNAL(name, message)</td>
<td>n/a</td>
<td>Signal the alert named <code>name</code> with <code>message</code>.</td>
</tr>
<tr>
<td>WAITANY(name OUT, message OUT, status OUT, timeout)</td>
<td>n/a</td>
<td>Wait for any registered alert to occur.</td>
</tr>
<tr>
<td>WAITONE(name, message OUT, status OUT, timeout)</td>
<td>n/a</td>
<td>Wait for the specified alert, <code>name</code>, to occur.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server's implementation of `DBMS_ALERT` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table are supported.

EDB Postgres Advanced Server allows a maximum of 500 concurrent alerts. You can use the `dbms_alert.max_alerts` GUC variable, located in the `postgresql.conf` file, to specify the maximum number of concurrent alerts allowed on a system.

To set a value for the `dbms_alert.max_alerts` variable, open the `postgresql.conf` file, which is located by default in `/opt/PostgresPlus/14AS/data`, with your choice of editor. Edit the `dbms_alert.max_alerts` parameter as shown:

```conf
dbms_alert.max_alerts = alert_count
```

`alert_count` specifies the maximum number of concurrent alerts. By default, the value of `dbms_alert.max_alerts` is 100. To disable this feature, set `dbms_alert.max_alerts` to 0.

For the `dbms_alert.max_alerts` GUC to function correctly, the `custom_variable_classes` parameter must contain `dbms_alerts`:

```conf
custom_variable_classes = 'dbms_alert, -'
```

After editing the `postgresql.conf` file parameters, you must restart the server for the changes to take effect.

REGISTER

The `REGISTER` procedure enables the current session to be notified of the specified alert.

```sql
REGISTER(<name> VARCHAR2)
```

Parameters

- `name`

Name of the alert to register.

Examples
This anonymous block registers for an alert named `alert_test` and then waits for the signal.

```sql
DECLARE
  v_name VARCHAR2(30) := 'alert_test';
  v_msg VARCHAR2(80);
  v_status INTEGER;
  v_timeout NUMBER(3) := 128;
BEGIN
  DBMS_ALERT.REGISTER(v_name);
  DBMS_OUTPUT.PUT_LINE('Registered for alert ' || v_name);
  DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  DBMS_ALERT.WAITONE(v_name, v_msg, v_status, v_timeout);
  DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
  DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
  DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
  DBMS_ALERT.REMOVE(v_name);
END;
```

Registered for alert alert_test
Waiting for signal...

**REMOVE**

The **REMOVE** procedure unregisters the session for the named alert.

```sql
REMOVE(<name> VARCHAR2)
```

**Parameters**

- **name**

  Name of the alert to unregister.

**REMOVEALL**

The **REMOVEALL** procedure unregisters the session for all alerts.

```sql
REMOVEALL
```

**SIGNAL**

The **SIGNAL** procedure signals the occurrence of the named alert.

```sql
SIGNAL(<name> VARCHAR2, <message> VARCHAR2)
```

**Parameters**

- **name**

  Name of the alert.

- **message**

  Message to be associated with the alert.
Information to pass with this alert.

Examples

This anonymous block signals an alert for `alert_test`.

```sql
DECLARE
    v_name VARCHAR2(30) := 'alert_test';
BEGIN
    DBMS_ALERT.SIGNAL(v_name, 'This is the message from ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;
```

**WAITANY**

The `WAITANY` procedure waits for any of the registered alerts to occur.

```sql
WAITANY(<name> OUT VARCHAR2, <message> OUT VARCHAR2, <status> OUT INTEGER, <timeout> NUMBER)
```

Parameters

**name**
Variable receiving the name of the alert.

**message**
Variable receiving the message sent by the `SIGNAL` procedure.

**status**
Status code returned by the operation. Possible values are: 0 - alert occurred; 1 - timeout occurred.

**timeout**
Time to wait for an alert in seconds.

Examples

This anonymous block uses the `WAITANY` procedure to receive an alert named `alert_test` or `any_alert`:

```sql
DECLARE
    v_name VARCHAR2(30);
    v_msg VARCHAR2(80);
    v_status INTEGER;
    v_timeout NUMBER(3) := 120;
BEGIN
    DBMS_ALERTREGISTER('alert_test');
    DBMS_ALERTREGISTER('any_alert');
    DBMS_OUTPUT.PUT_LINE('Registered for alert alert_test and any_alert');
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
    DBMS_ALERT.WAITANY(v_name, v_msg, v_status, v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
```
An anonymous block in a second session issues a signal for any_alert:

```
DECLARE
  v_name  VARCHAR2(30) := 'any_alert';
BEGIN
  DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;
```

Control returns to the first anonymous block and the remainder of the code is executed:

```
DECLARE
  v_name  VARCHAR2(30) := 'alert_test';
BEGIN
  DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;
```

WAITONE

The WAITONE procedure waits for the specified registered alert to occur.

```
WAITONE(<name> VARCHAR2, <message> OUT VARCHAR2, <status> OUT INTEGER, <timeout> NUMBER)
```

Parameters

name

Name of the alert.

message

Variable receiving the message sent by the SIGNAL procedure.

status

Status code returned by the operation. Possible values are: 0 - alert occurred; 1 - timeout occurred.

timeout

Time to wait for an alert in seconds.

Examples

This anonymous block is similar to the one used in the WAITANY example except the WAITONE procedure is used to receive the alert named alert_test.

```
DECLARE
  v_name  VARCHAR2(30) := 'alert_test';
  v_msg  VARCHAR2(80);
  v_status INTEGER;
  v_timeout NUMBER(3) := 120;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
  DBMS_ALERT.REMOVEALL;
END;
```

Registered for alert alert_test and any_alert
Waiting for signal...

Registered for alert alert_test and any_alert
Waiting for signal...

Alert name   : any_alert
Alert msg    : This is the message from any_alert
Alert status : 0
Alert timeout: 120 seconds

WAITONE

The WAITONE procedure waits for the specified registered alert to occur.
Signal sent for alert_test sent by an anonymous block in a second session:

DECLARE
  v_name VARCHAR2(30) := 'alert_test';
BEGIN
  DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;
END;

First session is alerted, control returns to the anonymous block, and the remainder of the code is executed:

Issued alert for alert_test

Comprehensive example

The following example uses two triggers to send alerts when the dept table or the emp table is changed. An anonymous block listens for these alerts and displays messages when an alert is received.

The following are the triggers on the dept and emp tables:

```
CREATE OR REPLACE TRIGGER dept_alert_trig
AFTER INSERT OR UPDATE OR DELETE ON dept
DECLARE
  v_action VARCHAR2(25);
BEGIN
  IF INSERTING THEN
    v_action := ' added department(s) '
  ELSIF UPDATING THEN
    v_action := ' updated department(s) '
  ELSIF DELETING THEN
    v_action := ' deleted department(s) '
  END IF;
  DBMS_ALERT.SIGNAL('dept_alert',USER || v_action || ' on ' || SYSDATE);
END;
```

```
CREATE OR REPLACE TRIGGER emp_alert_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
```
This anonymous block is executed in a session while updates to the `dept` and `emp` tables occur in other sessions:

The following changes are made by user, mary:

```
INSERT INTO dept VALUES (50,'FINANCE','CHICAGO');
INSERT INTO emp (empno,ename,deptno) VALUES (9001,'JONES',50);
INSERT INTO emp (empno,ename,deptno) VALUES (9002,'ALICE',50);
```

The following change is made by user, john:

```
INSERT INTO dept VALUES (60,'HR','LOS ANGELES');
```

The following is the output displayed by the anonymous block receiving the signals from the triggers:

```
Registered for alerts dept_alert and emp_alert
Waiting for signal...
```

```
DECLARE
  v_dept_alert VARCHAR2(30) := 'dept_alert';
  v_emp_alert VARCHAR2(30) := 'emp_alert';
  v_name VARCHAR2(30);
  v_msg VARCHAR2(80);
  v_status INTEGER;
  v_timeout NUMBER(3) := 60;
BEGIN
  DBMS_ALERT.REGISTER(v_dept_alert);
  DBMS_ALERT.REGISTER(v_emp_alert);
  DBMS_OUTPUT.PUT_LINE('Registered for alerts dept_alert and emp_alert');
  DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  LOOP
    DBMS_ALERT.WAITANY(v_name,v_msg,v_status,v_timeout);
    EXIT WHEN v_status != 0;
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('------------------------------------');
    END LOOP;
  DBMS_ALERT.REMOVEALL;
END;
```

Registered for alerts dept_alert and emp_alert
Waiting for signal...

```
BEGIN
  IF INSERTING THEN
    v_action := ' added employee(s)
  ELSEIF UPDATING THEN
    v_action := ' updated employee(s)
  ELSEIF DELETING THEN
    v_action := ' deleted employee(s)
  END IF;
  DBMS_ALERT.SIGNAL('emp_alert',USER || v_action || ' on ' || SYSDATE);
END;
```

The following is the output displayed by the anonymous block receiving the signals from the triggers:

```
Alert name   : dept_alert
```

```
v_action VARCHAR2(25);
BEGIN
  IF INSERTING THEN
    v_action := ' added employee(s)
  ELSEIF UPDATING THEN
    v_action := ' updated employee(s)
  ELSEIF DELETING THEN
    v_action := ' deleted employee(s)
  END IF;
  DBMS_ALERT.SIGNAL('emp_alert',USER || v_action || ' on ' || SYSDATE);
END;
```
14.3.3.1.2 DBMS_AQ

EDB Postgres Advanced Server Advanced Queueing provides message queueing and message processing for the EDB Postgres Advanced Server database. User-defined messages are stored in a queue, and a collection of queues is stored in a queue table. Procedures in the DBMS_AQADM package create and manage message queues and queue tables. Use the DBMS_AQ package to add messages to or remove them from a queue or to register or unregister a PL/SQL callback procedure.

EDB Postgres Advanced Server also provides extended (noncompatible) functionality for the DBMS_AQ package with SQL commands. See SQL reference for detailed information about the following SQL commands:

- ALTER QUEUE
- ALTER QUEUE TABLE
- CREATE QUEUE
- CREATE QUEUE TABLE
- DROP QUEUE
- DROP QUEUE TABLE

The DBMS_AQ package provides procedures that allow you to enqueue a message, dequeue a message, and manage callback procedures. The supported procedures are:

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENQUEUE</td>
<td>n/a</td>
<td>Post a message to a queue.</td>
</tr>
<tr>
<td>DEQUEUE</td>
<td>n/a</td>
<td>Retrieve a message from a queue if or when a message is available.</td>
</tr>
<tr>
<td>REGISTER</td>
<td>n/a</td>
<td>Register a callback procedure.</td>
</tr>
<tr>
<td>UNREGISTER</td>
<td>n/a</td>
<td>Unregister a callback procedure.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server’s implementation of DBMS_AQ is a partial implementation when compared to Oracle’s version. Only those procedures listed in the table above are supported.

EDB Postgres Advanced Server supports use of these constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
<th>For parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_AQ.BROWSE (0)</td>
<td>Read the message without locking.</td>
<td>dequeue_options_t.dequeue_mode</td>
</tr>
<tr>
<td>DBMS_AQ.LOCKED (1)</td>
<td>This constant is defined but returns an error if used.</td>
<td>dequeue_options_t.dequeue_mode</td>
</tr>
<tr>
<td>DBMS_AQ.REMOVE (2)</td>
<td>Delete the message after reading (the default).</td>
<td>dequeue_options_t.dequeue_mode</td>
</tr>
<tr>
<td>DBMS_AQ.REMOVE_NODATA (3)</td>
<td>This constant is defined but returns an error if used.</td>
<td>dequeue_options_t.dequeue_mode</td>
</tr>
<tr>
<td>DBMS_AQ.FIRST_MESSAGE (6)</td>
<td>Return the first available message that matches the search terms.</td>
<td>dequeue_options_t.navigation</td>
</tr>
<tr>
<td>DBMS_AQ.NEXT_MESSAGE (1)</td>
<td>Return the next available message that matches the search terms.</td>
<td>dequeue_options_t.navigation</td>
</tr>
<tr>
<td>DBMS_AQ.NEXT_TRANSACTION (2)</td>
<td>This constant is defined but returns an error if used.</td>
<td>dequeue_options_t.navigation</td>
</tr>
<tr>
<td>DBMS_AQ.FOREVER (-1)</td>
<td>Wait forever if a message that matches the search term isn’t found (the default).</td>
<td>dequeue_options_t.wait</td>
</tr>
<tr>
<td>DBMS_AQ.NO_WAIT (0)</td>
<td>Don’t wait if a message that matches the search term isn’t found.</td>
<td>dequeue_options_t.wait</td>
</tr>
<tr>
<td>DBMS_AQ.ON_COMMIT (0)</td>
<td>The dequeue is part of the current transaction.</td>
<td>enqueue_options.t.visibility, enqueue_options.t.visibility</td>
</tr>
<tr>
<td>DBMS_AQ.IMMEDIATE (1)</td>
<td>This constant is defined but returns an error if used.</td>
<td>enqueue_options.t.visibility, enqueue_options.t.visibility</td>
</tr>
<tr>
<td>DBMS_AQ.PERSISTENT (0)</td>
<td>Store the message in a table.</td>
<td>enqueue_options_t.delivery_node</td>
</tr>
<tr>
<td>DBMS_AQ.BUFFERED (1)</td>
<td>This constant is defined but returns an error if used.</td>
<td>enqueue_options_t.delivery_node</td>
</tr>
<tr>
<td>DBMS_AQ.READY (0)</td>
<td>Specifies that the message is ready to process.</td>
<td>message_properties.t.state</td>
</tr>
<tr>
<td>DBMS_AQ.WAITING (1)</td>
<td>Specifies that the message is waiting to be processed.</td>
<td>message_properties.t.state</td>
</tr>
<tr>
<td>DBMS_AQ.PROCESSED (2)</td>
<td>Specifies that the message was processed.</td>
<td>message_properties.t.state</td>
</tr>
</tbody>
</table>
The **DBMS_AQ** configuration parameters listed in the following table can be defined in the `postgresql.conf` file. After the configuration parameters are defined, you can invoke the **DBMS_AQ** package to use and manage messages held in queues and queue tables.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbms_aq.max_workers</code></td>
<td>The maximum number of workers to run.</td>
</tr>
<tr>
<td><code>dbms_aq.max_idle_time</code></td>
<td>The idle time a worker must wait before exiting.</td>
</tr>
<tr>
<td><code>dbms_aq.min_work_time</code></td>
<td>The minimum time a worker can run before exiting.</td>
</tr>
<tr>
<td><code>dbms_aq.launch_delay</code></td>
<td>The minimum time between creating workers.</td>
</tr>
<tr>
<td><code>dbms_aq.batch_size</code></td>
<td>The maximum number of messages to process in a single transaction. The default batch size is 10.</td>
</tr>
<tr>
<td><code>dbms_aq.max_databases</code></td>
<td>The size of the <strong>DBMS_AQ</strong> hash table of databases. The default value is 1024.</td>
</tr>
<tr>
<td><code>dbms_aq.max_pending_retries</code></td>
<td>The size of the <strong>DBMS_AQ</strong> hash table of pending retries. The default value is 1024.</td>
</tr>
</tbody>
</table>

### 14.3.3.1.2.1 ENQUEUE

The **ENQUEUE** procedure adds an entry to a queue. The signature is:

```sql
ENQUEUE(
  <queue_name> IN VARCHAR2,
  <enqueue_options> IN DBMS_AQ.ENQUEUE_OPTIONS_T,
  <message_properties> IN DBMS_AQ.MESSAGE_PROPERTIES_T,
  <payload> IN <type_name>,
  <msgid> OUT RAW)
```

#### Parameters

- **queue_name**

The name (optionally schema qualified) of an existing queue. If you omit the schema name, the server uses the schema specified in the `SEARCH_PATH`. Unlike Oracle, unquoted identifiers are converted to lower case before storing. To include special characters or use a case-sensitive name, enclose the name in double quotes.

For detailed information about creating a queue, see `DBMS_AQADM.CREATE_QUEUE`.

- **enqueue_options**

`DBMS_AQ.ENQUEUE_OPTIONS_T` is a value of the type `enqueue_options_t`:

```sql
DBMS_AQ.ENQUEUE_OPTIONS_T IS RECORD(
  visibility BINARY_INTEGER DEFAULT ON_COMMIT,
  relative_msgid RAW(16) DEFAULT NULL,
  sequence_deviation BINARY_INTEGER DEFAULT NULL,
  transformation VARCHAR2(61) DEFAULT NULL,
  delivery_mode PLS_INTEGER NOT NULL DEFAULT PERSISTENT);
```

Currently, the only supported parameter values for `enqueue_options_t` are:

- **visibility** `ON_COMMIT`
- **delivery_mode** `PERSISTENT`
- **sequence_deviation** `NULL`
message_properties

message_properties is a value of the type message_properties_t:

message_properties_t IS RECORD(
  priority INTEGER,
  delay INTEGER,
  expiration INTEGER,
  correlation CHARACTER VARYING(128) COLLATE pg_catalog."C",
  attempts INTEGER,
  recipient_list "AQ$_RECIPIENT_LIST_T",
  exception_queue CHARACTER VARYING(61) COLLATE pg_catalog."C",
  enqueue_time TIMESTAMP WITHOUT TIME ZONE,
  state INTEGER,
  original_msgid BYTEA,
  delivery_mode INTEGER DBMS_AQ.PERSISTENT);

The supported values for message_properties_t are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>If the queue table definition includes a sort_list that references priority, this parameter affects the order that messages are dequeued. A lower value indicates a higher dequeue priority.</td>
</tr>
<tr>
<td>delay</td>
<td>Specify the number of seconds to pass before a message is available for dequeuing or NO_DELAY.</td>
</tr>
<tr>
<td>expiration</td>
<td>Use the expiration parameter to specify the number of seconds until a message expires.</td>
</tr>
<tr>
<td>correlation</td>
<td>Use correlation to specify a message to associate with the entry. The default is NULL.</td>
</tr>
<tr>
<td>attempts</td>
<td>This is a system-maintained value that specifies the number of attempts to dequeue the message.</td>
</tr>
<tr>
<td>recipient_list</td>
<td>This parameter is not supported.</td>
</tr>
<tr>
<td>exception_queue</td>
<td>Use the exception_queue parameter to specify the name of an exception queue to which to move a message if it expires or is dequeued by a transaction that rolls back too many times.</td>
</tr>
<tr>
<td>enqueue_time</td>
<td>enqueue_time is the time the record was added to the queue. This value is provided by the system.</td>
</tr>
<tr>
<td>state</td>
<td>This parameter is maintained by DBMS_AQ. The state can be:</td>
</tr>
<tr>
<td></td>
<td>DBMS_AQ.WAITING - The delay wasn't reached.</td>
</tr>
<tr>
<td></td>
<td>DBMS_AQ.READY - The queue entry is ready for processing.</td>
</tr>
<tr>
<td></td>
<td>DBMS_AQ.PROCESSED - The queue entry was processed.</td>
</tr>
<tr>
<td></td>
<td>DBMS_AQ.EXPIRED - The queue entry was moved to the exception queue.</td>
</tr>
<tr>
<td>original_msgid</td>
<td>This parameter is accepted for compatibility and ignored.</td>
</tr>
<tr>
<td>transaction_group</td>
<td>This parameter is accepted for compatibility and ignored.</td>
</tr>
<tr>
<td>delivery_mode</td>
<td>This parameter isn't supported. Specify a value of DBMS_AQ.PERSISTENT.</td>
</tr>
</tbody>
</table>

payload

Use the payload parameter to provide the data to associate with the queue entry. The payload type must match the type specified when creating the corresponding queue table (see DBMS_AQADM.CREATE_QUEUE_TABLE).

msgid

Use the msgid parameter to retrieve a unique (system-generated) message identifier.

Example

The following anonymous block calls DBMS_AQ.ENQUEUE, adding a message to a queue named work_order:
14.3.3.1.2.2 DEQUEUE

The DEQUEUE procedure dequeues a message. The signature is:

```
DEQUEUE(
    <queue_name> IN VARCHAR2,
    <dequeue_options> IN DBMS_AQ.DEQUEUE_OPTIONS_T,
    <message_properties> OUT DBMS_AQ.MESSAGE_PROPERTIES_T,
    <payload> OUT <type_name>,
    <msgid> OUT RAW)
```

**Parameters**

*queue_name*

The name (optionally schema-qualified) of an existing queue. If you omit the schema name, the server uses the schema specified in the `SEARCH_PATH`. Unlike Oracle, unquoted identifiers are converted to lower case before storing. To include special characters or use a case-sensitive name, enclose the name in double quotes.

For detailed information about creating a queue, see `DBMS_AQADM.CREATE_QUEUE`.

*dequeue_options* is a value of the type, `dequeue_options_t`:

```
DEQUEUE_OPTIONS_T IS RECORD(
    consumer_name CHARACTER VARYING(30),
    dequeue_mode INTEGER,
    navigation INTEGER,
    visibility INTEGER,
    wait INTEGER,
   msgid BYTEA,
    correlation CHARACTER VARYING(128),
    deq_condition CHARACTER VARYING(4000),
    transformation CHARACTER VARYING(61),
    delivery_mode INTEGER);
```

Currently, the supported parameter values for `dequeue_options_t` are:

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>consumer_name</td>
<td><strong>Must be NULL</strong></td>
</tr>
</tbody>
</table>
The locking behavior of the dequeue operation. Must be either:

- DBMS_AQ.BROWSE - Read the message without obtaining a lock.
- DBMS_AQ.LOCKED - Read the message after acquiring a lock.
- DBMS_AQ.REMOVE - Read the message before deleting the message.
- DBMS_AQ.REMOVE_NODATA - Read the message, but don’t delete the message.

Identifies the message to retrieve. Must be either:

- FIRST_MESSAGE - The first message in the queue that matches the search term.
- NEXT_MESSAGE - The next message that’s available that matches the first term.

Must be ON_COMMIT. If you roll back the current transaction, the dequeued item remains in the queue.

Must be a number larger than 0, or:

- DBMS_AQ.FOREVER - Wait indefinitely.
- DBMS_AQ.NO_WAIT - Don’t wait.

The message ID of the message that to dequeue.

Accepted for compatibility and ignored.

A VARCHAR2 expression that evaluates to a BOOLEAN value indicating whether to dequeue the message.

Accepted for compatibility and ignored.

Must be PERSISTENT. Buffered messages aren’t supported.

The supported values for message_properties_t are:

- priority
- delay
- expiration
- correlation
- attempts
- recipient_list
- exception_queue
- enqueue_time
- state
- original_msgid
- delivery_mode
- message_properties_t

If the queue table definition includes a sort_list that references priority, this parameter affects the order that messages are dequeued. A lower value indicates a higher dequeue priority.

Specify the number of seconds that pass before a message is available for dequeueing or NO_DELAY.

Use the expiration parameter to specify the number of seconds until a message expires.

Use correlation to specify a message to associate with the entry. The default is NULL.

This is a system-maintained value that specifies the number of attempts to dequeue the message.

This parameter isn’t supported.

Use the exception_queue parameter to specify the name of an exception queue to which to move a message if it expires or is dequeued by a transaction that rolls back too many times.

This parameter is maintained by DBMS_AQ; state can be:

- DBMS_AQ.WAITING - the delay has not been reached.
- DBMS_AQ.READY - the queue entry is ready for processing.
- DBMS_AQ.PROCESSED - the queue entry has been processed.
- DBMS_AQ.EXPIRED - the queue entry has been moved to the exception queue.

This parameter is accepted for compatibility and ignored.
This parameter is accepted for compatibility and ignored.

**delivery_mode**
This parameter isn't supported. Specify a value of `DBMS_AQ.PERSISTENT`.

**payload**
Use the `payload` parameter to retrieve the payload of a message with a dequeue operation. The payload type must match the type specified when creating the queue table.

**msgid**
Use the `msgid` parameter to retrieve a unique message identifier.

**Example**

The following anonymous block calls `DBMS_AQ.DEQUEUE`, retrieving a message from the queue and a payload:

```sql
DECLARE
    dequeue_options DBMS_AQ.DEQUEUE_OPTIONS_T;
    message_properties DBMS_AQ.MESSAGE_PROPERTIES_T;
    message_handle raw(16);
    payload work_order;
BEGIN
    dequeue_options.dequeue_mode := DBMS_AQ.BROWSE;

    DBMS_AQ.DEQUEUE(
        queue_name => 'work_queue',
        dequeue_options,
        message_properties,
        payload,
        msgid,
        message_handle
    );

    DBMS_OUTPUT.PUT_LINE('The next work order is [' || payload.subject || '].');
END;
```

The payload is displayed by `DBMS_OUTPUT.PUT_LINE`.

### 14.3.3.1.2.3 REGISTER

Use the `REGISTER` procedure to register an email address, procedure, or URL to notify when an item is enqueued or dequeued. The signature is:

```
REGISTER(
    <reg_list> IN SYS.AQ$_REG_INFO_LIST,
    <count> IN NUMBER)
```

**Parameters**

**`reg_list`** is a list of type `AQ$_REG_INFO_LIST` that provides information about each subscription that you want to register. Each entry in the list is of the type `AQ$_REG_INFO` and can contain:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td>VARCHAR2 (128)</td>
<td>The (optionally schema-qualified) name of the subscription.</td>
</tr>
<tr>
<td><code>namespace</code></td>
<td>NUMERIC</td>
<td>The only supported value is <code>DBMS_AQ.NAMESPACE_AQ</code> (0).</td>
</tr>
</tbody>
</table>
callback VARCHAR2 (4000)  
Describes the action to perform on notification. Currently, only calls to PL/SQL procedures are supported. The call takes the form:

\[ \text{plsql://schema.procedure} \]

Where:

- schema specifies the schema in which the procedure resides.
- procedure specifies the name of the procedure to notify.

context RAW (16)  
Any user-defined value required by the procedure.

count  
count is the number of entries in reg_list.

Example

The following anonymous block calls DBMS_AQ.REGISTER, registering procedures to notify when an item is added to or removed from a queue. A set of attributes (of sys.aq$_reg_info type) is provided for each subscription identified in the DECLARE section:

```sql
DECLARE
    subscription1 sys.aq$_reg_info;
    subscription2 sys.aq$_reg_info;
    subscription3 sys.aq$_reg_info;
    subscriptionlist sys.aq$_reg_info_list;
BEGIN
    subscription1 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ, 'plsql://assign_worker?PR=0', HEXTORAW('FFFF'));
    subscription2 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ, 'plsql://add_to_history?PR=1', HEXTORAW('FFFF'));
    subscription3 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ, 'plsql://reserve_parts?PR=2', HEXTORAW('FFFF'));
    subscriptionlist := sys.aq$_reg_info_list(subscription1, subscription2, subscription3);
    dbms_aq.register(subscriptionlist, 3);
    commit;
END;
/
```

The subscriptionlist is of type sys.aq$_reg_info_list and contains the previously described sys.aq$_reg_info objects. The list name and an object count are passed to dbms_aq.register.

14.3.3.1.2.4 UNREGISTER

Use the UNREGISTER procedure to turn off notifications related to enqueuing and dequeuing. The signature is:

```sql
UNREGISTER(
    <reg_list> IN SYS.AQ$_REG_INFO_LIST,
    <count> IN NUMBER)
```

Parameter

**reg_list**

- reg_list is a list of type AQ$_REG_INFO_LIST that provides information about each subscription that you want to register. Each entry in the list is of the type AQ$_REG_INFO and can contain:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
### Attribute Table

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>VARCHAR2 (128)</td>
<td>The (optionally schema-qualified) name of the subscription.</td>
</tr>
<tr>
<td>namespace</td>
<td>NUMERIC</td>
<td>The only supported value is <code>DBMS_AQ.NAMESPACE_AQ (0)</code>.</td>
</tr>
</tbody>
</table>

Describes the action to perform on notification. Currently, only calls to PL/SQL procedures are supported. The call takes the form:

```
plsql://schema.procedure
```

Where:

- `schema` specifies the schema in which the procedure resides.
- `procedure` specifies the name of the procedure to notify.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callback</td>
<td>VARCHAR2 (4000)</td>
<td>Any user-defined value required by the procedure.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>RAW (16)</td>
<td><code>count</code> is the number of entries in <code>reg_list</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td></td>
<td><code>count</code> is the number of entries in <code>reg_list</code>.</td>
</tr>
</tbody>
</table>

### Example

The following anonymous block calls `DBMS_AQ.UNREGISTER`, disabling the notifications specified in the example for `DBMS_AQ.REGISTER`:

```sql
DECLARE
  subscription1 sys.aq$_reg_info;
  subscription2 sys.aq$_reg_info;
  subscription3 sys.aq$_reg_info;
  subscriptionlist sys.aq$_reg_info_list;
BEGIN
  subscription1 := sys.aq$_reg_info('q',
    DBMS_AQ.NAMESPACE_AQ,
    'plsql://assign_worker?PR=0',HEXTORAW('FFFF'));
  subscription2 := sys.aq$_reg_info('q',
    DBMS_AQ.NAMESPACE_AQ,
    'plsql://add_to_history?PR=1',HEXTORAW('FFFF'));
  subscription3 := sys.aq$_reg_info('q',
    DBMS_AQ.NAMESPACE_AQ,
    'plsql://reserve_parts?PR=2',HEXTORAW('FFFF'));
  subscriptionlist := sys.aq$_reg_info_list(subscription1, subscription2, subscription3);
  dbms_aq.unregister(subscriptionlist, 3);
  commit;
END;
/
```

The `subscriptionlist` is of type `sys.aq$_reg_info_list` and contains `sys.aq$_reg_info` objects. The list name and an object count are passed to `dbms_aq.unregister`.

#### 14.3.3.1.3 DBMS_AQADM

EDB Postgres Advanced Server advanced queueing provides message queueing and message processing for the EDB Postgres Advanced Server database. User-defined messages are stored in a queue, and a collection of queues is stored in a queue table. Procedures in the `DBMS_AQADM` package create and manage message queues and queue tables. Use the `DBMS_AQ` package to add messages to or remove messages from a queue or to register or unregister a PL/SQL callback procedure.

EDB Postgres Advanced Server also provides extended (non-compatible) functionality for the `DBMS_AQ` package with SQL commands. See SQL reference for detailed information about the following SQL commands:

- `ALTER QUEUE`
- `ALTER QUEUE TABLE`
- `CREATE QUEUE`
- `CREATE QUEUE TABLE`
- `DROP QUEUE`
- `DROP QUEUE TABLE`

The `DBMS_AQADM` package provides procedures that allow you to create and manage queues and queue tables. EDB Postgres Advanced Server's implementation of `DBMS_AQADM` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table are supported.
### Function/procedure

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER_QUEUE</td>
<td>n/a</td>
<td>Modify an existing queue.</td>
</tr>
<tr>
<td>ALTER_QUEUE_TABLE</td>
<td>n/a</td>
<td>Modify an existing queue table.</td>
</tr>
<tr>
<td>CREATE_QUEUE</td>
<td>n/a</td>
<td>Create a queue.</td>
</tr>
<tr>
<td>CREATE_QUEUE_TABLE</td>
<td>n/a</td>
<td>Create a queue table.</td>
</tr>
<tr>
<td>DROP_QUEUE</td>
<td>n/a</td>
<td>Drop an existing queue.</td>
</tr>
<tr>
<td>DROP_QUEUE_TABLE</td>
<td>n/a</td>
<td>Drop an existing queue table.</td>
</tr>
<tr>
<td>PURGE_QUEUE_TABLE</td>
<td>n/a</td>
<td>Remove one or more messages from a queue table.</td>
</tr>
<tr>
<td>START_QUEUE</td>
<td>n/a</td>
<td>Make a queue available for enqueueing and dequeueing procedures.</td>
</tr>
<tr>
<td>STOP_QUEUE</td>
<td>n/a</td>
<td>Make a queue unavailable for enqueueing and dequeueing procedures.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server supports use of the arguments listed in the table:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
<th>For parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_AQADM.TRANSACTIONAL(1)</td>
<td>This constant is defined but returns an error if used.</td>
<td>message_grouping</td>
</tr>
<tr>
<td>DBMS_AQADM.NONE(0)</td>
<td>Use to specify message grouping for a queue table.</td>
<td>message_grouping</td>
</tr>
<tr>
<td>DBMS_AQADM.NORMAL_QUEUE(0)</td>
<td>Use with create_queue to specify queue_type.</td>
<td>queue_type</td>
</tr>
<tr>
<td>DBMS_AQADM.EXCEPTION_QUEUE(1)</td>
<td>Use with create_queue to specify queue_type.</td>
<td>queue_type</td>
</tr>
<tr>
<td>DBMS_AQADM.INFINITE(-1)</td>
<td>Use with create_queue to specify retention_time.</td>
<td>retention_time</td>
</tr>
<tr>
<td>DBMS_AQADM.PERSISTENT(0)</td>
<td>Store the message in a table.</td>
<td>enqueue_options_t.delivery_mode</td>
</tr>
<tr>
<td>DBMS_AQADM.BUFFERED(1)</td>
<td>This constant is defined but returns an error if used.</td>
<td>enqueue_options_t.delivery_mode</td>
</tr>
<tr>
<td>DBMS_AQADM.PERSISTENT OR BUFFERED(2)</td>
<td>This constant is defined but returns an error if used.</td>
<td>enqueue_options_t.delivery_mode</td>
</tr>
</tbody>
</table>

### 14.3.3.1.3 ALTER_QUEUE

Use the **ALTER_QUEUE** procedure to modify an existing queue. The signature is:

```sql
ALTER_QUEUE(
  <max_retries> IN NUMBER DEFAULT NULL,
  <retry_delay> IN NUMBER DEFAULT 0,
  <retention_time> IN NUMBER DEFAULT 0,
  <auto_commit> IN BOOLEAN DEFAULT TRUE,
  <comment> IN VARCHAR2 DEFAULT NULL,
)
```

**Parameters**

- **queue_name**
  
  The name of the new queue.

- **max_retries**
  
  Specifies the maximum number of attempts to remove a message with a dequeue statement. The value of `max_retries` is incremented with each `ROLLBACK` statement. When the number of failed attempts reaches the value specified by `max_retries`, the message is moved to the exception queue. Specify `0` to indicate that no retries are allowed.

- **retry_delay**
  
  Specifies the number of seconds until a message is scheduled for reprocessing after a `ROLLBACK`. Specify `0` to retry the message immediately (the default).

- **retention_time**
  
  Specifies the length of time in seconds that a message is stored after being dequeued. You can also specify `INFINITE` to retain the message forever.

- **auto_commit**
  
  This parameter is accepted for compatibility and ignored.
comment

comment specifies a comment associated with the queue.

Example

The following command alters a queue named `work_order`, setting the `retry_delay` parameter to 5 seconds:

```sql
EXEC DBMS_AQADM.ALTER_QUEUE(queue_name => 'work_order', retry_delay => 5);
```

14.3.3.1.3.2 ALTER_QUEUE_TABLE

Use the `ALTER_QUEUE_TABLE` procedure to modify an existing queue table. The signature is:

```sql
ALTER_QUEUE_TABLE
(
    <queue_table> IN VARCHAR2,
    <comment> IN VARCHAR2 DEFAULT NULL,
    <primary_instance> IN BINARY_INTEGER DEFAULT 0,
    <secondary_instance> IN BINARY_INTEGER DEFAULT 0
);
```

Parameters

`queue_table`

The (optionally schema-qualified) name of the queue table.

`comment`

Use the `comment` parameter to provide a comment about the queue table.

`primary_instance`

`primary_instance` is accepted for compatibility and stored but is ignored.

`secondary_instance`

`secondary_instance` is accepted for compatibility but is ignored.

Example

The following command modifies a queue table named `work_order_table`:

```sql
EXEC DBMS_AQADM.ALTER_QUEUE_TABLE
    (queue_table => 'work_order_table', comment => 'This queue table contains work orders for the shipping department.');</sql>

The queue table is named `work_order_table`. The command adds a comment to the definition of the queue table.

14.3.3.1.3.3 CREATE_QUEUE

Use the `CREATE_QUEUE` procedure to create a queue in an existing queue table. The signature is:

```sql
CREATE_QUEUE
(
    <queue_name> IN VARCHAR2
    <queue_table> IN VARCHAR2
);
```
Parameters

queue_name

The name of the new queue.

queue_table

The name of the table in which the new queue resides.

queue_type

The type of the new queue. The valid values for queue_type are:

- DBMS_AQADM.NORMAL_QUEUE — This value specifies a normal queue (the default).
- DBMS_AQADM.EXCEPTION_QUEUE — This value specifies that the new queue is an exception queue. An exception queue supports only dequeue operations.

max_retries

max_retries specifies the maximum number of attempts to remove a message with a dequeue statement. The value of max_retries is incremented with each ROLLBACK statement. When the number of failed attempts reaches the value specified by max_retries, the message is moved to the exception queue. The default value for a system table is 0. The default value for a user-created table is 5.

retry_delay

retry_delay specifies the number of seconds until a message is scheduled for reprocessing after a ROLLBACK. Specify 0 to retry the message immediately (the default).

retention_time

retention_time specifies the length of time (in seconds) that a message is stored after being dequeued. You can also specify 0 (the default) to indicate not to retain the message after dequeuing or INFINITE to retain the message forever.

dependency_tracking

This parameter is accepted for compatibility and ignored.

comment

comment specifies a comment associated with the queue.

auto_commit

This parameter is accepted for compatibility and ignored.

Example

The following anonymous block creates a queue named work_order in the work_order_table table:

```sql
BEGIN
    DBMS_AQADM.CREATE_QUEUE ( queue_name => 'work_order', queue_table => 'work_order_table', comment => 'This queue contains pending work orders.');
END;
```

14.3.3.1.3.4 CREATE_QUEUE_TABLE
Use the `CREATE_QUEUE_TABLE` procedure to create a queue table. The signature is:

```
CREATE_QUEUE_TABLE
(
    <queue_table> IN VARCHAR2,
    <queue_payload_type> IN VARCHAR2 DEFAULT NULL,
    <storage_clause> IN VARCHAR2 DEFAULT NULL,
    <multiple_consumers> IN BOOLEAN DEFAULT FALSE,
    <message_grouping> IN BINARY_INTEGER DEFAULT NONE,
    <comment> IN VARCHAR2 DEFAULT NULL,
    <auto_commit> IN BOOLEAN DEFAULT TRUE,
    <primary_instance> IN BINARY_INTEGER DEFAULT 0,
    <secondary_instance> IN BINARY_INTEGER DEFAULT 0,
    <secure> IN VARCHAR2 DEFAULT FALSE
)
```

**Parameters**

- **<queue_table>**
  The (optionally schema-qualified) name of the queue table.

- **<queue_payload_type>**
  The user-defined type of the data that's stored in the queue table. To specify a RAW data type, you must create a user-defined type that identifies a RAW type.

- **<storage_clause>**
  Use the `storage_clause` parameter to specify attributes for the queue table. Only the TABLESPACE option is enforced. All others are accepted for compatibility and ignored. Use the TABLESPACE clause to specify the name of a tablespace in which to create the table.

  `storage_clause` can be one or more of the following:

  - **TABLESPACE**
    `tablespace_name`, `PCTFREE` integer, `PCTUSED` integer, `INITRANS` integer, `MAXTRANS` integer or `STORAGE` `storage_option`.

- **<storage_option>**
  `storage_option` can be one or more of the following:

  - `MINEXTENTS` integer, `MAXEXTENTS` integer, `PCTINCREASE` integer, `INITIAL size_clause`, `NEXT`, `FREELISTS` integer, `OPTIMAL size_clause`, `BUFFER_POOL` (KEEP|RECYCLE|DEFAULT).

- **<sort_list>**
  `sort_list` controls the dequeueing order of the queue. Specify the names of the columns to use to sort the queue in ascending order. The currently accepted values are the following combinations of `enq_time` and `priority`:

  - `enq_time, priority`
  - `priority, enq_time`
  - `priority`
  - `enq_time`

- **<multiple_consumers>**
  `multiple_consumers` queue tables isn't supported.

- **<message_grouping>**
  If specified, `message_grouping` must be `NONE`.

- **<comment>**
  Use the `comment` parameter to provide a comment about the queue table.
Example

The following anonymous block first creates a type (work_order) with attributes that hold a name (a VARCHAR2), and a project description (a TEXT). The block then uses that type to create a queue table:

```sql
BEGIN
    CREATE TYPE work_order AS (name VARCHAR2, project TEXT, completed BOOLEAN);
    EXEC DBMS_AQADM.CREATE_QUEUE_TABLE (queue_table => 'work_order_table',
                                            queue_payload_type => 'work_order',
                                            comment => 'Work order message queue table');
END;
```

The queue table is named work_order_table and contains a payload of a type work_order. A comment notes that this is the Work order message queue table.

### 14.3.3.1.3.5 DROP_QUEUE

Use the DROP_QUEUE procedure to delete a queue. The signature is:

```sql
DROP_QUEUE(
    <queue_name> IN VARCHAR2,
    <auto_commit> IN BOOLEAN DEFAULT TRUE
)
```

Parameters

- **queue_name**

  The name of the queue that you want to drop.

- **auto_commit**

  auto_commit is accepted for compatibility but is ignored.

Example

The following anonymous block drops the queue named work_order:
14.3.3.1.3.6 DROP_QUEUE_TABLE

Use the DROP_QUEUE_TABLE procedure to delete a queue table. The signature is:

```
DROP_QUEUE_TABLE(
  <queue_table> IN VARCHAR2,
  <force> IN BOOLEAN default FALSE,
  <auto_commit> IN BOOLEAN default TRUE)
```

Parameters

**queue_table**

The (optionally schema-qualified) name of the queue table.

**force**

The force keyword determines the behavior of the DROP_QUEUE_TABLE command when dropping a table that contain entries:

- If the target table contains entries and force is FALSE, the command fails, and the server issues an error.
- If the target table contains entries and force is TRUE, the command drops the table and any dependent objects.

**auto_commit**

auto_commit is accepted for compatibility but is ignored.

Example

The following anonymous block drops a table named work_order_table:

```
BEGIN
  DBMS_AQADM.DROP_QUEUE_TABLE (queue_name => 'work_order', force => TRUE);
END;
```

14.3.3.1.3.7 PURGE_QUEUE_TABLE

Use the PURGE_QUEUE_TABLE procedure to delete messages from a queue table. The signature is:

```
PURGE_QUEUE_TABLE(
  <queue_table> IN VARCHAR2,
  <purge_condition> IN VARCHAR2,
  <purge_options> IN aq$_purge_options_t)
```

Parameters

**queue_table**

**queue_table** specifies the name of the queue table from which you're deleting a message.

**purge_condition**

Use purge_condition to specify a condition (a SQL WHERE clause) that the server evaluates when deciding which messages to purge.
purge_options

purge_options is an object of the type `aq$_purge_options_t`. An `aq$_purge_options_t` object contains:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>Boolean</td>
<td>Specify <code>TRUE</code> to hold an exclusive lock on all queues in the table. The default is <code>FALSE</code>.</td>
</tr>
<tr>
<td>delivery_mode</td>
<td>INTEGER</td>
<td><code>delivery_mode</code> specifies the type of message to purge. The only accepted value is <code>DBMS_AQ.PERSISTENT</code>.</td>
</tr>
</tbody>
</table>

Example

The following anonymous block removes any messages from the `work_order_table` with a value in the completed column of `YES`:

```sql
DECLARE
    purge_options dbms_aqadm.aq$_purge_options_t;
BEGIN
    dbms_aqadm.purge_queue_table('work_order_table', 'completed = YES', purge_options);
END;
```

14.3.3.1.3.8 START_QUEUE

Use the `START_QUEUE` procedure to make a queue available for enqueueing and dequeueing. The signature is:

```sql
START_QUEUE(    <queue_name> IN VARCHAR2,    <enqueue> IN BOOLEAN DEFAULT TRUE,    <dequeue> IN BOOLEAN DEFAULT TRUE)
```

Parameters

queue_name

`queue_name` specifies the name of the queue that you're starting.

enqueue

Specify `TRUE` to enable enqueueing (the default) or `FALSE` to leave the current setting unchanged.

decqueue

Specify `TRUE` to enable dequeueing (the default) or `FALSE` to leave the current setting unchanged.

Example

The following anonymous block makes a queue named `work_order` available for enqueueing:

```sql
BEGIN
    DBMS_AQADM.START_QUEUE (queue_name => 'work_order);
END;
```

14.3.3.1.3.9 STOP_QUEUE

Use the `STOP_QUEUE` procedure to disable enqueueing or dequeueing on a specified queue. The signature is:
Parameters

queue_name

queue_name specifies the name of the queue that you’re stopping.

nenqueue

Specify TRUE to disable enqueueing (the default) or FALSE to leave the current setting unchanged.

dequeue

Specify TRUE to disable dequeueing (the default) or FALSE to leave the current setting unchanged.

wait

Specify TRUE to instruct the server to wait for any uncompleted transactions to complete before applying the specified changes. While waiting to stop the queue, no transactions are allowed to enqueue or dequeue from the specified queue. Specify FALSE to stop the queue immediately.

Example

The following anonymous block disables enqueueing and dequeueing from the queue named work_order:

```
BEGIN
  DBMS_AQADM.STOP_QUEUE(queue_name => 'work_order', enqueue => TRUE, dequeue => TRUE, wait => TRUE);
END;
```

Enqueuing and dequeueing stops after any outstanding transactions complete.

14.3.3.1.4 DBMS_CRYPTO

The DBMS_CRYPTO package provides functions and procedures that allow you to encrypt or decrypt RAW, BLOB, or CLOB data. You can also use DBMS_CRYPTO functions to generate cryptographically strong random values.

The table lists the DBMS_CRYPTO functions and procedures.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECRYPT(src, typ, key, iv)</td>
<td>RAW</td>
<td>Decrypts RAW data.</td>
</tr>
<tr>
<td>DECRYPT(dst INOUT, src, typ, key, iv)</td>
<td>N/A</td>
<td>Decrypts BLOB data.</td>
</tr>
<tr>
<td>DECRYPT(dst INOUT, src, typ, key, iv)</td>
<td>N/A</td>
<td>Decrypts CLOB data.</td>
</tr>
<tr>
<td>ENCRYPT(src, typ, key, iv)</td>
<td>RAW</td>
<td>Encrypts RAW data.</td>
</tr>
<tr>
<td>ENCRYPT(dst INOUT, src, typ, key, iv)</td>
<td>N/A</td>
<td>Encrypts BLOB data.</td>
</tr>
<tr>
<td>ENCRYPT(dst INOUT, src, typ, key, iv)</td>
<td>N/A</td>
<td>Encrypts CLOB data.</td>
</tr>
<tr>
<td>HASH(src, typ)</td>
<td>RAW</td>
<td>Applies a hash algorithm to RAW data.</td>
</tr>
<tr>
<td>MAC(src, typ, key)</td>
<td>RAW</td>
<td>Returns the hashed MAC value of the given RAW data using the specified hash algorithm and key.</td>
</tr>
<tr>
<td>MAC(src, typ, key)</td>
<td>RAW</td>
<td>Returns the hashed MAC value of the given CLOB data using the specified hash algorithm and key.</td>
</tr>
<tr>
<td>RANDOMBYTES(number_bytes)</td>
<td>RAW</td>
<td>Returns a specified number of cryptographically strong random bytes.</td>
</tr>
<tr>
<td>RANDOMINTEGER()</td>
<td>INTEGER</td>
<td>Returns a random integer.</td>
</tr>
</tbody>
</table>
Function/procedure | Return type | Description
--- | --- | ---
RANDONNUMBER() | NUMBER | Returns a random number.

**DBMS_CRYPTO** functions and procedures support the following error messages:

- ORA-28239 - DBMS_CRYPTO.KeyNull
- ORA-28829 - DBMS_CRYPTO.CipherSuiteNull
- ORA-28827 - DBMS_CRYPTO.CipherSuiteInvalid

Unlike Oracle, EDB Postgres Advanced Server doesn't return error ORA-28233 if you reencrypt previously encrypted information.

**RAW** and **BLOB** are synonyms for the PostgreSQL **BYTEA** data type. **CLOB** is a synonym for **TEXT**.

### 14.3.3.1.4.1 DECRYPT

The **DECRYPT** function or procedure decrypts data using a user-specified cipher algorithm, key, and optional initialization vector. The signature of the **DECRYPT** function is:

```sql
DECRYPT (
  <src> IN RAW, <typ> IN INTEGER, <key> IN RAW, <iv> IN RAW DEFAULT NULL) RETURN RAW
```

The signature of the **DECRYPT** procedure is:

```sql
DECRYPT (
  <dst> INOUT BLOB, <src> IN BLOB, <typ> IN INTEGER, <key> IN RAW, 
  <iv> IN RAW DEFAULT NULL)
```

or

```sql
DECRYPT (
  <dst> INOUT CLOB, <src> IN CLOB, <typ> IN INTEGER, <key> IN RAW, 
  <iv> IN RAW DEFAULT NULL)
```

When invoked as a procedure, **DECRYPT** returns **BLOB** or **CLOB** data to a user-specified **BLOB**.

#### Parameters

- **dst**
  - Specifies the name of a **BLOB** to which the output of the **DECRYPT** procedure is written. The **DECRYPT** procedure overwrites any existing data currently in **dst**.

- **src**
  - Specifies the source data to decrypt. If you're invoking **DECRYPT** as a function, specify **RAW** data. If invoking **DECRYPT** as a procedure, specify **BLOB** or **CLOB** data.

- **typ**
  - Specifies the block cipher type and any modifiers. Match the type specified when the **src** was encrypted. EDB Postgres Advanced Server supports the following block cipher algorithms, modifiers, and cipher suites:

  **Block cipher algorithms**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCRYPT_DES</td>
<td>1</td>
</tr>
<tr>
<td>ENCRYPT_3DES</td>
<td>3</td>
</tr>
<tr>
<td>ENCRYPT_AES</td>
<td>4</td>
</tr>
<tr>
<td>ENCRYPT_AES128</td>
<td>6</td>
</tr>
<tr>
<td>ENCRYPT_AES192</td>
<td>192</td>
</tr>
<tr>
<td>ENCRYPT_AES256</td>
<td>256</td>
</tr>
</tbody>
</table>

  **Block cipher modifiers**
Block cipher padding modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Constant integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAD_PKCS5</td>
<td>4096</td>
</tr>
<tr>
<td>PAD_NONE</td>
<td>8192</td>
</tr>
</tbody>
</table>

Block cipher suites

<table>
<thead>
<tr>
<th>Suite Name</th>
<th>Constant Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_CBC_PKCS5</td>
<td>256 + 768 + 4096</td>
</tr>
<tr>
<td>DES3_CBC_PKCS5</td>
<td>256 + 768 + 4096</td>
</tr>
<tr>
<td>AES_CBC_PKCS5</td>
<td>256 + 768 + 4096</td>
</tr>
</tbody>
</table>

**key** specifies the user-defined decryption key. Match the key specified when the src was encrypted.

**iv** (optional) specifies an initialization vector. If an initialization vector was specified when the src was encrypted, you must specify an initialization vector when decrypting the src. The default is NULL.

**Examples**

This example uses the DBMS_CRYPTO.DECRYPT function to decrypt an encrypted password retrieved from the passwords table:

```sql
CREATE TABLE passwords
(
    principal VARCHAR2(90) PRIMARY KEY,
    username VARCHAR2(90) -- encrypted
);

CREATE FUNCTION get_password(username VARCHAR2) RETURN RAW
AS
BEGIN
    SELECT ciphertext INTO password FROM passwords WHERE principal = username;
    RETURN dbms_crypto.decrypt(password, typ, key, iv);
END;
```

When calling DECRYPT, you must pass the same cipher type, key value, and initialization vector that was used when encrypting the target.

**14.3.3.1.4.2 ENCRYPT**

The ENCRYPT function or procedure uses a user-specified algorithm, key, and optional initialization vector to encrypt RAW, BLOB, or CLOB data. The signature of the ENCRYPT function is:

```sql
ENCRYPT ((<src> IN RAW, <typ> IN INTEGER, <key> IN RAW,
           <iv> IN RAW DEFAULT NULL)) RETURN RAW
```

The signature of the ENCRYPT procedure is:

```sql
ENCRYPT ((<dst> INOUT BLOB, <src> IN BLOB, <typ> IN INTEGER, <key> IN RAW,
           <iv> IN RAW DEFAULT NULL))
```
When invoked as a procedure, ENCRYPT returns BLOB or CLOB data to a user-specified BLOB.

**Parameters**

**dst**

dst specifies the name of a BLOB to which to write the output of the ENCRYPT procedure. The ENCRYPT procedure overwrites any existing data currently in dst.

**src**

src specifies the source data to encrypt. If you're invoking ENCRYPT as a function, specify RAW data. If invoking ENCRYPT as a procedure, specify BLOB or CLOB data.

**typ**

typ specifies the block cipher type used by ENCRYPT and any modifiers. EDB Postgres Advanced Server supports the block cipher algorithms, modifiers, and cipher suites shown in the table.

### Block cipher algorithms

- ENCRYPT_DES
- ENCRYPT_3DES
- ENCRYPT_AES
- ENCRYPT_AES128
- ENCRYPT_AES192
- ENCRYPT_AES256

### Block cipher modifiers

- CHAIN_CBC
- CHAIN_ECB

### Block cipher padding modifiers

- PAD_PKCS5
- PAD_NONE

### Block cipher suites

- DES_CBC_PKCS5
- DES3_CBC_PKCS5
- AES_CBC_PKCS5

**key**

key specifies the encryption key.

**iv**

iv (optional) specifies an initialization vector. By default, iv is NULL.

### Examples

This example uses the DBMS_CRYPTO.DES_CBC_PKCS5 Block Cipher Suite (a predefined set of algorithms and modifiers) to encrypt a value retrieved from the passwords table:

```sql
CREATE TABLE passwords
(
    principal VARCHAR2(90) PRIMARY KEY, --
    username VARCHAR2(90) -- encrypted
) AS
CREATE PROCEDURE set_password(username VARCHAR2, cleartext RAW)
```
ENCRYPT uses a key value of my secret key and an initialization vector of my initialization vector when encrypting the password. Specify the same key and initialization vector when decrypting the password.

14.3.3.1.4.3 HASH

The HASH function uses a user-specified algorithm to return the hash value of a RAW or CLOB value. The HASH function is available in three forms:

\[
\text{HASH} \left( \langle \text{src} \rangle \text{ IN RAW}, \langle \text{typ} \rangle \text{ IN INTEGER} \right) \text{ RETURN RAW}
\]

\[
\text{HASH} \left( \langle \text{src} \rangle \text{ IN CLOB}, \langle \text{typ} \rangle \text{ IN INTEGER} \right) \text{ RETURN RAW}
\]

Parameters

**src**

- **src** specifies the value for which the hash value is generated. You can specify a RAW, BLOB, or CLOB value.

**typ**

- **typ** specifies the HASH function type. EDB Postgres Advanced Server supports the HASH function types shown in the table.

**HASH functions**

- \text{HASH MD4} \quad \text{CONSTANT INTEGER} := 1;
- \text{HASH MD5} \quad \text{CONSTANT INTEGER} := 2;
- \text{HASH SH1} \quad \text{CONSTANT INTEGER} := 3;

Examples

This example uses \text{DBMS_CRYPTO.HASH} to find the MD5 hash value of the string, cleartext source:

```
DECLARE
    typ INTEGER := DBMS_CRYPTO.HASH_MD5;
    hash_value RAW(180);
BEGIN
    hash_value := DBMS_CRYPTO.HASH('cleartext source', typ);
END;
```

14.3.3.1.4.4 MAC

The MAC function uses a user-specified MAC function to return the hashed MAC value of a RAW or CLOB value. The MAC function is available in three forms:

```
DECLARE
    typ INTEGER := DBMS_CRYPTO.DES_CBC_PKCS;
    key RAW(128) := 'my secret key';
    iv RAW(100) := 'my initialization vector';
    encrypted RAW(2048);
BEGIN
    encrypted := dbms_crypto.encrypt(cleartext, typ, key, iv);
    UPDATE passwords SET ciphertext = encrypted WHERE principal = username;
END;
```

ENCRYPT uses a key value of my secret key and an initialization vector of my initialization vector when encrypting the password. Specify the same key and initialization vector when decrypting the password.
Parameters

src

src specifies the value for which the MAC value is generated. Specify a RAW, BLOB, or CLOB value.

typ

typ specifies the MAC function used. EDB Postgres Advanced Server supports the MAC functions shown in the table.

MAC functions

<table>
<thead>
<tr>
<th>HMAC_MD5</th>
<th>CONSTANT INTEGER := 1;</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMAC_SH1</td>
<td>CONSTANT INTEGER := 2;</td>
</tr>
</tbody>
</table>

key

key specifies the key used to calculate the hashed MAC value.

Examples

This example finds the hashed MAC value of the string cleartext source:

```sql
DECLARE
typ INTEGER := DBMS_CRYPTO.HMAC_MD5;
key RAW(100) := 'my secret key';
mac_value RAW(100);
BEGIN
  mac_value := DBMS_CRYPTO.MAC('cleartext source', typ, key);
END;
```

DBMS_CRYPTO.MAC uses a key value of my secret key when calculating the MAC value of cleartext source.

14.3.3.1.4.5 RANDOMBYTES

The RANDOMBYTES function returns a RAW value of the specified length, containing cryptographically random bytes. The signature is:

```sql
RANDOMBYTES
  (IN number_bytes INTEGER) RETURNS RAW
```

Parameter

number_bytes

number_bytes specifies the number of random bytes to return.

Examples
This example uses `RANDOMBYTES` to return a value that is 1024 bytes long:

```sql
DECLARE
  result RAW(1024);
BEGIN
  result := DBMS_CRYPTO.RANDOMBYTES(1024);
END;
```

### 14.3.3.1.4.6 RANDOMINTEGER

The `RANDOMINTEGER()` function returns a random integer between 0 and 2,684,354,555. The signature is:

```sql
RANDOMINTEGER() RETURNS INTEGER
```

**Examples**

This example uses the `RANDOMINTEGER` function to return a cryptographically strong random `INTEGER` value:

```sql
DECLARE
  result INTEGER;
BEGIN
  result := DBMS_CRYPTO.RANDOMINTEGER();
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

### 14.3.3.1.4.7 RANDOMNUMBER

The `RANDOMNUMBER()` function returns a random number between 0 and 2,684,354,555. The signature is:

```sql
RANDOMNUMBER() RETURNS NUMBER
```

**Examples**

This example uses the `RANDOMNUMBER` function to return a cryptographically strong random number:

```sql
DECLARE
  result NUMBER;
BEGIN
  result := DBMS_CRYPTO.RANDOMNUMBER();
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

### 14.3.3.1.5 DBMS_JOB

The `DBMS_JOB` package's jobs are scheduled and run in the background by the `edb_job_scheduler` extension. For more information, see [EDB Job Scheduler extension](#).

The `DBMS_JOB` package lets you create, schedule, and manage jobs. A job runs a stored procedure that was previously stored in the database. The `SUBMIT` procedure creates and stores a job definition. A job identifier is assigned to a job with a stored procedure and the attributes describing when and how often to run the job.

EDB Postgres Advanced Server's implementation of `DBMS_JOB` is a partial implementation when compared to Oracle's version. The following table lists the supported `DBMS_JOB` procedures.
### 14.3.3.1.5.1 BROKEN

The **BROKEN** procedure sets the state of a job to either broken or not broken. You can execute a broken job only by using the **RUN** procedure.

```sql
BROKEN(job, broken [, next_date ])  
```

**Parameters**

- `job`
  Identifier of the job to set as broken or not broken.

- `broken`
  If set to **TRUE**, the job's state is set to broken. If set to **FALSE**, the job's state is set to not broken. You can run broken jobs only by using the **RUN** procedure.

- `next_date`
  Date/time when the job is to be run. The default is **SYSDATE**.

**Examples**

Set the state of a job with job identifier 104 to broken:

```sql
BEGIN
  BROKEN(104, TRUE, SYSDATE);
END;
```
14.3.3.1.5.2 CHANGE

The CHANGE procedure modifies certain job attributes including the stored procedure to run, the next date/time the job runs, and how often it runs.

Change the state back to not broken:

BEGIN
  DBMS_JOB.BROKEN(104,true);
END;

BEGIN
  DBMS_JOB.BROKEN(104,false);
END;

The CHANGE procedure modifies certain job attributes including the stored procedure to run, the next date/time the job runs, and how often it runs.

```sql
CHANGE(<job> BINARY_INTEGER <what> VARCHAR2, <next_date> DATE,
      <interval> VARCHAR2, <instance> BINARY_INTEGER, <force> BOOLEAN)
```

Parameters

**job**
Identifier of the job to modify.

**what**
Stored procedure name. Set this parameter to null if you want the existing value to remain unchanged.

**next_date**
Date/time to run the job next. Set this parameter to null if you want the existing value to remain unchanged.

**interval**
Date function that, when evaluated, provides the next date/time to run the job. Set this parameter to null if you want the existing value to remain unchanged.

**instance**
This argument is ignored but is included for compatibility.

**force**
This argument is ignored but is included for compatibility.

Examples

Change the job to run next on December 13, 2007. Leave other parameters unchanged.

```sql
BEGIN
  DBMS_JOB.CHANGE(104,NULL,TO_DATE('13-DEC-07','DD-MON-YY'),NULL, NULL);
END;
```

14.3.3.1.5.3 INTERVAL

The INTERVAL procedure sets how often a job.

```sql
INTERVAL(<job> BINARY_INTEGER, <interval> VARCHAR2)
```
Parameters

**job**

Identifier of the job to modify.

**interval**

Date function that, when evaluated, provides the next date/time to run the job. If `interval` is **NULL** and the job is complete, the job is removed from the queue.

Examples

Change the job to run once a week:

```sql
BEGIN
  DBMS_JOB.INTERVAL(104, 'SYSDATE + 7');
END;
```

14.3.3.1.5.4 **NEXT_DATE**

The `NEXT_DATE` procedure sets the date/time to run the job next.

```sql
NEXT_DATE(<job> BINARY_INTEGER, <next_date> DATE)
```

Parameters

**job**

Identifier of the job whose next run date you want to set.

**next_date**

Date/time when you want the job run next.

Examples

Change the job to run next on December 14, 2007:

```sql
BEGIN
  DBMS_JOB.NEXT_DATE(104, TO_DATE('14-DEC-07', 'DD-MON-YY'));
END;
```

14.3.3.1.5.5 **REMOVE**

The `REMOVE` procedure deletes the specified job from the database. You must resubmit the job using the `SUBMIT` procedure to execute it again. The stored procedure that was associated with the job isn't deleted.

```sql
REMOVE(<job> BINARY_INTEGER)
```

Parameter

**job**
Identifier of the job to remove from the database.

Examples

Remove a job from the database:

```sql
BEGIN
  DBMS_JOB.REMOVE(104);
END;
```

14.3.3.1.5.6 **RUN**

The **RUN** procedure forces the job to run even if its state is broken.

```sql
RUN(<job> BINARY_INTEGER)
```

Parameter

**job**

Identifier of the job to run.

Examples

Force a job to run.

```sql
BEGIN
  DBMS_JOB.RUN(104);
END;
```

14.3.3.1.5.7 **SUBMIT**

The **SUBMIT** procedure creates a job definition and stores it in the database. A job consists of:

- A job identifier
- The stored procedure to execute
- When to first run the job
- A date function that calculates the next date/time for the job to run

```sql
SUBMIT(<job> OUT BINARY_INTEGER, <what> VARCHAR2
  [, <next_date> DATE [, <interval> VARCHAR2 [, <no_parse> BOOLEAN ]]])
```

Parameters

**job**

Identifier assigned to the job.

**what**

Name of the stored procedure for the job to execute.

**next_date**
Date/time to run the job next. The default is `SYSDATE`.

```
interval
```

Date function that, when evaluated, provides the next date/time for the job to run. If `interval` is set to null, then the job runs only once. Null is the default.

```
no_parse
```

If set to `TRUE`, don't syntax-check the stored procedure upon job creation. Check only when the job first executes. If set to `FALSE`, check the procedure upon job creation. The default is `FALSE`.

**Note**

The `no_parse` option isn't supported in this implementation of `SUBMIT()`. It's included only for compatibility.

---

**Examples**

This example creates a job using the stored procedure `job_proc`. The job executes immediately and runs once a day after that, as set by the `interval` parameter, `SYSDATE + 1`.

```sql
DECLARE
  jobid INTEGER;
BEGIN
  DBMS_JOB.SUBMIT(jobid,'job_proc;',SYSDATE,
    'SYSDATE + 1');
  DBMS_OUTPUT.PUT_LINE('jobid: ' || jobid);
END;
```

The job immediately executes the procedure `job_proc`, populating the table `jobrun` with a row:

```
SELECT * FROM jobrun;
```

```
-----------------------------------------------------
job_proc run at 2007-12-11 11:43:25
(1 row)
```

---

### 14.3.3.1.5.8 WHAT

The `WHAT` procedure changes the stored procedure that the job executes.

```
WHAT(<job> BINARY_INTEGER, <what> VARCHAR2)
```

**Parameters**

```
job
```

Identifier of the job whose stored procedure you want to change.

```
what
```

Name of the stored procedure to execute.

**Examples**

Change the job to run the `list_emp` procedure:

```
BEGIN
  DBMS_JOB.WHAT(104,'list_emp');
END;
```
The **DBMS_LOB** package lets you operate on large objects. The following table lists the supported functions and procedures. EDB Postgres Advanced Server's implementation of **DBMS_LOB** is a partial implementation when compared to Oracle's version. Only the functions and procedures listed in this table are supported.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND(dest_lob IN OUT, src_lob)</td>
<td>n/a</td>
<td>Appends one large object to another.</td>
</tr>
<tr>
<td>COMPAIR(lob_1, lob_2 [, amount [, offset_1 [, offset_2 ]]])</td>
<td>INTEGER</td>
<td>Compares two large objects.</td>
</tr>
<tr>
<td>CONVERTLOB(dest_lob IN OUT, src_clob, amount, dest_offset IN OUT, blob_csid, lang_context IN OUT, warning OUT)</td>
<td>n/a</td>
<td>Converts character data to binary.</td>
</tr>
<tr>
<td>CONVERTTOCLOB(dest_lob IN OUT, src_blob, amount, dest_offset IN OUT, blob_csid, lang_context IN OUT, warning OUT)</td>
<td>n/a</td>
<td>Converts binary data to character.</td>
</tr>
<tr>
<td>COPY(dest_lob IN OUT, src_lob, amount [, dest_offset [, src_offset ]])</td>
<td>n/a</td>
<td>Copies one large object to another.</td>
</tr>
<tr>
<td>ERASE(lob_loc IN OUT, amount IN OUT [, offset ])</td>
<td>n/a</td>
<td>Erases a large object.</td>
</tr>
<tr>
<td>GET_STORAGE_LIMIT(lob_loc)</td>
<td>INTEGER</td>
<td>Gets the storage limit for large objects.</td>
</tr>
<tr>
<td>GETLENGTH(lob_loc)</td>
<td>INTEGER</td>
<td>Gets the length of the large object.</td>
</tr>
<tr>
<td>INSTR(lob_loc, pattern [, offset [, nth ]])</td>
<td>INTEGER</td>
<td>Gets the position of the nth occurrence of a pattern in the large object starting at offset.</td>
</tr>
<tr>
<td>READ(lob_loc, amount IN OUT, offset, buffer OUT)</td>
<td>n/a</td>
<td>Reads a large object.</td>
</tr>
<tr>
<td>SUBSTR(lob_loc [, amount [, offset ]])</td>
<td>RAW, VARCHAR2</td>
<td>Gets part of a large object.</td>
</tr>
<tr>
<td>TRIM(lob_loc IN OUT, newlen)</td>
<td>n/a</td>
<td>Trims a large object to the specified length.</td>
</tr>
<tr>
<td>WRITE(lob_loc IN OUT, amount, offset, buffer)</td>
<td>n/a</td>
<td>Writes data to a large object.</td>
</tr>
<tr>
<td>WRITEAPPEND(lob_loc IN OUT, amount, buffer)</td>
<td>n/a</td>
<td>Writes data from the buffer to the end of a large object.</td>
</tr>
</tbody>
</table>

The following table lists the public variables available in the package.

<table>
<thead>
<tr>
<th>Public variables</th>
<th>Data type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress off</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>compress on</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>deduplicate off</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>deduplicate on</td>
<td>INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>default_csid</td>
<td>INTEGER</td>
<td>8</td>
</tr>
<tr>
<td>default_lang_ctx</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>encrypt off</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>encrypt on</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>file_readonly</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>lobmaxsize</td>
<td>INTEGER</td>
<td>1073741823</td>
</tr>
<tr>
<td>lob_readonly</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>lob_readwrite</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>no_warning</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>opt_compress</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>opt_deduplicate</td>
<td>INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>opt_encrypt</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>warn_inconvertible_char</td>
<td>INTEGER</td>
<td>1</td>
</tr>
</tbody>
</table>

Lengths and offsets are measured in bytes if the large objects are **BLOB**. Lengths and offsets are measured in characters if the large objects are **CLOB**.

### 14.3.3.1.6.1 APPEND

The **APPEND** procedure appends one large object to another. Both large objects must be the same type.
APPEND(<dest_lob> IN OUT { BLOB | CLOB }, <src_lob> { BLOB | CLOB })

Parameters

dest_lob
Large object locator for the destination object. Must be the same data type as src_lob.

src_lob
Large object locator for the source object. Must be the same data type as dest_lob.

14.3.3.1.6.2 COMPARE

The COMPARE procedure performs an exact byte-by-byte comparison of two large objects for a given length at given offsets. The large objects being compared must be the same data type.

<status> INTEGER COMPARE(<lob_1> { BLOB | CLOB }, <lob_2> { BLOB | CLOB }) [, <amount> INTEGER [, <offset_1> INTEGER [, <offset_2> INTEGER ]]]

Parameters

lob_1
Large object locator of the first large object to compare. Must be the same data type as lob_2.

lob_2
Large object locator of the second large object to compare. Must be the same data type as lob_1.

amount
If the data type of the large objects is BLOB, then the comparison is made for amount bytes. If the data type of the large objects is CLOB, then the comparison is made for amount characters. The default is the maximum size of a large object.

offset_1
Position in the first large object to begin the comparison. The first byte/character is offset 1. The default is 1.

offset_2
Position in the second large object to begin the comparison. The first byte/character is offset 1. The default is 1.

status
Zero if both large objects are exactly the same for the specified length for the specified offsets. Nonzero if the objects aren't the same. NULL if amount, offset_1, or offset_2 are less than zero.

14.3.3.1.6.3 CONVERTTOBLOB

The CONVERTTOBLOB procedure converts character data to binary.

CONVERTTOBLOB(<dest_lob> IN OUT BLOB, <src_clob> CLOB, <amount> INTEGER, <dest_offset> IN OUT INTEGER, <src_offset> IN OUT INTEGER, <blob_csid> NUMBER, <lang_context> IN OUT INTEGER, <warning> OUT INTEGER)
Parameters

dest_lob

BLOB large object locator to which to convert the character data.

clob

CLOB large object locator of the character data to convert.

amount

Number of characters of src_clob to convert.

dest_offset IN

Position in bytes in the destination BLOB where you want to begin writing the source CLOB. The first byte is offset 1.

dest_offset OUT

Position in bytes in the destination BLOB after the write operation completes. The first byte is offset 1.

src_offset IN

Position in characters in the source CLOB where you want to begin conversion to the destination BLOB. The first character is offset 1.

src_offset OUT

Position in characters in the source CLOB after the conversion operation completes. The first character is offset 1.

blob_csid

Character set ID of the converted destination BLOB.

lang_context IN

Language context for the conversion. The default value of 0 is typically used for this setting.

lang_context OUT

Language context after the conversion completes.

warning

0 if the conversion was successful, 1 if a character can't be converted.

14.3.3.1.6.4 CONVERTTOCLOB

The CONVERTTOCLOB procedure converts binary data to character.

CONVERTTOCLOB(<dest_lob> IN OUT CLOB, <src_blob> BLOB, <amount> INTEGER, <dest_offset> IN OUT INTEGER, <src_offset> IN OUT INTEGER, <blob_csid> NUMBER, <lang_context> IN OUT INTEGER, <warning> OUT INTEGER)

Parameters

dest_lob

CLOB large object locator to which to convert the binary data.

src_blob
BLOB large object locator of the binary data to convert.

Number of bytes of src_blob to convert.

dest_offset IN

Position in characters in the destination CLOB where you want to begin writing the source BLOB. The first character is offset 1.

dest_offset OUT

Position in characters in the destination CLOB after the write operation completes. The first character is offset 1.

src_offset IN

Position in bytes in the source BLOB where you want the conversion to the destination CLOB to begin. The first byte is offset 1.

src_offset OUT

Position in bytes in the source BLOB after the conversion operation completes. The first byte is offset 1.

blob_csid

Character set ID of the converted destination CLOB.

lang_context IN

Language context for the conversion. The default value of 0 is typically used for this setting.

lang_context OUT

Language context after the conversion completes.

warning

0 if the conversion was successful, 1 if a character can’t be converted.

14.3.3.1.6.5 COPY

The COPY procedure copies one large object to another. The source and destination large objects must be the same data type.

COPY(<dest_lob> IN OUT ( BLOB | CLOB ),
     <src_lob> IN OUT ( BLOB | CLOB ),
     <amount> INTEGER,
     [, <dest_offset> INTEGER [, <src_offset> INTEGER ]])

Parameters

dest_lob

Large object locator of the large object to which you want to copy src_lob. Must be the same data type as src_lob.

src_lob

Large object locator of the large object to copy to dest_lob. Must be the same data type as dest_lob.

amount

Number of bytes/characters of src_lob to copy.

dest_offset

Position in characters in the destination CLOB after the write operation completes. The first character is offset 1.
Position in the destination large object where you want writing of the source large object to begin. The first position is offset 1. The default is 1.

Position in the source large object where you want copying to the destination large object to begin. The first position is offset 1. The default is 1.

### 14.3.3.1.6.6 ERASE

The ERASE procedure erases a portion of a large object. To erase a large object means to replace the specified portion with zero-byte fillers for BLOB or with spaces for CLOB. The actual size of the large object isn’t altered.

```sql
ERASE(<lob_loc> IN OUT ( BLOB | CLOB ), <amount> IN OUT
INTEGER [, <offset> INTEGER ])
```

**Parameters**

- **lob_loc**
  Large object locator of the large object to erase.

- **amount IN**
  Number of bytes/characters to erase.

- **amount OUT**
  Number of bytes/characters erased. This value can be smaller than the input value if the end of the large object is reached before `amount` bytes/characters are erased.

- **offset**
  Position in the large object where erasing begins. The first byte/character is position 1. The default is 1.

### 14.3.3.1.6.7 GET_STORAGE_LIMIT

The GET_STORAGE_LIMIT function returns the limit on the largest allowable large object.

```sql
<size> INTEGER GET_STORAGE_LIMIT(<lob_loc> BLOB)
<size> INTEGER GET_STORAGE_LIMIT(<lob_loc> CLOB)
```

**Parameters**

- **size**
  Maximum allowable size of a large object in this database.

- **lob_loc**
  This parameter is ignored but is included for compatibility.

### 14.3.3.1.6.8 GETLENGTH

The GETLENGTH function returns the length of a large object.

```sql
<amount> INTEGER GETLENGTH(<lob_loc> BLOB)
```
Parameters

**lob_loc**

Large object locator of the large object whose length to obtain.

**amount**

Length of the large object for **BLOB** or characters for **CLOB**, in bytes.

### 14.3.3.1.6.9 INSTR

The **INSTR** function returns the location of the nth occurrence of a given pattern in a large object.

```sql
INSTR(lob_loc, pattern, offset [nth])
```

**Parameters**

**lob_loc**

Large object locator of the large object in which to search for **pattern**.

**pattern**

Pattern of bytes or characters to match against the large object. **lob**. **pattern** must be **RAW** if **lob_loc** is a **BLOB**. **pattern** must be **VARCHAR2** if **lob_loc** is a **CLOB**.

**offset**

Position in **lob_loc** to start search for **pattern**. The first byte/character is position 1. The default is 1.

**nth**

Search for **pattern**, nth number of times starting at the position given by **offset**. The default is 1.

**position**

Position in the large object where **pattern** appears the nth time, starting from the position given by **offset**.

### 14.3.3.1.6.10 READ

The **READ** procedure reads a portion of a large object into a buffer.

```sql
READ(lob_loc, amount) INTO OUT buffer
```

**Parameters**

**lob_loc**

Large object locator of the large object to read.
amount IN

Number of bytes/characters to read.

amount OUT

Number of bytes/characters read. If there’s no more data to read, then amount returns 0 and a DATA_NOT_FOUND exception is thrown.

offset

Position to begin reading. The first byte/character is position 1.

buffer

Variable to receive the large object. If lob_loc is a BLOB, then buffer must be RAW. If lob_loc is a CLOB, then buffer must be VARCHAR2.

14.3.3.1.6.11 SUBSTR

The SUBSTR function returns a portion of a large object.

```
<data> { RAW | VARCHAR2 } SUBSTR(<lob_loc> { BLOB | CLOB }
| [{, <amount> INTEGER [, <offset> INTEGER ]})
```

Parameters

lob_loc

Large object locator of the large object to read.

amount

Number of bytes/characters to return. Default is 32,767.

offset

Position in the large object to begin returning data. The first byte/character is position 1. The default is 1.

data

Returned portion of the large object to read. If lob_loc is a BLOB, the return data type is RAW. If lob_loc is a CLOB, the return data type is VARCHAR2.

14.3.3.1.6.12 TRIM

The TRIM procedure truncates a large object to the specified length.

```
TRIM(<lob_loc> IN OUT { BLOB | CLOB }, <newlen> INTEGER)
```

Parameters

lob_loc

Large object locator of the large object to trim.

newlen

Number of bytes/characters to which you want to trim the large object.
14.3.3.1.6.13  WRITE

The WRITE procedure writes data into a large object. Any existing data in the large object at the specified offset for the given length is overwritten by data given in the buffer.

```sql
WRITE(<lob_loc> IN OUT ( BLOB | CLOB ),
       <amount> BINARY_INTEGER,
       <offset> INTEGER, <buffer> ( RAW | VARCHAR2 ))
```

**Parameters**

- **lob_loc**
  Large object locator of the large object to write.

- **amount**
  The number of bytes/characters in `buffer` to write to the large object.

- **offset**
  The offset in bytes/characters from the beginning of the large object (origin is 1) for the write operation to begin.

- **buffer**
  Contains data to write to the large object. If `lob_loc` is a BLOB, then `buffer` must be RAW. If `lob_loc` is a CLOB, then `buffer` must be VARCHAR2.

14.3.3.1.6.14  WRITEAPPEND

The WRITEAPPEND procedure adds data to the end of a large object.

```sql
WRITEAPPEND(<lob_loc> IN OUT ( BLOB | CLOB ),
             <amount> BINARY_INTEGER, <buffer> ( RAW | VARCHAR2 ))
```

**Parameters**

- **lob_loc**
  Large object locator of the large object to which you want to append the data.

- **amount**
  Number of bytes/characters from `buffer` to append to the large object.

- **buffer**
  Data to append to the large object. If `lob_loc` is a BLOB, then `buffer` must be RAW. If `lob_loc` is a CLOB, then `buffer` must be VARCHAR2.

14.3.3.1.7  DBMS_LOCK

EDB Postgres Advanced Server provides support for the `DBMS_LOCK.SLEEP` procedure.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLEEP(seconds)</td>
<td>n/a</td>
<td>Suspends a session for the specified number of seconds.</td>
</tr>
</tbody>
</table>
EDB Postgres Advanced Server’s implementation of DBMS_LOCK is a partial implementation when compared to Oracle’s version. Only DBMS_LOCK.SLEEP is supported.

SLEEP

The SLEEP procedure suspends the current session for the specified number of seconds.

SLEEP(<seconds> NUMBER)

Parameters

seconds

seconds specifies the number of seconds for which you want to suspend the session. seconds can be a fractional value. For example, enter 1.75 to specify one and three-fourths of a second.

14.3.3.1.8 DBMS_MVIEW

Use procedures in the DBMS_MVIEW package to manage and refresh materialized views and their dependencies. EDB Postgres Advanced Server provides support for the following DBMS_MVIEW procedures:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_MV_DEPENDENCIES(list VARCHAR2, deplist VARCHAR2);</td>
<td>n/a</td>
<td>The GET_MV_DEPENDENCIES procedure returns a list of dependencies for a specified view.</td>
</tr>
<tr>
<td>REFRESH(list VARCHAR2, method VARCHAR2, rollback_seg VARCHAR2 , push_deferred_rpc BOOLEAN, refresh_after_errors BOOLEAN, purge_option NUMBER, parallelism NUMBER, heap_size NUMBER, atomic_refresh BOOLEAN, nested BOOLEAN);</td>
<td>n/a</td>
<td>This variation of the REFRESH procedure refreshes all views named in a comma-separated list of view names.</td>
</tr>
<tr>
<td>REFRESH(tab dbms_utility.uncl_array, method VARCHAR2, rollback_seg VARCHAR2, push_deferred_rpc BOOLEAN, refresh_after_errors BOOLEAN, purge_option NUMBER, parallelism NUMBER, heap_size NUMBER, atomic_refresh BOOLEAN, nested BOOLEAN);</td>
<td>n/a</td>
<td>This variation of the REFRESH procedure refreshes all views named in a table of dbms_utility.uncl_array values.</td>
</tr>
<tr>
<td>REFRESH_ALL_MVIEWS(number_of_failures BINARY_INTEGER, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN);</td>
<td>n/a</td>
<td>The REFRESH_ALL_MVIEWS procedure refreshes all materialized views.</td>
</tr>
<tr>
<td>REFRESH_DEPENDENT(number_of_failures BINARY_INTEGER, list VARCHAR2, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN, nested BOOLEAN);</td>
<td>n/a</td>
<td>This variation of the REFRESH_DEPENDENT procedure refreshes all views that depend on the views listed in a comma-separated list.</td>
</tr>
<tr>
<td>REFRESH_DEPENDENT(number_of_failures BINARY_INTEGER, tab dbms_utility.uncl_array, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN, nested BOOLEAN);</td>
<td>n/a</td>
<td>This variation of the REFRESH_DEPENDENT procedure refreshes all views that depend on the views listed in a table of dbms_utility.uncl_array values.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server’s implementation of DBMS_MVIEW is a partial implementation when compared to Oracle’s version. Only those functions and procedures listed in the table are supported.

14.3.3.1.8.1 GET_MV_DEPENDENCIES

When given the name of a materialized view, GET_MV_DEPENDENCIES returns a list of items that depend on the specified view. The signature is:

GET_MV_DEPENDENCIES(
    <list> IN VARCHAR2,
    <deplist> OUT VARCHAR2);

Parameters

list

list specifies the name of a materialized view or a comma-separated list of materialized view names.

deplist
**deplist** is a comma-separated list of schema-qualified dependencies. **deplist** is a VARCHAR2 value.

**Examples**

This example displays a list of the dependencies on a materialized view named `public.emp_view`.

```sql
DECLARE
    deplist VARCHAR2(1000);
BEGIN
    DBMS_MVIEW.GET_MV_DEPENDENCIES('public.emp_view', deplist);
    DBMS_OUTPUT.PUT_LINE('deplist: ' || deplist);
END;
```

**14.3.3.1.8.2 REFRESH**

Use the `REFRESH` procedure to refresh all views specified in either a comma-separated list of view names or a table of `DBMS_UTILITY.UNCL_ARRAY` values. The procedure has two signatures. Use the first form when specifying a comma-separated list of view names:

```sql
REFRESH(
    <list> IN VARCHAR2,
    <method> IN VARCHAR2 DEFAULT NULL,
    <rollback_seg> IN VARCHAR2 DEFAULT NULL,
    <push_deferred_rpc> IN BOOLEAN DEFAULT TRUE,
    <refresh_after_errors> IN BOOLEAN DEFAULT FALSE,
    <purge_option> IN NUMBER DEFAULT 1,
    <parallelism> IN NUMBER DEFAULT 0,
    <heap_size> IN NUMBER DEFAULT 0,
    <atomic_refresh> IN BOOLEAN DEFAULT TRUE,
    <nested> IN BOOLEAN DEFAULT FALSE);
```

Use the second form to specify view names in a table of `DBMS_UTILITY.UNCL_ARRAY` values:

```sql
REFRESH(
    <tab> IN OUT
    DBMS_UTILITY.UNCL_ARRAY,
    <method> IN VARCHAR2 DEFAULT NULL,
    <rollback_seg> IN VARCHAR2 DEFAULT NULL,
    <push_deferred_rpc> IN BOOLEAN DEFAULT TRUE,
    <refresh_after_errors> IN BOOLEAN DEFAULT FALSE,
    <purge_option> IN NUMBER DEFAULT 1,
    <parallelism> IN NUMBER DEFAULT 0,
    <heap_size> IN NUMBER DEFAULT 0,
    <atomic_refresh> IN BOOLEAN DEFAULT TRUE,
    <nested> IN BOOLEAN DEFAULT FALSE);
```

**Parameters**

- **list**
  - **list** is a VARCHAR2 value that specifies the name of a materialized view or a comma-separated list of materialized view names. The names can be schema-qualified.

- **tab**
  - **tab** is a table of `DBMS_UTILITY.UNCL_ARRAY` values that specify names of a materialized view.

- **method**
  - **method** is a VARCHAR2 value that specifies the refresh method to apply to the specified views. The only supported method is `C`, which performs a complete refresh of the view.

- **rollback_seg**
  - **rollback_seg** is accepted for compatibility and ignored. The default is `NULL`.

- **push_deferred_rpc**
push_deferred_rpc is accepted for compatibility and ignored. The default is TRUE.

refresh_after_errors

refresh_after_errors is accepted for compatibility and ignored. The default is FALSE.

purge_option

purge_option is accepted for compatibility and ignored. The default is 1.

parallelism

parallelism is accepted for compatibility and ignored. The default is 0.

heap_size IN NUMBER DEFAULT 0,

heap_size is accepted for compatibility and ignored. The default is 0.

atomic_refresh

atomic_refresh is accepted for compatibility and ignored. The default is TRUE.

nested

nested is accepted for compatibility and ignored. The default is FALSE.

Examples

This example uses DBMS_MVIEW.REFRESH to perform a complete refresh on the public.emp_view materialized view:

```sql
EXEC DBMS_MVIEW.REFRESH(list => 'public.emp_view', method => 'C');
```

14.3.3.1.8.3 REFRESH_ALL_MVIEWS

Use the REFRESH_ALL_MVIEWS procedure to refresh any materialized views that weren't refreshed since the table or view on which the view depends was modified. The signature is:

```sql
REFRESH_ALL_MVIEWS(
  <number_of_failures> OUT BINARY_INTEGER,
  <method> IN VARCHAR2 DEFAULT NULL,
  <rollback_seg> IN VARCHAR2 DEFAULT NULL,
  <refresh_after_errors> IN BOOLEAN DEFAULT FALSE,
  <atomic_refresh> IN BOOLEAN DEFAULT TRUE);
```

Parameters

number_of_failures

number_of_failures is a BINARY_INTEGER that specifies the number of failures that occurred during the refresh.

method

method is a VARCHAR2 value that specifies the refresh method to apply to the specified views. The only supported method is C, which performs a complete refresh of the view.

rollback_seg

rollback_seg is accepted for compatibility and ignored. The default is NULL.

refresh_after_errors

refresh_after_errors is accepted for compatibility and ignored. The default is FALSE.
atomic_refresh

atomic_refresh is accepted for compatibility and ignored. The default is TRUE.

Examples

This example performs a complete refresh on all materialized views:

```
DECLARE
   errors INTEGER;
BEGIN
   DBMS_MVIEW.REFRESH_ALL_MVIEWS(errors, method => 'C');
END;
```

Upon completion, errors contains the number of failures.

14.3.3.1.8.4 REFRESH_DEPENDENT

Use the REFRESH_DEPENDENT procedure to refresh all material views that depend on the views specified in the call to the procedure. You can specify a comma-separated list or provide the view names in a table of DBMS_UTILITY.UNCL_ARRAY values.

Use the first form of the procedure to refresh all material views that depend on the views specified in a comma-separated list:

```
REFRESH_DEPENDENT(
   <number_of_failures> OUT BINARY_INTEGER,
   <list> IN VARCHAR2,
   <method> IN VARCHAR2 DEFAULT NULL,
   <rollback_seg> IN VARCHAR2 DEFAULT NULL,
   <refresh_after_errors> IN BOOLEAN DEFAULT FALSE,
   <atomic_refresh> IN BOOLEAN DEFAULT TRUE,
   <nested> IN BOOLEAN DEFAULT FALSE);
```

Use the second form of the procedure to refresh all material views that depend on the views specified in a table of DBMS_UTILITY.UNCL_ARRAY values:

```
REFRESH_DEPENDENT(
   <number_of_failures> OUT BINARY_INTEGER,
   <tab> IN DBMS_UTILITY.UNCL_ARRAY,
   <method> IN VARCHAR2 DEFAULT NULL,
   <rollback_seg> IN VARCHAR2 DEFAULT NULL,
   <refresh_after_errors> IN BOOLEAN DEFAULT FALSE,
   <atomic_refresh> IN BOOLEAN DEFAULT TRUE,
   <nested> IN BOOLEAN DEFAULT FALSE);
```

Parameters

- **number_of_failures**
  - A BINARY_INTEGER that contains the number of failures that occurred during the refresh operation.
- **list**
  - A VARCHAR2 value that specifies the name of a materialized view or a comma-separated list of materialized view names. The names can be schema-qualified.
- **tab**
  - A table of DBMS_UTILITY.UNCL_ARRAY values that specify the names of a materialized view.
- **method**
  - A VARCHAR2 value that specifies the refresh method to apply to the specified views. The only supported method is 'C', which performs a complete refresh of the view.
- **rollback_seg**
  - A VARCHAR2 value that specifies the refresh method to apply to the specified views. The only supported method is 'C', which performs a complete refresh of the view.
rollback_seg is accepted for compatibility and ignored. The default is NULL.

refresh_after_errors

refresh_after_errors is accepted for compatibility and ignored. The default is FALSE.

atomic_refresh

atomic_refresh is accepted for compatibility and ignored. The default is TRUE.

nested

nested is accepted for compatibility and ignored. The default is FALSE.

Examples

This example performs a complete refresh on all materialized views that depend on a materialized view named emp_view that resides in the public schema:

```sql
DECLARE
    errors INTEGER;
BEGIN
    DBMS_MVIEW.REFRESH_DEPENDENT(errors, list => 'public.emp_view', method => 'C');
END;
```

Upon completion, errors contains the number of failures.

14.3.3.19 DBMS_OUTPUT

The DBMS_OUTPUT package sends messages (lines of text) to a message buffer or gets messages from the message buffer. A message buffer is local to a single session. Use the DBMS_PIPE package to send messages between sessions.

The procedures and functions available in the DBMS_OUTPUT package are listed in the following table.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE</td>
<td>n/a</td>
<td>Disable the capability to send and receive messages.</td>
</tr>
<tr>
<td>ENABLE(buffer_size)</td>
<td>n/a</td>
<td>Enable the capability to send and receive messages.</td>
</tr>
<tr>
<td>GET_LINE(line OUT, status OUT)</td>
<td>n/a</td>
<td>Get a line from the message buffer.</td>
</tr>
<tr>
<td>GET_LINES(lines OUT, numlines IN OUT)</td>
<td>n/a</td>
<td>Get multiple lines from the message buffer.</td>
</tr>
<tr>
<td>NEW_LINE</td>
<td>n/a</td>
<td>Puts an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT(item)</td>
<td>n/a</td>
<td>Puts a partial line without an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT_LINE(item)</td>
<td>n/a</td>
<td>Puts a complete line with an end-of-line character sequence.</td>
</tr>
<tr>
<td>SERVEROUTPUT(stdout)</td>
<td>n/a</td>
<td>Direct messages from PUT, PUT_LINE, or NEW_LINE to either standard output or the message buffer.</td>
</tr>
</tbody>
</table>

The following table lists the public variables available in the DBMS_OUTPUT package.

<table>
<thead>
<tr>
<th>Public variables</th>
<th>Data type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chararr</td>
<td>TABLE</td>
<td></td>
<td>For message lines.</td>
</tr>
</tbody>
</table>

CHARARR

The CHARARR is for storing multiple message lines.

```sql
TYPE chararr IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```
**DISABLE**

The `DISABLE` procedure clears out the message buffer. You can no longer access any messages in the buffer at the time the `DISABLE` procedure is executed. Any messages later sent with the `PUT`, `PUT_LINE`, or `NEW_LINE` procedures are discarded. No error is returned to the sender when the `PUT`, `PUT_LINE`, or `NEW_LINE` procedures are executed and messages were disabled.

Use the `ENABLE` or `SERVEROUTPUT(TRUE)` procedure to reenable sending and receiving messages.

**ENABLE**

The `ENABLE` procedure enables you to send messages to or retrieve messages from the message buffer. Running `SERVEROUTPUT(TRUE)` also implicitly performs the `ENABLE` procedure.

The destination of a message sent with `PUT`, `PUT_LINE`, or `NEW_LINE` depends on the state of `SERVEROUTPUT`.

- If the last state of `SERVEROUTPUT` is `TRUE`, the message goes to standard output of the command line.
- If the last state of `SERVEROUTPUT` is `FALSE`, the message goes to the message buffer.

```sql
BEGIN
    DBMS_OUTPUT.DISABLE;
END;
```

**Examples**

This anonymous block disables sending and receiving messages in the current session.

```sql
BEGIN
    DBMS_OUTPUT.DISABLE;
END;
```

This anonymous block enables messages. Setting `SERVEROUTPUT(TRUE)` forces them to standard output.

```sql
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Messages enabled!');
END;
```

You can achieve the same effect by using `SERVEROUTPUT(TRUE)`.

```sql
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Messages enabled');
END;
```

This anonymous block enables messages, but setting `SERVEROUTPUT(FALSE)` directs messages to the message buffer.

```sql
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    DBMS_OUTPUT.PUT_LINE('Message sent to buffer');
END;
```
GET_LINE

The **GET_LINE** procedure retrieves a line of text from the message buffer. Only text that was terminated by an end-of-line character sequence is retrieved. That includes complete lines generated using **PUT_LINE** or a series of **PUT** calls followed by a **NEW_LINE** call.

**GET_LINE**(〈line〉 OUT VARCHAR2, 〈status〉 OUT INTEGER)

**Parameters**

- **line**: Variable receiving the line of text from the message buffer.
- **status**: 0 if a line was returned from the message buffer, 1 if there was no line to return.

**Examples**

This anonymous block writes the **emp** table out to the message buffer as a comma-delimited string for each row.

```sql
EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);
DECLARE
  v_emprec VARCHAR2(128);
  CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
  DBMS_OUTPUT.ENABLE;
  FOR i IN emp_cur LOOP
    v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' || NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate || ',' || i.sal || ',' || NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
  DBMS_OUTPUT.PUT_LINE(v_emprec);
END LOOP;
END;
```

This anonymous block reads the message buffer and inserts the messages written by the prior example into a table named **messages**. The rows in **messages** are then displayed.

```sql
CREATE TABLE messages
(
  status INTEGER,
  msg VARCHAR2(108)
);
DECLARE
  v_line VARCHAR2(108);
  v_status INTEGER := 0;
BEGIN
  DBMS_OUTPUT.GET_LINE(v_line,v_status);
  WHILE v_status = 0 LOOP
    INSERT INTO messages VALUES(v_status, v_line);
    DBMS_OUTPUT.GET_LINE(v_line,v_status);
  END LOOP;
END;
SELECT msg FROM messages;
```
GET_LINES

The `GET_LINES` procedure retrieves one or more lines of text from the message buffer into a collection. Only text that was terminated by an end-of-line character sequence is retrieved. That includes complete lines generated using `PUT_LINE` or by a series of `PUT` calls followed by a `NEW_LINE` call.

**Parameters**

- `lines`  
  Table receiving the lines of text from the message buffer. See `CHARARR` for a description of `lines`.

- `numlines`  
  Number of lines to retrieve from the message buffer.

- `numlines`  
  Actual number of lines retrieved from the message buffer. If the output value of `numlines` is less than the input value, then no more lines are left in the message buffer.

**Examples**

This example uses the `GET_LINES` procedure to store all rows from the `emp` table that were placed in the message buffer into an array.

```sql
EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE
  v_emprec VARCHAR2(128);
  CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
  DBMS_OUTPUT.ENABLE;
  FOR i IN emp_cur LOOP
    v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' || NVL(LTRIM(TO_CHAR(i.mgr, '9999')),'') || ',' || i.hiredate || ',' || i.sal || ',' || NVL(LTRIM(TO_CHAR(i.comm, '9990.99')),'') || ',' || i.deptno;
  DBMS_OUTPUT.PUT_LINE(v_emprec);
  END LOOP;
END;

DECLARE
  v_lines DBMS_OUTPUT.CHARARR;
  v_numlines INTEGER := 14;
  v_status INTEGER := 0;
END;
```

---

**EDB Postgres Advanced Server (EPAS)**

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.

808
BEGIN
DBMS_OUTPUT.GET_LINES(v_lines,v_numlines);
FOR i IN 1..v_numlines
LOOP
    INSERT INTO messages VALUES(v_numlines,
v_lines(i));
END LOOP;
END;
SELECT msg FROM messages;

 thứ tự

The NEW_LINE procedure writes an end-of-line character sequence in the message buffer.

NEW_LINE

Parameter

The NEW_LINE procedure expects no parameters.

NEW_LINE

PUT

The PUT procedure writes a string to the message buffer. No end-of-line character sequence is written at the end of the string. Use the NEW_LINE procedure to add an end-of-line character sequence.

PUT(<item> VARCHAR2)

Parameter

item

Text written to the message buffer.

Examples

The following example uses the PUT procedure to display a comma-delimited list of employees from the emp table.

DECLARE
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    FOR i IN emp_cur
    LOOP
        DBMS_OUTPUT.PUT(1.empno);
    END LOOP;
END;
The **PUT_LINE** procedure writes a single line to the message buffer including an end-of-line character sequence.

**PUT_LINE(item VARCHAR2)**

**Parameter**

**item**

Text to write to the message buffer.

**Examples**

This example uses the **PUT_LINE** procedure to display a comma-delimited list of employees from the **emp** table.

```sql
DECLARE
  v_emprec VARCHAR2(128);
BEGIN
  CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
  FOR i IN emp_cur LOOP
    v_emprec := i.empno || ',' || i.ename || ',' || i.job || ','
    || NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
    ',' || i.sal || ',' ||
    NVL(LTRIM(TO_CHAR(i.comm,'9999.99')),'') || ',' || i.deptno;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(v_emprec);
END;
```
If, in the same session, the following anonymous block is executed, the message stored in the message buffer from the prior example is flushed. It’s displayed on the command line along with the new message.

```sql
BEGIN
  DBMS_OUTPUT.SERVEROUTPUT(TRUE);
  DBMS_OUTPUT.PUT_LINE('This message goes to the command line');
  DBMS_OUTPUT.SERVEROUTPUT(FALSE);
  DBMS_OUTPUT.PUT_LINE('This message goes to the message buffer');
END;
```

This message goes to the command line

If, in the same session, the following anonymous block is executed, the message stored in the message buffer from the prior example is flushed. It’s displayed on the command line along with the new message.

```sql
BEGIN
  DBMS_OUTPUT.SERVEROUTPUT(TRUE);
  DBMS_OUTPUT.PUT_LINE('Flush messages from the buffer');
END;
```

This message goes to the message buffer

Flush messages from the buffer

### ServerOutput

The **SERVEROUTPUT** procedure directs messages to standard output of the command line or to the message buffer. Setting **SERVEROUTPUT(TRUE)** also performs an implicit execution of **ENABLE**.

The default setting of **SERVEROUTPUT** depends on the implementation. For example, in Oracle SQL*Plus, **SERVEROUTPUT(FALSE)** is the default. In PSQL, **SERVEROUTPUT(TRUE)** is the default. Also, in Oracle SQL*Plus, you control this setting using the SQL*Plus **SET** comman, not by a stored procedure as implemented in EDB Postgres Advanced Server.

**SERVEROUTPUT(<stdout> BOOLEAN)**

To get an Oracle-style display output, you can set the `dbms_output.serveroutput` to **FALSE** in the `postgresql.conf` file, which disables the message output. The default is **TRUE**, which enables the message output.

Parameter

- **stdout**

Set to **TRUE** if you want subsequent **PUT**, **PUT_LINE**, or **NEW_LINE** to send text directly to standard output of the command line. Set to **FALSE** to send text to the message buffer.

Examples

This anonymous block sends the first message to the command line and the second message to the message buffer.

```sql
BEGIN
  DBMS_OUTPUT.SERVEROUTPUT(TRUE);
  DBMS_OUTPUT.PUT_LINE('This message goes to the command line');
  DBMS_OUTPUT.SERVEROUTPUT(FALSE);
  DBMS_OUTPUT.PUT_LINE('This message goes to the message buffer');
END;
```

This message goes to the command line

**14.3.3.1.10 DBMS_PIPE**
The `DBMS_PIPE` package lets you send messages through a pipe in or between sessions connected to the same database cluster.

The table shows the procedures and functions available in the `DBMS_PIPE` package:

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CREATE_PIPE(pipename [, maxpipesize ] [, private ] )</code></td>
<td>INTEGER</td>
<td>Explicitly create a private pipe if <code>private</code> is <code>true</code> (the default) or a public pipe if <code>private</code> is <code>false</code>.</td>
</tr>
<tr>
<td><code>NEXT_ITEM_TYPE</code></td>
<td>INTEGER</td>
<td>Determine the data type of the next item in a received message.</td>
</tr>
<tr>
<td><code>PACK_MESSAGE(item)</code></td>
<td>n/a</td>
<td>Place <code>item</code> in the session’s local message buffer.</td>
</tr>
<tr>
<td><code>PURGE(pipename)</code></td>
<td>n/a</td>
<td>Remove unreceived messages from the specified pipe.</td>
</tr>
<tr>
<td><code>RECEIVE_MESSAGE(pipename [, timeout ])</code></td>
<td>INTEGER</td>
<td>Get a message from a specified pipe.</td>
</tr>
<tr>
<td><code>REMOVE_PIPE(pipename)</code></td>
<td>INTEGER</td>
<td>Delete an explicitly created pipe.</td>
</tr>
<tr>
<td><code>RESET_BUFFER</code></td>
<td>n/a</td>
<td>Reset the local message buffer.</td>
</tr>
<tr>
<td><code>SEND_MESSAGE(pipename [, timeout ] [, maxpipesize ] )</code></td>
<td>INTEGER</td>
<td>Send a message on a pipe.</td>
</tr>
<tr>
<td><code>UNIQUE_SESSION_NAME</code></td>
<td>VARCHAR2</td>
<td>Obtain a unique session name.</td>
</tr>
<tr>
<td><code>UNPACK_MESSAGE(item OUT)</code></td>
<td>n/a</td>
<td>Retrieve the next data item from a message into a type-compatible variable, <code>item</code>.</td>
</tr>
</tbody>
</table>

Pipes are categorized as `implicit` or `explicit`. An implicit pipe is created if a reference is made to a pipe name that wasn’t previously created by the `CREATE_PIPE` function. For example, if the `SEND_MESSAGE` function is executed using a nonexistent pipe name, a new implicit pipe is created with that name. An explicit pipe is created using the `CREATE_PIPE` function in which the first parameter specifies the pipe name for the new pipe.

Pipes are also categorized as `private` or `public`. Only the user who created the pipe can access a private pipe. Even a superuser can’t access a private pipe that was created by another user. Any user who has access to the `DBMS_PIPE` package can access a public pipe.

You can create a public pipe only by using the `CREATE_PIPE` function with the third parameter set to `FALSE`. You can use the `CREATE_PIPE` function to create a private pipe by setting the third parameter to `TRUE` or by omitting the third parameter. All implicit pipes are private.

The individual data items, or lines, of a message are first built in a local message buffer, unique to the current session. The `PACK_MESSAGE` procedure builds the message in the session’s local message buffer. You then use the `SEND_MESSAGE` function to send the message through the pipe.

Receipt of a message involves the reverse operation. You use the `RECEIVE_MESSAGE` function to get a message from the specified pipe. The message is written to the session’s local message buffer. You then use the `UNPACK_MESSAGE` procedure to transfer the message data items from the message buffer to program variables. If a pipe contains multiple messages, `RECEIVE_MESSAGE` gets the messages in FIFO (first-in-first-out) order.

Each session maintains separate message buffers for messages created with the `PACK_MESSAGE` procedure and messages retrieved by the `RECEIVE_MESSAGE` function. Thus messages can be both built and received in the same session. However, if consecutive `RECEIVE_MESSAGE` calls are made, only the message from the last `RECEIVE_MESSAGE` call is preserved in the local message buffer.

### 14.3.3.10.1 CREATE_PIPE

The `CREATE_PIPE` function creates an explicit public pipe or an explicit private pipe with a specified name.

```
<status> INTEGER CREATE_PIPE(<pipename> VARCHAR2 [
   [, <maxpipesize> INTEGER ] [, <private> BOOLEAN ]
])
```

**Parameters**

- `pipename`
  - Name of the pipe.

- `maxpipesize`
  - Maximum capacity of the pipe in bytes. Default is 8192 bytes.

- `private`
  - Create a public pipe if set to `FALSE`. Create a private pipe if set to `TRUE`. This is the default.
Status code returned by the operation. 0 indicates successful creation.

Examples

This example creates a private pipe named messages:

```sql
DECLARE
  v_status
  INTEGER;
BEGIN
  v_status := DBMS_PIPE.CREATE_PIPE('messages');
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END;
CREATE_PIPE status: 0
```

This example creates a public pipe named mailbox:

```sql
DECLARE
  v_status
  INTEGER;
BEGIN
  v_status := DBMS_PIPE.CREATE_PIPE('mailbox', 8192, FALSE);  
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END;
CREATE_PIPE status: 0
```

14.3.3.1.10.2 NEXT_ITEM_TYPE

The NEXT_ITEM_TYPE function returns an integer code identifying the data type of the next data item in a message that was retrieved into the session’s local message buffer. As each item is moved off of the local message buffer with the UNPACK_MESSAGE procedure, the NEXT_ITEM_TYPE function returns the data type code for the next available item. A code of 0 is returned when no more items are left in the message.

```sql
<typecode> INTEGER
NEXT_ITEM_TYPE
```

Parameters

typecode

Code identifying the data type of the next data item is shown in the following table.

<table>
<thead>
<tr>
<th>Type code</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No more data items</td>
</tr>
<tr>
<td>9</td>
<td>NUMBER</td>
</tr>
<tr>
<td>11</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>13</td>
<td>DATE</td>
</tr>
<tr>
<td>23</td>
<td>RAW</td>
</tr>
</tbody>
</table>

!! Note The type codes listed in the table aren’t compatible with Oracle databases. Oracle assigns a different numbering sequence to the data types.

Examples

This example shows a pipe packed with a NUMBER item, a VARCHAR2 item, a DATE item, and a RAW item. A second anonymous block then uses the NEXT_ITEM_TYPE function to display the type code of each item.

```sql
DECLARE
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
v_number NUMBER := 123;

v_varchar VARCHAR2(20) := 'Character data';

v_date DATE := SYSDATE;

v_raw RAW(4) := '2122324';

v_status INTEGER;

BEGIN
  DBMS_PIPE.PACK_MESSAGE(v_number);
  DBMS_PIPE.PACK_MESSAGE(v_varchar);
  DBMS_PIPE.PACK_MESSAGE(v_date);
  DBMS_PIPE.PACK_MESSAGE(v_raw);

  v_status := DBMS_PIPE.SEND_MESSAGE('datatypes');
  DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
  END;
END;

SEND_MESSAGE status: 0

DECLARE
  v_number NUMBER;
  v_varchar VARCHAR2(20);
  v_date DATE;
  v_timestamp TIMESTAMP;
  v_raw RAW(4);
  v_status INTEGER;

BEGIN
  v_status := DBMS_PIPE.RECEIVE_MESSAGE('datatypes');
  DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  DBMS_OUTPUT.PUT_LINE('----------------------------------');
  DBMS_OUTPUT.PUT_LINE('NUMBER Item   : ' || v_number);
  DBMS_OUTPUT.PUT_LINE('----------------------------------');

  v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
  DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
  DBMS_PIPE.UNPACK_MESSAGE(v_number);
  DBMS_OUTPUT.PUT_LINE('NUMBER Item   : ' || v_number);
  DBMS_OUTPUT.PUT_LINE('----------------------------------');

  v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
  DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
  DBMS_PIPE.UNPACK_MESSAGE(v_varchar);
  DBMS_OUTPUT.PUT_LINE('VARCHAR2 Item : ' || v_varchar);
  DBMS_OUTPUT.PUT_LINE('----------------------------------');

  v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
  DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
  DBMS_PIPE.UNPACK_MESSAGE(v_date);
  DBMS_OUTPUT.PUT_LINE('DATE Item     : ' || v_date);
  DBMS_OUTPUT.PUT_LINE('----------------------------------');

  v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
  DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
  DBMS_PIPE.UNPACK_MESSAGE(v_raw);
  DBMS_OUTPUT.PUT_LINE('RAW Item      : ' || v_raw);
  DBMS_OUTPUT.PUT_LINE('----------------------------------');
14.3.3.10.3 PACK_MESSAGE

The PACK_MESSAGE procedure places an item of data in the session's local message buffer. You must execute PACK_MESSAGE at least once before issuing a SEND_MESSAGE call.

Parameters

item

An expression evaluating to any of the acceptable parameter data types. The value is added to the session's local message buffer.

14.3.3.10.4 PURGE

The PURGE procedure removes the unreceived messages from a specified implicit pipe.

Parameters

pipename

Name of the pipe.
Examples

Two messages are sent on a pipe:

```sql
DECLARE
  v_status INTEGER;
BEGIN
  DBMS_PIPE.PACK_MESSAGE('Message #1');
  v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
  DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
  DBMS_PIPE.PACK_MESSAGE('Message #2');
  v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
  DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END;
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

Receive the first message and unpack it:

```sql
DECLARE
  v_item VARCHAR2(80);
  v_status INTEGER;
BEGIN
  v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe', 1);
  DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  DBMS_PIPE.UNPACK_MESSAGE(v_item);
  DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END;
RECEIVE_MESSAGE status: 0
Item: Message #1
```

Purge the pipe:

```sql
EXEC DBMS_PIPE.PURGE('pipe');
```

Try to retrieve the next message. The `RECEIVE_MESSAGE` call returns status code 1 indicating it timed out because no message was available.

```sql
DECLARE
  v_item VARCHAR2(80);
  v_status INTEGER;
BEGIN
  v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe', 1);
  DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END;
RECEIVE_MESSAGE status: 1
```

14.3.3.10.5 **RECEIVE_MESSAGE**

The `RECEIVE_MESSAGE` function obtains a message from a specified pipe.

```sql
<status> INTEGER RECEIVE_MESSAGE(<pipename>[
  [, <timeout> INTEGER]])
```
Parameters

- **pipename**
  Name of the pipe.

- **timeout**
  Wait time (seconds). Default is 86400000 (1000 days).

- **status**
  Status code returned by the operation.

The possible status codes are:

<table>
<thead>
<tr>
<th>Status code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>1</td>
<td>Time out</td>
</tr>
<tr>
<td>2</td>
<td>Message too large for the buffer</td>
</tr>
</tbody>
</table>

### 14.3.3.1.10.6 REMOVE_PIPE

The **REMOVE_PIPE** function deletes an explicit private or explicit public pipe.

```sql
<status> INTEGER REMOVE_PIPE(<pipename> VARCHAR2)
```

Use the **REMOVE_PIPE** function to delete explicitly created pipes, that is, pipes created with the **CREATE_PIPE** function.

Parameters

- **pipename**
  Name of the pipe.

- **status**
  Status code returned by the operation. A status code of 0 is returned even if the named pipe is nonexistent.

Examples

Two messages are sent on a pipe:

```sql
DECLARE
  v_status INTEGER;
BEGIN
  v_status := DBMS_PIPE.CREATE_PIPE('pipe');
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
  DBMS_PIPE.PACK_MESSAGE('Message #1');
  v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
  DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
  DBMS_PIPE.PACK_MESSAGE('Message #2');
  v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
  DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Receive the first message and unpack it:

```sql
DECLARE
  v_item VARCHAR2(80);
  v_status INTEGER;
BEGIN
  v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe', 1);
  DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  DBMS_PIPE.UNPACK_MESSAGE(v_item);
  DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END;
```

Remove the pipe:

```sql
SELECT DBMS_PIPE.REMOVE_PIPE('pipe') FROM DUAL;
```

Try to retrieve the next message. The `RECEIVE_MESSAGE` call returns status code 1 indicating it timed out because the pipe was deleted.

```
DECLARE
  v_item VARCHAR2(80);
  v_status INTEGER;
BEGIN
  v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe', 1);
  DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END;
```

**14.3.3.1.10.7  RESET_BUFFER**

The `RESET_BUFFER` procedure resets a pointer to the session's local message buffer back to the beginning of the buffer. This causes later `PACK_MESSAGE` calls to overwrite any data items that were in the message buffer before the `RESET_BUFFER` call.

```
RESET_BUFFER
```

**Examples**

A message to John is written to the local message buffer. It is replaced by a message to Bob by calling `RESET_BUFFER`. The message is sent on the pipe.

```
DECLARE
  v_status INTEGER;
BEGIN
  DBMS_PIPE.PACK_MESSAGE('Hi, John');
  DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?');
  DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?');
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The message to Bob is in the received message:

```
DECLARE
  v_item VARCHAR2(80);
  v_status INTEGER;
BEGIN
  v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe');
  DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  DBMS_PIPE.UNPACK_MESSAGE(v_item);
  DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  DBMS_PIPE.UNPACK_MESSAGE(v_item);
END;
RECEIVE_MESSAGE status: 0
Item: Hi, Bob
Item: Can you attend a meeting at 9:30, tomorrow?
```

### 14.3.3.10.8 SEND_MESSAGE

The `SEND_MESSAGE` function sends a message from the session's local message buffer to the specified pipe.

```
<status> SEND_MESSAGE(<pipename> VARCHAR2 [, <timeout> INTEGER [, <maxpipesize> INTEGER ]])
```

**Parameters**

- **pipename**
  
  Name of the pipe.

- **timeout**
  
  Wait time (seconds). Default is 86400000 (1000 days).

- **maxpipesize**
  
  Maximum capacity of the pipe in bytes. Default is 8192 bytes.

- **status**
  
  Status code returned by the operation.

The possible status codes are:

<table>
<thead>
<tr>
<th>Status code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>1</td>
<td>Time out</td>
</tr>
<tr>
<td>3</td>
<td>Function Interrupted</td>
</tr>
</tbody>
</table>
14.3.3.1.10.9  UNIQUE_SESSION_NAME

The `UNIQUE_SESSION_NAME` function returns a name that is unique to the current session.

```sql
DECLARE
  v_session VARCHAR2(30);
BEGIN
  v_session := DBMS_PIPE.UNIQUE_SESSION_NAME;
  DBMS_OUTPUT.PUT_LINE('Session Name: ' || v_session);
END;
```

**Parameters**

- `name` 
  Unique session name.

**Examples**

The following anonymous block retrieves and displays a unique session name:

14.3.3.1.10.10  UNPACK_MESSAGE

The `UNPACK_MESSAGE` procedure copies the data items of a message from the local message buffer to a specified program variable. You must place the message in the local message buffer with the `RECEIVE_MESSAGE` function before using `UNPACK_MESSAGE`.

```sql
UNPACK_MESSAGE(<item> OUT (DATE | NUMBER | VARCHAR2 | RAW))
```

**Parameters**

- `item` 
  Type-compatible variable that receives a data item from the local message buffer.

14.3.3.1.10.11  Comprehensive example

This example uses a pipe as a "mailbox." Three procedures are enclosed in a package named `mailbox`. These procedures:

1. Create the mailbox.
2. Add a multi-item message to the mailbox (up to three items).
3. Display the full contents of the mailbox.

```sql
CREATE OR REPLACE PACKAGE mailbox
IS
  PROCEDURE create_mailbox;
  PROCEDURE add_message
  ( p_mailbox VARCHAR2,
    p_item_1 VARCHAR2 DEFAULT 'END',
    p_item_2 VARCHAR2 DEFAULT 'END',
    p_item_3 VARCHAR2 DEFAULT 'END' )
END;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
CREATE OR REPLACE PACKAGE BODY mailbox
IS
PROCEDURE create_mailbox
IS
  v_mailbox VARCHAR2(30);
v_status INTEGER;
BEGIN
  v_mailbox := DBMS_PIPE.UNIQUE_SESSION_NAME;
  v_status := DBMS_PIPE.CREATE_PIPE(v_mailbox,1000,FALSE);
  IF v_status = 0 THEN
    DBMS_OUTPUT.PUT_LINE('Created mailbox: ' || v_mailbox);
  ELSE
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE failed - status: ' || v_status);
  END IF;
END create_mailbox;
PROCEDURE add_message
(IS
  p_mailbox VARCHAR2,
p_item_1 VARCHAR2,
  p_item_2 VARCHAR2 DEFAULT 'END',
  p_item_3 VARCHAR2 DEFAULT 'END'
) IS
  v_item_cnt INTEGER := 0;
v_status INTEGER;
BEGIN
  DBMS_PIPE.PACK_MESSAGE(p_item_1);
  v_item_cnt := 1;
  IF p_item_2 != 'END' THEN
    DBMS_PIPE.PACK_MESSAGE(p_item_2);
    v_item_cnt := v_item_cnt + 1;
  END IF;
  IF p_item_3 != 'END' THEN
    DBMS_PIPE.PACK_MESSAGE(p_item_3);
    v_item_cnt := v_item_cnt + 1;
  END IF;
  v_status := DBMS_PIPE.SEND_MESSAGE(p_mailbox);
  IF v_status = 0 THEN
    DBMS_OUTPUT.PUT_LINE('Added message with ' || v_item_cnt || ' item(s) to mailbox ' || p_mailbox);
  ELSE
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE in add_message failed - ' || 'status: ' || v_status);
  END IF;
END add_message;
PROCEDURE empty_mailbox
(IS
  p_mailbox VARCHAR2,
p_waittime INTEGER DEFAULT 10
) IS
v_msgno INTEGER DEFAULT 0;
The following shows executing the procedures in mailbox. The first procedure creates a public pipe using a name generated by the UNIQUE_SESSION_NAME function.

```sql
BEGIN
  DBMS_PIPE.RECEIVE_MESSAGE(p_mailbox,p_waittime);
  WHILE v_status = 0
    LOOP
      v_msgno := v_msgno + 1;
      DBMS_OUTPUT.PUT_LINE('****** Start message #' || v_msgno || ' ******');
      BEGIN
        LOOP
          v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
          EXIT WHEN v_status = 0;
          DBMS_PIPE.UNPACK_MESSAGE(v_item);
          v_itemno := v_itemno + 1;
          DBMS_OUTPUT.PUT_LINE('Item #' || v_itemno || ': ' || v_item);
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('******* End message #' || v_msgno || ' *******');
      END;
    END LOOP;
  END;
  v_status := DBMS_PIPE.REMOVE_PIPE(p_mailbox);
END
END
END

empty_mailbox;
```

Finally, the contents of the mailbox can be emptied:

```sql
EXEC mailbox.empty_mailbox('PG$PIPE$13$3940');
```
14.3.3.11 DBMS_PRIVILEGE_CAPTURE

EDB Postgres Advanced Server provides support for capturing and analyzing the privilege usage by the users.

There are two ways to capture and analyze the privilege usage:

- Using the DBMS_PRIVILEGE_CAPTURE package compatible to Oracle.
- Using SQL commands.

See SQL reference for detailed information about the SQL commands:

- CREATE_CAPTURE_PRIVILEGES_POLICY
- ALTER_CAPTURE_PRIVILEGES_POLICY
- DROP_CAPTURE_PRIVILEGES_POLICY

The DBMS_PRIVILEGE_CAPTURE package provides an interface for database privilege analysis.

The procedures available in the DBMS_PRIVILEGE_CAPTURE package are:

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Function or procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_CAPTURE</td>
<td>Procedure</td>
<td>n/a</td>
<td>Creates a policy that specifies the conditions for analyzing privilege use.</td>
</tr>
<tr>
<td>ENABLE_CAPTURE</td>
<td>Procedure</td>
<td>n/a</td>
<td>Starts capturing the privilege usage for a specific privilege analysis policy.</td>
</tr>
<tr>
<td>DISABLE_CAPTURE</td>
<td>Procedure</td>
<td>n/a</td>
<td>Stops capturing the privilege use for a specific privilege analysis policy.</td>
</tr>
<tr>
<td>DROP_CAPTURE</td>
<td>Procedure</td>
<td>n/a</td>
<td>Removes a privilege analysis policy along with the data captured.</td>
</tr>
<tr>
<td>GENERATE_RESULT</td>
<td>Procedure</td>
<td>n/a</td>
<td>Populates the data dictionary views with the privilege analysis data.</td>
</tr>
<tr>
<td>DELETE_Run</td>
<td>Procedure</td>
<td>n/a</td>
<td>Deletes a privilege analysis capture run.</td>
</tr>
</tbody>
</table>

This package enables you to create a policy that captures the use of system and object privileges granted to the users. It helps you determine that users are using the privileges or not. After analysis, you can revoke the unused privileges to help reducing the number of excess privilege grants for users.

A super user or the non-super user with role CAPTURE_ADMIN can access the DBMS_PRIVILEGE_CAPTURE package.

Configure parameters in postgresql.conf file

To enable the feature, set this parameter value greater than zero in postgresql.conf file:

```
edb_max_capture_privileges_policies = 2
```

This parameter limits the maximum number of policies at cluster level. By default the value is 0. Set this parameter to greater than zero. Also, only one policy can be enabled per database even if `edb_max_capture_privileges_policies` value is set to more than 1. Restart the server after changing the value of this parameter.
edb_capture_privileges_hash_entries = 2000

This parameter sets the hash table size for each privilege policy. Whatever value is set, once that many records of privilege use are captured it gets flushed to DBA_USED_PRIVS and DBA_UNUSED_PRIVS system catalogs. Minimum value is 1 and the default value is 2000. Restart the server after changing the value of this parameter.

CREATE_CAPTURE

The CREATE_CAPTURE procedure creates a privilege analysis policy that specifies the conditions for analyzing privilege use. It also, optionally specifies the roles for which privilege use is to be analyzed, and the conditions under which privilege use is to be analyzed.

```sql
DBMS_PRIVILEGE_CAPTURE.CREATE_CAPTURE
(
    name IN VARCHAR2,
    description IN VARCHAR2 DEFAULT NULL,
    type IN NUMBER DEFAULT G_DATABASE,
    roles IN ROLE_NAME_LIST DEFAULT ROLE_NAME_LIST(),
    condition IN VARCHAR2 DEFAULT NULL);
```

Parameters

name

Name of the privilege analysis policy. A string of size up to 30 characters.

description

Description of the policy up to 1024 characters.

type

Type of the privilege analysis policy. Possible values are:

- G_DATABASE — Captures all privilege use in the database, except privileges used by superuser.
- G_ROLE — Captures the use of a privilege if the privilege is part of a specified role or list of roles.
- G_CONTEXT — Captures the use of a privilege if the context specified by the condition parameter evaluates to true.
- G_ROLE_AND_CONTEXT — Captures the use of a privilege if the privilege is part of the specified list of roles and when the condition specified by the condition parameter is true.

roles

The roles whose privileges are to be analyzed. Required if the type is G_ROLE or G_ROLE_AND_CONTEXT.

condition

Boolean expression containing up to 4000 characters. Required if type is G_CONTEXT or G_ROLE_AND_CONTEXT.

ENABLE_CAPTURE

ENABLE_CAPTURE procedure starts the recording of privilege analysis for a specified privilege analysis policy and optionally provides a capture run for this policy. After a policy is enabled, all privilege use under the policy condition is recorded.

```sql
DBMS_PRIVILEGE_CAPTURE.ENABLE_CAPTURE
(
    name IN VARCHAR2,
    run_name IN VARCHAR2 DEFAULT NULL);
```

Parameters

name

Name of the privilege analysis policy to be enabled.
run_name

Name of the capture run to associate with this policy, less than 128 characters. Enclose exotic characters in double quotation marks.

DISABLE_CAPTURE

DISABLE_CAPTURE procedure stops the recording of privilege use for a specified privilege analysis policy. When a policy is disabled, privilege use meeting the policy condition is no longer needed.

```sql
DBMS_PRIVILEGE_CAPTURE.DISABLE_CAPTURE
(
   name IN VARCHAR2,
   run_name IN VARCHAR2 DEFAULT NULL
);
```

Parameters

- **name**
  Name of the privilege analysis policy to be disabled.

- **run_name**
  Name of the capture run.

DROP_CAPTURE

DROP_CAPTURE procedure removes a privilege analysis policy together with the data recorded. When a policy is removed, all previously recorded privilege use data associated with the policy is deleted.

```sql
DBMS_PRIVILEGE_CAPTURE.DROP_CAPTURE
(
   name IN VARCHAR2
);
```

Parameters

- **name**
  Name of the privilege analysis policy to be removed.

GENERATE_RESULT

GENERATE_RESULT procedure populates the data dictionary views with the privilege analysis data.

```sql
DBMS_PRIVILEGE_CAPTURE.GENERATE_RESULT
(
   name IN VARCHAR2,
   run_name IN VARCHAR2 DEFAULT NULL,
   dependency IN BOOLEAN DEFAULT NULL /* This is just for redwood compatibility */
);
```

Parameters

- **name**
  Name of the privilege analysis policy for which the data dictionary views are populated.

- **run_name**
  Name of the capture run.
Name of the capture run associated with the privilege analysis policy. If you skip this parameter, then the records of all the created runs are analyzed. Specify this parameter to analyze the records of specific run.

dependency

This parameter is added for Oracle compatibility. The value is boolean Y (yes) or N (no).

DELETE_RUN

DELETE_RUN procedure deletes a privilege analysis capture run.

```sql
DBMS_PRIVILEGE_CAPTURE.DELETE_RUN
(
    name  IN VARCHAR2,
    run_name IN VARCHAR2);
```

Parameters

name

Name of the privilege analysis policy with which the capture run is associated.

run_name

Name of the capture run.

Examples

To enable the privilege analysis feature, set the parameter to a value greater than zero in `postgresql.conf` file:

```sql
edb_max_capture_privileges_policies = 2
```

Login as a superuser or grant the CAPTURE_ADMIN role to the non-super user:

```sql
GRANT CAPTURE_ADMIN TO user1;
```

Create a table:

```sql
CREATE TABLE tab1 (a INT);
```

Policy on database

```sql
## Create a privilege analysis policy on the database:
BEGIN

DBMS_PRIVILEGE_CAPTURE.CREATE_CAPTURE
(
    name => 'policy_on_database',
    type => DBMS_PRIVILEGE_CAPTURE.G_DATABASE);

DBMS_PRIVILEGE_CAPTURE.ENABLE_CAPTURE
(
    name => 'policy_on_database');

END;
```

## Insert data into the table:

```sql
INSERT INTO tab1 VALUES(1);
```

## Select data from the table:

```sql
SELECT * FROM tab1;
```

## Disable the policy:

```sql
DBMS_PRIVILEGE_CAPTURE.DISABLE_CAPTURE
(
    name => 'policy_on_database');
```
## Run `GENERATE_RESULT` procedure to populate the dictionary views:

```
BEGIN
DBMS_PRIVILEGE_CAPTURE.GENERATE_RESULT(
  name => 'policy_on_database');
END;
```

Query the data dictionary views to find used privileges:

```
SELECT * FROM DBA_USED_PRIVS ORDER BY object_name COLLATE 'C', privilege_type COLLATE 'C';
```

<table>
<thead>
<tr>
<th>policy_name</th>
<th>run_name</th>
<th>object_class</th>
<th>object_name                                      column_name</th>
<th>application</th>
<th>role_name</th>
<th>privilege_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy_on_database</td>
<td></td>
<td>procedure</td>
<td>dbms_privilege_capture._validate_input_parameter(character varying,character varying)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edb-psql</td>
<td>edb</td>
<td>EXECUTE</td>
<td>dbms_privilege_capture.disable_capture(character varying,character varying)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>policy_on_database</td>
<td></td>
<td>procedure</td>
<td>dbms_privilege_capture.capture(character varying,character varying)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edb-psql</td>
<td>edb</td>
<td>INSERT</td>
<td>dbms_privilege_capture.capture(character varying,character varying)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>policy_on_database</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edb-psql</td>
<td>edb</td>
<td>SELECT</td>
<td>public.tab1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(4 rows)

Query the data dictionary views to find unused privileges:

```
SELECT * FROM DBA_UNUSED_PRIVS ORDER BY object_name COLLATE 'C', privilege_type COLLATE 'C';
```

<table>
<thead>
<tr>
<th>policy_name</th>
<th>run_name</th>
<th>object_class</th>
<th>object_name</th>
<th>column_name</th>
<th>application</th>
<th>role_name</th>
<th>privilege_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy_on_database</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>edb-psql</td>
<td>edb</td>
<td>DELETE</td>
</tr>
<tr>
<td>policy_on_database</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>edb-psql</td>
<td>edb</td>
<td>REFERENCES</td>
</tr>
<tr>
<td>policy_on_database</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>edb-psql</td>
<td>edb</td>
<td>TRIGGER</td>
</tr>
<tr>
<td>policy_on_database</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>edb-psql</td>
<td>edb</td>
<td>TRUNCATE</td>
</tr>
<tr>
<td>policy_on_database</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>edb-psql</td>
<td>edb</td>
<td>UPDATE</td>
</tr>
</tbody>
</table>

(5 rows)

Drop the policy:

```
BEGIN
DBMS_PRIVILEGE_CAPTURE.drop_capture(
  name => 'policy_on_database');
END;
```

Query the data dictionary views to check policy and its related data is removed:

```
SELECT * FROM DBA_PRIV_CAPTURES;
SELECT * FROM DBA_USED_PRIVS;
```

(0 rows)

(0 rows)

### Policy on context

**Create a function:**

```
CREATE OR REPLACE FUNCTION func1(i INTEGER) RETURNS INTEGER AS $$
BEGIN
  RETURN i + 1;
END;
$$ LANGUAGE plpgsql;
```

**Create a policy on context:**
BEGIN
   DBMS_PRIVILEGE_CAPTURE.CREATE_CAPTURE(
      name => 'policy_on_db_context',
      type => DBMS_PRIVILEGE_CAPTURE.G_CONTEXT,
      condition => '1=1');

   DBMS_PRIVILEGE_CAPTURE.ENABLE_CAPTURE(
      name => 'policy_on_db_context');

END;

## Access objects:
SELECT func1();

DELETE FROM tab1;

## Disable policy:
BEGIN
   DBMS_PRIVILEGE_CAPTURE.DISABLE_CAPTURE(
      name => 'policy_on_db_context');

END;

## Run 'GENERATE_RESULT' procedure to populate the dictionary views:
BEGIN
   DBMS_PRIVILEGE_CAPTURE.GENERATE_RESULT(
      name => 'policy_on_db_context');

END;

Query the data dictionary views to find used privileges:

```
SELECT * FROM DBA_USED_PRIVS ORDER BY object_name COLLATE "C", privilege_type COLLATE "C";
```

<table>
<thead>
<tr>
<th>policy_name</th>
<th>run_name</th>
<th>object_class</th>
<th>object_name</th>
<th>column_name</th>
<th>application</th>
<th>role_name</th>
<th>privilege_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy_on_db_context</td>
<td></td>
<td>procedure</td>
<td>dbms_privilege_capture._validate_input_parameter(character varying, character varying)</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edb-psql</td>
<td>edb</td>
<td>EXECUTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>policy_on_db_context</td>
<td></td>
<td>procedure</td>
<td>dbms_privilege_capture.disable_capture(character varying, character varying)</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edb-psql</td>
<td>edb</td>
<td>EXECUTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>policy_on_db_context</td>
<td></td>
<td>function</td>
<td>public.func1(integer)</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edb-psql</td>
<td>edb</td>
<td>EXECUTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>policy_on_db_context</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edb-psql</td>
<td>edb</td>
<td>DELETE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(4 rows)

Query the data dictionary views to find unused privileges:

```
SELECT * FROM DBA_UNUSED_PRIVS ORDER BY object_name COLLATE "C", privilege_type COLLATE "C";
```

<table>
<thead>
<tr>
<th>policy_name</th>
<th>run_name</th>
<th>object_class</th>
<th>object_name</th>
<th>column_name</th>
<th>application</th>
<th>role_name</th>
<th>privilege_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy_on_db_context</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>eedb-psql</td>
<td>edb</td>
<td>INSERT</td>
</tr>
<tr>
<td>policy_on_db_context</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>eedb-psql</td>
<td>edb</td>
<td>REFERENCES</td>
</tr>
<tr>
<td>policy_on_db_context</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>eedb-psql</td>
<td>edb</td>
<td>TRIGGER</td>
</tr>
<tr>
<td>policy_on_db_context</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>eedb-psql</td>
<td>edb</td>
<td>TRUNCATE</td>
</tr>
<tr>
<td>policy_on_db_context</td>
<td></td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>eedb-psql</td>
<td>edb</td>
<td>UPDATE</td>
</tr>
</tbody>
</table>

(6 rows)

Drop the policy:

```
BEGIN
   DBMS_PRIVILEGE_CAPTURE.drop_capture(
      name => 'policy_on_db_context');

END;
```

Query the data dictionary views to check policy and its related data is removed:

```
SELECT * FROM DBA_PRIV_CAPTURES;
```

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>type</th>
<th>enabled</th>
<th>roles</th>
<th>context</th>
<th>run_name</th>
</tr>
</thead>
</table>

(0 rows)
## Policy on role

**Create a role and grant privileges to role:**

```sql
CREATE ROLE ROLE1;
GRANT SELECT, UPDATE, DELETE on tab1 to user1;
```

**Create a privilege analysis policy on role:**

```sql
BEGIN
DBMS_PRIVILEGE_CAPTURE.create_capture(
    name => 'policy_on_role',
    type => DBMS_PRIVILEGE_CAPTURE.G_ROLE,
    roles => ROLE_NAME_LIST('role1'));

DBMS_PRIVILEGE_CAPTURE.enable_capture(
    name => 'policy_on_role',
    run_name => 'run1');
END;
```

**Set role and delete data from the table:**

```sql
SET ROLE role1;
DELETE FROM tab1;
SET ROLE edb;
```

**Disable the policy:**

```sql
BEGIN
    DBMS_PRIVILEGE_CAPTURE.disable_capture(
        name => 'policy_on_role');
END;
```

**Run Generate_result procedure to populate the dictionary views:**

```sql
BEGIN
    DBMS_PRIVILEGE_CAPTURE.generate_result(
        name => 'policy_on_role');
END;
```

---

Query the data dictionary views to find captured policy information:

```sql
SELECT * FROM DBA_PRIV_CAPTURES;
```

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>type</th>
<th>enabled</th>
<th>roles</th>
<th>context</th>
<th>run_name</th>
<th>privilege_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy_on_role</td>
<td></td>
<td>G_ROLE</td>
<td>f</td>
<td>(role1)</td>
<td></td>
<td>run1</td>
<td></td>
</tr>
</tbody>
</table>
(1 row)

Query the data dictionary views to find used privileges:

```sql
SELECT * FROM DBA_USED_PRIVS;
```

<table>
<thead>
<tr>
<th>policy_name</th>
<th>run_name</th>
<th>object_class</th>
<th>object_name</th>
<th>column_name</th>
<th>application</th>
<th>role_name</th>
<th>privilege_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy_on_role</td>
<td>run1</td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>edb-psql</td>
<td>role1</td>
<td>DELETE</td>
</tr>
</tbody>
</table>
(1 row)

Query the data dictionary views to find unused privileges:

```sql
SELECT * FROM DBA_UNUSED_PRIVS ORDER BY object_name COLLATE "C", role_name, privilege_type COLLATE "C";
```

<table>
<thead>
<tr>
<th>policy_name</th>
<th>run_name</th>
<th>object_class</th>
<th>object_name</th>
<th>column_name</th>
<th>application</th>
<th>role_name</th>
<th>privilege_type</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy_on_role</td>
<td>run1</td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>edb-psql</td>
<td>role1</td>
<td>SELECT</td>
</tr>
<tr>
<td>policy_on_role</td>
<td>run1</td>
<td>table</td>
<td>public.tab1</td>
<td>-</td>
<td>edb-psql</td>
<td>role1</td>
<td>UPDATE</td>
</tr>
</tbody>
</table>
### 14.3.3.1.12 DBMS_PROFILER

The **DBMS_PROFILER** package collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a performance profiling session. Use these functions and procedures to control the profiling tool.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLUSH_DATA</td>
<td>Status Code or Exception</td>
<td>Flushes performance data collected in the current session without terminating the session. Profiling continues.</td>
</tr>
<tr>
<td>GET_VERSION(major OUT, minor OUT)</td>
<td>n/a</td>
<td>Returns the version number of this package.</td>
</tr>
<tr>
<td>INTERNAL_VERSION_CHECK</td>
<td>Status Code</td>
<td>Confirms that the current version of the profiler works with the current database.</td>
</tr>
<tr>
<td>PAUSE_PROFILER</td>
<td>Status Code or Exception</td>
<td>Pauses data collection.</td>
</tr>
<tr>
<td>RESUME_PROFILER</td>
<td>Status Code or Exception</td>
<td>Resumes data collection.</td>
</tr>
<tr>
<td>START_PROFILER</td>
<td>Status Code or Exception</td>
<td>Starts data collection.</td>
</tr>
<tr>
<td>STOP_PROFILER</td>
<td>Status Code or Exception</td>
<td>Stops data collection and flushes performance data to the <strong>PLSQL_PROFILER_RAWDATA</strong> table.</td>
</tr>
</tbody>
</table>

The functions in the **DBMS_PROFILER** package return a status code to indicate success or failure. The **DBMS_PROFILER** procedures raise an exception only if they encounter a failure. The table shows the status codes and messages returned by the functions and the exceptions raised by the procedures.

<table>
<thead>
<tr>
<th>Status code</th>
<th>Message</th>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>error version</td>
<td>version_mismatch</td>
<td>The profiler version and the database are incompatible.</td>
</tr>
<tr>
<td>0</td>
<td>success</td>
<td>n/a</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>1</td>
<td>error_param</td>
<td>profiler_error</td>
<td>The operation received an incorrect parameter.</td>
</tr>
<tr>
<td>2</td>
<td>error_io</td>
<td>profiler_error</td>
<td>The data flush operation failed.</td>
</tr>
</tbody>
</table>

### FLUSH_DATA

The **FLUSH_DATA** function/procedure flushes the data collected in the current session without terminating the profiler session. The data is flushed to the tables described in the EDB Postgres Advanced Server Performance Features Guide. The function and procedure signatures are:

```sql
<status> INTEGER FLUSH_DATA
```

#### Parameters

- **status**: Status code returned by the operation.

### GET_VERSION

The **GET_VERSION** procedure returns the version of **DBMS_PROFILER**. The procedure signature is:

```sql
GET_VERSION(<major> OUT INTEGER, <minor> OUT INTEGER)
```
Parameters

**major**

The major version number of `DBMS_PROFILER`.

**minor**

The minor version number of `DBMS_PROFILER`.

**INTERNAL_VERSION_CHECK**

The `INTERNAL_VERSION_CHECK` function confirms that the current version of `DBMS_PROFILER` works with the current database. The function signature is:

```
<status> INTEGER INTERNAL_VERSION_CHECK
```

Parameters

**status**

Status code returned by the operation.

**PAUSE_PROFILER**

The `PAUSE_PROFILER` function/procedure pauses a profiling session. The function and procedure signatures are:

```
<status> INTEGER PAUSE_PROFILER
```

Parameters

**status**

Status code returned by the operation.

**RESUME_PROFILER**

The `RESUME_PROFILER` function/procedure resumes a profiling session. The function and procedure signatures are:

```
<status> INTEGER RESUME_PROFILER
```

Parameters

**status**

Status code returned by the operation.

**START_PROFILER**

The `START_PROFILER` function/procedure starts a data collection session. The function and procedure signatures are:
Parameters

run_comment

A user-defined comment for the profiler session. The default value is SYSDATE.

run_comment1

An additional user-defined comment for the profiler session. The default value is ".

run_number

The session number of the profiler session.

status

Status code returned by the operation.

STOP_PROFILER

The STOP_PROFILER function/procedure stops a profiling session and flushes the performance information to the DBMS_PROFILER tables and view. The function and procedure signatures are:

```sql
INTEGER STOP_PROFILER
```

Parameters

status

Status code returned by the operation.

Using DBMS_PROFILER

The DBMS_PROFILER package collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a profiling session. You can review the performance information in the tables and views provided by the profiler.

DBMS_PROFILER works by recording a set of performance-related counters and timers for each line of PL/pgSQL or SPL statement that executes in a profiling session. The counters and timers are stored in a table named SYS.PLSQL_PROFILER_DATA. When you complete a profiling session, DBMS_PROFILER writes a row to the performance statistics table for each line of PL/pgSQL or SPL code that executed in the session. For example, suppose you execute the following function:

```sql
CREATE OR REPLACE FUNCTION getBalance(acctNumber INTEGER)
RETURNS NUMERIC AS $$
DECLARE
result NUMERIC;
BEGIN
SELECT INTO result balance FROM acct WHERE id = acctNumber;
IF (result IS NULL) THEN
RAISE INFO 'Balance is null';
END IF;
RETURN result;
END;
$$ LANGUAGE 'plpgsql';
```
DBMS_PROFILER adds one PLSQL_PROFILER_DATA entry for each line of code in the `getBalance()` function, including blank lines and comments. The entry corresponding to the SELECT statement executed exactly one time and required a very small amount of time to execute. On the other hand, the entry corresponding to the RAISE INFO statement executed once or not at all, depending on the value for the balance column.

Some of the lines in this function contain no executable code so the performance statistics for those lines always contains zero values.

To start a profiling session, invoke the `DBMS_PROFILER.START_PROFILER` function or procedure. Once you've invoked `START_PROFILER`, EDB Postgres Advanced Server profiles every PL/pgSQL or SPL function, procedure, trigger, or anonymous block that your session executes until you either stop or pause the profiler by calling `STOP_PROFILER` or `PAUSE_PROFILER`.

**Note**

When you start or resume the profiler, the profiler gathers performance statistics only for functions/procedures/triggers that start after the call to `START_PROFILER` (or `RESUME_PROFILER`).

While the profiler is active, EDB Postgres Advanced Server records a large set of timers and counters in memory. When you invoke the `STOP_PROFILER` or `FLUSH_DATA` function/procedure, `DBMS_PROFILER` writes those timers and counters to a set of three tables:

- **SYS.PLSQL_PROFILER_RAWDATA**
  
  Contains the performance counters and timers for each statement executed in the session.

- **SYS.PLSQL_PROFILER_RUNS**
  
  Contains a summary of each run, aggregating the information found in PLSQL_PROFILER_RAWDATA.

- **SYS.PLSQL_PROFILER_UNITS**
  
  Contains a summary of each code unit (function, procedure, trigger, or anonymous block) executed in a session.

In addition, `DBMS_PROFILER` defines a view, `SYS.PLSQL_PROFILER_DATA`, which contains a subset of the PLSQL_PROFILER_RAWDATA table.

A user who is not a superuser can gather profiling information but can't view that profiling information unless a superuser grants specific privileges on the profiling tables stored in the `SYS` schema. This permits a nonprivileged user to gather performance statistics without exposing information that the administrator might want to keep secret.

**Querying the DBMS_PROFILER tables and view**

The following example uses `DBMS_PROFILER` to retrieve performance information for procedures, functions, and triggers included in the sample data distributed with EDB Postgres Advanced Server.

1. Open the EDB-PSQL command line, and establish a connection to the EDB Postgres Advanced Server database. Use an EXEC statement to start the profiling session:

```
acctg=# EXEC dbms_profiler.start_profiler('profile list_emp');
EDB-SPL Procedure successfully completed
```

**Note**

The call to `start_profiler()` includes a comment that `DBMS_PROFILER` associates with the profiler session.

2. Call the `list_emp` function:

```
acctg=# SELECT list_emp();
```

```
INFO: | EMPNO | ENAME |
INFO: | ------ | ------ |
INFO: | 7369   | SMITH |
INFO: | 7499   | ALLEN |
INFO: | 7521   | WARD  |
INFO: | 7566   | JONES |
INFO: | 7654   | MARTIN|
INFO: | 7698   | BLAKE |
INFO: | 7782   | CLARK |
INFO: | 7788   | SCOTT |
INFO: | 7839   | KING  |
INFO: | 7844   | TURNER|
INFO: | 7876   | ADAMS |
INFO: | 7980   | JAMES |
INFO: | 7982   | FORD  |
INFO: | 7934   | MILLER|
```

---

**Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.**
3. Stop the profiling session with a call to `dbms_profiler.stop_profiler`:

```
acctg=# EXEC dbms_profiler.stop_profiler;
EDB-SPL Procedure successfully completed
```

4. Start a new session with the `dbms_profiler.start_profiler` function followed by a new comment:

```
acctg=# EXEC dbms_profiler.start_profiler('profile get_dept_name and emp_sal_trig');
EDB-SPL Procedure successfully completed
```

5. Invoke the `get_dept_name` function:

```
acctg=# SELECT get_dept_name(10);

get_dept_name
---------------
ACCOUNTING
(1 row)
```

6. Execute an `UPDATE` statement that causes a trigger to execute:

```
acctg=# UPDATE memp SET sal = 500 WHERE empno = 7982;
INFO: Updating employee 7982
INFO: ..Old salary: 3800.00
INFO: ..New salary: 580.00
INFO: ..Raise: -250.00
INFO: User enterprisedb updated employee(s) on 04-FEB-14
UPDATE 1
```

7. End the profiling session and flush the performance information to the profiling tables:

```
acctg=# EXEC dbms_profiler.stop_profiler;
EDB-SPL Procedure successfully completed
```

8. Query the `plsql_profiler_runs` table to view a list of the profiling sessions, arranged by `runid`:

```
acctg=# SELECT * FROM plsql_profiler_runs;

<table>
<thead>
<tr>
<th>runid</th>
<th>related_run</th>
<th>run_owner</th>
<th>run_date</th>
<th>run_comment</th>
<th>run_total_time</th>
<th>run_system_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>enterprisedb</td>
<td>04-FEB-14 09:32:48.874315</td>
<td>profile list_emp</td>
<td>4154</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>enterprisedb</td>
<td>04-FEB-14 09:41:38.546503</td>
<td>profile get_dept_name and emp_sal_trig</td>
<td>2088</td>
<td></td>
</tr>
</tbody>
</table>

(2 rows)
```

9. Query the `plsql_profiler_units` table to view the amount of time consumed by each unit (function, procedure, or trigger):

```
acctg=# SELECT * FROM plsql_profiler_units;

<table>
<thead>
<tr>
<th>runid</th>
<th>unit_number</th>
<th>unit_type</th>
<th>unit_owner</th>
<th>unit_name</th>
<th>unit_timestamp</th>
<th>total_time</th>
<th>spare1</th>
<th>spare2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16999</td>
<td>FUNCTION</td>
<td>enterprisedb</td>
<td>list_emp()</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>17802</td>
<td>FUNCTION</td>
<td>enterprisedb</td>
<td>user_audit_trig()</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>17808</td>
<td>FUNCTION</td>
<td>enterprisedb</td>
<td>get_dept_name(p_deptno numeric)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>17804</td>
<td>FUNCTION</td>
<td>enterprisedb</td>
<td>emp_sal_trig()</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(4 rows)
```
10. Query the `plsql_profiler_rawdata` table to view a list of the wait-event counters and wait-event times:

```sql
acctg=# SELECT runid, sourcecode, func_oid, line_number, exec_count, tuples_returned, time_total FROM plsql_profiler_rawdata;
```

```
runid | sourcecode | func_oid | line_number | exec_count | tuples_returned | time_total
-----+------------+----------+-------------+------------+-----------------+------------
1 | DECLARE | 16999 | 1 | 0 | 0 | 0
1 | v_empno | 16999 | 2 | 0 | 0 | 0
1 | v_ename | 16999 | 3 | 0 | 0 | 0
1 | emp_cur | CURSOR FOR 16999 | 4 | 0 | 0 | 0
1 | SELECT | empno, ename | FROM | memp ORDER | BY | empno; 16999 | 5 | 0 | 0 | 0
1 | BEGIN | 16999 | 6 | 0 | 0 | 0
1 | OPEN | emp_cur; 16999 | 7 | 0 | 0 | 0
1 | RAISE INFO | 'EMPNO NAMEN'; 16999 | 8 | 1 | 0 | 0.001621
1 | RAISE INFO | ------ ------'; 16999 | 9 | 1 | 0 | 0.000301
1 | LOOP | 16999 | 10 | 1 | 0 | 4.6e-05
1 | FETCH | emp_cur INTO | v_empno, v_ename; 16999 | 11 | 1 | 0 | 0.001114
1 | EXIT | WHEN NOT FOUND; 16999 | 12 | 15 | 0 | 0.000206
1 | RAISE INFO | %N', v_empno, v_ename; 16999 | 13 | 15 | 0 | 8.3e-05
1 | END | LOOP; 16999 | 14 | 14 | 0 | 0.008773
1 | CLOSE | emp_cur; 16999 | 15 | 0 | 0 | 0
1 | RETURN; 16999 | 16 | 1 | 0 | 1e-05
1 | END; 16999 | 17 | 1 | 0 | 0 | 0
1 | DECLARE | 17002 | 1 | 0 | 0 | 0
1 | v_action | VARCHAR(24); 17002 | 2 | 0 | 0 | 0
1 | v_text | TEXT; 17002 | 3 | 0 | 0 | 0
1 | BEGIN | 17002 | 4 | 0 | 0 | 0
1 | IF | TG_OP = 'INSERT' THEN 17002 | 5 | 0 | 0 | 0
1 | v_action | ::= ' added employee(s) on '; 17002 | 6 | 1 | 0 | 0.008143
1 | ELSEIF | TG_OP = 'UPDATE' THEN 17002 | 7 | 0 | 0 | 0
1 | v_action | ::= ' updated employee(s) on '; 17002 | 8 | 0 | 0 | 0
1 | ELSEIF | TG_OP = 'DELETE' THEN 17002 | 9 | 1 | 0 | 3.2e-05
1 | v_action | ::= ' deleted employee(s) on '; 17002 | 10 | 0 | 0 | 0
1 | END | IF; 17002 | 11 | 0 | 0 | 0
1 | v_text | ::= 'User || USER || v_action || CURRENT_DATE; 17002 | 12 | 0 | 0 | 0
1 | RAISE INFO | '\n', v_text; 17002 | 13 | 1 | 0 | 0.008383
1 | RETURN NULL; 17002 | 14 | 1 | 0 | 6.3e-05
1 | END; 17002 | 15 | 1 | 0 | 3.6e-05
1 | DECLARE | 17002 | 16 | 0 | 0 | 0
1 | DECLARE | 17000 | 1 | 0 | 0 | 0
```
```sql
BEGIN
    SELECT INTO v_dname dname FROM dept WHERE deptno = p_deptno;
    RETURN v_dname;
    IF NOT FOUND THEN
        RAISE INFO 'Invalid department number %', p_deptno;
        RETURN '';
    END IF;
END;

DECLARE
    sal_diff NUMERIC(7,2);
BEGIN
    IF TG_OP = 'INSERT' THEN
        RAISE INFO 'Inserting employee %', NEW.empno;
        RAISE INFO '..New salary: %', NEW.sal;
        RETURN NEW;
    END IF;

    IF TG_OP = 'UPDATE' THEN
        sal_diff := NEW.sal - OLD.sal;
        RAISE INFO 'Updating employee %', OLD.empno;
        RAISE INFO '..Old salary: %', OLD.sal;
        RAISE INFO '..New salary: %', NEW.sal;
        RAISE INFO '..Raise     : %', sal_diff;
        RETURN NEW;
    END IF;

    IF TG_OP = 'DELETE' THEN
        RAISE INFO 'Deleting employee %', OLD.empno;
        RAISE INFO '..Old salary: %', OLD.sal;
        RETURN OLD;
    END IF;
END;
```

11. Query the `plsql_profiler_data` view to review a subset of the information found in `plsql_profiler_rawdata` table:

```
cacct# SELECT * FROM plsql_profiler_data;
```

```sql
| runid | unit_number | line# | total_occur | total_time | min_time | max_time |
|-------+-------------+-------+-------------+------------+----------+---------|
| 1     | spare1      |       |             |            |          |         |
|       | spare2      |       |             |            |          |         |
|       | spare3      |       |             |            |          |         |
|       | spare4      |       |             |            |          |         |
```

(68 rows)
The EDB Postgres Advanced Server installer creates the following tables and views that you can query to review PL/SQL performance profile information:

<table>
<thead>
<tr>
<th>Table name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLSQL_PROFILER_RUNS</td>
<td>Table containing information about all profiler runs, organized by runid.</td>
</tr>
<tr>
<td>PLSQL_PROFILER_UNITS</td>
<td>Table containing information about all profiler runs, organized by unit.</td>
</tr>
<tr>
<td>PLSQL_PROFILER_DATA</td>
<td>View containing performance statistics.</td>
</tr>
<tr>
<td>PLSQL_PROFILER_RAWDATA</td>
<td>Table containing the performance statistics and the extended performance statistics for DRITA counters and timers.</td>
</tr>
</tbody>
</table>

The **PLSQL_PROFILER_RUNS** table contains the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>runid</td>
<td>INTEGER (NOT NULL)</td>
<td>Unique identifier (plsql_profiler_runnumber)</td>
</tr>
<tr>
<td>related_run</td>
<td>INTEGER</td>
<td>The runid of a related run</td>
</tr>
<tr>
<td>run_owner</td>
<td>TEXT</td>
<td>The role that recorded the profiling session</td>
</tr>
<tr>
<td>run_date</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>The profiling session start time</td>
</tr>
<tr>
<td>run_comment</td>
<td>TEXT</td>
<td>User comments relevant to this run</td>
</tr>
<tr>
<td>run_total_time</td>
<td>BIGINT</td>
<td>Run time (in microseconds)</td>
</tr>
<tr>
<td>run_system_info</td>
<td>TEXT</td>
<td>Currently unused</td>
</tr>
<tr>
<td>run_comment1</td>
<td>TEXT</td>
<td>Additional user comments</td>
</tr>
<tr>
<td>spare1</td>
<td>TEXT</td>
<td>Currently unused</td>
</tr>
</tbody>
</table>

### PLSQL_PROFILER_UNITS

The **PLSQL_PROFILER_UNITS** table contains the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runid</td>
<td>INTEGER</td>
<td>Unique identifier (plsql_profiler_runnumber)</td>
</tr>
<tr>
<td>unit_number</td>
<td>OID</td>
<td>Corresponds to the OID of the row in the pg_proc table that identifies the unit</td>
</tr>
<tr>
<td>unit_type</td>
<td>TEXT</td>
<td>PL/SQL function, procedure, trigger or anonymous block</td>
</tr>
<tr>
<td>unit_owner</td>
<td>TEXT</td>
<td>The identity of the role that owns the unit</td>
</tr>
<tr>
<td>unit_name</td>
<td>TEXT</td>
<td>The complete signature of the unit</td>
</tr>
<tr>
<td>unit_timestamp</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>Creation date of the unit (currently NULL)</td>
</tr>
<tr>
<td>total_time</td>
<td>BIGINT</td>
<td>Time spent within the unit (in milliseconds)</td>
</tr>
<tr>
<td>spare1</td>
<td>BIGINT</td>
<td>Currently unused</td>
</tr>
<tr>
<td>spare2</td>
<td>BIGINT</td>
<td>Currently unused</td>
</tr>
</tbody>
</table>

### PLSQL_PROFILER_DATA

The **PLSQL_PROFILER_DATA** view contains the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runid</td>
<td>INTEGER</td>
<td>The run identifier (plsql_profiler_runnumber)</td>
</tr>
<tr>
<td>unit_number</td>
<td>OID</td>
<td>Object ID of the unit that contains the current line</td>
</tr>
<tr>
<td>line#</td>
<td>INTEGER</td>
<td>Current line number of the profiled workload</td>
</tr>
<tr>
<td>total_occur</td>
<td>BIGINT</td>
<td>The number of times that the line was executed</td>
</tr>
<tr>
<td>total_time</td>
<td>DOUBLE PRECISION</td>
<td>The amount of time spent executing the line (in seconds)</td>
</tr>
<tr>
<td>min_time</td>
<td>DOUBLE PRECISION</td>
<td>The minimum execution time for the line</td>
</tr>
<tr>
<td>max_time</td>
<td>DOUBLE PRECISION</td>
<td>The maximum execution time for the line</td>
</tr>
<tr>
<td>spare1</td>
<td>NUMBER</td>
<td>Currently unused</td>
</tr>
<tr>
<td>spare2</td>
<td>NUMBER</td>
<td>Currently unused</td>
</tr>
<tr>
<td>spare3</td>
<td>NUMBER</td>
<td>Currently unused</td>
</tr>
<tr>
<td>spare4</td>
<td>NUMBER</td>
<td>Currently unused</td>
</tr>
</tbody>
</table>

### PLSQL_PROFILER_RAWDATA

The **PLSQL_PROFILER_RAWDATA** table contains the statistical and wait-events information found in the **PLSQL_PROFILER_DATA** view, as well as the performance statistics returned by the DRITA counters and timers.

<table>
<thead>
<tr>
<th>Column</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runid</td>
<td>INTEGER</td>
<td>The run identifier (plsql_profiler_runnumber)</td>
</tr>
<tr>
<td>sourcecode</td>
<td>TEXT</td>
<td>The individual line of profiled code.</td>
</tr>
<tr>
<td>func_oid</td>
<td>OID</td>
<td>Object ID of the unit that contains the current line</td>
</tr>
<tr>
<td>line_number</td>
<td>INTEGER</td>
<td>Current line number of the profiled workload</td>
</tr>
</tbody>
</table>

---

EDB Postgres Advanced Server (EPAS)

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
<table>
<thead>
<tr>
<th>Column</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exec_count</td>
<td>BIGINT</td>
<td>The number of times that the line was executed.</td>
</tr>
<tr>
<td>tuples_returned</td>
<td>BIGINT</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>time_total</td>
<td>DOUBLE</td>
<td>The amount of time spent executing the line, in seconds.</td>
</tr>
<tr>
<td>time_shortest</td>
<td>DOUBLE</td>
<td>The minimum execution time for the line.</td>
</tr>
<tr>
<td>time_longest</td>
<td>DOUBLE</td>
<td>The maximum execution time for the line.</td>
</tr>
<tr>
<td>num_scans</td>
<td>BIGINT</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>tuples_fetched</td>
<td>BIGINT</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>tuples_inserted</td>
<td>BIGINT</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>tuples_updated</td>
<td>BIGINT</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>tuples_deleted</td>
<td>BIGINT</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>blocks_fetched</td>
<td>BIGINT</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>blocks_hit</td>
<td>BIGINT</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>wal_write</td>
<td>BIGINT</td>
<td>A server has waited for a write to the write-ahead log buffer. Expect this value to be high.</td>
</tr>
<tr>
<td>wal_flush</td>
<td>BIGINT</td>
<td>A server has waited for the write-ahead log to flush to disk.</td>
</tr>
<tr>
<td>wal_file_sync</td>
<td>BIGINT</td>
<td>A server has waited for the write-ahead log to sync to disk (related to the wal sync method parameter which, by default, is 'sync' - better performance can be gained by changing this parameter to open sync).</td>
</tr>
<tr>
<td>db_file_read</td>
<td>BIGINT</td>
<td>A server has waited for the completion of a read (from disk).</td>
</tr>
<tr>
<td>db_file_write</td>
<td>BIGINT</td>
<td>A server has waited for the completion of a write (to disk).</td>
</tr>
<tr>
<td>db_file_sync</td>
<td>BIGINT</td>
<td>A server has waited for the operating system to flush all changes to disk.</td>
</tr>
<tr>
<td>db_file_extend</td>
<td>BIGINT</td>
<td>A server has waited for the operating system while adding a new page to the end of a file.</td>
</tr>
<tr>
<td>sql_parse</td>
<td>BIGINT</td>
<td>Currently unused.</td>
</tr>
<tr>
<td>query_plan</td>
<td>BIGINT</td>
<td>A server has generated a query plan.</td>
</tr>
<tr>
<td>other_lwlock_acquire</td>
<td>BIGINT</td>
<td>A server has waited for other light-weight lock to protect data.</td>
</tr>
<tr>
<td>shared_plan_cache_collision</td>
<td>BIGINT</td>
<td>A server has waited for the completion of the shared_plan_cache_collision event.</td>
</tr>
<tr>
<td>shared_plan_cache_insert</td>
<td>BIGINT</td>
<td>A server has waited for the completion of the shared_plan_cache_insert event.</td>
</tr>
<tr>
<td>shared_plan_cache_hit</td>
<td>BIGINT</td>
<td>A server has waited for the completion of the shared_plan_cache_hit event.</td>
</tr>
<tr>
<td>shared_plan_cache_miss</td>
<td>BIGINT</td>
<td>A server has waited for the completion of the shared_plan_cache_miss event.</td>
</tr>
<tr>
<td>shared_plan_cache_lock</td>
<td>BIGINT</td>
<td>A server has waited for the completion of the shared_plan_cache_lock event.</td>
</tr>
<tr>
<td>shared_plan_cache_busy</td>
<td>BIGINT</td>
<td>A server has waited for the completion of the shared_plan_cache_busy event.</td>
</tr>
<tr>
<td>shmemindexlock</td>
<td>BIGINT</td>
<td>A server has waited to find or allocate space in the shared memory.</td>
</tr>
<tr>
<td>oidgenlock</td>
<td>BIGINT</td>
<td>A server has waited to allocate or assign an OID.</td>
</tr>
<tr>
<td>xidgenlock</td>
<td>BIGINT</td>
<td>A server has waited to allocate or assign a transaction ID.</td>
</tr>
<tr>
<td>procarraylock</td>
<td>BIGINT</td>
<td>A server has waited to get a snapshot or clearing a transaction ID at transaction end.</td>
</tr>
<tr>
<td>sinvalreadlock</td>
<td>BIGINT</td>
<td>A server has waited to retrieve or remove messages from shared invalidation queue.</td>
</tr>
<tr>
<td>sinvalwritelock</td>
<td>BIGINT</td>
<td>A server has waited to add a message to the shared invalidation queue.</td>
</tr>
<tr>
<td>walbufmappinglock</td>
<td>BIGINT</td>
<td>A server has waited to replace a page in WAL buffers.</td>
</tr>
<tr>
<td>walwrite_lock</td>
<td>BIGINT</td>
<td>A server has waited for WAL buffers to be written to disk.</td>
</tr>
<tr>
<td>controlfilelock</td>
<td>BIGINT</td>
<td>A server has waited to read or update the control file or creation of a new WAL file.</td>
</tr>
<tr>
<td>checkcontrollock</td>
<td>BIGINT</td>
<td>A server has waited to perform a checkpoint.</td>
</tr>
<tr>
<td>subtranscontrollock</td>
<td>BIGINT</td>
<td>A server has waited to read or update the sub-transaction information.</td>
</tr>
<tr>
<td>multixactgenlock</td>
<td>BIGINT</td>
<td>A server has waited to read or update the shared multixact state.</td>
</tr>
<tr>
<td>multixactoffsetcontrlock</td>
<td>BIGINT</td>
<td>A server has waited to read or update multixact offset mappings.</td>
</tr>
<tr>
<td>multixactmembercontrlock</td>
<td>BIGINT</td>
<td>A server has waited to read or update multixact member mappings.</td>
</tr>
<tr>
<td>relcacheinitlock</td>
<td>BIGINT</td>
<td>A server has waited to read or write the relation cache initialization file.</td>
</tr>
<tr>
<td>Column</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>checkpointercomlock</td>
<td>BIGINT</td>
<td>A server has waited to manage the fsync requests.</td>
</tr>
<tr>
<td>twophasestatelock</td>
<td>BIGINT</td>
<td>A server has waited to read or update the state of prepared transactions.</td>
</tr>
<tr>
<td>tablespacecreatelock</td>
<td>BIGINT</td>
<td>A server has waited to create or drop the tablespace.</td>
</tr>
<tr>
<td>btreevacuumlock</td>
<td>BIGINT</td>
<td>A server has waited to read or update the vacuum related information for a B-tree index.</td>
</tr>
<tr>
<td>addinsmeminitlock</td>
<td>BIGINT</td>
<td>A server has waited to manage space allocation in shared memory.</td>
</tr>
<tr>
<td>autovacuumlock</td>
<td>BIGINT</td>
<td>The autovacuum launcher waiting to read or update the current state of autovacuum workers.</td>
</tr>
<tr>
<td>autovacuumschedulelock</td>
<td>BIGINT</td>
<td>A server has waited to ensure that the table selected for a vacuum still needs vacuuming.</td>
</tr>
<tr>
<td>syncscanlock</td>
<td>BIGINT</td>
<td>A server has waited to get the start location of a scan on a table for synchronized scans.</td>
</tr>
<tr>
<td>relationmappinglock</td>
<td>BIGINT</td>
<td>A server has waited to update the relation map file used to store catalog to file node mapping.</td>
</tr>
<tr>
<td>asyncctllock</td>
<td>BIGINT</td>
<td>A server has waited to read or update shared notification state.</td>
</tr>
<tr>
<td>asynctimequeuelock</td>
<td>BIGINT</td>
<td>A server has waited to read or update the notification messages.</td>
</tr>
<tr>
<td>serializableaexitashocklock</td>
<td>BIGINT</td>
<td>A server has waited to retrieve or store information about serializable transactions.</td>
</tr>
<tr>
<td>serializablefinishedlistlock</td>
<td>BIGINT</td>
<td>A server has waited to access the list of finished serializable transactions.</td>
</tr>
<tr>
<td>serializable predicatlock</td>
<td>BIGINT</td>
<td>A server has waited to perform an operation on a list of locks held by serializable transactions.</td>
</tr>
<tr>
<td>oldserxidlock</td>
<td>BIGINT</td>
<td>A server has waited to read or record the conflicting serializable transactions.</td>
</tr>
<tr>
<td>syncreloak</td>
<td>BIGINT</td>
<td>A server has waited to read or update information about synchronous replicas.</td>
</tr>
<tr>
<td>backgroundworkerlock</td>
<td>BIGINT</td>
<td>A server has waited to read or update the background worker state.</td>
</tr>
<tr>
<td>dynamicsharedmemorylock</td>
<td>BIGINT</td>
<td>A server has waited to read or update the dynamic shared memory state.</td>
</tr>
<tr>
<td>autofilelock</td>
<td>BIGINT</td>
<td>A server has waited to update the <code>postgresql.auto.conf</code> file.</td>
</tr>
<tr>
<td>replicationslotallocationlock</td>
<td>BIGINT</td>
<td>A server has waited to allocate or free a replication slot.</td>
</tr>
<tr>
<td>replicationslotcontrollock</td>
<td>BIGINT</td>
<td>A server has waited to read or update replication slot state.</td>
</tr>
<tr>
<td>commitscontrollock</td>
<td>BIGINT</td>
<td>A server has waited to read or update transaction commit timestamps.</td>
</tr>
<tr>
<td>commitslock</td>
<td>BIGINT</td>
<td>A server has waited to read or update the last value set for the transaction timestamp.</td>
</tr>
<tr>
<td>replicationoriginlock</td>
<td>BIGINT</td>
<td>A server has waited to set up, drop, or use replication origin.</td>
</tr>
<tr>
<td>multixacttruncationlock</td>
<td>BIGINT</td>
<td>A server has waited to read or truncate multixact information.</td>
</tr>
<tr>
<td>oldsnapshottimemaplock</td>
<td>BIGINT</td>
<td>A server has waited to read or update old snapshot control information.</td>
</tr>
<tr>
<td>backendrandomlock</td>
<td>BIGINT</td>
<td>A server has waited to generate a random number.</td>
</tr>
<tr>
<td>logicalreplicatorlock</td>
<td>BIGINT</td>
<td>A server has waited for the action on logical replication worker to finish.</td>
</tr>
<tr>
<td>clogtruncationlock</td>
<td>BIGINT</td>
<td>A server has waited to truncate the write-ahead log or waiting for write-ahead log truncation to finish.</td>
</tr>
<tr>
<td>bulkloadlock</td>
<td>BIGINT</td>
<td>A server has waited for the bulkloadlock to bulk load the data.</td>
</tr>
<tr>
<td>edbresourcemanagerlock</td>
<td>BIGINT</td>
<td>The edbresourcemanagerlock provides detail about edb resource manager lock module.</td>
</tr>
<tr>
<td>wal_write_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a wal_write wait event to write to the write-ahead log buffer (expect this value to be high).</td>
</tr>
<tr>
<td>wal_flush_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a wal_flush wait event to write-ahead log to flush to disk.</td>
</tr>
<tr>
<td>wal_file_sync_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a wal_file_sync wait event to write-ahead log to sync to disk (related to the wal_sync_method parameter which, by default, is 'Sync' - better performance can be gained by changing this parameter to open_sync).</td>
</tr>
<tr>
<td>db_file_read_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the db_file_read wait event for completion of a read (from disk).</td>
</tr>
<tr>
<td>db_file_write_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the db_file_write wait event for completion of a write (to disk).</td>
</tr>
<tr>
<td>db_file_sync_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the db_file_sync wait event to sync all changes to disk.</td>
</tr>
<tr>
<td>db_fileextend_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the db_fileextend wait event while adding a new page to the end of a file.</td>
</tr>
<tr>
<td>sql_parse_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the sql_parse wait event to parse a SQL statement.</td>
</tr>
<tr>
<td>query_plan_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the query_plan wait event to compute the execution plan for a SQL statement.</td>
</tr>
<tr>
<td>other_lwlockacquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the other_lwlock_acquire wait event to protect data.</td>
</tr>
<tr>
<td>sharedplan_cache_collision_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the sharedplan_cache_collision wait event.</td>
</tr>
<tr>
<td>sharedplan_cache_insert_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the sharedplan_cache_insert wait event.</td>
</tr>
<tr>
<td>sharedplan_cache_hit_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the sharedplan_cache_hit wait event.</td>
</tr>
<tr>
<td>Column</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>shared_plan_cache_miss_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>shared_plan_cache_miss</code> wait event.</td>
</tr>
<tr>
<td>shared_plan_cache_lock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>shared_plan_cache_lock</code> wait event.</td>
</tr>
<tr>
<td>shared_plan_cache_busy_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>shared_plan_cache_busy</code> wait event.</td>
</tr>
<tr>
<td>shmemindexlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>shmemindexlock</code> wait event to find or allocate space in the shared memory.</td>
</tr>
<tr>
<td>oidgenlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>oidgenlock</code> wait event to allocate or assign an OID.</td>
</tr>
<tr>
<td>xidgenlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for <code>xidgenlock</code> wait event to allocate or assign a transaction ID.</td>
</tr>
<tr>
<td>proccarraylock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a <code>proccarraylock</code> wait event to clear a transaction ID at transaction end.</td>
</tr>
<tr>
<td>sinalreadlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a <code>sinalreadlock</code> wait event to retrieve or remove messages from shared invalidation queue.</td>
</tr>
<tr>
<td>sinalritelock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a <code>sinalritelock</code> wait event to add a message to the shared invalidation queue.</td>
</tr>
<tr>
<td>walbufmappinglock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a <code>walbufmappinglock</code> wait event to replace a page in WAL buffers.</td>
</tr>
<tr>
<td>walwriteock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a <code>walwritelock</code> wait event to write the WAL buffers to disk.</td>
</tr>
<tr>
<td>controlfilelock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a <code>controlfilelock</code> wait event to read or update the control file or to create a new WAL file.</td>
</tr>
<tr>
<td>checkpointlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a <code>checkpointlock</code> wait event to perform a checkpoint.</td>
</tr>
<tr>
<td>clogcontrollock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a <code>clogcontrollock</code> wait event to read or update the transaction status.</td>
</tr>
<tr>
<td>subtranscontrollock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>subtranscontrollock</code> wait event to read or update the sub-transaction information.</td>
</tr>
<tr>
<td>multixactgenlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>multixactgenlock</code> wait event to read or update the shared multixact state.</td>
</tr>
<tr>
<td>multixactoffsetcontrollock_</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>multixactoffsetcontrollock</code> wait event to read or update multixact offset mappings.</td>
</tr>
<tr>
<td>multixactmembercontrollock_</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>multixactmembercontrollock</code> wait event to read or update multixact member mappings.</td>
</tr>
<tr>
<td>relcacheinitlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>relcacheinitlock</code> wait event to read or write the relation cache initialization file.</td>
</tr>
<tr>
<td>twophasestatelow_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>twophasestatelow</code> wait event to manage the fsync requests.</td>
</tr>
<tr>
<td>tablespacecreatelock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>tablespacecreatelock</code> wait event to create or drop the tablespace.</td>
</tr>
<tr>
<td>btreevacuumlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>btreevacuumlock</code> wait event to read or update the vacuum related information for a B-tree index.</td>
</tr>
<tr>
<td>addinsmexinitlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>addinsmexinitlock</code> wait event to manage space allocation in shared memory.</td>
</tr>
<tr>
<td>autovaccumlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>autovaccumlock</code> wait event to read or update the current state of autovacuum workers.</td>
</tr>
<tr>
<td>autocavacumschedulelock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>autocavacumschedulelock</code> wait event to ensure that the table selected for a vacuum still needs vacuuming.</td>
</tr>
<tr>
<td>syncscanlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>syncscanlock</code> wait event to get the start location of a scan on a table for synchronized scans.</td>
</tr>
<tr>
<td>relationmappinglock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>relationmappinglock</code> wait event to update the relation map file used to store catalog to file node mapping.</td>
</tr>
<tr>
<td>asyncctllock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>asyncctllock</code> wait event to read or update shared notification state.</td>
</tr>
<tr>
<td>asyncqueueunlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>asyncqueueunlock</code> wait event to read or update the notification messages.</td>
</tr>
<tr>
<td>serializableeaxthashlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>serializableeaxthashlock</code> wait event to retrieve or store information about serializable transactions.</td>
</tr>
<tr>
<td>serializablefinishedlistlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>serializablefinishedlistlock</code> wait event to access the list of finished serializable transactions.</td>
</tr>
<tr>
<td>serializablepredicatelocklistlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>serializablepredicatelocklistlock</code> wait event to perform an operation on a list of locks held by serializable transactions.</td>
</tr>
<tr>
<td>oldserxidlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>oldserxidlock</code> wait event to read or record the conflicting serializable transactions.</td>
</tr>
<tr>
<td>syncreplock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>syncreplock</code> wait event to read or update information about synchronous replicas.</td>
</tr>
<tr>
<td>backgroundworkerlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>backgroundworkerlock</code> wait event to read or update the background worker state.</td>
</tr>
<tr>
<td>dynamicsharedmemorycontrollock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>dynamicsharedmemorycontrollock</code> wait event to read or update the dynamic shared memory state.</td>
</tr>
<tr>
<td>autofilelock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the <code>autofilelock</code> wait event to update the <code>postgresql.auto.conf</code> file.</td>
</tr>
<tr>
<td>Column</td>
<td>Data type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>replicationslotallocationlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{replicationslotallocationlock} wait event to allocate or free a replication slot.</td>
</tr>
<tr>
<td>replicationslotcontrollock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{replicationslotcontrollock} wait event to read or update replication slot state.</td>
</tr>
<tr>
<td>committscontrollock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{committscontrollock} wait event to read or update transaction commit timestamps.</td>
</tr>
<tr>
<td>committslock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{committslock} wait event to read or update the last value set for the transaction timestamp.</td>
</tr>
<tr>
<td>replicationoriginlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{replicationoriginlock} wait event to set up, drop, or use replication origin.</td>
</tr>
<tr>
<td>multixacttruncationlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{multixacttruncationlock} wait event to read or truncate multixact information.</td>
</tr>
<tr>
<td>oldsnapshottimenaplock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{oldsnapshottimenaplock} wait event to read or update old snapshot control information.</td>
</tr>
<tr>
<td>backendrandomlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{backendrandomlock} wait event to generate a random number.</td>
</tr>
<tr>
<td>logicalrepworkerlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{logicalrepworkerlock} wait event for an action on logical replication worker to finish.</td>
</tr>
<tr>
<td>clogtruncationlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{clogtruncationlock} wait event to truncate the write-ahead log or waiting for write-ahead log truncation to finish.</td>
</tr>
<tr>
<td>bulkloadlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{bulkloadlock} wait event to bulk upload the data.</td>
</tr>
<tr>
<td>edbresourcemanagerlock_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the \texttt{edbresourcemanagerlock} wait event.</td>
</tr>
<tr>
<td>totalwaits</td>
<td>BIGINT</td>
<td>The total number of event waits.</td>
</tr>
<tr>
<td>totalwaittime</td>
<td>BIGINT</td>
<td>The total time spent waiting for an event.</td>
</tr>
</tbody>
</table>

\subsection*{14.3.3.13 DBMS\_RANDOM}

The \texttt{DBMS\_RANDOM} package provides methods to generate random values. The procedures and functions available in the \texttt{DBMS\_RANDOM} package are listed in the following table.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZE ((val))</td>
<td>n/a</td>
<td>Initializes the \texttt{DBMS_RANDOM} package with the specified seed \texttt{val}. Deprecated, but supported for backward compatibility.</td>
</tr>
<tr>
<td>\texttt{NORMAL()}</td>
<td>NUMBER</td>
<td>Returns a random \texttt{NUMBER}.</td>
</tr>
<tr>
<td>\texttt{RANDOM}</td>
<td>INTEGER</td>
<td>Returns a random \texttt{INTEGER} with a value greater than or equal to (-2^{31}) and less than (2^{31}). Deprecated, but supported for backward compatibility.</td>
</tr>
<tr>
<td>\texttt{SEED} ((val))</td>
<td>n/a</td>
<td>Resets the seed with the specified \texttt{val}.</td>
</tr>
<tr>
<td>\texttt{STRING} ((opt, len))</td>
<td>VARCHAR2</td>
<td>Returns a random string.</td>
</tr>
<tr>
<td>\texttt{TERMINATE}</td>
<td>n/a</td>
<td>\texttt{TERMINATE} has no effect. Deprecated, but supported for backward compatibility.</td>
</tr>
<tr>
<td>\texttt{VALUE}</td>
<td>NUMBER</td>
<td>Returns a random number with a value greater than or equal to (8) and less than (1), with 38-digit precision.</td>
</tr>
<tr>
<td>\texttt{VALUE} ((low, high))</td>
<td>NUMBER</td>
<td>Returns a random number with a value greater than or equal to \texttt{low} and less than \texttt{high}.</td>
</tr>
</tbody>
</table>

\subsection*{INITIALIZE}

The \texttt{INITIALIZE} procedure initializes the \texttt{DBMS\_RANDOM} package with a seed value. The signature is:

\begin{verbatim}
INITIALIZE(<val> IN INTEGER)
\end{verbatim}

This procedure is deprecated. It is included for backward compatibility.

\textbf{Parameters}

\texttt{val}

\texttt{val} is the seed value used by the \texttt{DBMS\_RANDOM} package algorithm.

\textbf{Example}
The following code shows a call to the `INITIALIZE` procedure that initializes the `DBMS_RANDOM` package with the seed value 6475:

```sql
DBMS_RANDOM.INITIALIZE(6475);
```

**NORMAL**

The `NORMAL` function returns a random number of type `NUMBER`. The signature is:

```sql
<result> NUMBER NORMAL()
```

**Parameters**

- `result`
  
  `result` is a random value of type `NUMBER`.

**Example**

The following code shows a call to the `NORMAL` function:

```sql
x := DBMS_RANDOM.NORMAL();
```

**RANDOM**

The `RANDOM` function returns a random `INTEGER` value that is greater than or equal to `-2 ^31` and less than `2 ^31`. The signature is:

```sql
<result> INTEGER RANDOM()
```

This function is deprecated. It is included for backward compatibility.

**Parameters**

- `result`
  
  `result` is a random value of type `INTEGER`.

**Example**

The following code shows a call to the `RANDOM` function. The call returns a random number:

```sql
x := DBMS_RANDOM.RANDOM();
```

**SEED**

The first form of the `SEED` procedure resets the seed value for the `DBMS_RANDOM` package with an `INTEGER` value. The `SEED` procedure is available in two forms. The signature of the first form is:

```sql
SEED(<val> IN INTEGER)
```

**Parameters**

- `val`
  
  `val` is the seed value used by the `DBMS_RANDOM` package algorithm.
Example

The following code shows a call to the `SEED` procedure. The call sets the seed value at 8495.

```sql
DBMS_RANDOM.SEED(8495);
```

SEED

The second form of the `SEED` procedure resets the seed value for the `DBMS_RANDOM` package with a string value. The `SEED` procedure is available in two forms. The signature of the second form is:

```sql
SEED (<val> IN VARCHAR2)
```

Parameters

`val`

`val` is the seed value used by the `DBMS_RANDOM` package algorithm.

Example

The following code shows a call to the `SEED` procedure. The call sets the seed value to `abc123`.

```sql
DBMS_RANDOM.SEED('abc123');
```

STRING

The `STRING` function returns a random `VARCHAR2` string in a user-specified format. The signature of the `STRING` function is:

```sql
<result> VARCHAR2 STRING(<opt> IN CHAR, <len> IN NUMBER)
```

Parameters

`opt`

Formatting option for the returned string. `option` can be:

<table>
<thead>
<tr>
<th>Option</th>
<th>Specifies formatting option</th>
</tr>
</thead>
<tbody>
<tr>
<td>u or U</td>
<td>Uppercase alpha string</td>
</tr>
<tr>
<td>l or L</td>
<td>Lowercase alpha string</td>
</tr>
<tr>
<td>a or A</td>
<td>Mixed-case string</td>
</tr>
<tr>
<td>x or X</td>
<td>Uppercase alphanumeric string</td>
</tr>
<tr>
<td>p or P</td>
<td>Any printable characters</td>
</tr>
</tbody>
</table>

`len`

The length of the returned string.

`result`

`result` is a random value of type `VARCHAR2`.

Example

The following code shows a call to the `STRING` function. The call returns a random alphanumeric character string that is 10 characters long.
TERMINATE

The `TERMINATE` procedure has no effect. The signature is:

`TERMINATE`

The `TERMINATE` procedure is deprecated. The procedure is supported for compatibility.

VALUE

The `VALUE` function returns a random `NUMBER` that is greater than or equal to 0 and less than 1, with 38-digit precision. The `VALUE` function has two forms. The signature of the first form is:

```plaintext
<result> NUMBER VALUE()
```

Parameters

- `result`
  - `result` is a random value of type `NUMBER`.

Example

The following code shows a call to the `VALUE` function. The call returns a random `NUMBER`:

```sql
x := DBMS_RANDOM.VALUE();
```

VALUE

The `VALUE` function returns a random `NUMBER` with a value that is between boundaries that you specify. The `VALUE` function has two forms. The signature of the second form is:

```plaintext
<result> NUMBER VALUE(<low> IN NUMBER, <high> IN NUMBER)
```

Parameters

- `low`
  - `low` specifies the lower boundary for the random value. The random value can be equal to `low`.
- `high`
  - `high` specifies the upper boundary for the random value. The random value is less than `high`.
- `result`
  - `result` is a random value of type `NUMBER`.

Example

The following code shows a call to the `VALUE` function. The call returns a random `NUMBER` with a value that is greater than or equal to 1 and less than 100:

```sql
x := DBMS_RANDOM.VALUE(1, 100);
```
The `DBMS_REDACT` package enables you to redact or mask data returned by a query. The `DBMS_REDACT` package provides a procedure to create, alter, enable, disable, and drop policies. The procedures available in the `DBMS_REDACT` package are listed in the following table.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_POLICY</td>
<td>Procedure</td>
<td>n/a</td>
<td>Adds a data redaction policy.</td>
</tr>
<tr>
<td>ALTER_POLICY</td>
<td>Procedure</td>
<td>n/a</td>
<td>Alters the existing data redaction policy.</td>
</tr>
<tr>
<td>DISABLE_POLICY</td>
<td>Procedure</td>
<td>n/a</td>
<td>Disables the existing data redaction policy.</td>
</tr>
<tr>
<td>ENABLE_POLICY</td>
<td>Procedure</td>
<td>n/a</td>
<td>Enables a previously disabled data redaction policy.</td>
</tr>
<tr>
<td>DROP_POLICY</td>
<td>Procedure</td>
<td>n/a</td>
<td>Drops a data redaction policy.</td>
</tr>
<tr>
<td>UPDATE_FULL_REDACTION_VALUES</td>
<td>Procedure</td>
<td>n/a</td>
<td>Updates the full redaction default values for the specified datatype.</td>
</tr>
</tbody>
</table>

The data redaction feature uses the `DBMS_REDACT` package to define policies or conditions to redact data in a column based on the table column type and redaction type. You must be the owner of the table to create or change the data redaction policies. The users are exempted from all the column redaction policies, which the table owner or superuser is by default.

### Using DBMS_REDACT constants and function parameters

The `DBMS_REDACT` package uses the constants and redacts the column data by using any one of the data redaction types. The redaction type can be decided based on the `function_type` parameter of `dbms_redact.add_policy` and `dbms_redact.alter_policy` procedure. The table highlights the values for `function_type` parameters of `dbms_redact.add_policy` and `dbms_redact.alter_policy`.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>INTEGER</td>
<td>0</td>
<td>No redaction, zero effect on the result of a query against table.</td>
</tr>
<tr>
<td>FULL</td>
<td>INTEGER</td>
<td>1</td>
<td>Full redaction, redacts full values of the column data.</td>
</tr>
<tr>
<td>PARTIAL</td>
<td>INTEGER</td>
<td>2</td>
<td>Partial redaction, redacts a portion of the column data.</td>
</tr>
<tr>
<td>RANDOM</td>
<td>INTEGER</td>
<td>4</td>
<td>Random redaction, each query results in a different random value depending on the datatype of the column.</td>
</tr>
<tr>
<td>REGEXP</td>
<td>INTEGER</td>
<td>5</td>
<td>Regular-expression-based redaction, searches for the pattern of data to redact.</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>INTEGER</td>
<td>99</td>
<td>Custom redaction type.</td>
</tr>
</tbody>
</table>

The following table shows the values for the `action` parameter of `dbms_redact.alter_policy`.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_COLUMN</td>
<td>INTEGER</td>
<td>1</td>
<td>Adds a column to the redaction policy.</td>
</tr>
<tr>
<td>DROP_COLUMN</td>
<td>INTEGER</td>
<td>2</td>
<td>Drops a column from the redaction policy.</td>
</tr>
<tr>
<td>MODIFY_EXPRESSION</td>
<td>INTEGER</td>
<td>3</td>
<td>Modifies the expression of a redaction policy. The redaction is applied when the expression evaluates to the <code>BOOLEAN</code> value to <code>TRUE</code>.</td>
</tr>
<tr>
<td>MODIFY_COLUMN</td>
<td>INTEGER</td>
<td>4</td>
<td>Modifies a column in the redaction policy to change the redaction function type or function parameter.</td>
</tr>
<tr>
<td>SET_POLICY_DESCRIPTION</td>
<td>INTEGER</td>
<td>5</td>
<td>Sets the redaction policy description.</td>
</tr>
<tr>
<td>SET_COLUMN_DESCRIPTION</td>
<td>INTEGER</td>
<td>6</td>
<td>Sets a description for the redaction performed on the column.</td>
</tr>
</tbody>
</table>

The partial data redaction enables you to redact only a portion of the column data. To use partial redaction, you must set the `dbms_redact.add_policy` procedure `function_type` parameter to `dbms_redact.partial` and use the `function_parameters` parameter to specify the partial redaction behavior.
The data redaction feature provides a predefined format to configure policies that use the following datatypes:

- Character
- Number
- Datetime

The following table highlights the format descriptor for partial redaction with respect to datatype. The example shows how to perform a redaction for a string datatype (in this scenario, a Social Security Number (SSN)), a Number datatype, and a DATE datatype.

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Format descriptor</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>REDACT_PARTIAL_INPUT_FORMAT</td>
<td>Specifies the input format. Enter V for each character from the input string to be possibly redacted. Enter F for each character from the input string that can be considered as a separator such as blank spaces or hyphens.</td>
<td>VVVVFVFVVVFVVFVVF for masking first 5 digits of SSN strings such as 123-45-6789, adding a hyphen to format it and thereby resulting in strings such as XXX-XX-6789. The field value VVFVFVFVVVV for matching SSN strings such as 123-45-6789.</td>
</tr>
<tr>
<td>Character</td>
<td>REDACT_PARTIAL_OUTPUT_FORMAT</td>
<td>Specifies the output format. Enter V for each character from the input string to be possibly redacted. Replace each F character from the input format with a character such as a hyphen or any other separator.</td>
<td>The field value VVV-VV-VVVV can be used to redact SSN strings into XXX-XX-6789 where X comes from REDACT_PARTIAL_MASKCHAR field.</td>
</tr>
<tr>
<td>Character</td>
<td>REDACT_PARTIAL_MASKCHAR</td>
<td>Specifies the character to use for redaction.</td>
<td>X for redacting SSN strings into XXX-XX-6789.</td>
</tr>
<tr>
<td>Character</td>
<td>REDACT_PARTIAL_MASKFROM</td>
<td>Specifies the character to display in the range between 0 and 9.</td>
<td>'9, 1, 5' for redacting the first five digits of the Social Security Number 123456789 into 999996789.</td>
</tr>
</tbody>
</table>
| Number   | REDACT_PARTIAL_MASKFROM | Specifies the start-digit position for redaction. | |}

The following table represents function parameters values that you can use in partial redaction.

<table>
<thead>
<tr>
<th>Datetime</th>
<th>Format descriptor</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datetime</td>
<td>REDACT_PARTIAL_DATE_MONTH</td>
<td>'m' redacts the month. To mask a specific month, specify 'm#', where # indicates the month specified by its number between 1 and 12.</td>
<td>n3 displays as March.</td>
</tr>
<tr>
<td>Datetime</td>
<td>REDACT_PARTIAL_DATE_DAY</td>
<td>'d' redacts the day of the month. To mask with a day of the month, append 1-31 to a lowercase d.</td>
<td>d3 displays as 03.</td>
</tr>
<tr>
<td>Datetime</td>
<td>REDACT_PARTIAL_DATE_YEAR</td>
<td>'y' redacts the year. To mask with a year, append 1-9999 to a lowercase y.</td>
<td>y1968 displays as 68.</td>
</tr>
<tr>
<td>Datetime</td>
<td>REDACT_PARTIAL_DATE_HOUR</td>
<td>'h' redacts the hour. To mask with an hour, append 0-23 to a lowercase h.</td>
<td>h18 displays as 18.</td>
</tr>
<tr>
<td>Datetime</td>
<td>REDACT_PARTIAL_DATE_MINUTE</td>
<td>'m' redacts the minute. To mask with a minute, append 0-59 to a lowercase m.</td>
<td>n28 displays as 20.</td>
</tr>
<tr>
<td>Datetime</td>
<td>REDACT_PARTIAL_DATE_SECOND</td>
<td>'s' redacts the second. To mask with a second, append 0-59 to a lowercase s.</td>
<td>s48 displays as 40.</td>
</tr>
</tbody>
</table>

The following table represents function parameters values that you can use in partial redaction.
<table>
<thead>
<tr>
<th>Function parameter</th>
<th>Data type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDACT_US_SSN_F_5</td>
<td>VAR CHAR 2</td>
<td>'VVVVFVFVVVV,VVV-VVV-VVV-VVV,X,1,5'</td>
<td>Redacts the first 5 numbers of SSN. Example: The number 123-45-6789 becomes XXX-XX-6789.</td>
</tr>
<tr>
<td>REDACT_US_SSN_L_4</td>
<td>VAR CHAR 2</td>
<td>'VVVVFVFVVVV,VVV-VVV-VVV-VVV,X,6,9'</td>
<td>Redacts the last 4 numbers of SSN. Example: The number 123-45-6789 becomes 123-45-XXXX.</td>
</tr>
<tr>
<td>REDACT_US_SSN_ENTIRE</td>
<td>VAR CHAR 2</td>
<td>'VVVVFVFVVVV,VVV-VVV-VVV-VVV,X,1,9'</td>
<td>Redacts the entire SSN. Example: The number 123-45-6789 becomes XXX-XX-XXXX.</td>
</tr>
<tr>
<td>REDACT_NUM_US_SNS_F</td>
<td>VAR CHAR 2</td>
<td>'9,1,5'</td>
<td>Redacts the first 5 numbers of SSN when the column is a number datatype. Example: The number 123456789 becomes 999996789.</td>
</tr>
<tr>
<td>REDACT_NUM_US_SNS_L4</td>
<td>VAR CHAR 2</td>
<td>'9,6,9'</td>
<td>Redacts the last 4 numbers of SSN when the column is a number datatype. Example: The number 123456789 becomes 123459999.</td>
</tr>
<tr>
<td>REDACT_NUM_US_SNS_ENTIRE</td>
<td>VAR CHAR 2</td>
<td>'9,1,5'</td>
<td>Redacts the entire SSN when the column is a number datatype. Example: The number 123456789 becomes 999999999.</td>
</tr>
<tr>
<td>REDACT_ZIP_CODE</td>
<td>VAR CHAR 2</td>
<td>'VVVVV,VVVVVX,1,5'</td>
<td>Redacts a 5 digit zip code. Example: 12345 becomes XXXX.</td>
</tr>
<tr>
<td>REDACT_NUM_ZIP_CODE</td>
<td>VAR CHAR 2</td>
<td>'9,1,5'</td>
<td>Redacts a 5 digit zip code when the column is a number datatype. Example: 12345 becomes 99999.</td>
</tr>
<tr>
<td>REDACT_CCN16_FI_2</td>
<td>VAR CHAR 2</td>
<td>'VVVVVVVVVVVVVVVVV,VVVV-VVV-VVV-VVV+,1,12'</td>
<td>Redacts a 16 digit credit card number and displays only 4 digits. Example: 1234 5678 9000 2358 becomes <strong><strong>-</strong></strong>-****-2358.</td>
</tr>
<tr>
<td>REDACT_DATE_MILLENNIUM</td>
<td>VAR CHAR 2</td>
<td>'mld1y2080'</td>
<td>Redacts a date that is in the DD-MM-YY format. Example: Redacts all dates to 01-JAN-2080.</td>
</tr>
<tr>
<td>REDACT_DATE_EPOCH</td>
<td>VAR CHAR 2</td>
<td>'mld1y1970'</td>
<td>Redacts all dates to 01-JAN-78.</td>
</tr>
<tr>
<td>REDACT_AMEX_CCN_FORMATTED</td>
<td>VAR CHAR 2</td>
<td>'VVVVVVVVVVVVVVVVVVVVVVVVV+,VVVV-VVV-VVV-VVV+,+1,10'</td>
<td>Redacts the American Express credit card number and replaces the digit with * except for the last 5 digits. Example: The credit card number 1234 567890 34508 becomes **** ******* 34508.</td>
</tr>
<tr>
<td>REDACT_AMEX_CCN_NUMBER</td>
<td>VAR CHAR 2</td>
<td>'9,1,10'</td>
<td>Redacts the American Express credit card number and replaces the digit with 0 except for the last 5 digits. Example: The credit card number 1234 567890 34508 becomes 0000 000000 34508.</td>
</tr>
<tr>
<td>REDACT_SIN_FORMATTED</td>
<td>VAR CHAR 2</td>
<td>'VVVVFVFVVVV,VVV-VVV-VVV-VVV+,*,1,6'</td>
<td>Redacts the Social Insurance Number by replacing the first 6 digits by *. Example: 123-456-789 becomes <em><strong>-</strong></em>-789.</td>
</tr>
<tr>
<td>REDACT_SIN_NUMBER</td>
<td>VAR CHAR 2</td>
<td>'9,1,6'</td>
<td>Redacts the Social Insurance Number by replacing the first 6 digits by 0. Example: 123456789 becomes 999999789.</td>
</tr>
<tr>
<td>REDACT_SIN_UNFORMATTED</td>
<td>VAR CHAR 2</td>
<td>'VVVVVVVVVVVVV,VVVVVVV+,*,1,6'</td>
<td>Redacts the Social Insurance Number by replacing the first 6 digits by *. Example: 123456789 becomes ** ** ** ** **.</td>
</tr>
<tr>
<td>REDACT_CCN_FORMATTED</td>
<td>VAR CHAR 2</td>
<td>'VVVVVVVVVVVVVVVVVVVVVVVVV+,VVVV-VVV-VVV-VVV+,+1,12'</td>
<td>Redacts a credit card number by * and displays only 4 digits. Example: The credit card number 1234-5678-9000-4671 becomes ***<strong><strong>-</strong></strong>-4671.</td>
</tr>
<tr>
<td>REDACT_CCN_NUMBER</td>
<td>VAR CHAR 2</td>
<td>'9,1,12'</td>
<td>Redacts a credit card number by 0 except for the last 4 digits. Example: The credit card number 1234567890084671 becomes 00000008004671.</td>
</tr>
<tr>
<td>REDACT_NA_PHONE_FORMATTED</td>
<td>VAR CHAR 2</td>
<td>'VVVVFVFVVVVV,VVV-VVV-VVV-VVV,X,4,10'</td>
<td>Redacts the North American phone number by * leaving the area code. Example: 123-456-7890 becomes 123-XXX-XXXX.</td>
</tr>
<tr>
<td>REDACT_NA_PHONE_NUMBER</td>
<td>VAR CHAR 2</td>
<td>'9,4,10'</td>
<td>Redacts the North American phone number by 0 leaving the area code. Example: 1234567890 becomes 1230000008.</td>
</tr>
<tr>
<td>REDACT_NA_PHONE_UNFORMATTED</td>
<td>VAR CHAR 2</td>
<td>'VVVVVVVVVVVVVVVVVVVVVVVVV+,VVVV-VVV-VVV-VVV+,+1,8'</td>
<td>Redacts the North American phone number by * leaving the area code. Example: 1234567890 becomes 123XXXXXXXX.</td>
</tr>
<tr>
<td>REDACT_UK_NIN_FORMATTED</td>
<td>VAR CHAR 2</td>
<td>'VVVVVVVVVVVVVVVVVVVVVVVVV,+X,3,0'</td>
<td>Redacts the UK National Insurance Number by X but leaving the alphabetic characters. Example: NY 22 81 34 D becomes NY XX XX XX D.</td>
</tr>
</tbody>
</table>
This table shows the `regexp_replace_string` values that you can use during `REGEXP` based redaction.

<table>
<thead>
<tr>
<th>Function parameter</th>
<th>Data type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REDACT UK NIN UNFORMATTED</td>
<td>VAR CHAR 2</td>
<td>'VVVVVVVV,VVVVVVVV,X,3,8'</td>
<td>Redacts the UK National Insurance Number by X but leaving the alphabetic characters. Example: <code>NY228134D</code></td>
</tr>
<tr>
<td>RE_PATTERN_CC_L6_T4</td>
<td>VARCH AR2</td>
<td>'(\d{6})(\d{4})'</td>
<td>Searches for the middle digits of a credit card number that includes 6 leading digits and 4 trailing digits. The <code>regexp_replace_string</code> setting to use with the format is <code>RE_REDACT_CC_MIDDLE_DIGITS</code> that replaces the identified pattern with the characters specified by the <code>RE_PATTERN_CC_L6_T4</code> parameter.</td>
</tr>
<tr>
<td>RE_PATTERN_ANY_DIGIT</td>
<td>VARCH AR2</td>
<td>'d'</td>
<td>Searches for any digit and replaces the identified pattern with the characters specified by the <code>RE_PATTERN_ANY_DIGIT</code> parameter. The <code>regexp_replace_string</code> parameter with the <code>\d</code> value will return one digit for each of the data digits as specified by setting the <code>\d</code> parameter with the <code>\d{1,3}</code> value. Example: <code>123-45-6789</code> becomes <code>111-111-1111</code>.</td>
</tr>
<tr>
<td>RE_PATTERN_US_PHONE</td>
<td>VARCH AR2</td>
<td>'(\d{3})-(\d{3})-(\d{4})'</td>
<td>Searches for the U.S. phone number and replaces the identified pattern with the characters specified by the <code>RE_PATTERN_US_PHONE</code> parameter. The <code>regexp_replace_string</code> parameter with the <code>\d{1,3}</code> value will return one digit for each of the data digits as specified by setting the <code>\d</code> parameter with the <code>\d{1,3}</code> value. Example: <code>123-444-5900</code> becomes <code>XXX-XXX-XXXX</code>.</td>
</tr>
<tr>
<td>RE_PATTERN_EMAIL_ADDRESS</td>
<td>VARCH AR2</td>
<td>'.<em>@.</em>'</td>
<td>Searches for the email address and replaces the identified pattern with the characters specified by the <code>RE_PATTERN_EMAIL_ADDRESS</code> parameter. The <code>regexp_replace_string</code> parameter with the <code>\d</code> value will return one digit for each of the data digits as specified by setting the <code>\d</code> parameter with the <code>\d{1,3}</code> value. Example: <code>example@domain.com</code> becomes <code>example@domain.com</code>.</td>
</tr>
<tr>
<td>RE_PATTERN_IP_ADDRESS</td>
<td>VARCH AR2</td>
<td>'.*'</td>
<td>Searches for an IP address and replaces the identified pattern with the characters specified by the <code>RE_PATTERN_IP_ADDRESS</code> parameter. The <code>regexp_replace_string</code> parameter to be used is <code>RE_REDACT_IP_L3</code> that replaces the last section of an IP address with <code>999</code> and indicates it is redacted.</td>
</tr>
<tr>
<td>RE_PATTERN_CC_L6_T3</td>
<td>VARCH AR2</td>
<td>'.*'</td>
<td>Searches for the middle digits of a credit card number that includes 6 leading digits and 4 trailing digits. The <code>regexp_replace_string</code> parameter to be used is <code>RE_REDACT_CC_MIDDLE_DIGITS</code> that replaces the identified pattern with the characters specified by the <code>RE_PATTERN_CC_L6_T3</code> parameter.</td>
</tr>
<tr>
<td>RE_PATTERN_CC_L6_T5</td>
<td>VARCH AR2</td>
<td>'.*'</td>
<td>Searches for the middle digits of a credit card number that includes 6 leading digits and 4 trailing digits. The <code>regexp_replace_string</code> parameter to be used is <code>RE_REDACT_CC_MIDDLE_DIGITS</code> that replaces the identified pattern with the characters specified by the <code>RE_PATTERN_CC_L6_T5</code> parameter.</td>
</tr>
<tr>
<td>RE_PATTERN_CC_L6_T6</td>
<td>VARCH AR2</td>
<td>'.*'</td>
<td>Searches for the middle digits of a credit card number that includes 6 leading digits and 4 trailing digits. The <code>regexp_replace_string</code> parameter to be used is <code>RE_REDACT_CC_MIDDLE_DIGITS</code> that replaces the identified pattern with the characters specified by the <code>RE_PATTERN_CC_L6_T6</code> parameter.</td>
</tr>
<tr>
<td>REDACT_CC_MIDDLE_DIGITS</td>
<td>VARCH AR2</td>
<td>'11-XXX-XX'</td>
<td>Redacts the middle digits of a credit card number according to the <code>regexp_pattern</code> parameter with the <code>RE_PATTERN_CC_L6_T4</code> format and replaces each redacted character with an X. Example: The credit card number 1234 5678 9000 2490 becomes 1234 56XX 9000 2490.</td>
</tr>
<tr>
<td>REDACT_CC_MIDDLE_DIGITS</td>
<td>VARCH AR2</td>
<td>'11-XXX-XXX'</td>
<td>Redacts the middle digits of a credit card number according to the <code>regexp_pattern</code> parameter with the <code>RE_PATTERN_CC_L6_T4</code> format and replaces each redacted character with an X. Example: The credit card number 1234 5678 9000 2490 becomes 1111 1111 1111.</td>
</tr>
<tr>
<td>REDACT_CC_MIDDLE_DIGITS</td>
<td>VARCH AR2</td>
<td>'11-XXX-XXXX'</td>
<td>Redacts the middle digits of a credit card number according to the <code>regexp_pattern</code> parameter with the <code>RE_PATTERN_CC_L6_T4</code> format and replaces each redacted character with an X. Example: The credit card number 1234 5678 9000 2490 becomes 123-45-6789.</td>
</tr>
<tr>
<td>REDACT_CC_MIDDLE_DIGITS</td>
<td>VARCH AR2</td>
<td>'11-XXX-XXXX'</td>
<td>Redacts the middle digits of a credit card number according to the <code>regexp_pattern</code> parameter with the <code>RE_PATTERN_CC_L6_T4</code> format and replaces each redacted character with an X. Example: The credit card number 1234 5678 9000 2490 becomes 123-444-5900.</td>
</tr>
</tbody>
</table>

A regular expression-based redaction searches for patterns of data to redact. The `regexp_pattern` search the values for the `regexp_replace_string` to change the value. The following table shows the `regexp_pattern` values that you can use during `REGEXP` based redaction.
Redacts the email name according to the `regexp_pattern` parameter with the `RE_PATTERN_EMAIL_ADDRESS` format and replaces the email username with the four `x` characters.

**Example:** The email address `sjohn@example.com` becomes `xxxx@example.com`.

Redacts the email domain name according to the `regexp_pattern` parameter with the `RE_PATTERN_EMAIL_ADDRESS` format and replaces the domain with the five `x` characters.

**Example:** The email address `sjohn@example.com` becomes `sjohn@xxxxx.com`.

Redacts the entire email address according to the `regexp_pattern` parameter with the `RE_PATTERN_EMAIL_ADDRESS` format and replaces the email address with the `x` characters.

**Example:** The email address `sjohn@example.com` becomes `xxxx@xxxxx.com`.

Redacts the last 3 digits of an IP address according to the `regexp_pattern` parameter with the `RE_PATTERN_IP_ADDRESS` format.

**Example:** The IP address `172.0.1.258` becomes `172.0.1.999`, which is an invalid IP address.

Redacts the first 10 digits of an American Express credit card number according to the `regexp_pattern` parameter with the `RE_PATTERN_AMEX_CCN` format.

**Example:** `123456789062816` becomes `**********62816`.

Redacts the first 12 digits of a credit card number as specified by the `regexp_pattern` parameter with the `RE_PATTERN_CCN` format.

**Example:** `8749012678345671` becomes `************5671`.

The following tables show the `regexp_position` value and `regexp_occurrence` values that you can use during `REGEXP` based redaction.

<table>
<thead>
<tr>
<th>Function parameter</th>
<th>Data type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE_BEGINNING</td>
<td>INTEGER</td>
<td>1</td>
<td>Specifies the position of a character where search must begin. By default, the value is 1 that indicates the search begins at the first character of <code>source_char</code>.</td>
</tr>
<tr>
<td>RE_ALL</td>
<td>INTEGER</td>
<td>0</td>
<td>Specifies the replacement occurrence of a substring. If the value is 0, then the replacement of each matching substring occurs.</td>
</tr>
<tr>
<td>RE_FIRST</td>
<td>INTEGER</td>
<td>1</td>
<td>Specifies the replacement occurrence of a substring. If the value is 1, then the replacement of the first matching substring occurs.</td>
</tr>
</tbody>
</table>

The following table shows the `regexp_match_parameter` values that you can use during `REGEXP` based redaction which lets you change the default matching behavior of a function.

<table>
<thead>
<tr>
<th>Function parameter</th>
<th>Data type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE_CASE_SENSITIVE</td>
<td>VARCHAR2</td>
<td>'c'</td>
<td>Specifies the case-sensitive matching.</td>
</tr>
<tr>
<td>RE_CASE_INSENSITIVE</td>
<td>VARCHAR2</td>
<td>'i'</td>
<td>Specifies the case-insensitive matching.</td>
</tr>
<tr>
<td>RE_MULTIPLE_LINES</td>
<td>VARCHAR2</td>
<td>'n'</td>
<td>Treats the source string as multiple lines but if you omit this parameter, then it indicates as a single line.</td>
</tr>
<tr>
<td>RE_NEWLINE_WILDCARD</td>
<td>VARCHAR2</td>
<td>'n'</td>
<td>Specifies the period (.), but if you omit this parameter, then the period does not match the newline character.</td>
</tr>
<tr>
<td>RE_IGNORE_WHITESPACE</td>
<td>VARCHAR2</td>
<td>'x'</td>
<td>Ignores the whitespace characters.</td>
</tr>
</tbody>
</table>

**Note**

If you create a redaction policy based on a numeric-type column, then make sure that the result after redaction is a number and set the replacement string accordingly to avoid runtime errors.

**Note**

If you create a redaction policy based on a character-type column, then make sure that a length of the result after redaction is compatible with the column type and set the replacement string accordingly to avoid runtime errors.

**ADD_POLICY**

The `add_policy` procedure creates a new data redaction policy for a table.
Parameters

**object_schema**

Specifies the name of the schema in which the object resides and on which the data redaction policy is applied. If you specify `NULL`, then the given object is searched by the order specified by the `search_path` setting.

**object_name**

Name of the table on which the data redaction policy is created.

**policy_name**

Name of the policy to add. Ensure that the `policy_name` is unique for the table on which the policy is created.

**policy_description**

Specify the description of a redaction policy.

**column_name**

Name of the column to which the redaction policy applies. To redact more than one column, use the `alter_policy` procedure to add more columns.

**column_description**

Description of the column to redact. The `column_description` isn't supported, but if you specify the description for a column, a warning message appears.

**function_type**

The type of redaction function to use. The possible values are `NONE`, `FULL`, `PARTIAL`, `RANDOM`, `REGEXP`, and `CUSTOM`.

**function_parameters**

Specifies the function parameters for the partition redaction and is applicable only for partial redaction.

**expression**

Specifies the Boolean expression for the table and determines how to apply the policy. The redaction occurs if this policy expression evaluates to `TRUE`.

**enable**

When set to `TRUE`, the policy is enabled upon creation. The default is `TRUE`. When set to `FALSE`, the policy is disabled, but you can enable the policy by calling the `enable_policy` procedure.

**regexp_pattern**

Specifies the regular expression pattern to redact data. If the `regexp_pattern` doesn't match, then the `NULL` value is returned.

**regexp_replace_string**

Specifies the replacement string value.
**regexp_position**

Specifies the position of a character where search must begin. By default, the function parameter is `RE_BEGINNING`.

**regexp_occurrence**

Specifies the replacement occurrence of a substring. If the constant is `RE_ALL`, then the replacement of each matching substring occurs. If the constant is `RE_FIRST`, then the replacement of the first matching substring occurs.

**regexp_match_parameter**

Changes the default matching behavior of a function. The possible `regexp_match_parameter` constants can be `RE_CASE_SENSITIVE`, `RE_CASE_INSENSITIVE`, `RE_MULTIPLE_LINES`, `RE_NEWLINE_WILDCARD`, and `RE_IGNORE_WHITESPACE`.

**custom_function_expression**

The `custom_function_expression` applies only for the `CUSTOM` redaction type. The `custom_function_expression` is a function expression, that is, a schema-qualified function with a parameter such as `schema_name.function_name (argument1, ..)` that allows a user to use their redaction logic to redact the column data.

**Example**

This example shows how to create a policy and use full redaction for values in the `payment_details_tab` table `customer_id` column.

```sql
CREATE TABLE payment_details_tab
(
  customer_id NUMBER NOT NULL,
  card_string VARCHAR2(19) NOT NULL);
BEGIN
  INSERT INTO payment_details_tab VALUES (4000, '1234-1234-1234-1234');
  INSERT INTO payment_details_tab VALUES (4001, '2345-2345-2345-2345');
END;
```

**Redacted Result:**

```
customer_id
------------
edb= edb=
```

```sql
CREATE TABLE payment_details_tab
(
  customer_id NUMBER NOT NULL,
  card_string VARCHAR2(19) NOT NULL);
BEGIN
  INSERT INTO payment_details_tab VALUES (4000, '1234-1234-1234-1234');
  INSERT INTO payment_details_tab VALUES (4001, '2345-2345-2345-2345');
END;
```

**Example**

This example shows how to create a policy and use full redaction for values in the `payment_details_tab` table `customer_id` column.

```sql
CREATE TABLE payment_details_tab
(
  customer_id NUMBER NOT NULL,
  card_string VARCHAR2(19) NOT NULL);
BEGIN
  INSERT INTO payment_details_tab VALUES (4000, '1234-1234-1234-1234');
  INSERT INTO payment_details_tab VALUES (4001, '2345-2345-2345-2345');
END;
```

**Redacted Result:**

```
customer_id
------------
edb= edb=
```

```sql
CREATE TABLE payment_details_tab
(
  customer_id NUMBER NOT NULL,
  card_string VARCHAR2(19) NOT NULL);
BEGIN
  INSERT INTO payment_details_tab VALUES (4000, '1234-1234-1234-1234');
  INSERT INTO payment_details_tab VALUES (4001, '2345-2345-2345-2345');
END;
```

**Redacted Result:**

```
customer_id
------------
edb= edb=
```

```sql
CREATE TABLE payment_details_tab
(
  customer_id NUMBER NOT NULL,
  card_string VARCHAR2(19) NOT NULL);
BEGIN
  INSERT INTO payment_details_tab VALUES (4000, '1234-1234-1234-1234');
  INSERT INTO payment_details_tab VALUES (4001, '2345-2345-2345-2345');
END;
```

**Redacted Result:**

```
customer_id
------------
edb= edb=
```

```sql
CREATE TABLE payment_details_tab
(
  customer_id NUMBER NOT NULL,
  card_string VARCHAR2(19) NOT NULL);
BEGIN
  INSERT INTO payment_details_tab VALUES (4000, '1234-1234-1234-1234');
  INSERT INTO payment_details_tab VALUES (4001, '2345-2345-2345-2345');
END;
```
ALTER_POLICY

The `alter_policy` procedure alters or modifies an existing data redaction policy for a table.

**PROCEDURE** alter_policy

```
PROCEDURE alter_policy
(
  <object_schema> IN VARCHAR2 DEFAULT NULL,
  <object_name> IN VARCHAR2,
  <policy_name> IN VARCHAR2,
  <action> IN INTEGER DEFAULT DBMS_REDACT.ADD_COLUMN,
  <column_name> IN VARCHAR2 DEFAULT NULL,
  <function_type> IN INTEGER DEFAULT DBMS_REDACT.FULL,
  <function_parameters> IN VARCHAR2 DEFAULT NULL,
  <expression> IN VARCHAR2 DEFAULT NULL,
  <regexp_pattern> IN VARCHAR2 DEFAULT NULL,
  <regexp_replace_string> IN VARCHAR2 DEFAULT NULL,
  <regexp_position> IN INTEGER DEFAULT DBMS_REDACT.RE_BEGINNING,
  <regexp_occurrence> IN INTEGER DEFAULT DBMS_REDACT.RE_ALL,
  <regexp_match_parameter> IN VARCHAR2 DEFAULT NULL,
  <policy_description> IN VARCHAR2 DEFAULT NULL,
  <column_description> IN VARCHAR2 DEFAULT NULL,
  <custom_function_expression> IN VARCHAR2 DEFAULT NULL
)
```

**Parameters**

- **object_schema**
  Specifies the name of the schema in which the object resides and on which to alter the data redaction policy. If you specify `NULL`, then the given object is searched by the order specified by `search_path` setting.

- **object_name**
  Name of the table to which to alter a data redaction policy.

- **policy_name**
  Name of the policy to alter.

- **action**
  The action to perform. For more information about action parameters see, Using DBMS_REDACT Constants and Function Parameters

- **column_name**
  Name of the column to which the redaction policy applies.

- **function_type**
  The type of redaction function to use. The possible values are `NONE`, `FULL`, `PARTIAL`, `RANDOM`, `REGEXP`, and `CUSTOM`.

- **function_parameters**
  Specifies the function parameters for the redaction function.

- **expression**
  Specifies the Boolean expression for the table and determines how to apply the policy. The redaction occurs if this policy expression evaluates to `TRUE`.

- **regexp_pattern**
  Enables the use of regular expressions to redact data. If the `regexp_pattern` doesn't match the data, then the `NULL` value is returned.
**regexp_replace_string**

Specifies the replacement string value.

**regexp_position**

Specifies the position of a character where search must begin. By default, the function parameter is `RE_BEGINNING`.

**regexp_occurrence**

Specifies the replacement occurrence of a substring. If the constant is `RE_ALL`, then the replacement of each matching substring occurs. If the constant is `RE_FIRST`, then the replacement of the first matching substring occurs.

**regexp_match_parameter**

Changes the default matching behavior of a function. The possible `regexp_match_parameter` constants can be `RE_CASE_SENSITIVE`, `RE_CASE_INSENSITIVE`, `RE_MULTIPLE_LINES`, `RE_NEWLINE_WILDCARD`, and `RE_IGNORE_WHITESPACE`.

!!Note For more information on constants, function_parameters, or `regexp`, see Using DBMS_REDACT Constants and Function Parameters.

**policy_description**

Specify the description of a redaction policy.

**column_description**

Description of the column to redact. The `column_description` isn't supported, but if you specify the description for a column, a warning message appears.

**custom_function_expression**

The `custom_function_expression` applies only for the CUSTOM redaction type. The `custom_function_expression` is a function expression, that is, a schema-qualified function with a parameter such as `schema_name.function_name (argument1, …)` that allows a user to use their redaction logic to redact the column data.

**Example**

This example shows how to alter a policy using partial redaction for values in the `payment_details_tab` table `card_string` (usually a credit card number) column.

```sql
\c edb base
__user
BEGIN
DBMS_REDACT.alter_policy
  (object_schema => 'public',
   object_name => 'payment_details_tab',
   policy_name => 'redactPolicy_001',
   action => DBMS_REDACT.ADD_COLUMN,
   column_name => 'card_string',
   function_type => DBMS_REDACT.partial,
   function_parameters => DBMS_REDACT.REDACT_CCN16_F12);
END;
```

Redacted Result:

```sql
EDB Postgres Advanced Server (EPAS)  Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
```

```sql
<table>
<thead>
<tr>
<th>customer_id</th>
<th>card_string</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>************-1234</td>
</tr>
<tr>
<td>0</td>
<td>************-2345</td>
</tr>
</tbody>
</table>
(2 rows)
```
The `disable_policy` procedure disables an existing data redaction policy.

```sql
PROCEDURE disable_policy
(
  <object_schema> IN VARCHAR2 DEFAULT NULL,
  <object_name> IN VARCHAR2,
  <policy_name> IN VARCHAR2
)
```

**Parameters**

- **object_schema**
  Specifies the name of the schema in which the object resides and on which to apply the data redaction policy. If you specify `NULL`, then the given object is searched by the order specified by `search_path` setting.

- **object_name**
  Name of the table for which to disable a data redaction policy.

- **policy_name**
  Name of the policy to disable.

**Example**

This example shows how to disable a policy.

```
c edb
base_user
BEGIN
  DBMS_REDACT.disable_policy(
    object_schema => 'public',
    object_name => 'payment_details_tab',
    policy_name => 'redactPolicy_001');
END;
```

Redacted Result: Data is no longer redacted after disabling a policy.

---

**ENABLE_POLICY**

The `enable_policy` procedure enables the previously disabled data redaction policy.

```sql
PROCEDURE enable_policy
(
  <object_schema> IN VARCHAR2 DEFAULT NULL,
  <object_name> IN VARCHAR2,
  <policy_name> IN VARCHAR2
)
```

**Parameters**

- **object_schema**
  Specifies the name of the schema in which the object resides and on which to apply the data redaction policy. If you specify `NULL`, then the given object is searched by the order specified by `search_path` setting.

- **object_name**
  Name of the table to which to enable a data redaction policy.

- **policy_name**
  Name of the policy to enable.
Name of the policy to enable.

Example

This example shows how to enable a policy.

```sql
\c edb
base_user
BEGIN
DBMS_REDACT.enable_policy(
    object_schema => 'public',
    object_name => 'payment_details_tab',
    policy_name => 'redactPolicy_001');
END;
```

Redacted Result: Data is redacted after enabling a policy.

DROP_POLICY

The `drop_policy` procedure drops a data redaction policy by removing the masking policy from a table.

```sql
PROCEDURE drop_policy
(
    <object_schema> IN VARCHAR2 DEFAULT NULL,
    <object_name> IN VARCHAR2,
    <policy_name> IN VARCHAR2
)
```

Parameters

- **object_schema**
  Specifies the name of the schema in which the object resides and on which to apply the data redaction policy. If you specify `NULL`, then the given object is searched by the order specified by `search_path` setting.

- **object_name**
  Name of the table from which to drop a data redaction policy.

- **policy_name**
  Name of the policy to drop.

Example

This example shows how to drop a policy.

```sql
\c edb
base_user
BEGIN
DBMS_REDACT.drop_policy(
    object_schema => 'public',
    object_name => 'payment_details_tab',
    policy_name => 'redactPolicy_001');
END;
```

Redacted Result: The server drops the specified policy.
UPDATE_FULL_REDACTION_VALUES

The `update_full_redaction_values` procedure updates the default displayed values for a data redaction policy. You can view these default values using the `redaction_values_for_type_full` view that uses the full redaction type.

```sql
PROCEDURE update_full_redaction_values
(
    <number_val>
    IN NUMBER DEFAULT NULL,
    <binfloat_val>
    IN FLOAT4 DEFAULT NULL,
    <bindouble_val>
    IN FLOAT8 DEFAULT NULL,
    <char_val>
    IN CHAR DEFAULT NULL,
    <varchar_val>
    IN VARCHAR2 DEFAULT NULL,
    <nchar_val>
    IN NCHAR DEFAULT NULL,
    <nvarchar_val>
    IN NVARCHAR2 DEFAULT NULL,
    <datecol_val>
    IN DATE DEFAULT NULL,
    <ts_val>
    IN TIMESTAMP DEFAULT NULL,
    <tswtz_val>
    IN TIMESTAMPTZ DEFAULT NULL,
    <blob_val>
    IN BLOB DEFAULT NULL,
    <clob_val>
    IN CLOB DEFAULT NULL,
    <nclob_val>
    IN CLOB DEFAULT NULL
)
```

Parameters

`number_val`

Updates the default value for columns of the `NUMBER` datatype.

`binfloat_val`

The `FLOAT4` datatype is a random value. The binary float datatype isn't supported.

`bindouble_val`

The `FLOAT8` datatype is a random value. The binary double datatype isn't supported.

`char_val`

Updates the default value for columns of the `CHAR` datatype.

`varchar_val`

Updates the default value for columns of the `VARCHAR2` datatype.

`nchar_val`

The `nchar_val` is mapped to `CHAR` datatype and returns the `CHAR` value.

`nvarchar_val`

The `nvarchar_val` is mapped to `VARCHAR2` datatype and returns the `VARCHAR` value.

`datecol_val`

Updates the default value for columns of the `DATE` datatype.

`ts_val`

Updates the default value for columns of the `TIMESTAMP` datatype.

`tswtz_val`

Updates the default value for columns of the `TIMESTAMPTZ` datatype.

`blob_val`
Updates the default value for columns of the **BLOB** datatype.

```
clob_val
```

Updates the default value for columns of the **CLOB** datatype.

```
cmp_val
```

The `ncmp_val` is mapped to **CLOB** datatype and returns the **CLOB** value.

**Example**

This example shows how to update the full redaction values. Before updating the values, you can view the default values using the `redaction_values_for_type_full` view.

```
edb=# \x
Expanded display is on.
edb=# SELECT number_value, char_value, varchar_value, date_value, timestamp_value, timestamp_with_time_zone_value, blob_value, clob_value FROM redaction_values_for_type_full;

<table>
<thead>
<tr>
<th>number_value</th>
<th>char_value</th>
<th>varchar_value</th>
<th>date_value</th>
<th>timestamp_value</th>
<th>timestamp_with_time_zone_value</th>
<th>blob_value</th>
<th>clob_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>01-JAN-01 00:00:00</td>
<td>01-JAN-01 01:00:00</td>
<td>31-DEC-00 20:00:00 -05:00</td>
<td>\x5b72656461637465645d</td>
<td>[redacted]</td>
</tr>
</tbody>
</table>
(1 row)
```

Update the default values for full redaction type. The **NULL** values are ignored.

```
\c edb
base_user

edb=# BEGIN
DBMS_REDACT.update_full_redaction_values
(
  number_val => 9999999,
  char_val => 'Z',
  varchar_val => 'V',
  datecol_val => to_date('17/10/2018', 'DD/MM/YYYY'),
  ts_val => to_timestamp('17/10/2018 11:12:13', 'DD/MM/YYYY HH24:MI:SS'),
  tswtz_val => NULL,
  blob_val => 'NEW REDACTED VALUE',
  clob_val => 'NEW REDACTED VALUE');
END;
```

You can now see the updated values using the `redaction_values_for_type_full` view.

```
EDB-SPL Procedure successfully completed
edb=# SELECT number_value, char_value, varchar_value, date_value, timestamp_value, timestamp_with_time_zone_value, blob_value, clob_value FROM redaction_values_for_type_full;

<table>
<thead>
<tr>
<th>number_value</th>
<th>char_value</th>
<th>varchar_value</th>
<th>date_value</th>
<th>timestamp_value</th>
<th>timestamp_with_time_zone_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999999</td>
<td>Z</td>
<td>V</td>
<td>17-OCT-18 00:00:00</td>
<td>17-OCT-18 11:12:13</td>
<td>31-DEC-00 20:00:00 -05:00</td>
</tr>
</tbody>
</table>
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The **DBMS_RLS** package enables you to implement Virtual Private Database on certain EDB Postgres Advanced Server database objects.

EDB Postgres Advanced Server's implementation of **DBMS_RLS** is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table are supported.

### Function/procedure

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Function or procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_POLICY(object_schema, object_name, policy_name, function_schema, policy_function[, statement_types[, update_check[, enable[, static_policy[, policy_type[, long_predicate[, sec_relevant_cols[, sec_relevant_cols_opt]]]]]]]]))</td>
<td>Procedure</td>
<td>n/a</td>
<td>Add a security policy to a database object.</td>
</tr>
<tr>
<td>DROP_POLICY(object_schema, object_name, policy_name)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Remove a security policy from a database object.</td>
</tr>
<tr>
<td>ENABLE_POLICY(object_schema, object_name, policy_name, enable)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Enable or disable a security policy.</td>
</tr>
</tbody>
</table>

Virtual Private Database is a type of fine-grained access control using security policies. Fine-grained access control in Virtual Private Database means that you can control access to data down to specific rows as defined by the security policy.

The rules that encode a security policy are defined in a policy function, which is an SPL function with certain input parameters and return values. The security policy is the named association of the policy function to a particular database object, typically a table.

#### Note

In EDB Postgres Advanced Server, you can write the policy function in any language supported by EDB Postgres Advanced Server, such as SQL, PL/pgSQL, and SPL.

#### Note

The database objects currently supported by EDB Postgres Advanced Server Virtual Private Database are tables. You can't apply policies to views or synonyms.

Virtual Private Database has these advantages:

- Provides a fine-grained level of security. Database object-level privileges given by the **GRANT** command determine access privileges to the entire instance of a database object, while Virtual Private Database provides access control for the individual rows of a database object.
- You can apply a different security policy depending on the type of SQL command (INSERT, UPDATE, DELETE, or SELECT).
- The security policy can vary dynamically for each applicable SQL command affecting the database object, depending on factors such as the session user of the application accessing the database object.
- Invoking the security policy is transparent to all applications that access the database object. Thus, you don't have to modify individual applications to apply the security policy.
- Once a security policy is enabled, applications (including new applications) can't circumvent the security policy except by the system privilege described in the note that follows. Even superusers can't circumvent the security policy except by the system privilege described in the note that follows.

#### Note

The only way to circumvent security policies is if the **EXEMPT ACCESS POLICY** system privilege is granted to a user. Grant the **EXEMPT ACCESS POLICY** privilege with extreme care as a user with this privilege is exempted from all policies in the database.

The **DBMS_RLS** package provides procedures to create policies, remove policies, enable policies, and disable policies.

---

**Redacted Result:**

```sql
edb# c edb
You are now connected to database "edb" as user "redact_user".

edb> select * from payment_details_tab order by 1;

<table>
<thead>
<tr>
<th>customer_id</th>
<th>card_string</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999999</td>
<td>V</td>
</tr>
<tr>
<td>9999999</td>
<td>V</td>
</tr>
</tbody>
</table>
```

14.3.3.1.15 **DBMS_RLS**

The **DBMS_RLS** package provides procedures to create policies, remove policies, enable policies, and disable policies.
The process for implementing Virtual Private Database is as follows:

1. Create a policy function. The function must have two input parameters of type VARCHAR2. The first input parameter is for the schema containing the database object to which the policy applies. The second input parameter is for the name of that database object. The function must have a VARCHAR2 return type and return a string in the form of a WHERE clause predicate. This predicate is dynamically appended as an AND condition to the SQL command that acts on the database object. Thus, rows that don't satisfy the policy function predicate are filtered out from the SQL command result set.

2. Use the ADD_POLICY procedure to define a new policy, which is the association of a policy function with a database object. With the ADD_POLICY procedure, you can also specify:
   - The types of SQL commands (INSERT, UPDATE, DELETE, or SELECT) to which the policy applies
   - Whether to enable the policy at the time of its creation
   - Whether the policy applies to newly inserted rows or the modified image of updated rows

3. Use the ENABLE_POLICY procedure to disable or enable an existing policy.

4. Use the DROP_POLICY procedure to remove an existing policy. The DROP_POLICY procedure doesn't drop the policy function or the associated database object.

Once you create policies, you can view them in the catalog views compatible with Oracle databases: ALL_POLICIES, DBA_POLICIES, or USER_POLICIES. The supported compatible views are listed in Catalog views.

The SYS_CONTEXT function is often used with DBMS_RLS. The signature is:

```
SYS_CONTEXT(<namespace>, <attribute>)
```

Where:

- namespace is a VARCHAR2. The only accepted value is USERENV. Any other value returns NULL.
- attribute is a VARCHAR2. Possible values are:

<table>
<thead>
<tr>
<th>attribute</th>
<th>Equivalent value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESSION_USER</td>
<td>pg_catalog.session_user</td>
</tr>
<tr>
<td>CURRENT_USER</td>
<td>pg_catalog.current_user</td>
</tr>
<tr>
<td>CURRENT_SCHEMA</td>
<td>pg_catalog.current_schema</td>
</tr>
<tr>
<td>HOST</td>
<td>pg_catalog.inet_host</td>
</tr>
<tr>
<td>IP_ADDRESS</td>
<td>pg_catalog.inet_client_addr</td>
</tr>
<tr>
<td>SERVER_HOST</td>
<td>pg_catalog.inet_server_addr</td>
</tr>
</tbody>
</table>

The examples used to illustrate the DBMS_RLS package are based on a modified copy of the sample emp table provided with EDB Postgres Advanced Server along with a role named salesmgr that is granted all privileges on the table. You can create the modified copy of the emp table named vpemp and the salesmgr role as follows:

```
CREATE TABLE public.vpemp AS SELECT empno, ename, job, sal, comm, deptno FROM emp;
ALTER TABLE vpemp ADD authid VARCHAR2(12);
UPDATE vpemp SET authid = 'researchmgr' WHERE deptno = 20;
UPDATE vpemp SET authid = 'salesmgr' WHERE deptno = 30;
SELECT * FROM vpemp;
```

The examples used to illustrate the DBMS_RLS package are based on a modified copy of the sample emp table provided with EDB Postgres Advanced Server along with a role named salesmgr that is granted all privileges on the table. You can create the modified copy of the emp table named vpemp and the salesmgr role as follows:

```
CREATE ROLE salesmgr WITH LOGIN PASSWORD 'password';
GRANT ALL ON vpemp TO salesmgr;
```

EDB Postgres Advanced Server (EPAS)  
Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.  
861
The `ADD_POLICY` procedure creates a new policy by associating a policy function with a database object.

You must be a superuser to execute this procedure.

```sql
ADD_POLICY(<object_schema> VARCHAR2, <object_name> VARCHAR2,
            <policy_name> VARCHAR2, <function_schema> VARCHAR2,
            <policy_function> VARCHAR2
            [, <statement_types> VARCHAR2
            [, <update_check> BOOLEAN
            [, <enable> BOOLEAN
            [, <static_policy> BOOLEAN
            [, <policy_type> INTEGER
            [, <long_predicate> BOOLEAN
            [, <sec_relevant_cols> VARCHAR2
            [, <sec_relevant_cols_opt> INTEGER ]]]]]])
```

### Parameters

- **object_schema**
  - Name of the schema containing the database object to which to apply the policy.

- **object_name**
  - Name of the database object to which to apply the policy. A given database object can have more than one policy applied to it.

- **policy_name**
  - Name assigned to the policy. The combination of database object (identified by `object_schema` and `object_name`) and policy name must be unique in the database.

- **function_schema**
  - Name of the schema containing the policy function.

  **Note:** The policy function might belong to a package. In this case `function_schema` must contain the name of the schema in which the package is defined.

- **policy_function**
  - Name of the SPL function that defines the rules of the security policy. You can specify the same function in more than one policy.

  **Note:** The policy function might belong to a package. In this case `policy_function` must also contain the package name in dot notation (that is, `package_name.function_name`).

- **statement_types**
  - Comma-separated list of SQL commands to which the policy applies. Valid SQL commands are `INSERT`, `UPDATE`, `DELETE`, and `SELECT`. The default is `INSERT,UPDATE,DELETE,SELECT`.

  **Note:** EDB Postgres Advanced Server accepts `INDEX` as a statement type but it is ignored. Policies aren't applied to index operations in EDB Postgres Advanced Server.

- **update_check**
  - Applies to `INSERT` and `UPDATE` SQL commands only.
    - When set to `TRUE`, the policy is applied to newly inserted rows and to the modified image of updated rows. If any of the new or modified rows don't qualify according to the policy function predicate, then the `INSERT` or `UPDATE` command throws an exception and no rows are inserted or modified by the `INSERT` or `UPDATE` command.
    - When set to `FALSE`, the policy isn't applied to newly inserted rows or the modified image of updated rows. Thus, a newly inserted row might not appear in the result set of a subsequent SQL command that invokes the same policy. Similarly, rows that qualified according to the policy prior to an `UPDATE` command might not appear in the result set of a subsequent SQL command that invokes the same policy.
    - The default is `FALSE`.

- **enable**
  - When set to `TRUE`, the policy is enabled and applied to the SQL commands given by the `statement_types` parameter. When set to `FALSE`, the policy is disabled and not applied to any SQL commands. You can enable the policy using the `ENABLE_POLICY` procedure. The default is `TRUE`.

- **static_policy**
  - In Oracle, when set to `TRUE`, the policy is static, which means the policy function is evaluated once per database object the first time it's invoked by a policy on that database object. The resulting policy function predicate string is saved in memory and reused for all invocations of that policy on that database object while the database server instance is running.
    - When set to `FALSE`, the policy is dynamic, which means the policy function is reevaluated and the policy function predicate string regenerated for all invocations of the policy.

---

```
Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
```

862
The default is FALSE.

In Oracle 10g, the policy_type parameter was introduced, which is intended to replace the static_policy parameter. In Oracle, if the policy_type parameter isn't set to its default value of NULL, the policy_type parameter setting overrides the static_policy setting.

The setting of static_policy is ignored by EDB Postgres Advanced Server. EDB Postgres Advanced Server implements only the dynamic policy, regardless of the setting of the static_policy parameter.

The default is NULL. The policy_type parameter determines when the policy function is reevaluated and, hence, if and when the predicate string returned by the policy function changes. The default is NULL.

This parameter setting is ignored by EDB Postgres Advanced Server. The policy_type setting is ignored by EDB Postgres Advanced Server. EDB Postgres Advanced Server always assumes a dynamic policy.

In Oracle, allows predicates up to 32K bytes if set to TRUE. Otherwise predicates are limited to 4000 bytes. The default is FALSE.

This parameter setting is ignored by EDB Postgres Advanced Server. An EDB Postgres Advanced Server policy function can return a predicate of unlimited length for all practical purposes.

Comma-separated list of columns of object_name. Provides column-level Virtual Private Database for the listed columns. The policy is enforced if any of the listed columns are referenced in a SQL command of a type listed in statement_types. The policy isn't enforced if no such columns are referenced.

The default is NULL, which has the same effect as if all of the database object's columns were included in sec_relevant_cols.

In Oracle, if sec_relevant_cols_opt is set to DBMS_RLS.ALL_ROWS (INTEGER constant of value 1), then the columns listed in sec_relevant_cols return NULL on all rows where the applied policy predicate is false. (If sec_relevant_cols_opt isn't set to DBMS_RLS.ALL_ROWS, these rows aren't returned at all in the result set.) The default is NULL.

Note EDB Postgres Advanced Server doesn't support DBMS_RLS.ALL_ROWS. EDB Postgres Advanced Server throws an error if sec_relevant_cols_opt is set to DBMS_RLS.ALL_ROWS (INTEGER value of 1).

This example uses the following policy function:

```sql
CREATE OR REPLACE FUNCTION verify_session_user (
  p_schema VARCHAR2,
  p_object VARCHAR2
) RETURN VARCHAR2 IS BEGIN RETURN 'authid = SYS_CONTEXT(''USERENV'',''
SESSION_USER'');
END;
```

This function generates the predicate authid = SYS_CONTEXT(''USERENV'',''
SESSION_USER''), which is added to the WHERE clause of any SQL command of the type specified in the ADD POLICY procedure.

This limits the effect of the SQL command to those rows where the content of the authid column is the same as the session user.

Note

This example uses the SYS_CONTEXT function to return the login user name. In Oracle the SYS_CONTEXT function is used to return attributes of an application context. The first parameter of the SYS_CONTEXT function is the name of an application context. The second parameter is the name of an attribute set in the application context. USERENV is a special built-in namespace that describes the current session. EDB Postgres Advanced Server doesn't support application contexts. It supports only this specific usage of the SYS_CONTEXT function.

The following anonymous block calls the ADD POLICY procedure to create a policy named secure_update. The policy applies to the vpemp table using the function verify_session_user whenever an INSERT, UPDATE, or DELETE SQL command is given referencing the vpemp table.
After successfully creating the policy, a terminal session is started by user salesmgr. The following query shows the content of the vpemp table:

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
<th>authid</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>2450.00</td>
<td></td>
<td>10</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>5000.00</td>
<td></td>
<td>10</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>1300.00</td>
<td></td>
<td>10</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>800.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>2975.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>1100.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>3000.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>1600.00</td>
<td></td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>1250.00</td>
<td>300.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7598</td>
<td>BLAKE</td>
<td>SALESMAN</td>
<td>1250.00</td>
<td></td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>1500.00</td>
<td>0.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>950.00</td>
<td></td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>(14 rows)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

An unqualified UPDATE command (no WHERE clause) is issued by the salesmgr user:

```
edb> UPDATE vpemp SET comm = sal * 0.75;
6 rows updated
```

Instead of updating all rows in the table, the policy restricts the effect of the update to only those rows where the authid column contains the value salesmgr, as specified by the policy function predicate authid = SYS_CONTEXT('USERENV', 'SESSION_USER').

The following query shows that the comm column was changed only for those rows where authid contains salesmgr. All other rows are unchanged.

```
edb> SELECT * FROM vpemp;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
<th>authid</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>2450.00</td>
<td></td>
<td>10</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>5000.00</td>
<td></td>
<td>10</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>1300.00</td>
<td></td>
<td>10</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>800.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>2975.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>1100.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>3000.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>(14 rows)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Furthermore, since the update_check parameter was set to TRUE in the ADD_POLICY procedure, the following INSERT command throws an exception. The value given for the authid column, researchmgr, doesn't match the session user (salesmgr) and hence fails the policy.

```
EDB Postgres Advanced Server (EPAS)
Edition: Enterprise
Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
865
```
If the query references the `sal` or `comm` columns, then the policy is applied to the query, eliminating any rows where `sal` is greater than or equal to 2000:

```sql
SELECT empno, ename, job, sal, comm, deptno, authid FROM vpemp;
```

### DROP_POLICY

The `DROP_POLICY` procedure deletes an existing policy. The `DROP_POLICY` procedure doesn't delete the policy function and database object associated with the policy.

You must be a superuser to execute this procedure.

```sql
DROP_POLICY(<object_schema> VARCHAR2, <object_name> VARCHAR2, <policy_name> VARCHAR2);
```

#### Parameters

- **object_schema**
  
  Name of the schema containing the database object to which the policy applies.

- **object_name**
  
  Name of the database object to which the policy applies.

- **policy_name**
  
  Name of the policy to delete.

#### Examples

This example deletes policy `secure_update` on table `public.vpemp`:

```sql
DECLARE
  v_object_schema VARCHAR2(30) := 'public';
  v_object_name VARCHAR2(30) := 'vpemp';
  v_policy_name VARCHAR2(30) := 'secure_update';
BEGIN
  DBMS_RLS.DROP_POLICY(
    v_object_schema,
    v_object_name,
    v_policy_name
  );
END;
```
**ENABLE_POLICY**

The `ENABLE_POLICY` procedure enables or disables an existing policy on the specified database object.

You must be a superuser to execute this procedure.

```sql
ENABLE_POLICY(<object_schema> VARCHAR2, <object_name> VARCHAR2, <policy_name> VARCHAR2, <enable> BOOLEAN)
```

**Parameters**

- `object_schema`:
  Name of the schema containing the database object to which the policy applies.

- `object_name`:
  Name of the database object to which the policy applies.

- `policy_name`:
  Name of the policy to enable or disable.

- `enable`:
  When set to `TRUE`, the policy is enabled. When set to `FALSE`, the policy is disabled.

**Examples**

This example disables policy `secure_update` on table `public.vpemp`:

```sql
DECLARE
  v_object_schema VARCHAR2(30) := 'public';
  v_object_name VARCHAR2(30) := 'vpemp';
  v_policy_name VARCHAR2(30) := 'secure_update';
  v_enable BOOLEAN := FALSE;
BEGIN
  DBMS_RLS.ENABLE_POLICY(v_object_schema,
                           v_object_name,
                           v_policy_name,
                           v_enable);
END;
```

**14.3.3.16 DBMS_SCHEDULER**

The `DBMS_SCHEDULER` package's jobs are scheduled and run in the background by the `edb_jobScheduler` extension. For more information, see EDB Job Scheduler extension.

The `DBMS_SCHEDULER` package provides a way to create and manage Oracle-style jobs, programs, and job schedules.

The `DBMS_SCHEDULER` package implements the following functions and procedures:

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CREATE_JOB(job_name, job_type, job_action, number_of_arguments, start_date, repeat_interval, end_date, job_class, enabled, auto_drop, comments)</code></td>
<td>n/a</td>
<td>Use the first form of the <code>CREATE_JOB</code> procedure to create a job, specifying program and schedule details by means of parameters.</td>
</tr>
<tr>
<td><code>CREATE_JOB(job_name, program_name, schedule_name, job_class, enabled, auto_drop, comments)</code></td>
<td>n/a</td>
<td>Use the second form of <code>CREATE_JOB</code> to create a job that uses a named program and named schedule.</td>
</tr>
</tbody>
</table>
Token type  | Syntax | Valid values
----------|--------|-------------
FREQ      | FREQ=presdefined_interv\al | Where predefined_interval is one of the following: YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY. The SECONDLY keyword isn't supported.
BYMONTH   | BYMONTH=month(, month)\... | Where month is the three-letter abbreviation of the month name: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
BYMONTH   | BYMONTH=month(, month)\... | Where month is the numeric value representing the month: 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12
BYMONTHDAYS | BYMONTHDAY=day_of_month | Where day_of_month is a value from 1 through 31
BYDAY     | BYDAY=weekday | Where weekday is a three-letter abbreviation or single-digit value representing the day of the week.

EDB Postgres Advanced Server supports the token types and syntax described in the table.
14.3.3.1.16.2  CREATE_JOB

Use the CREATE_JOB procedure to create a job. The procedure comes in two forms. The first form of the procedure specifies a schedule in the job definition as well as a job action to invoke when the job executes:

```
CREATE_JOB(
    <job_name> IN VARCHAR2,
    <job_type> IN VARCHAR2,
    <job_action> IN VARCHAR2,
    <number_of_arguments> IN PL SQL_INTEGER DEFAULT 0,
    <start_date> IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
    <repeat_interval> IN VARCHAR2 DEFAULT NULL,
    <end_date> IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
    <job_class> IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
    <enabled> IN BOOLEAN DEFAULT FALSE,
    <auto_drop> IN BOOLEAN DEFAULT TRUE,
    <comments> IN VARCHAR2 DEFAULT NULL)
```

The second form uses a job schedule to specify the schedule on which the job executes and specifies the name of a program to execute when the job runs:

```
CREATE_JOB(
    <job_name> IN VARCHAR2,
    <program_name> IN VARCHAR2,
    <schedule_name> IN VARCHAR2,
    <job_class> IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
    <enabled> IN BOOLEAN DEFAULT FALSE,
    <auto_drop> IN BOOLEAN DEFAULT TRUE,
    <comments> IN VARCHAR2 DEFAULT NULL)
```

Parameters

**job_name**

job_name specifies the optionally schema-qualified name of the job being created.

**job_type**

job_type specifies the type of job. The current implementation of CREATE_JOB supports a job type of PL/SQL_BLOCK or STORED_PROCEDURE.

**job_action**

- if job_type is PL/SQL_BLOCK, job_action specifies the content of the PL/SQL block to invoke when the job executes. The block must be terminated with a semi-colon (;).
- if job_type is STORED_PROCEDURE, job_action specifies the optionally schema-qualified name of the procedure.

**number_of_arguments**

number_of_arguments is an integer value that specifies the number of arguments expected by the job. The default is 0.

**start_date**

start_date is a TIMESTAMP WITH TIME ZONE value that specifies the first time that the job is scheduled to execute. The default value is NULL, indicating to schedule the job to execute when the job is enabled.
**repeat_interval**

*repeat_interval* is a `VARCHAR2` value that specifies how often the job repeats. If you don’t specify a `repeat_interval`, the job executes only once. The default value is `NULL`.

**end_date**

*end_date* is a `TIMESTAMP WITH TIME ZONE` value that specifies a time after which the job no longer executes. If you specify a date, the *end_date* must be after *start_date*. The default value is `NULL`.

If you don’t specify an *end_date* and you do specify a `repeat_interval`, the job repeats indefinitely until you disable it.

**program_name**

*program_name* is the name of a program for the job to execute.

**schedule_name**

*schedule_name* is the name of the schedule associated with the job.

**job_class**

*job_class* is accepted for compatibility and ignored.

**enabled**

*enabled* is a Boolean value that specifies if the job is enabled when created. By default, a job is created in a disabled state, with `enabled` set to `FALSE`. To enable a job, specify a value of `TRUE` when creating the job, or enable the job with the `DBMS_SCHEDULER.ENABLE` procedure.

**auto_drop**

The `auto_drop` parameter is accepted for compatibility and is ignored. By default, a job’s status is changed to `DISABLED` after the time specified in *end_date*.

**comments**

Use the `comments` parameter to specify a comment about the job.

**Example**

This example shows a call to the `CREATE_JOB` procedure:

```sql
EXEC
  DBMS_SCHEDULER.CREATE_JOB
  (job_name => 'update_log',
   job_type => 'PLSQL_BLOCK',
   job_action => 'BEGIN INSERT INTO my_log
VALUES(current_timestamp);
END;',
   start_date => '01-JUN-15 09:00:00.000000',
   repeat_interval => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU, FRI;BYHOUR=17;',
   end_date => NULL,
   enabled => TRUE,
   comments => 'This job adds a row to the my_log table.');
```

The code fragment creates a job named *update_log* that executes each weeknight at 5:00. The job executes a PL/SQL block that inserts the current timestamp into a logfile (*my_log*). Since no *end_date* is specified, the job executes until disabled by the `DBMS_SCHEDULER.DISABLE` procedure.

**14.3.1.16.3 CREATE_PROGRAM**

Use the `CREATE_PROGRAM` procedure to create a `DBMS_SCHEDULER` program. The signature is:

```sql
CREATE_PROGRAM(
  <program_name> IN VARCHAR2,
```
Parameters

**program_name**

Specifies the name of the program that's being created.

**program_type**

Specifies the type of program. The current implementation of `CREATE_PROGRAM` supports a `program_type` of `PLSQL_BLOCK` or `PROCEDURE`.

**program_action**

- If `program_type` is `PLSQL_BLOCK`, `program_action` contains the PL/SQL block that executes when the program is invoked. The PL/SQL block must be terminated with a semi-colon (`;`).
- If `program_type` is `PROCEDURE`, `program_action` contains the name of the stored procedure.

**number_of_arguments**

- If `program_type` is `PLSQL_BLOCK`, this argument is ignored.
- If `program_type` is `PROCEDURE`, `number_of_arguments` specifies the number of arguments required by the procedure. The default value is 0.

**enabled**

Specifies if the program is created enabled or disabled:

- If `enabled` is `TRUE`, the program is created enabled.
- If `enabled` is `FALSE`, the program is created disabled. Use the `DBMS_SCHEDULER.ENABLE` program to enable a disabled program.

The default value is `FALSE`.

**comments**

Use the `comments` parameter to specify a comment about the program. By default, this parameter is `NULL`.

Example

The following call to the `CREATE_PROGRAM` procedure creates a program named `update_log`:

```sql
EXEC DBMS_SCHEDULER.CREATE_PROGRAM
(
  program_name => 'update_log',
  program_type => 'PLSQL_BLOCK',
  program_action => 'BEGIN INSERT INTO my_log
VALUES(current_timestamp);
  END;',
  enabled => TRUE,
  comment => 'This program adds a row to the my_log table.');
```

`update_log` is a PL/SQL block that adds a row containing the current date and time to the `my_log` table. The program is enabled when the `CREATE_PROGRAM` procedure executes.

14.3.3.16.4 CREATE_SCHEDULER

Use the `CREATE_SCHEDULER` procedure to create a job schedule. The signature of the `CREATE_SCHEDULER` procedure is:
CREATE_SCHEDULE(
  <schedule_name> IN VARCHAR2,
  <start_date> IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
  <repeat_interval> IN VARCHAR2,
  <end_date> IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
  <comments> IN VARCHAR2 DEFAULT NULL)

Parameters

schedule_name

schedule_name specifies the name of the schedule.

start_date

start_date is a TIMESTAMP WITH TIME ZONE value that specifies the date and time that the schedule is eligible to execute. If you don't specify a start_date, the date that the job is enabled is used as the start_date. By default, start_date is NULL.

repeat_interval

repeat_interval is a VARCHAR2 value that specifies how often the job repeats. If you don't specify a repeat_interval, the job executes only once, on the date specified by start_date.!!! Note You must provide a value for either start_date or repeat_interval. If both start_date and repeat_interval are NULL, the server returns an error.

end_date

end_date is a TIMESTAMP WITH TIME ZONE value that specifies a time after which the schedule no longer executes. If you specify a date, the end_date must be after the start_date. The default value is NULL.!!! Note If you specify a repeat_interval and don't specify an end_date, the schedule repeats indefinitely until you disable it.

comments

Use the comments parameter to specify a comment about the schedule. By default, this parameter is NULL.

Example

This code fragment calls CREATE_SCHEDULE to create a schedule named weeknights_at_5:

EXEC
  DBMS_SCHEDULER.CREATE_SCHEDULE
  (
    schedule_name => 'weeknights_at_5',
    start_date => '01-JUN-13 09:00:00.000000',
    repeat_interval => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',
    comments => 'This schedule executes each weeknight at 5:00');

The schedule executes each weeknight, at 5:00, effective after June 1, 2013. Since no end_date is specified, the schedule executes indefinitely until disabled with DBMS_SCHEDULER.DISABLE.

14.3.3.1.16.5 DEFINE_PROGRAM_ARGUMENT

Use the DEFINE_PROGRAM_ARGUMENT procedure to define a program argument. The DEFINE_PROGRAM_ARGUMENT procedure comes in two forms. The first form defines an argument with a default value:

DEFINE_PROGRAM_ARGUMENT(
  <program_name> IN VARCHAR2,
  <argument_position> IN PLS_INTEGER,
  <argument_name> IN VARCHAR2 DEFAULT NULL,
  <argument_type> IN VARCHAR2,
  <default_value> IN VARCHAR2,
  <out_argument> IN BOOLEAN DEFAULT FALSE)

The second form defines an argument without a default value:
Parameters

**program_name**

*program_name* is the name of the program to which the arguments belong.

**argument_position**

*argument_position* specifies the position of the argument as it's passed to the program.

**argument_name**

*argument_name* specifies the optional name of the argument. By default, *argument_name* is *NULL*.

**argument_type** in *VARCHAR2*

*argument_type* specifies the data type of the argument.

**default_value**

*default_value* specifies the default value assigned to the argument. *default_value* is overridden by a value specified by the job when the job executes.

**out_argument** in *BOOLEAN DEFAULT FALSE*

*out_argument* isn't currently used. If specified, the value must be *FALSE*.

Example

This code fragment uses the **DEFINE_PROGRAM_ARGUMENT** procedure to define the first and second arguments in a program named *add_emp*:

```sql
EXEC DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
  program_name => 'add_emp',
  argument_position => 1,
  argument_name => 'dept_no',
  argument_type => 'INTEGER',
  default_value => '20');
EXEC DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
  program_name => 'add_emp',
  argument_position => 2,
  argument_name => 'emp_name',
  argument_type => 'VARCHAR2');
```

The first argument is an *INTEGER* value named *dept_no* that has a default value of *20*. The second argument is a *VARCHAR2* value named *emp_name* and doesn’t have a default value.

14.3.3.1.16.6 **DISABLE**

Use the **DISABLE** procedure to disable a program or a job. The signature of the **DISABLE** procedure is:

```sql
DISABLE(
  <name> IN VARCHAR2,
  <force> IN BOOLEAN DEFAULT FALSE,
)```
Parameters

name

name specifies the name of the program or job that's being disabled.

force

force is accepted for compatibility and ignored.

commit_semantics

commit_semantics tells the server how to handle an error encountered while disabling a program or job. By default, commit_semantics is set to STOP_ON_FIRST_ERROR, which means to stop when an error is encountered. Any programs or jobs that were successfully disabled before the error are committed to disk.

The TRANSACTIONAL and ABSORB_ERRORS keywords are accepted for compatibility and ignored.

Example

The following call to the DISABLE procedure disables a program named update_emp:

```sql
DBMS_SCHEDULER.DISABLE('update_emp');
```

14.3.3.1.16.7 DROP_JOB

Use the DROP_JOB procedure to drop a job, drop any arguments that belong to the job, and eliminate any future job executions. The signature of the procedure is:

```
DROP_JOB(
    <job_name> IN VARCHAR2,
    <force> IN BOOLEAN DEFAULT FALSE,
    <defer> IN BOOLEAN DEFAULT FALSE,
    <commit_semantics> IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```

Parameters

job_name

job_name specifies the name of the job that's being dropped.

force

force is accepted for compatibility and ignored.

defer

defer is accepted for compatibility and ignored.

commit_semantics

commit_semantics tells the server how to handle an error encountered while dropping a program or job. By default, commit_semantics is set to STOP_ON_FIRST_ERROR, which means to stop when an error is encountered.

The TRANSACTIONAL and ABSORB_ERRORS keywords are accepted for compatibility and ignored.

Example


The following call to **DROP_JOB** drops a job named `update_log`:

```
DBMS_SCHEDULER.DROP_JOB('update_log');
```

### 14.3.3.16.8 DROP_PROGRAM

The **DROP_PROGRAM** procedure drops a program. The signature of the **DROP_PROGRAM** procedure is:

```sql
DROP_PROGRAM(
    <program_name> IN VARCHAR2,
    <force> IN BOOLEAN DEFAULT FALSE)
```

**Parameters**

- **program_name**
  - Specifies the name of the program that's being dropped.

- **force**
  - `force` is a Boolean value that tells the server how to handle programs with dependent jobs.
    - Specify `FALSE` to return an error if the program is referenced by a job.
    - Specify `TRUE` to disable any jobs that reference the program before dropping the program.
    - The default value is `FALSE`.

**Example**

The following call to **DROP_PROGRAM** drops a job named `update_emp`:

```
DBMS_SCHEDULER.DROP_PROGRAM('update_emp');
```

### 14.3.3.16.9 DROP_PROGRAM_ARGUMENT

Use the **DROP_PROGRAM_ARGUMENT** procedure to drop a program argument. The **DROP_PROGRAM_ARGUMENT** procedure comes in two forms. The first form uses an argument position to specify the argument to drop:

```sql
DROP_PROGRAM_ARGUMENT(
    <program_name> IN VARCHAR2,
    <argument_position> IN PLS_INTEGER)
```

The second form takes the argument name:

```sql
DROP_PROGRAM_ARGUMENT(
    <program_name> IN VARCHAR2,
    <argument_name> IN VARCHAR2)
```

**Parameters**

- **program_name**
  - Specifies the name of the program that's being modified.

- **argument_position**
  - Specifies the position of the argument to drop.

- **argument_name**
  - Specifies the name of the argument to drop.
**DROP_PROGRAM_ARGUMENT**

*argument_position* specifies the position of the argument that's being dropped.

*argument_name* specifies the name of the argument that's being dropped.

**Examples**

The following call to `DROP_PROGRAM_ARGUMENT` drops the first argument in the `update_emp` program:

```sql
DBMS_SCHEDULER.DROP_PROGRAM_ARGUMENT('update_emp', 1);
```

The following call to `DROP_PROGRAM_ARGUMENT` drops an argument named `emp_name`:

```sql
DBMS_SCHEDULER.DROP_PROGRAM_ARGUMENT('update_emp', 'emp_name');
```

### 14.3.3.16.10 DROP_SCHEDULE

Use the `DROP_SCHEDULE` procedure to drop a schedule. The signature is:

```sql
DROP_SCHEDULE(
    <schedule_name> IN VARCHAR2,
    <force> IN BOOLEAN DEFAULT FALSE)
```

**Parameters**

- **schedule_name** specifies the name of the schedule that's being dropped.

- **force** specifies the behavior of the server if the specified schedule is referenced by any job:
  
  - Specify `FALSE` to return an error if the specified schedule is referenced by a job. This is the default behavior.
  - Specify `TRUE` to disable any jobs that use the specified schedule before dropping the schedule. Any running jobs are allowed to complete before the schedule is dropped.

**Example**

The following call to `DROP_SCHEDULE` drops a schedule named `weeknights_at_5`:

```sql
DBMS_SCHEDULER.DROP_SCHEDULE('weeknights_at_5', TRUE);
```

The server disables any jobs that use the schedule before dropping the schedule.

### 14.3.3.16.11 ENABLE

Use the `ENABLE` procedure to enable a disabled program or job.

The signature of the `ENABLE` procedure is:

```sql
ENABLE(
    <name> IN VARCHAR2,
    <commit_semantics> IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```
Parameters

name
	names the name of the program or job that's being enabled.

commit_semantics

tells the server how to handle an error encountered while enabling a program or job. By default, commit_semantics is set to STOP_ON_FIRST_ERROR, telling the server to stop when it encounters an error.

The TRANSACTIONAL and ABSORB_ERRORS keywords are accepted for compatibility and ignored.

Example

The following call to DBMS_SCHEDULER.ENABLE enables the update_emp program:

```
DBMS_SCHEDULER.ENABLE('update_emp');
```

14.3.3.1.16.12 EVALUATE_CALENDAR_STRING

Use the EVALUATE_CALENDAR_STRING procedure to evaluate the repeat_interval value specified when creating a schedule with the CREATE_SCHEDULE procedure. The EVALUATE_CALENDAR_STRING procedure returns the date and time that a specified schedule executes without actually scheduling the job.

The signature of the EVALUATE_CALENDAR_STRING procedure is:

```
EVALUATE_CALENDAR_STRING(
  <calendar_string> IN VARCHAR2,
  <start_date> IN TIMESTAMP WITH TIME ZONE,
  <return_date_after> IN TIMESTAMP WITH TIME ZONE,
  <next_run_date> OUT TIMESTAMP WITH TIME ZONE)
```

Parameters

calendar_string

calendar_string is the calendar string that describes a repeat_interval that's being evaluated.

start_date IN TIMESTAMP WITH TIME ZONE

start_date is the date and time after which the repeat_interval becomes valid.

return_date_after

Use the return_date_after parameter to specify the date and time for EVALUATE_CALENDAR_STRING to use as a starting date when evaluating the repeat_interval.

For example, if you specify a return_date_after value of 01-APR-13 09.00.00.000000, EVALUATE_CALENDAR_STRING returns the date and time of the first iteration of the schedule after April 1st, 2013.

next_run_date OUT TIMESTAMP WITH TIME ZONE

next_run_date is an OUT parameter that contains the first occurrence of the schedule after the date specified by the return_date_after parameter.

Example

This example evaluates a calendar string and returns the first date and time that the schedule will execute after June 15, 2013:

```
DECLARE
```

```
Because June 15, 2013 is a Saturday, the schedule will execute on Monday, June 17, 2013 at 5:00 pm.

14.3.3.16.13  

**RUN_JOB**

Use the **RUN_JOB** procedure to execute a job immediately. The signature of the **RUN_JOB** procedure is:

```sql
RUN_JOB(
    <job_name> IN VARCHAR2,
    <use_current_session> IN BOOLEAN DEFAULT TRUE
);
```

**Parameters**

**job_name**

**job_name** specifies the name of the job to execute.

**use_current_session**

By default, the job executes in the current session. If specified, **use_current_session** must be set to **TRUE**. If **use_current_session** is set to **FALSE**, EDB Postgres Advanced Server returns an error.

**Example**

The following call to **RUN_JOB** executes a job named **update_log**:

```sql
DBMS_SCHEDULER.RUN_JOB('update_log', TRUE);
```

Passing a value of **TRUE** as the second argument instructs the server to invoke the job in the current session.

14.3.3.16.14  

**SET_JOB_ARGUMENT_VALUE**

Use the **SET_JOB_ARGUMENT_VALUE** procedure to specify a value for an argument. The **SET_JOB_ARGUMENT_VALUE** procedure comes in two forms. The first form specifies the argument to modify by position:

```sql
SET_JOB_ARGUMENT_VALUE(
    <job_name> IN VARCHAR2,
    <argument_position> IN PLS_INTEGER,
    <argument_value> IN VARCHAR2)
```

The second form uses an argument name to specify the argument to modify:

```sql
SET_JOB_ARGUMENT_VALUE(
    <job_name> IN VARCHAR2,
    <argument_name> IN VARCHAR2,
    <argument_value> IN VARCHAR2)
```
Argument values set by the `SET_JOB_ARGUMENT_VALUE` procedure override any values set by default.

**Parameters**

*job_name*

Specifies the name of the job to which the modified argument belongs.

*argument_position*

Use `argument_position` to specify the argument position for which the value is set.

*argument_name*

Use `argument_name` to specify the argument by name for which the value is set.

*argument_value*

`argument_value` specifies the new value of the argument.

**Examples**

This example assigns a value of `30` to the first argument in the `update_emp` job:

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE('update_emp', 1, '30');
```

This example sets the `emp_name` argument to `SMITH`:

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE('update_emp', 'emp_name', 'SMITH');
```

### 14.3.3.1.17 DBMS_SESSION

EDB Postgres Advanced Server provides support for the following `DBMS_SESSION.SET_ROLE` procedure:

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SET_ROLE(role_cmd)</code></td>
<td>n/a</td>
<td>Executes a <code>SET ROLE</code> statement followed by the string value specified in <code>role_cmd</code>.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server's implementation of `DBMS_SESSION` is a partial implementation when compared to Oracle's version. Only `DBMS_SESSION.SET_ROLE` is supported.

**SET_ROLE**

The `SET_ROLE` procedure sets the current session user to the role specified in `role_cmd`. After invoking the `SET_ROLE` procedure, the current session uses the permissions assigned to the specified role. The signature of the procedure is:

```
SET_ROLE(<role_cmd>)
```

The `SET_ROLE` procedure appends the value specified for `role_cmd` to the `SET ROLE` statement and then invokes the statement.

**Parameters**

*role_cmd*

`role_cmd` specifies a role name in the form of a string value.

**Example**
This call to the `SET_ROLE` procedure invokes the `SET ROLE` command to set the identity of the current session user to manager:

```
edb# exec DBMS_SESSION.SET_ROLE('manager');
```

### 14.3.3.18 DBMS_SQL

The `DBMS_SQL` package provides an application interface compatible with Oracle databases to the EDB dynamic SQL functionality. With `DBMS_SQL`, you can construct queries and other commands at runtime rather than when you write the application. EDB Postgres Advanced Server offers native support for dynamic SQL. `DBMS_SQL` provides a way to use dynamic SQL in a way that's compatible with Oracle databases without modifying your application.

`DBMS_SQL` assumes the privileges of the current user when executing dynamic SQL statements.

EDB Postgres Advanced Server’s implementation of `DBMS_SQL` is a partial implementation when compared to Oracle’s version. Only those functions and procedures listed in the table are supported.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Function or procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIND_ARRAY(c IN, name IN, table_variable IN [index1 IN, index2 IN])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Binds a value or set of values to a variable.</td>
</tr>
<tr>
<td>BIND_VARIABLE(c, name, value [, out_value_size ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Bind a value to a variable.</td>
</tr>
<tr>
<td>BIND_VARIABLE_CHAR(c, name, value [, out_value_size ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Bind a <code>CHAR</code> value to a variable.</td>
</tr>
<tr>
<td>BIND_VARIABLE_RAW(c, name, value [, out_value_size ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Bind a <code>RAW</code> value to a variable.</td>
</tr>
<tr>
<td>CLOSE_CURSOR(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Close a cursor.</td>
</tr>
<tr>
<td>COLUMN_VALUE(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Return a column value into a variable.</td>
</tr>
<tr>
<td>COLUMN_VALUE_CHAR(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Return a <code>CHAR</code> column value into a variable.</td>
</tr>
<tr>
<td>COLUMN_VALUE_RAW(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Return a <code>RAW</code> column value into a variable.</td>
</tr>
<tr>
<td>COLUMN_VALUE_LONG(c, position, length, offset, value OUT, value_length OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Return a part of the <code>LONG</code> column value into a variable.</td>
</tr>
<tr>
<td>COLUMN_VALUE_ROWID(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Return a <code>ROWID</code> column in a cursor.</td>
</tr>
<tr>
<td>DEFINE_ARRAY(c IN, position IN, table_variable IN, cnt IN, lower_bnd IN)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define collection for column into which to fetch rows.</td>
</tr>
<tr>
<td>DEFINE_COLUMN(c, position, column [, column_size ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define a column in the <code>SELECT</code> list.</td>
</tr>
<tr>
<td>DEFINE_COLUMN_CHAR(c, position, column, column_size)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define a <code>CHAR</code> column in the <code>SELECT</code> list.</td>
</tr>
<tr>
<td>DEFINE_COLUMN_RAW(c, position, column, column_size)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define a <code>RAW</code> column in the <code>SELECT</code> list.</td>
</tr>
<tr>
<td>DEFINE_COLUMN_LONG(c, position)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define a <code>LONG</code> column in the <code>SELECT</code> list.</td>
</tr>
<tr>
<td>DEFINE_COLUMN_ROWID(c IN, position IN, column IN)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define a <code>ROWID</code> column in the <code>SELECT</code> list.</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS(c IN, col_cnt OUT, desc_t OUT, DESC_TAB)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define columns to hold a cursor result set.</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS2(c IN, col_cnt OUT, desc_t OUT, DESC_TAB)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define columns to hold a cursor result set.</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS3(c IN, col_cnt OUT, desc_t OUT, DESC_TAB)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define columns to hold a cursor result set.</td>
</tr>
<tr>
<td>EXECUTE(c)</td>
<td>Function</td>
<td>INTEGER</td>
<td>Execute a cursor.</td>
</tr>
<tr>
<td>EXECUTE_AND_FETCH(c [, exact ])</td>
<td>Function</td>
<td>INTEGER</td>
<td>Execute a cursor and fetch a single raw.</td>
</tr>
<tr>
<td>FETCH_ROWS(c)</td>
<td>Function</td>
<td>INTEGER</td>
<td>Fetch rows from the cursor.</td>
</tr>
<tr>
<td>IS_OPEN(c)</td>
<td>Function</td>
<td>BOOLEAN</td>
<td>Check if a cursor is open.</td>
</tr>
<tr>
<td>LAST_ROW_COUNT</td>
<td>Function</td>
<td>INTEGER</td>
<td>Return cumulative number of rows fetched.</td>
</tr>
<tr>
<td>LAST_ERROR_POSITION</td>
<td>Function</td>
<td>INTEGER</td>
<td>Return byte offset in the SQL statement text where the error occurred.</td>
</tr>
<tr>
<td>OPEN_CURSOR</td>
<td>Function</td>
<td>INTEGER</td>
<td>Open a cursor.</td>
</tr>
<tr>
<td>PARSE(c, statement, language_flag)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Parse a statement.</td>
</tr>
<tr>
<td>TO_CURSOR_NUMBER (cc IN OUT )</td>
<td>Function</td>
<td>INTEGER</td>
<td>Transform a ref cursor into a SQL cursor number.</td>
</tr>
<tr>
<td>TO_REFCURSOR (cursor_number IN OUT)</td>
<td>Function</td>
<td>SYS_REFCURSOR</td>
<td>Transform an open cursor into a REF CURSOR.</td>
</tr>
<tr>
<td>VARIABLE_VALUE</td>
<td>Function</td>
<td></td>
<td>Not supported in EPAS</td>
</tr>
</tbody>
</table>

The following table lists the public variables available in the `DBMS_SQL` package.

<table>
<thead>
<tr>
<th>Public variables</th>
<th>Data type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>edb=</td>
<td>=</td>
<td># =</td>
<td>exec</td>
</tr>
</tbody>
</table>
14.3.3.1.18.1  BIND_ARRAY

The **BIND_ARRAY** procedure binds a value or set of values to a variable in a cursor, based on the name of the variable.

**BIND_ARRAY** (\(<\>\) IN NUMBER, \(<\>\) IN VARCHAR2, 
\(<\>\) IN \{ FLOAT_TABLE | BLOB_TABLE | CLOB_TABLE | VARCHAR2_TABLE | NUMBER_TABLE | TIMESTAMP_TABLE | INTEGER_TABLE | ROWID_TABLE | TEXT_TABLE \} [,\(<\>\) IN NUMBER, \(<\>\) IN NUMBER]);

**Parameters**

**c**

Cursor ID of the cursor for the SQL command with the values to be bound.

**name**

Name of the collection. The name must be less than or equal to 30 bytes.

**table_variable**

A local variable that is declared to be one of the following datatypes:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bflt_tab&gt;</td>
<td>FLOAT_TABLE</td>
</tr>
<tr>
<td>&lt;bl_tab&gt;</td>
<td>BLOB_TABLE</td>
</tr>
<tr>
<td>&lt;cl_tab&gt;</td>
<td>CLOB_TABLE</td>
</tr>
<tr>
<td>&lt;c_tab&gt;</td>
<td>VARCHAR2_TABLE</td>
</tr>
<tr>
<td>&lt;n_tab&gt;</td>
<td>NUMBER_TABLE</td>
</tr>
<tr>
<td>&lt;tms_tab&gt;</td>
<td>TIMESTAMP_TABLE</td>
</tr>
<tr>
<td>&lt;int_tab&gt;</td>
<td>INTEGER_TABLE</td>
</tr>
<tr>
<td>&lt;rd_tab&gt;</td>
<td>ROWID_TABLE</td>
</tr>
<tr>
<td>&lt;txt_tab&gt;</td>
<td>TEXT_TABLE</td>
</tr>
</tbody>
</table>

**index1**

Index for the table element that marks the lower bound of the range if you are binding a range. This value must be less than or equal to the value of **index2**. All elements between **index1** and **index2** are used in the bind. This is an optional parameter that can be used with three parameters or five parameters but not four.

**index2**

Index for the table element that marks the upper bound of the range. This is an optional parameter that can be used with three parameters or five parameters but not four.

**Notes**

*If you don’t specify indexes in the bind call, and two different binds in a statement specify tables that contain a different number of elements, then the number of elements actually used is the minimum number between all tables. This is also the case if you specify indexes. The minimum range is selected between the two indexes for all tables.*

Not all bind variables in a query must be array binds. Some can be regular binds, and the same value is used for each element of the collections in expression evaluations and so forth.

**Example**
The **BIND_VARIABLE** procedure associates a value with an **IN** or **IN OUT** bind variable in a SQL command.

**BIND_VARIABLE**

```plaintext
BIND_VARIABLE(c<*> NUMBER, <name> VARCHAR2,
    <value> [ BLOB | CLOB | FLOAT | INTEGER | NUMBER | ROWID | TIMESTAMP | VARCHAR2 ]
    [, <out_value_size> NUMBER ]
```

### Parameters

- **c**
  - Cursor ID of the cursor for the SQL command with bind variables.

- **name**
  - Name of the bind variable in the SQL command.

- **value**
  - Value to be bound to the bind variable.

**Example**

```sql
CREATE TABLE tab_test (col_int INTEGER, col_charactervarying VARCHAR(40));

-- BIND_ARRAY (used for multiple insert/update/delete in single command
DECLARE
    c INTEGER;
    dbms_return INTEGER;
    res NUMBER;
    fetch_ret INTEGER;
    varint DBMS_SQL.INTEGER_TABLE;
BEGIN
    c := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE (c, 'INSERT into tab_test(col_int, col_charactervarying) values (:bvarint, :bvarname)
    ', DBMS_SQL.NATIVE);
    varint(1) := 10;
    varint(2) := 11;
    varint(4) := 12;
    DBMS_SQL.BIND_ARRAY (c, 'bvarint', varint);
    varint(1) := 10;
    varint(2) := 11;
    varint(4) := 12;
    DBMS_SQL.BIND_VARIABLE (c, 'bvarname', 'edb_user');
    dbms_return := DBMS_SQL.EXECUTE(c);
    DBMS_OUTPUT.PUT_LINE(' RETURN VALUE ...' || dbms_return);
    COMMIT;
    DBMS_SQL.CLOSE_CURSOR (c);
END;
/
```

This inserts three rows using bind_array so verify them in the table.

```sql
SELECT * FROM tab_test;
```

<table>
<thead>
<tr>
<th>col_int</th>
<th>col_charactervarying</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>edb_user</td>
</tr>
<tr>
<td>11</td>
<td>edb_user</td>
</tr>
<tr>
<td>12</td>
<td>edb_user</td>
</tr>
</tbody>
</table>

For **col_int** we are passing bind_array with the size of 3 elements, and as **col_charactervarying** we are passing bind_variable. So for all the insertions, we are using same bind_variable (**edb_user**).

(3 rows)
Value to be assigned.

\[ \text{out\_value\_size} \]

If \text{name} is an \text{IN OUT} variable, defines the maximum length of the output value. If not specified, the length of \text{value} is assumed.

### Examples

The following anonymous block uses bind variables to insert a row into the \text{emp} table.

```sql
DECLARE
  curid NUMBER;
  v_sql VARCHAR2(150) := 'INSERT INTO emp VALUES ('
  ':p_empno, :p_ename, :p_job, :p_mgr, '
  ':p_hiredate, :p_sal, :p_comm,
  ':p_deptno);
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
  v_job emp.job%TYPE;
  v_mgr emp.mgr%TYPE;
  v_hiredate emp.hiredate%TYPE;
  v_sal emp.sal%TYPE;
  v_comm emp.comm%TYPE;
  v_deptno emp.deptno%TYPE;
  v_status INTEGER;
BEGIN
  curid := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
  v_empno := 9001;
  v_ename := 'JONES';
  v_job := 'SALESMAN';
  v_mgr := 7369;
  v_hiredate := TO_DATE('13-DEC-07','DD-MON-YY');
  v_sal := 8500.00;
  v_comm := 1500.00;
  v_deptno := 40;
  DBMS_SQL.BIND_VARIABLE(curid,':p_empno',v_empno);
  DBMS_SQL.BIND_VARIABLE(curid,':p_ename',v_ename);
  DBMS_SQL.BIND_VARIABLE(curid,':p_job',v_job);
  DBMS_SQL.BIND_VARIABLE(curid,':p_mgr',v_mgr);
  DBMS_SQL.BIND_VARIABLE(curid,':p_hiredate',v_hiredate);
  DBMS_SQL.BIND_VARIABLE(curid,':p_sal',v_sal);
  DBMS_SQL.BIND_VARIABLE(curid,':p_comm',v_comm);
  DBMS_SQL.BIND_VARIABLE(curid,':p_deptno',v_deptno);
  v_status := DBMS_SQL.EXECUTE(curid);
  DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
END;
```

Number of rows processed: 1

### 14.3.3.18.3 \text{BIND\_VARIABLE\_CHAR}

The \text{BIND\_VARIABLE\_CHAR} procedure associates a \text{CHAR} value with an \text{IN} or \text{IN OUT} bind variable in a SQL command.

\text{BIND\_VARIABLE\_CHAR}(<c> \text{NUMBER, <name> VARCHAR2, <value> CHAR}
Parameters

\( c \)

Cursor ID of the cursor for the SQL command with bind variables.

\( \text{name} \)

Name of the bind variable in the SQL command.

\( \text{value} \)

Value of type \text{CHAR} to be assigned.

\( \text{out_value_size} \)

If \text{name} is an \text{IN OUT} variable, defines the maximum length of the output value. If not specified, the length of \text{value} is assumed.

14.3.3.1.18.4  BIND_VARIABLE_RAW

The \text{BIND_VARIABLE_RAW} procedure associates a \text{RAW} value with an \text{IN} or \text{IN OUT} bind variable in a SQL command.

\text{BIND_VARIABLE_RAW}(c NUMBER, \text{name} VARCHAR2, \text{value} RAW
[, \text{out_value_size} NUMBER])

Parameters

\( c \)

Cursor ID of the cursor for the SQL command with bind variables.

\( \text{name} \)

Name of the bind variable in the SQL command.

\( \text{value} \)

Value of type \text{RAW} to be assigned.

\( \text{out_value_size} \)

If \text{name} is an \text{IN OUT} variable, defines the maximum length of the output value. If not specified, the length of \text{value} is assumed.

14.3.3.18.5  CLOSE_CURSOR

The \text{CLOSE_CURSOR} procedure closes an open cursor. The resources allocated to the cursor are released and you can no longer use it.

Note

When you convert a cursor using \text{TO_REFCURSOR}, you cannot call \text{CLOSE_CURSOR} for that cursor.

\text{CLOSE_CURSOR}(c \text{ IN OUT NUMBER})

Parameters
Cursor ID of the cursor to be closed.

Examples

This example closes an open cursor:

```sql
DECLARE
    curid          NUMBER;
    v_sql           VARCHAR2(150);
    v_status        INTEGER;
BEGIN
    v_sql := 'INSERT INTO emp VALUES (9001, ''JONES'', ''SALESMAN'', 7369, TO_DATE(''13-DEC-87'', ''DD-MON-YY''), 8500.00, 1500.00, 50)';
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    v_status := DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

14.3.3.18.6 COLUMN_VALUE

The COLUMN_VALUE procedure defines a variable to receive a value from a cursor.

```sql
COLUMN_VALUE(<c> NUMBER, <position> NUMBER, <value> OUT { BLOB | CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 | VARCHAR2 | }) [, <column_error> OUT NUMBER [, <actual_length> OUT INTEGER ]])
```

Parameters

Cursor id of the cursor returning data to the variable being defined.

```sql
position
```

Position of the returned data in the cursor. The first value in the cursor is position 1.

```sql
value
```

Variable receiving the data returned in the cursor by a prior fetch call.

```sql
column_error
```

Error number associated with the column, if any.

```sql
actual_length
```

Actual length of the data prior to any truncation.

Examples

This example shows the portion of an anonymous block that receives the values from a cursor using the COLUMN_VALUE procedure.
The `COLUMN_VALUE_CHAR` procedure defines a variable to receive a `CHAR` value from a cursor.

### Parameters

- **c**: Cursor id of the cursor returning data to the variable being defined.
- **position**: Position of the returned data in the cursor. The first value in the cursor is position 1.
- **value**: Variable of data type `CHAR` receiving the data returned in the cursor by a prior fetch call.
column_error
Error number associated with the column, if any.

actual_length
Actual length of the data prior to any truncation.

### 14.3.3.1.18.8 COLUMN_VALUE_RAW

The **COLUMN_VALUE_RAW** procedure defines a variable to receive a `RAW` value from a cursor.

```sql
COLUMN_VALUE_RAW(<c> NUMBER, <position> NUMBER, <value> OUT RAW
[, <column_error> OUT NUMBER [, <actual_length> OUT INTEGER]
])
```

#### Parameters

- **c**
  
  Cursor id of the cursor returning data to the variable being defined.

- **position**
  
  Position of the returned data in the cursor. The first value in the cursor is position 1.

- **value**
  
  Variable of data type `RAW` receiving the data returned in the cursor by a prior fetch call.

- **column_error**
  
  Error number associated with the column, if any.

- **actual_length**
  
  Actual length of the data prior to any truncation.

### COLUMN_VALUE_LONG

The **COLUMN_VALUE_LONG** procedure returns a part of the value of a `LONG` column.

```sql
COLUMN_VALUE_LONG(<c> NUMBER, <position> NUMBER, <length> NUMBER,
<offset> NUMBER, <value> OUT VARCHAR2, <value_length> OUT INTEGER)
```

#### Parameters

- **c**
  
  Cursor id of the cursor from which to get a value.

- **position**
  
  Position of the column of which to get a value.

- **length**
  
  Number of bytes of the long value to fetch.

- **offset**
  
  Offset of the part of the `LONG` value to fetch.
Offset into the long field for start of fetch.

value

Value of the column.

value_length

Number of bytes returned in value.

For an example, see DEFINE_COLUMN_LONG.

### 14.3.3.18.9 COLUMN_VALUE_LONG

The **COLUMN_VALUE_LONG** procedure gets part of a value of a **LONG** column to be returned and retrieved in a cursor.

```
COLUMN_VALUE_LONG(<c> IN NUMBER, <position> IN NUMBER, <length> IN NUMBER,
<offset> IN NUMBER, <value> OUT VARCHAR2, <value_length> OUT NUMBER);
```

**Parameters**

**c**

Cursor ID of the cursor returning data.

**position**

Position of the returned data in the cursor. The first value in the cursor is position 1.

**length**

Number of bytes of the long value to fetch.

**offset**

Offset into the long field at which point the fetch begins.

**value**

Value of the column as a VARCHAR2.

**value_length**

Number of bytes returned.

**Examples**

```sql
-- Creating the required object
CREATE TABLE rm44142_dtypes
(
  col_long LONG,
  col_int INTEGER);

-- Insert data into table
INSERT INTO rm44142_dtypes VALUES ('TestingForDefineColumnValueLong', 1);

-- Use zero offset
DECLARE
  cur_id INTEGER;
  v_long VARCHAR(28);
```
The `COLUMN_VALUE_ROWID` procedure defines a column of the type `ROWID` to be returned and retrieved in a cursor.

```sql
COLUMN_VALUE_ROWID(<c> NUMBER, <position> NUMBER, <value> ROWID [, <column_error> OUT NUMBER [, <actual_length> OUT INTEGER ]]);
```

Parameters
Cursor ID of the cursor returning data to the variable being defined.

**position**

Position of the returned data in the cursor. The first value in the cursor is position 1.

**value**

The ROWID the data returned in the cursor by a prior fetch call.

**column_error**

Error number associated with the column, if any.

**actual_length**

Actual length of the data prior to any truncation.

### Examples

```sql
DECLARE
  curid    NUMBER;
  v_rid    ROWID;
  v_empno  NUMBER(4);
  v_ename  VARCHAR2(18);
  v_hiredate DATE;
  v_sal    NUMBER(7,2);
  v_comm   NUMBER(7,2);
  v_sql    VARCHAR2(55) := 'SELECT rid, empno, ename, hiredate, sal, ' ||
                      'comm FROM emp';
  v_status INTEGER;
BEGIN
  curid := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
  DBMS_SQL.DEFINE_COLUMN(curid,1,v_rid);
  DBMS_SQL.DEFINE_COLUMN(curid,2,v_empno);
  DBMS_SQL.DEFINE_COLUMN(curid,3,v_ename,18);
  DBMS_SQL.DEFINE_COLUMN(curid,4,v_hiredate);
  DBMS_SQL.DEFINE_COLUMN(curid,5,v_sal);
  DBMS_SQL.DEFINE_COLUMN(curid,6,v_comm);
  v_status := DBMS_SQL.EXECUTE(curid);
  LOOP
    v_status := DBMS_SQL.FETCH_ROWS(curid);
    EXIT WHEN v_status = 0;
    DBMS_SQL.COLUMN_VALUE(curid,1,v_rid);
    DBMS_SQL.COLUMN_VALUE(curid,2,v_empno);
    DBMS_SQL.COLUMN_VALUE(curid,3,v_ename);
    DBMS_SQL.COLUMN_VALUE(curid,4,v_hiredate);
    DBMS_SQL.COLUMN_VALUE(curid,5,v_sal);
    DBMS_SQL.COLUMN_VALUE(curid,6,v_comm);
    DBMS_OUTPUT.PUT_LINE( 
      TO_CHAR(v_hiredate,'yyyy-mm-dd') || ' ' ||
      TO_CHAR(v_sal,'9,999.99') || ' ' ||
      TO_CHAR(NVL(v_comm,'9,999.99')));  
  END LOOP;
END;
```

---

**Cursor ID of the cursor returning data to the variable being defined.**

**position**

Position of the returned data in the cursor. The first value in the cursor is position 1.

**value**

The ROWID the data returned in the cursor by a prior fetch call.

**column_error**

Error number associated with the column, if any.

**actual_length**

Actual length of the data prior to any truncation.

### Examples

```sql
DECLARE
  curid NUMBER;
  v_rid ROWID;
  v_empno NUMBER(4);
  v_ename VARCHAR2(18);
  v_hiredate DATE;
  v_sal NUMBER(7,2);
  v_comm NUMBER(7,2);
  v_sql VARCHAR2(55) := 'SELECT rid, empno, ename, hiredate, sal, ' ||
                      'comm FROM emp';
  v_status INTEGER;
BEGIN
  curid := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
  DBMS_SQL.DEFINE_COLUMN_rowid(curid,1,v_rid);
  DBMS_SQL.DEFINE_COLUMN(curid,2,v_empno);
  DBMS_SQL.DEFINE_COLUMN(curid,3,v_ename,10);
  DBMS_SQL.DEFINE_COLUMN(curid,4,v_hiredate);
  DBMS_SQL.DEFINE_COLUMN(curid,5,v_sal);
  v_status := DBMS_SQL.EXECUTE(curid);
  LOOP
    v_status := DBMS_SQL.FETCH_ROWS(curid);
    EXIT WHEN v_status = 0;
    DBMS_SQL.COLUMN_VALUE_rowid(curid,1,v_rid);
    DBMS_SQL.COLUMN_VALUE(curid,2,v_empno);
    DBMS_SQL.COLUMN_VALUE(curid,3,v_ename);
    DBMS_SQL.COLUMN_VALUE(curid,4,v_hiredate);
    DBMS_SQL.COLUMN_VALUE(curid,5,v_sal);
    DBMS_SQL.COLUMN_VALUE(curid,6,v_comm);
    DBMS_OUTPUT.PUT_LINE(v_rid || ' ' || v_empno || ' ' || v_ename || ' ' ||
      TO_CHAR(v_hiredate,'yyyy-mm-dd') || ' ' ||
      TO_CHAR(v_sal,'9,999.99') || ' ' ||
      TO_CHAR(NVL(v_comm,'9,999.99')));
  END LOOP;
END;
```
14.3.3.18.11  DEFINE_ARRAY

The `DEFINE_ARRAY` procedure defines the collection for a column into which you want to fetch rows with a FETCH_ROWS call. You can use this procedure to fetch multiple rows from a single SELECT statement.

When you fetch rows, they are copied into DBMS_SQL buffers until you run a COLUMN_VALUE call, which copies the rows into the table that was passed as an argument to the COLUMN_VALUE call.

```sql
DEFINE_ARRAY(<c> IN NUMBER, <position> IN NUMBER,
               <table_variable> IN { FLOAT_TABLE | BLOB_TABLE | CLOB_TABLE | VARCHAR2_TABLE | NUMBER_TABLE | TIMESTAMP_TABLE |
               INTEGER_TABLE | ROWID_TABLE | TEXT_TABLE },
               <cnt> IN NUMBER, <lower_bnd> IN NUMBER);
```

**Parameters**

- **c**
  ID of the cursor to which you want to bind an array.

- **position**
  Relative position of the column in the array being defined. The first column is position 1.

- **table_variable**
  A datatype that can be any one of the following matching pairs:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;bl_tab&gt;</code></td>
<td>BLOB_TABLE</td>
</tr>
<tr>
<td><code>&lt;cl_tab&gt;</code></td>
<td>CLOB_TABLE</td>
</tr>
<tr>
<td><code>&lt;c_tab&gt;</code></td>
<td>VARCHAR2_TABLE</td>
</tr>
<tr>
<td><code>&lt;n_tab&gt;</code></td>
<td>NUMBER_TABLE</td>
</tr>
<tr>
<td><code>&lt;tms_tab&gt;</code></td>
<td>TIMESTAMP_TABLE</td>
</tr>
<tr>
<td><code>&lt;int_tab&gt;</code></td>
<td>INTEGER_TABLE</td>
</tr>
<tr>
<td><code>&lt;rd_tab&gt;</code></td>
<td>ROWID_TABLE</td>
</tr>
<tr>
<td><code>&lt;txt_tab&gt;</code></td>
<td>TEXT_TABLE</td>
</tr>
</tbody>
</table>

- **cnt**
  Number of rows to fetch. Must be a positive integer.

- **lower_bnd**
  Results are copied into the collection, starting at this lower bound index.

**Example**

```sql
CREATE TABLE tab_test (col_int INTEGER, col_charactervarying VARCHAR(40));
INSERT into tab_test(col_int, col_charactervarying) values (10, 'user1'), (10, 'user2'), (10, 'user3'), (10, 'user4');
INSERT into tab_test(col_int, col_charactervarying) values (10, 'user1'), (10, 'user2'), (10, 'user3'), (10, 'user4');

-- DEFINE_ARRAY (used in SELECT to get multiple rows)
DECLARE names DBMS_SQL.VARCHAR2_TABLE;
dpt_id DBMS_SQL.INTEGER_TABLE;
c NUMBER;
```
14.3.3.1.18.12  DEFINE_COLUMN

The `DEFINE_COLUMN` procedure defines a column or expression in the `SELECT` list to be returned and retrieved in a cursor.

```
DEFINE_COLUMN(<c> NUMBER, <position> NUMBER, <column> { BLOB | CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 } [, <column_size> NUMBER ])
```

**Parameters**

`c`

Cursor id of the cursor associated with the `SELECT` command.

`position`

Position of the column or expression in the `SELECT` list that's being defined.
A variable of the same data type as the column or expression in position `position` of the `SELECT` list.

The maximum length of the returned data. Specify `column_size` only if `column` is `VARCHAR2`. Returned data exceeding `column_size` is truncated to `column_size` characters.

Examples

This example shows how the `empno`, `ename`, `hiredate`, `sal`, and `comm` columns of the `emp` table are defined with the `DEFINE_COLUMN` procedure.

```sql
DECLARE
    curid NUMBER;
    v_empno NUMBER(4);
    v_ename VARCHAR2(10);
    v_hiredate DATE;
    v_sal NUMBER(7,2);
    v_comm NUMBER(7,2);
    v_sql VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';
    v_status INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_ename,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_comm);
    v_status := DBMS_SQL.EXECUTE(curid);
    LOOP
        v_status := DBMS_SQL.FETCH_ROWS(curid);
        EXIT WHEN v_status = 0;
        DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
        DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
        DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || v_ename || ' ' || TO_CHAR(v_hiredate,'yyyymm-dd') || ' ' || TO_CHAR(v_sal,'9,999.99') || ' ' || TO_CHAR(NVL(v_comm,0),'9,999.99'));
    END LOOP;
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

The following shows an alternative that produces the same results. The lengths of the data types are irrelevant. The `empno`, `sal`, and `comm` columns still return data equivalent to `NUMBER(4)` and `NUMBER(7,2)`, respectively, even though `v_num` is defined as `NUMBER(1)` (assuming the declarations in the `COLUMN_VALUE` procedure are of the appropriate maximum sizes). The `ename` column returns data up to 10 characters in length as defined by the `length` parameter in the `DEFINE_COLUMN` call, not by the data type declaration, `VARCHAR2(1)` declared for `v_varchar`. The actual size of the returned data is dictated by the `COLUMN_VALUE` procedure.
14.3.3.18.13  DEFINE_COLUMN_CHAR

The `DEFINE_COLUMN_CHAR` procedure defines a CHAR column or expression in the SELECT list to be returned and retrieved in a cursor.

\[
\text{DEFINE_COLUMN_CHAR}(\text{c}, \text{position}, \text{column}, \text{column_size})
\]

**Parameters**

- **c**: Cursor id of the cursor associated with the SELECT command.
- **position**: Position of the column or expression in the SELECT list being defined.
- **column**: A CHAR variable.
- **column_size**: The maximum length of the returned data. Returned data exceeding `column_size` is truncated to `column_size` characters.

---

14.3.3.18.14  DEFINE_COLUMN_RAW

The `DEFINE_COLUMN_RAW` procedure defines a RAW column or expression in the SELECT list to be returned and retrieved in a cursor.

\[
\text{DEFINE_COLUMN_RAW}(\text{c}, \text{position}, \text{column}, \text{column_size})
\]
Parameters

\[ c \]

Cursor id of the cursor associated with the `SELECT` command.

\[ position \]

Position of the column or expression in the `SELECT` list being defined.

\[ column \]

A `RAW` variable.

\[ column\_size \]

The maximum length of the returned data. Returned data exceeding `column_size` is truncated to `column_size` characters.

**DEFINE_COLUMN_LONG**

The `DEFINE_COLUMN_LONG` procedure defines a long column for a `SELECT` cursor.

```
DEFINE_COLUMN_LONG(<c> NUMBER, <position> NUMBER)
```

Parameters

\[ c \]

Cursor id of the cursor for a row defined to be selected.

\[ position \]

Position of the column in a row being defined.

Examples

This example shows an anonymous block that defines a long column in the `SELECT` list using `DEFINE_COLUMN_LONG` procedure. It returns a part of the `LONG` column value into a variable using procedure `COLUMN_VALUE_LONG`.

```
DECLARE
  curid NUMBER;
  v_ename VARCHAR2(20);
  sql_stmt VARCHAR2(50) := 'SELECT ename ' || ' FROM emp WHERE empno = 7844';
  v_status INTEGER;
  v_length INTEGER;
BEGIN
  curid := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(curid, sql_stmt, DBMS_SQL.native);
  DBMS_SQL.DEFINE_COLUMN_LONG(curid, 1);
  v_status := DBMS_SQL.EXECUTE(curid);
  DBMS_SQL.FETCH_ROWS(curid);
  DBMS_SQL.COLUMN_VALUE_LONG(curid, 1, 7, 0, v_ename, v_length);
  DBMS_OUTPUT.PUT_LINE('ename: ' || v_ename || ' & length: ' || v_length);
  DBMS_SQL.CLOSE_CURSOR(curid);
END;
```
14.3.3.18.15  DEFINE_COLUMN_LONG

The DEFINE_COLUMN_LONG procedure defines a column of the type LONG in the SELECT list to be returned and retrieved in a cursor. When you use this procedure, the column you are defining is identified by its relative position in the SELECT list of the statement in the specified cursor. The COLUMN value type determines the type of column being defined.

```
DEFINE_COLUMN_LONG(<c> IN NUMBER, <position> IN NUMBER);
```

Parameters

c

Cursor ID of the cursor from which a row is selected.

position

Relative position of the column in the row being defined. The first column in a statement has position 1.

Examples

```sql
-- Creating the required object
CREATE TABLE rm44142_dtypes
(
  col_long LONG,
  col_int INTEGER
);

-- Insert data into table
INSERT INTO rm44142_dtypes VALUES ('TestingForDefineColumnValueLong', 1);

-- Use zero offset
DECLARE
  cur_id INTEGER;
  v_long VARCHAR(20);
  sql_stat VARCHAR2(50) := 'SELECT col_long ' || ' FROM rm44142_dtypes';
  status INTEGER;
  length INTEGER;
BEGIN
  cur_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cur_id, sql_stat, DBMS_SQL.native);
  DBMS_SQL.DEFINE_COLUMN_LONG(cur_id, 1);
  status := DBMS_SQL.EXECUTE(cur_id);
  DBMS_SQL.FETCH_ROWS(cur_id);
  DBMS_SQL.COLUMN_VALUE_LONG(cur_id, 1, 7, 0, v_long, length);
  DBMS_OUTPUT.PUT_LINE('col_long: ' || v_long);
  DBMS_OUTPUT.PUT_LINE('value_length: ' || length);
  DBMS_SQL.CLOSE_CURSOR(cur_id);
END;
```

Output:

```
col_long: Testing
value_length: 7
```

EDB-SPL Procedure successfully completed
14.3.3.1.18.16  DEFINE_COLUMN_ROWID

The `DEFINE_COLUMN_ROWID` procedure defines a column of the type `ROWID` in the `SELECT` list to be returned and retrieved in a cursor.

**Note**

In EDB Postgres Advanced Server, `ROWID` is a domain rather than a bigint.

When you use this procedure the column you are defining is identified by its relative position in the `SELECT` list of the statement in the specified cursor. The `COLUMN` value type determines the type of column being defined.

```sql
DEFINE_COLUMN_ROWID(<c> IN NUMBER, <position> IN NUMBER, <column> IN NUMBER);
```

**Parameters**

- `c`
  
  Cursor ID of the cursor from which a row is selected.

- `position`
  
  Relative position of the column in the row being defined. The first column in a statement has position 1.

- `column`
  
  Value of the column. The type of this value determines the type for the column being defined.

**Examples**

This example shows how the `empno`, `ename`, `hiredate`, `sal`, and `comm` columns of the `emp` table are defined with the `DEFINE_COLUMN_ROWID` procedure.

```sql
DECLARE
  cur_id INTEGER;
  v_long VARCHAR2(20);
BEGIN
  DECLARE
    sql_stmt VARCHAR2(50) := 'SELECT col_int, col_long ' || ' FROM rm44142_dtypes';
    status INTEGER;
    length INTEGER;
  BEGIN
    cur_id := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(cur_id, sql_stmt, DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN_LONG(cur_id, 2);
    status := DBMS_SQL.EXECUTE(cur_id);
    DBMS_SQL.FETCH_ROWS(cur_id);
    DBMS_SQL.COLUMN_VALUE_LONG(cur_id, 2, 12, 10, v_long, length);
    DBMS_OUTPUT.PUT_LINE('col_long: ' || v_long);
    DBMS_OUTPUT.PUT_LINE('value_length: ' || length);
    DBMS_SQL.CLOSE_CURSOR(cur_id);
  END;
END;
```

Output:

```
col_long: DefineColumn
value_length: 12
```

EDB SPL Procedure successfully completed
14.3.3.1.18.17  DESCRIBE_COLUMNS

The `DESCRIBE_COLUMNS` procedure describes the columns returned by a cursor.

```
DESCRIBE_COLUMNS(<c> NUMBER, <col_cnt> OUT NUMBER, <desc_t> OUT DESC_TAB);
```

**Parameters**

- `<c>`
  
  The cursor ID of the cursor.

- `<col_cnt>`
  
  The number of columns in the cursor result set.

- `<desc_t>`
  
  The table that contains a description of each column returned by the cursor. The descriptions are of type `DESC_REC` and contain the following values:
14.3.3.1.18.18 DESCRIBE_COLUMNS2

The DESCRIBE_COLUMNS2 procedure describes specified columns returned by a cursor. This procedure provides an alternative to DESCRIBE_COLUMN.

DESCRIBE_COLUMNS2(<c> IN NUMBER, <col_cnt> OUT NUMBER, <desc_tab2> OUT DESC_TAB2);

Parameters

$c$

The cursor ID of the cursor.

$\text{col_cnt}$

The number of columns in the cursor result set.

$\text{desc_tab2}$

The table that contains a description of each column returned by the cursor. The descriptions are of type DESC_REC2 and contain the following values:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{col_type}$</td>
<td>INTEGER</td>
</tr>
<tr>
<td>$\text{col_maxlen}$</td>
<td>INTEGER</td>
</tr>
<tr>
<td>$\text{col_name}$</td>
<td>VARCHAR2(128)</td>
</tr>
<tr>
<td>$\text{col_name_len}$</td>
<td>INTEGER</td>
</tr>
<tr>
<td>$\text{col_schema_name}$</td>
<td>VARCHAR2(128)</td>
</tr>
<tr>
<td>$\text{col_schema_name_len}$</td>
<td>INTEGER</td>
</tr>
<tr>
<td>$\text{col_precision}$</td>
<td>INTEGER</td>
</tr>
<tr>
<td>$\text{col_scale}$</td>
<td>INTEGER</td>
</tr>
<tr>
<td>$\text{col_charsetid}$</td>
<td>INTEGER</td>
</tr>
<tr>
<td>$\text{col_charsetform}$</td>
<td>INTEGER</td>
</tr>
<tr>
<td>$\text{col_null_ok}$</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>

Examples

```
DECLARE
    c INTEGER := DBMS_SQL.OPEN_CURSOR;
    l_columns DBMS_SQL.desc_tab2;
    l_numcols INTEGER;
    dbms_return INTEGER;
```
The `DESCRIBE_COLUMNS3` procedure describes specified columns returned by a cursor. This procedure provides an alternative to `DESCRIBE_COLUMN`.

```
BEGIN
  DBMS_SQL.parse ( c, 'SELECT * FROM emp', DBMS_SQL.native );
  dbms_return := DBMS_SQL.EXECUTE ( c);
  DBMS_SQL.describe_columns3 ( c, l_numcols, l_columns );
  DBMS_OUTPUT.PUT_LINE ( 'Total # of Columns     :    ' || l_numcols);
  DBMS_OUTPUT.put_line ( '--------------------------------------' );
  FOR colind IN 1..l_numcols LOOP
    DBMS_OUTPUT.PUT_LINE( 'Columns # :     ' || colind);
    DBMS_OUTPUT.put_line ( 'Column Name :     ' || l_columns ( colind ).col_name);
    DBMS_OUTPUT.put_line ( 'Column Type :     ' || l_columns ( colind ).col_type);
    DBMS_OUTPUT.put_line ( 'Column Max Length :     ' || l_columns ( colind ).col_max_len);
    DBMS_OUTPUT.put_line ( 'Column Name Length :     ' || l_columns ( colind ).col_name_len);
    DBMS_OUTPUT.put_line ( 'Column Schema Name :     ' || l_columns ( colind ).col_schema_name);
    DBMS_OUTPUT.put_line ( 'Column Schema Name Length :     ' || l_columns ( colind ).col_schema_name_len);
    DBMS_OUTPUT.put_line ( 'Column Precision :     ' || l_columns ( colind ).col_precision);
    DBMS_OUTPUT.put_line ( 'Column Scale :     ' || l_columns ( colind ).col_scale);
    DBMS_OUTPUT.put_line ( 'Character Set ID :     ' || l_columns ( colind ).col_charsetid);
    DBMS_OUTPUT.put_line ( 'Character Set Form :     ' || l_columns ( colind ).col_charsetform);
    DBMS_OUTPUT.put ( 'Column Can be NULL : ');
    IF l_columns ( colind ).col_null_ok THEN
      DBMS_OUTPUT.put_line ( 'NULL' );
    ELSE
      DBMS_OUTPUT.put_line ( 'NOT NULL' );
    END IF;
    DBMS_OUTPUT.put_line ( '--------------------------------------' );
  END LOOP;
  DBMS_SQL.close_cursor ( c );
END;
```

### Parameters

- **c**
  - The cursor ID of the cursor.

- **col_cnt**
  - The number of columns in the cursor result set.

- **desc_tab3**
  - The table that contains a description of each column returned by the cursor. The descriptions are of type `DESC_REC3` and contain the following values:

<table>
<thead>
<tr>
<th>Column name</th>
<th>Type</th>
</tr>
</thead>
</table>
Examples

```
CREATE TYPE PROJECT_DETAILS AS OBJECT
    ( projname VARCHAR2(20),
      mgr VARCHAR2(20));
CREATE TABLE projecttab (deptno NUMBER, project PROJECT_DETAILS);

DECLARE
    curid NUMBER;
    desctab DBMS_SQL.DESC_TAB3;
    colcnt NUMBER;
    sql_stmt VARCHAR2(200) := 'SELECT * FROM projecttab';
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid, sql_stmt,
                   DBMS_SQL.NATIVE);
    DBMS_SQL.DESCRIBE_COLUMNS3(curid, colcnt, desctab);
    FOR i IN 1 .. colcnt LOOP
        IF desctab(i).col_type = 109 THEN
            DBMS_OUTPUT.PUT(desctab(i).col_name || ' is user-defined type: ');
            DBMS_OUTPUT.PUT_LINE('COL_TYPE_NAME = ' ||
                                   desctab(i).col_type_name ||
                                   ' COL_TYPE_NAME_LEN = ' ||
                                   desctab(i).col_type_name_len);
        ELSE
            DBMS_OUTPUT.PUT(desctab(i).col_name || ' is not user-defined type: ');
            DBMS_OUTPUT.PUT_LINE('COL_TYPE_NAME is NULL and ' ||
                                    'COL_TYPE_NAME_LEN = ' ||
                                    desctab(i).col_type_name_len);
        END IF;
    END LOOP;
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

14.3.3.18.20 EXECUTE

The `EXECUTE` function executes a parsed SQL command or SPL block.

```
<status> INTEGER EXECUTE(<c> NUMBER)
```

Parameters
Cursor ID of the parsed SQL command or SPL block to execute.

**status**

Number of rows processed if the SQL command was `DELETE`, `INSERT`, or `UPDATE`. `status` is meaningless for all other commands.

**Examples**

The following anonymous block inserts a row into the `dept` table.

```sql
DECLARE
  curid NUMBER;
  v_sql VARCHAR2(50);
  v_status INTEGER;
BEGIN
  curid := DBMS_SQL.OPEN_CURSOR;
  v_sql := 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
  DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  v_status := DBMS_SQL.EXECUTE(curid);
  DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

### 14.3.3.18.21 EXECUTE_AND_FETCH

Function `EXECUTE_AND_FETCH` executes a parsed `SELECT` command and fetches one row.

```sql
INTEGER EXECUTE_AND_FETCH(c NUMBER [, exact BOOLEAN ])
```

**Parameters**

- `c` Cursor id of the cursor for the `SELECT` command to execute.
- `exact` If set to `TRUE`, an exception is thrown if the number of rows in the result set isn't exactly equal to 1. If set to `FALSE`, no exception is thrown. The default is `FALSE`. A `NO_DATA_FOUND` exception is thrown if `exact` is `TRUE` and there are no rows in the result set. A `TOO_MANY_ROWS` exception is thrown if `exact` is `TRUE` and more than one row is in the result set.

**status**

Returns 1 if a row was successfully fetched, 0 if no rows to fetch. If an exception is thrown, no value is returned.

**Examples**

This stored procedure uses the `EXECUTE_AND_FETCH` function to retrieve one employee using the employee's name. An exception is thrown if the employee isn't found or more than one employee has the same name.

```sql
CREATE OR REPLACE PROCEDURE select_by_name(
  p_ename emp.ename%TYPE
) IS
  curid NUMBER;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
v_empno  emp.empno%TYPE;

v_hiredate emp.hiredate%TYPE;

v_sal     emp.sal%TYPE;

v_comm    emp.comm%TYPE;

v_dname   dept.dname%TYPE;

v_disp_date VARCHAR2(10);

v_sql     VARCHAR2(120) := 'SELECT empno, hiredate, sal, ' ||
                          'NVL(comm, 0), dname ' ||
                          'FROM emp e, dept d ' ||
                          'WHERE ename = :p_ename ' ||
                          'AND e.deptno = d.deptno';

v_status  INTEGER;

BEGIN
  curid := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
  DBMS_SQL.BIND_VARIABLE(curid,':p_ename',UPPER(p_ename));
  DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
  DBMS_SQL.DEFINE_COLUMN(curid,2,v_hiredate);
  DBMS_SQL.DEFINE_COLUMN(curid,3,v_sal);
  DBMS_SQL.DEFINE_COLUMN(curid,4,v_comm);
  DBMS_SQL.DEFINE_COLUMN(curid,5,v_dname,14);
  v_status := DBMS_SQL.EXECUTE_AND_FETCH(curid,TRUE);
    DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
    DBMS_SQL.COLUMN_VALUE(curid,2,v_hiredate);
    DBMS_SQL.COLUMN_VALUE(curid,3,v_sal);
    DBMS_SQL.COLUMN_VALUE(curid,4,v_comm);
    DBMS_SQL.COLUMN_VALUE(curid,5,v_dname);

  v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
  DBMS_OUTPUT.PUT_LINE('Number    : ' ||
                        v_empno);
  DBMS_OUTPUT.PUT_LINE('Name      : ' ||
                        UPPER(p_ename));
  DBMS_OUTPUT.PUT_LINE('Hire Date : ' ||
                        v_disp_date);
  DBMS_OUTPUT.PUT_LINE('Salary    : ' ||
                        v_sal);
  DBMS_OUTPUT.PUT_LINE('Commission: ' ||
                        v_comm);
  DBMS_OUTPUT.PUT_LINE('Department: ' ||
                        v_dname);
  DBMS_SQL.CLOSE_CURSOR(curid);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_ename || ' not found');
  DBMS_SQL.CLOSE_CURSOR(curid);
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('Too many employees named, ' || p_ename || ', found');
  DBMS_SQL.CLOSE_CURSOR(curid);
  WHEN OTHERS
    DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:' || SQLCODE);
  DBMS_SQL.CLOSE_CURSOR(curid);
END;
EXEC select_by_name('MARTIN')
14.3.3.1.18.22 FETCH_ROWS

The `FETCH_ROWS` function retrieves a row from a cursor.

```sql
<status> INTEGER FETCH_ROWS(<c>
NUMBER)
```

**Parameters**

- **c**
  Cursor ID of the cursor from which to fetch a row.

- **status**
  Returns 1 if a row was successfully fetched, 0 if no more rows to fetch.

**Examples**

These examples fetch the rows from the `emp` table and display the results.

```sql
DECLARE
  curid NUMBER;
  v_empno NUMBER(4);
  v_ename VARCHAR2(10);
  v_hiredate DATE;
  v_sal NUMBER(7,2);
  v_comm NUMBER(7,2);
  v_sql VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                     'comm FROM emp';
  v_status INTEGER;
BEGIN
  curid := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
  DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
  DBMS_SQL.DEFINE_COLUMN(curid,2,v_ename,10);
  DBMS_SQL.DEFINE_COLUMN(curid,3,v_hiredate);
  DBMS_SQL.DEFINE_COLUMN(curid,4,v_sal);
  DBMS_SQL.DEFINE_COLUMN(curid,5,v_comm);
  v_status :=
    DBMS_SQL.EXECUTE(curid);
  DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL            COMM');
  DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------  ------')
  LOOP
    v_status :=
      DBMS_SQL.FETCH_ROWS(curid);
    v_empno..v_salary :=
      DBMS_SQL.COLUMN_VALUE(curid,1..5);
    DBMS_OUTPUT.PUT_LINE(v_empno..v_salary'::CHAR(20));
  EXIT;
END;
```

---

Number : 7654
Name : MARTIN
Hire Date : 09/28/1981
Salary : 1250
Commission: 1400
Department: SALES
### 14.3.3.18.23 IS_OPEN

The **IS_OPEN** function tests whether the given cursor is open.

```sql
DECLARE
    CURSOR curid IS
        SELECT * FROM employees;
    CURSOR curid2 IS
        SELECT * FROM employees2;
    CURSOR curid3 IS
        SELECT * FROM employees3;
    v_status BOOLEAN;
    v_empno NUMBER;
    v_ename VARCHAR2(10);
    v_hiredate VARCHAR2(18);
    v_sal NUMBER;
    v_comm NUMBER;
BEGIN
    v_status := DBMS_SQL.IS_OPEN(curid);
    DBMS_OUTPUT.PUT_LINE('IS_OPEN: ' || v_status);
    v_status := DBMS_SQL.IS_OPEN(curid2);
    DBMS_OUTPUT.PUT_LINE('IS_OPEN: ' || v_status);
    v_status := DBMS_SQL.IS_OPEN(curid3);
    DBMS_OUTPUT.PUT_LINE('IS_OPEN: ' || v_status);
END;
```

**Parameters**

- `<c>` **NUMBER**
  - Cursor ID of the cursor to test.

**status**

Set to `TRUE` if the cursor is open, set to `FALSE` if the cursor isn't open.

### 14.3.3.18.24 LAST_ROW_COUNT

The **LAST_ROW_COUNT** function returns the number of rows that were currently fetched.

```sql
EXIT WHEN v_status = 0;
DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
DBMS_OUTPUT.PUT_LINE(v_empno || ' | ' || RPAD(v_ename,10,' ') || ' | ' || TO_CHAR(v_hiredate,'yyyy-mm-dd') || ' | ' || TO_CHAR(v_sal,'9,999.99') || ' | ' || TO_CHAR(NVL(v_comm,0),'9,999.99'));
END LOOP;
DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>1980-12-17</td>
<td>800.00</td>
<td></td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>1980-12-20</td>
<td>1,600.00</td>
<td>300.80</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>1980-12-22</td>
<td>1,250.00</td>
<td>508.00</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>1980-12-22</td>
<td>2,075.00</td>
<td></td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>1980-09-28</td>
<td>1,250.00</td>
<td></td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>1980-08-01</td>
<td>2,850.00</td>
<td></td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>1980-06-09</td>
<td>2,450.00</td>
<td></td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>1987-04-19</td>
<td>3,008.00</td>
<td></td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>1981-11-17</td>
<td>5,800.00</td>
<td></td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>1981-09-08</td>
<td>1,508.00</td>
<td></td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>1987-05-23</td>
<td>1,100.00</td>
<td></td>
</tr>
<tr>
<td>7980</td>
<td>JAMES</td>
<td>1981-12-03</td>
<td>950.00</td>
<td></td>
</tr>
<tr>
<td>7982</td>
<td>FORD</td>
<td>1981-12-03</td>
<td>3,608.00</td>
<td></td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>1982-01-23</td>
<td>1,308.00</td>
<td></td>
</tr>
</tbody>
</table>
Parameters

\texttt{rowcnt}

Number of rows fetched thus far.

Examples

This example uses the \texttt{LAST\_ROW\_COUNT} function to display the total number of rows fetched in the query.

```sql
DECLARE
    curid NUMBER;
    v_empno NUMBER(4);
    v_ename VARCHAR2(18);
    v_hiredate DATE;
    v_sal NUMBER(7,2);
    v_comm NUMBER(7,2);
    v_sql VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                  'comm FROM emp'::VARCHAR2(50);
    v_status INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    DBMS_SQL.DEFINE_COLUMN(curid,1,v_empno);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_ename,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_hiredate);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_sal);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_comm);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL    COMM');
    DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------  --------');
    LOOP
        v_status := DBMS_SQL.FETCH_ROWS(curid);
        EXIT WHEN v_status = 0;
        DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
        DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
        DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,18) || '   ' ||
                        TO_CHAR(v_hiredate,'yyyy-mm-dd') || '   ' ||
                        TO_CHAR(v_sal,'9,999.99') || '   ' ||
                        TO_CHAR(NVL(v_comm,0),'9,999.99'));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Number of rows: ' || DBMS_SQL.LAST_ROW_COUNT);
END;
END;
```
The `LAST_ERROR_POSITION` function returns an integer value indicating the byte offset in the SQL statement text where the error occurred. The error position of the first character in the SQL statement is at 1.

**Examples**

This example shows an anonymous block that returns an error position with the `LAST_ERROR_POSITION` function.

```sql
DECLARE
    curid NUMBER;
    sql_stmt VARCHAR2(50) := 'SELECT empno FROM not_exist_table';
    v_position INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid, sql_stmt, DBMS_SQL.native);
    EXCEPTION WHEN OTHERS THEN
        v_position := DBMS_SQL.LAST_ERROR_POSITION;
        DBMS_OUTPUT.PUT_LINE('error position = ' || v_position);
        DBMS_SQL.CLOSE_CURSOR(curid);
    END;
END;
```

**LAST_ERROR_POSITION** returns 19.

**OPEN_CURSOR**

The `OPEN_CURSOR` function creates a cursor. A cursor must be used to parse and execute any dynamic SQL statement. Once a cursor is open, you can reuse it with the same or different SQL statements. You don't have to close the cursor and reopen it to reuse it.
Parameters

`c`

Cursor ID number associated with the newly created cursor.

Examples

This example creates a new cursor:

```
DECLARE
  curid NUMBER;
  v_sql VARCHAR2(150);
  v_status INTEGER;
BEGIN
  v_sql := 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
  curid := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  v_status := DBMS_SQL.EXECUTE(curid);
  DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

The `PARSE` procedure parses a SQL command or SPL block. If the SQL command is a DDL command, it executes immediately and doesn't require that you run the `EXECUTE` function.

```
PARSE(<c> NUMBER, <statement> VARCHAR2, <language_flag> NUMBER)
```

Parameters

`c`

Cursor ID of an open cursor.

`statement`

SQL command or SPL block to parse. A SQL command must not end with the semi-colon terminator. However an SPL block does require the semi-colon terminator.

`language_flag`

Language flag provided for compatibility with Oracle syntax. Use `DBMS_SQL.V6`, `DBMS_SQL.V7` or `DBMS_SQL.native`. This flag is ignored, and all syntax is assumed to be in EDB EDB Postgres Advanced Server form.

Examples

This anonymous block creates a table named `job`. DDL statements are executed immediately by the `PARSE` procedure and don't require a separate `EXECUTE` step.

```
DECLARE
  curid NUMBER;
BEGIN
  curid := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno NUMBER(3), ' ||
           'jname VARCHAR2(9))', DBMS_SQL.native);
END;
```
The following inserts two rows into the `job` table.

```plsql
DECLARE
    curid NUMBER;
    v_sql VARCHAR2(58);
    v_status INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    v_sql := 'INSERT INTO job VALUES (100, ''ANALYST'');';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    v_sql := 'INSERT INTO job VALUES (200, ''CLERK'');';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

This anonymous block uses the `DBMS_SQL` package to execute a block containing two `INSERT` statements. The end of the block contains a terminating semi-colon. In the prior example, each `INSERT` statement doesn't have a terminating semi-colon.

```plsql
DECLARE
    curid NUMBER;
    v_sql VARCHAR2(100);
    v_status INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    v_sql := 'BEGIN ' || 'INSERT INTO job VALUES (300, ''MANAGER''); ' || 'INSERT INTO job VALUES (400, ''SALESMAN''); ' || 'END;';
    DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

### 14.3.3.1.18.27 TO_CURSOR_NUMBER

The `TO_CURSOR_NUMBER` function transforms a ref cursor into a SQL cursor number.

This function is useful to retrieve the metadata (columns, column types, and so forth) from a ref cursor, when you have a SYS_REFCURSOR variable with an unknown return structure.

```plsql
TO_CURSOR_NUMBER (<rc> IN OUT SYS_REFCURSOR)
RETURN INTEGER;
```

**Parameters**

- `<rc>`
The ref cursor to be transformed into a cursor number.

### Examples

```sql
DECLARE
cur1 SYS_REFCURSOR;
cur_id INTEGER;
tbl_desc test_tbl%ROWTYPE;
totcols INTEGER;
ret INTEGER;
rowcnt NUMBER;
coll_data VARCHAR(30);
coll2_data VARCHAR(30);
BEGIN
  OPEN cur1 FOR 'SELECT * FROM test_tbl';
  cur_id := dbms_sql.to_cursor_number(cur1);
  dbms_sql.define_column(cur_id, 1, tbl_desc.col1);
  dbms_sql.define_column(cur_id, 2, tbl_desc.col2);
  LOOP
    ret := dbms_sql.FETCH_ROWS(cur_id);
    EXIT WHEN ret = 0;
    dbms_sql.column_value(cur_id, 1, tbl_desc.col1);
    dbms_sql.column_value(cur_id, 2, tbl_desc.col2);
    dbms_output.put_line('Col1: ' || tbl_desc.col1 || ' Col2: ' ||
                           tbl_desc.col2);
  END LOOP;
END;
```

### 14.3.3.1.18.28 TO_REFCURSOR

The **TO_REFCURSOR** function transforms an open cursor into a REF CURSOR for use in SQL statements. This program is only used with SELECT cursors.

**Note**

When you convert a cursor using **TO_REFCURSOR**, you cannot call **CLOSE_CURSOR** for that cursor.

```sql
TO_REFCURSOR (<cursor_number> IN OUT INTEGER)
RETURN SYS_REFCURSOR;
```

#### Parameter

`cursor_number`

Cursor ID number to be transformed into a ref cursor.

#### Examples

```sql
CREATE TABLE student_info (id INTEGER PRIMARY KEY, first_name VARCHAR(20), last_name VARCHAR(20), dept_id int);
INSERT INTO student_info VALUES
  (1, 'Tia', 'Jackson', 10),
  (2, 'Raj', 'Malhotra', 12),
  (3, 'Nash', 'Cadogan', 13),
  (4, 'Sam', 'Wordsworth', 10),
  (5, 'Roger', 'Zen', 10),
```
The **DBMS_UTILITY** package provides support for the following utility programs. EDB Postgres Advanced Server’s implementation of **DBMS_UTILITY** is a partial implementation when compared to Oracle’s version. Only the functions and procedures listed in the table are supported.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Function or procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANALYZE_DATABASE()</td>
<td>Procedure</td>
<td>n/a</td>
<td>Analyze database tables.</td>
</tr>
<tr>
<td>ANALYZE_PART_OBJECT()</td>
<td>Procedure</td>
<td>n/a</td>
<td>Analyze a partitioned table.</td>
</tr>
<tr>
<td>ANALYZE_SCHEMA()</td>
<td>Procedure</td>
<td>n/a</td>
<td>Analyze schema tables.</td>
</tr>
<tr>
<td>CANONICALIZE()</td>
<td>Procedure</td>
<td>n/a</td>
<td>Canonicalize a string, e.g., strip off white space.</td>
</tr>
</tbody>
</table>

---

{6, 'Cale', 'Kingston', 12},
{7, 'Caleb', 'Johnson', 13};

**CREATE TABLE** department_master (id number PRIMARY KEY, dept_name VARCHAR (20));
**INSERT INTO** department_master VALUES
{10, 'Science'},
{12, 'Maths'},
{13, 'Computers'};

**CREATE OR REPLACE FUNCTION** list_dept_student_info (qual IN VARCHAR2) RETURN sys_refcursor IS
dbmsql_cur NUMBER;
curvar sys_refcursor;
execvar PLS_INTEGER;
BEGIN
dbmsql_cur := DBMS_SQL.open_cursor;
DBMS_SQL.parse (dbmsql_cur
, 'SELECT first_name, last_name FROM student_info WHERE dept_id = (SELECT id FROM department_master WHERE dept_name LIKE :dept_id)' , DBMS_SQL.native);
DBMS_SQL.bind_variable (dbmsql_cur
, 'dept_id' , qual);
execvar := DBMS_SQL.EXECUTE (dbmsql_cur);
curvar := DBMS_SQL.to_refcursor (dbmsql_cur);
RETURN curvar;
END;

DECLARE
  TYPE strings_t IS TABLE OF VARCHAR2 (200);
  fnames strings_t;
  lnames strings_t;
BEGIN
  curvar := list_dept_student_info ('Science');
  FETCH curvar BULK COLLECT INTO fnames, lnames;
  DBMS_OUTPUT.put_line ('Students in Science department');
  FOR idx IN 1 .. fnames.COUNT LOOP
    DBMS_OUTPUT.put_line (fnames(idx) || ' ' || lnames(idx));
  END LOOP;
  CLOSE curvar;
END;

Students in Science department
Tia
Jackson
Sam
Wordsworth
Roger
Zen

**EDB-SPL Procedure** successfully completed
<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Function or procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMA_TO_TABLE(list, tablen OUT, tab OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Convert a comma-delimited list of names to a table of names.</td>
</tr>
<tr>
<td>DB_VERSION(version OUT, compatibility OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Get the database version.</td>
</tr>
<tr>
<td>EXEC_DDL_STATEMENT(parse_string)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Execute a DDL statement.</td>
</tr>
<tr>
<td>EXPAND_SQL_TEXT(input_sql_text, output_sql_text)</td>
<td>Function</td>
<td>TEXT</td>
<td>Returns expanded SQL references to view.</td>
</tr>
<tr>
<td>FORMAT_CALL_STACK</td>
<td>Function</td>
<td>TEXT</td>
<td>Formats the current call stack.</td>
</tr>
<tr>
<td>FORMAT_ERROR_BACKTRACE</td>
<td>Function</td>
<td>TEXT</td>
<td>Formats the current error call backtrace.</td>
</tr>
<tr>
<td>FORMAT_ERROR_STACK</td>
<td>Function</td>
<td>TEXT</td>
<td>Get the exception name.</td>
</tr>
<tr>
<td>GET_CPU_TIME</td>
<td>Function</td>
<td>NUMBER</td>
<td>Get the current CPU time.</td>
</tr>
<tr>
<td>GETDEPENDENCY(type, schema, name)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Get objects that depend on the given object.</td>
</tr>
<tr>
<td>GET_HASH_VALUE(name, base, hash_size)</td>
<td>Function</td>
<td>NUMBER</td>
<td>Compute a hash value.</td>
</tr>
<tr>
<td>GET_PARAMETER_VALUE(parnam, intval OUT, strval OUT)</td>
<td>Procedure</td>
<td>BINARY_INTEGER</td>
<td>Get database initialization parameter settings.</td>
</tr>
<tr>
<td>GET_TIME</td>
<td>Function</td>
<td>NUMBER</td>
<td>Get the current time.</td>
</tr>
<tr>
<td>NAME_TOKENIZE(name, a OUT, b OUT, c OUT, dblink OUT, nextpos OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Parse the given name into its component parts.</td>
</tr>
<tr>
<td>TABLE_TO_COMMA(tab, tablen OUT, list OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Convert a table of names to a comma-delimited list.</td>
</tr>
</tbody>
</table>

The following table lists the public variables available in the DBMS_UTILITY package.

<table>
<thead>
<tr>
<th>Public variables</th>
<th>Data type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv_error_on_restrictions</td>
<td>PLS_INTEGER</td>
<td>1</td>
<td>Used by the INVALIDATE procedure.</td>
</tr>
<tr>
<td>lname_array</td>
<td>TABLE</td>
<td></td>
<td>For lists of long names.</td>
</tr>
<tr>
<td>uncl_array</td>
<td>TABLE</td>
<td></td>
<td>For lists of users and names.</td>
</tr>
</tbody>
</table>

**LNAME_ARRAY**

The LNAME_ARRAY is for storing lists of long names including fully-qualified names.

```
TYPE lname_array IS TABLE OF VARCHAR2(4800) INDEX BY BINARY_INTEGER;
```

**UNCL_ARRAY**

The UNCL_ARRAY is for storing lists of users and names.

```
TYPE uncl_array IS TABLE OF VARCHAR2(227) INDEX BY BINARY_INTEGER;
```

**ANALYZE_DATABASE, ANALYZE_SCHEMA and ANALYZE PART_OBJECT**

The ANALYZE_DATABASE(), ANALYZE_SCHEMA() and ANALYZE PART_OBJECT() procedures gather statistics on tables in the database. When you execute the ANALYZE statement, Postgres samples the data in a table and records distribution statistics in the pg_statistic system table.

The ANALYZE_DATABASE, ANALYZE_SCHEMA, and ANALYZE PART_OBJECT differ primarily in the number of tables that are processed:

- **ANALYZE_DATABASE** analyzes all tables in all schemas in the current database.
- **ANALYZE_SCHEMA** analyzes all tables in a given schema (in the current database).
- **ANALYZE PART OBJECT** analyzes a single table.

The syntax for the ANALYZE commands are:

```
ANALYZE_DATABASE(<method> VARCHAR2 [, <estimate_rows> NUMBER [, <estimate_percent> NUMBER [, <method_opt> VARCHAR2 ]]])
```
Parameters for `ANALYZE_DATABASE` and `ANALYZE_SCHEMA`

- **method**
  Determines whether the `ANALYZE` procedure populates the `pg_statistics` table or removes entries from the `pg_statistics` table. If you specify a method of `DELETE`, the `ANALYZE` procedure removes the relevant rows from `pg_statistics`. If you specify a method of `COMPUTE` or `ESTIMATE`, the `ANALYZE` procedure analyzes a table (or multiple tables) and records the distribution information in `pg_statistics`. There's no difference between `COMPUTE` and `ESTIMATE`. Both methods execute the Postgres `ANALYZE` statement. All other parameters are validated and then ignored.

- **estimate_rows**
  Number of rows upon which to base estimated statistics. One of `estimate_rows` or `estimate_percent` must be specified if the method is `ESTIMATE`. This argument is ignored but is included for compatibility.

- **estimate_percent**
  Percentage of rows upon which to base estimated statistics. One of `estimate_rows` or `estimate_percent` must be specified if the method is `ESTIMATE`. This argument is ignored but is included for compatibility.

- **method_opt**
  Object types to analyze. Any combination of the following:

  ```
  [ FOR TABLE ]
  [ FOR ALL [ INDEXED ] COLUMNS ] [ SIZE n ]
  [ FOR ALL INDEXES ]
  ```

  This argument is ignored but is included for compatibility.

Parameters for `ANALYZE_PART_OBJECT`

- **schema**
  Name of the schema whose objects to analyze.

- **object_name**
  Name of the partitioned object to analyze.

- **object_type**
  Type of object to analyze. Valid values are: `T` - table, `I` - index. This argument is ignored but is included for compatibility.

- **command_type**
  Type of analyze functionality to perform. Valid values are:

  - `E` — Gather estimated statistics based on a specified number of rows or a percentage of rows in the `sample_clause` clause.
  - `C` — Compute exact statistics.
  - `V` — Validate the structure and integrity of the partitions.

  This argument is ignored but is included for compatibility.
For command type C or E, can be any combination of:

- [FOR TABLE]
- [FOR ALL COLUMNS]
- [FOR ALL LOCAL INDEXES]

For command type V, can be CASCADE if object_type is T.

This argument is ignored but is included for compatibility.

**SAMPLE n { ROWS | PERCENT }**

This argument is ignored but is included for compatibility.

---

**CANONICALIZE**

The CANONICALIZE procedure performs the following operations on an input string:

- If the string isn't double quoted, verifies that it uses the characters of a legal identifier. If not, an exception is thrown. If the string is double quoted, all characters are allowed.
- If the string isn't double quoted and doesn't contain periods, uppercases all alphabetic characters and eliminates leading and trailing spaces.
- If the string is double quoted and doesn't contain periods, strips off the double quotes.
- If the string contains periods and no portion of the string is double quoted, uppercases each portion of the string and encloses each portion in double quotes.
- If the string contains periods and portions of the string are double quoted, returns:
  - The double-quoted portions unchanged, including the double quotes
  - The non-double-quoted portions uppercased and enclosed in double quotes.

```sql
CANONICALIZE(<name> VARCHAR2, <canon_name> OUT VARCHAR2,
<canon_len> BINARY_INTEGER)
```

**Parameters**

- **name**
  - String to canonicalize.

- **canon_name**
  - The canonicalized string.

- **canon_len**
  - Number of bytes in name to canonicalize starting from the first character.

**Examples**

This procedure applies the CANONICALIZE procedure on its input parameter and displays the results.

```sql
CREATE OR REPLACE PROCEDURE canonicalize
(
  p_name VARCHAR2,
  p_length BINARY_INTEGER DEFAULT 30
) IS
  v_canon VARCHAR2(100);
BEGIN
  DBMS_UTILITY.CANONICALIZE(p_name, v_canon, p_length);
  DBMS_OUTPUT.PUT_LINE('Canonicalized name ==> ' || v_canon || '<<=');
END;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The `COMMA_TO_TABLE` procedure converts a comma-delimited list of names into a table of names. Each entry in the list becomes a table entry. Format the names as valid identifiers.

```
COMMA_TO_TABLE(<list> VARCHAR2, <tablen> OUT BINARY_INTEGER, <tab> OUT { LNAME_ARRAY | UNCL_ARRAY })
```

**Parameters**

- **list**
  - Comma-delimited list of names.

- **tablen**
  - Number of entries in `tab`.

- **tab**
  - Table containing the individual names in `list`.

**Parameters**

- **LNAME_ARRAY**
  - A `DBMS_UTILITY LNAME_ARRAY`, as described in `LNAME_ARRAY`.

- **UNCL_ARRAY**
  - A `DBMS_UTILITY UNCL_ARRAY`, as described in `UNCL_ARRAY`.

**Examples**
This procedure uses the `COMMA_TO_TABLE` procedure to convert a list of names to a table. It then displays the table entries.

```sql
CREATE OR REPLACE PROCEDURE comma_to_table
(p_list VARCHAR2)
IS
    r_lname DBMS_UTILITY.LNAME_ARRAY;
    v_length BINARY_INTEGER;
BEGIN
    DBMS_UTILITY.COMMA_TO_TABLE(p_list,v_length,r_lname);
    FOR i IN 1..v_length LOOP
        DBMS_OUTPUT.PUT_LINE(r_lname(i));
    END LOOP;
END;
EXEC comma_to_table('edb.dept, edb.emp, edb.jobhist')
edb.dept
edb.emp
edb.jobhist
```

### DB_VERSION

The `DB_VERSION` procedure returns the version number of the database.

```sql
DB_VERSION(<version> OUT VARCHAR2, <compatibility> OUT VARCHAR2)
```

**Parameters**

- `version`  
  Database version number.

- `compatibility`  
  Compatibility setting of the database (to be implementation-defined as to its meaning).

**Examples**

The following anonymous block displays the database version information.

```sql
DECLARE
    v_version VARCHAR2(150);
    v_compat VARCHAR2(150);
BEGIN
    DBMS_UTILITY.DB_VERSION(v_version,v_compat);
    DBMS_OUTPUT.PUT_LINE('Version: ' || v_version);
    DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END;
```

Version: PostgreSQL 15.2 (EnterpriseDB Advanced Server 15.2.0 (Debian 15.2.0-1.bullseye)) on x86_64-pc-linux-gnu, 64-bit  
Compiled by gcc (Debian 10.2.1-6) 10.2.1 20210118

Compatibility: PostgreSQL 15.2 (EnterpriseDB Advanced Server 15.2.0 (Debian 15.2.0-1.bullseye)) on x86_64-pc-linux-gnu, 64-bit  
Compiled by gcc (Debian 10.2.1-6) 10.2.1 20210118
EXEC_DDL_STATEMENT executes a DDL command.

EXEC_DDL_STATEMENT(<parse_string> VARCHAR2)

Parameters

parse_string

The DDL command to execute.

Examples

The following anonymous block creates the `job` table.

```
BEGIN
    DBMS_UTILITY.EXEC_DDL_STATEMENT(
        'CREATE TABLE job ('
        ||
        'jobno NUMBER(3),'
        ||
        'jname VARCHAR2(9))'
    );
END;
```

If the `parse_string` doesn't include a valid DDL statement, EDB Postgres Advanced Server returns an error:

```
edb# exec dbms_utility.exec_ddl_statement('select rownum from dual');
ERROR: EDB-20001: 'parse_string' must be a valid DDL statement
```

In this case, EDB Postgres Advanced Server's behavior differs from Oracle's. Oracle accepts the invalid `parse_string` without complaint.

EXPAND_SQL_TEXT

The `EXPAND_SQL_TEXT` function returns an expanded version of a given SQL query by replacing view references with its definition.

```
DBMS_UTILITY.EXPAND_SQL_TEXT(<input_sql_text> CLOB, <output_sql_text> OUT CLOB)
```

Parameters

input_sql_text

The SQL query to expand.

output_sql_text

The expanded version of the given SQL query.

Example

The following anonymous block returns an expanded version of the SQL query `SELECT * from vemp`:

```
BEGIN
    DBMS_UTILITY.expand_sql_text (
    input_sql_text => 'SELECT * from vemp',
    output_sql_text => result
    );
    DBMS_OUTPUT.put_line(result);
    SELECT empno,
    ename,
    job,
    mgr,
    hiredate,
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The `DBMS_UTILITY.FORMAT_CALL_STACK` function returns the formatted contents of the current call stack.

```
DBMS_UTILITY.FORMAT_CALL_STACK
```

You can use this function in a stored procedure, function, or package to return the current call stack in a readable format. This function is useful for debugging.

---

**FORMAT_ERROR_BACKTRACE**

The `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE` function returns the current error call stack, that is, function name and lines that lead up to the exception.

```
DBMS_UTILITY.FORMAT_ERROR_BACKTRACE
```

You can use this function in a stored procedure, function, or package to return the current error call backtrace in a readable format. This function is useful for debugging.

---

**FORMAT_ERROR_STACK**

The `DBMS_UTILITY.FORMAT_ERROR_STACK` function returns the current exception name.

```
DBMS_UTILITY.FORMAT_ERROR_STACK
```

You can use this function in a stored procedure, function, or package to return the current exception name. This function is useful for debugging.

---

**GET_CPU_TIME**

The `DBMS_UTILITY.GET_CPU_TIME` function returns the CPU time in hundredths of a second from some arbitrary point in time.

```
GET_CPU_TIME
```

**Parameters**

- `cputime`:
  - Type: `NUMBER`
  - Description: Number of hundredths of a second of CPU time.

---

*Note*

The output of the functions `DBMS_UTILITY.FORMAT_ERROR_STACK` and `DBMS_UTILITY.FORMAT_ERROR_BACKTRACE` is partially compatible with Oracle. However, it eases the migration from Oracle to EPAS.
Examples

This `SELECT` command retrieves the current CPU time, which is 603 hundredths of a second or .0603 seconds.

```
SELECT DBMS_UTILITY.GET_CPU_TIME FROM DUAL;
```

get_cpu_time
----------------
603

GET_DEPENDENCY

The `GET_DEPENDENCY` procedure lists the objects that depend on the specified object. `GET_DEPENDENCY` doesn't show dependencies for functions or procedures.

```
GET_DEPENDENCY(<type> VARCHAR2, <schema> VARCHAR2, <name> VARCHAR2)
```

Parameters

type

The object type of `name`. Valid values are `INDEX`, `PACKAGE`, `PACKAGE BODY`, `SEQUENCE`, `TABLE`, `TRIGGER`, `TYPE`, and `VIEW`.

schema

Name of the schema in which `name` exists.

name

Name of the object for which to obtain dependencies.

Examples

The following anonymous block finds dependencies on the `EMP` table:

```
BEGIN
  DBMS_UTILITY.GET_DEPENDENCY('TABLE', 'public', 'EMP');
END;
```

```
DEPENDENCIES ON public.EMP
---------------------------------------------
*TABLE public.EMP()
  + CONSTRAINT c public.emp()
  + CONSTRAINT f public.emp()
  + CONSTRAINT p public.emp()
* TYPE public.emp()
  + CONSTRAINT c public.emp()
  + CONSTRAINT f public.jobhist()
  + VIEW .empname_view()
```  

GET_HASH_VALUE

The `GET_HASH_VALUE` function computes a hash value for a given string.

```
<hash> NUMBER GET_HASH_VALUE(<name> VARCHAR2, <base> NUMBER, <hash_size> NUMBER)
```

Parameters

name

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The string for which to compute a hash value.

base

Starting value at which to generate hash values.

hash_size

The number of hash values for the desired hash table.

hash

The generated hash value.

Examples

The following anonymous block creates a table of hash values using the ename column of the emp table and then displays the key along with the hash value. The hash values start at 100 with a maximum of 1024 distinct values.

```
DECLARE
  v_hash NUMBER;
  TYPE hash_tab IS TABLE OF NUMBER INDEX BY VARCHAR2(10);
  r_hash HASH_TAB;
  CURSOR emp_cur IS SELECT ename FROM emp;
BEGIN
  FOR r_emp IN emp_cur LOOP
    r_hash(r_emp.ename) := DBMS_UTILITY.GET_HASH_VALUE(r_emp.ename, 100, 1024);
  END LOOP;
END;
```

SMITH 377
ALLEN 740
WARD 718
JONES 131
MARTIN 176
BLAKE 568
CLARK 621
SCOTT 1097
KING 235
TURNER 850
ADAMS 156
JAMES 942
FORD 775
MILLER 148

GET_PARAMETER_VALUE

The GET_PARAMETER_VALUE procedure retrieves database initialization parameter settings.

```
<status> BINARY_INTEGER GET_PARAMETER_VALUE(<parnam> VARCHAR2,
<intval> OUT INTEGER, <strval> OUT VARCHAR2)
```

Parameters

parnam

Name of the parameter whose value to return. The parameters are listed in the pg_settings system view.
Value of an integer parameter or the length of `strval`.

`strval`

Value of a string parameter.

`status`

Returns 0 if the parameter value is `INTEGER` or `BOOLEAN`. Returns 1 if the parameter value is a string.

**Examples**

The following anonymous block shows the values of two initialization parameters.

```sql
DECLARE
    v_intval INTEGER;
    v_strval VARCHAR2(80);
BEGIN
    DBMS_UTILITY.GET_PARAMETER_VALUE('max_fsm_pages', v_intval, v_strval);
    DBMS_OUTPUT.PUT_LINE('max_fsm_pages': v_intval);
    DBMS_UTILITY.GET_PARAMETER_VALUE('client_encoding', v_intval, v_strval);
    DBMS_OUTPUT.PUT_LINE('client_encoding': v_strval);
END;
```

```
max_fsm_pages: 72625
client_encoding: SQL_ASCII
```

### GET_TIME

The `GET_TIME` function returns the current time in hundredths of a second.

```sql
SELECT DBMS_UTILITY.GET_TIME FROM DUAL;
```

```
get_time
---------
1555860
```

```sql
SELECT DBMS_UTILITY.GET_TIME FROM DUAL;
```

```
get_time
---------
1556037
```

### NAME_TOKENIZE

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The `NAME_TOKENIZE` procedure parses a name into its component parts. Names without double quotes are uppercased. The double quotes are stripped from names with double quotes.

```
NAME_TOKENIZE(name> VARCHAR2, <a> OUT VARCHAR2,
<o> OUT VARCHAR2, <o> OUT VARCHAR2, <o> OUT VARCHAR2,
<nextpos> OUT BINARY_INTEGER)
```

**Parameters**

- **name**

String containing a name in the following format:

```
a[.b[.c]][@dblink ]
```

- **a**

Returns the leftmost component.

- **b**

Returns the second component, if any.

- **c**

Returns the third component, if any.

- **dblink**

Returns the database link name.

- **nextpos**

Position of the last character parsed in name.

**Examples**

This stored procedure displays the returned parameter values of the `NAME_TOKENIZE` procedure for various names.

```sql
CREATE OR REPLACE PROCEDURE name_tokenize
(
    p_name VARCHAR2
)
IS
    v_a VARCHAR2(30);
    v_b VARCHAR2(30);
    v_c VARCHAR2(30);
    v_dblink VARCHAR2(30);
    v_nextpos BINARY_INTEGER;
BEGIN
    DBMS_UTILITY.NAME_TOKENIZE(p_name, v_a, v_b, v_c, v_dblink, v_nextpos);
    DBMS_OUTPUT.PUT_LINE('name : ' || p_name);
    DBMS_OUTPUT.PUT_LINE('a      : ' || v_a);
    DBMS_OUTPUT.PUT_LINE('b      : ' || v_b);
    DBMS_OUTPUT.PUT_LINE('c      : ' || v_c);
    DBMS_OUTPUT.PUT_LINE('dblink : ' || v_dblink);
    DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
END;
```

Tokenize the name, `emp`:

```sql
BEGIN
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
name_tokenize('emp');
END;

name : emp
a : EMP
b : c :
dblink :
nextpos: 3

Tokenize the name, eadb.list_emp:

BEGIN
name_tokenize('edb.list_emp');
END;

name : eadb.list_emp
a : EDB
b : LIST_EMP
c :
dblink :
nextpos: 12

Tokenize the name, "edb."Emp_Admin.update_emp_sal:

BEGIN
name_tokenize("edb."Emp_Admin.update_emp_sal");
END;

name : "edb"."Emp_Admin.update_emp_sal
a : eadb
b : Emp_Admin
c :
UPDATE_EMP_SAL
dblink :
nextpos: 32

Tokenize the name eadb.emp@edb_dblink:

BEGIN
name_tokenize('edb.emp@edb_dblink');
END;

name : eadb.emp@edb_dblink
a : EDB
b : EMP
c :
dblink :
edb_dblink
nextpos: 18

TABLE_TO_COMMA

The `TABLE_TO_COMMA` procedure converts table of names into a comma-delimited list of names. Each table entry becomes a list entry. Format the names as valid identifiers.

TABLE_TO_COMMA(<tab> { LNAME_ARRAY | UNCL_ARRAY }
, <table> OUT BINARY_INTEGER, <list> OUT VARCHAR2)
Parameters

**tab**

Table containing names.

**LNAME_ARRAY**

A DBMS_UTILITY.LNAME_ARRAY, as described in **LNAME ARRAY**.

**UNCL_ARRAY**

A DBMS_UTILITY.UNCL_ARRAY, as described **UNCL ARRAY**.

**tablen**

Number of entries in **list**.

**list**

Comma-delimited list of names from **tab**.

Examples

This example first uses the **COMMA_TO_TABLE** procedure to convert a comma-delimited list to a table. The **TABLE_TO_COMMA** procedure then converts the table back to a comma-delimited list that it displays.

```
CREATE OR REPLACE PROCEDURE table_to_comma
(p_list VARCHAR2)
IS
    r_lname DBMS_UTILITY.LNAME_ARRAY;
    v_length BINARY_INTEGER;
    v_listlen BINARY_INTEGER;
    v_list VARCHAR2(80);
BEGIN
    DBMS_UTILITY.COMMA_TO_TABLE(p_list,v_length,r_lname);
    DBMS_OUTPUT.PUT_LINE('Table Entries: ');
    FOR i IN 1..v_length LOOP
        DBMS_OUTPUT.PUT_LINE(r_lname(i));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END;
EXEC table_to_comma('edb.dept, edb.emp, edb.jobhist')
```

**Table Entries**

```
edb.dept  
edb.emp   
edb.jobhist
```

Comma-Delimited List: edb.dept, edb.emp, edb.jobhist

14.3.3.1.20 **UTL_ENCODE**

The **UTL_ENCODE** package provides a way to encode and decode data. EDB Postgres Advanced Server supports the following functions and procedures:
<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE64_DECODE(r)</td>
<td>RAW</td>
<td>Use the BASE64_DECODE function to translate a Base64 encoded string to the original RAW value.</td>
</tr>
<tr>
<td>BASE64_ENCODE(r)</td>
<td>RAW</td>
<td>Use the BASE64_ENCODE function to translate a RAW string to an encoded Base64 value.</td>
</tr>
<tr>
<td>BASE64_ENCODE(loid)</td>
<td>TEXT</td>
<td>Use the BASE64_ENCODE function to translate a TEXT string to an encoded Base64 value.</td>
</tr>
<tr>
<td>MIMEHEADER_DECODE(buf)</td>
<td>VARCHAR</td>
<td>Use the MIMEHEADER_DECODE function to translate an encoded MIMEHEADER formatted string to its original value.</td>
</tr>
<tr>
<td>MIMEHEADER_ENCODE(buf, encode_charset, encoding)</td>
<td>VARCHAR</td>
<td>Use the MIMEHEADER_ENCODE function to convert and encode a string in MIMEHEADER format.</td>
</tr>
<tr>
<td>QUOTED_PRINTABLE_DECODE(r)</td>
<td>RAW</td>
<td>Use the QUOTED_PRINTABLE_DECODE function to translate an encoded string to a RAW value.</td>
</tr>
<tr>
<td>QUOTED_PRINTABLE_ENCODE(r)</td>
<td>RAW</td>
<td>Use the QUOTED_PRINTABLE_ENCODE function to translate an input string to a quoted-printable formatted RAW value.</td>
</tr>
<tr>
<td>TEXT_DECODE(buf, encode_charset, encoding)</td>
<td>VARCHAR</td>
<td>Use the TEXT_DECODE function to decode a string encoded by TEXT_ENCODE.</td>
</tr>
<tr>
<td>TEXT_ENCODE(buf, encode_charset, encoding)</td>
<td>VARCHAR</td>
<td>Use the TEXT_ENCODE function to translate a string to a user-specified character set, and then encode the string.</td>
</tr>
<tr>
<td>UUDECODE(r)</td>
<td>RAW</td>
<td>Use the UUDECODE function to translate a uuencode encoded string to a RAW value.</td>
</tr>
<tr>
<td>UU ENCODE(r, type, filename, permission)</td>
<td>RAW</td>
<td>Use the UU ENCODE function to translate a RAW string to an encoded uuencode value.</td>
</tr>
</tbody>
</table>

### 14.3.3.1.20.1 BASE64_DECODE

Use the BASE64_DECODE function to translate a Base64 encoded string to the original value originally encoded by BASE64_ENCODE. The signature is:

```
BASE64_DECODE (<r> IN RAW)
```

This function returns a RAW value.

**Parameters**

`r`

`r` is the string that contains the Base64 encoded data to translate to RAW form.

**Examples**

Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

This command instructs the server to escape any non-printable characters and to display BYTEA or RAW values onscreen in readable form. For more information, see the Postgres core documentation.

[https://www.postgresql.org/docs/current/static/datatype-binary.html](https://www.postgresql.org/docs/current/static/datatype-binary.html)

This example encodes and then decodes a string that contains the SQL abc. It uses BASE64_ENCODE for encoding and BASE64_DECODE for decoding.

```
edb\# SELECT UTL_ENCODE.BASE64_ENCODE(CAST ('abc' AS RAW));

    base64_encode
    ---------------------
        YWJj
    (1 row)

edb\# SELECT UTL_ENCODE.BASE64_DECODE(CAST ('YWJj' AS RAW));

    base64_decode
    ---------------------
        abc
    (1 row)
```
14.3.3.1.20.2  BASE64_ENCODE

Use the BASE64_ENCODE function to translate and encode a string in Base64 format, as described in RFC 4648. This function can be useful when composing Mime email that you intend to send using the UTL_SMTP package. The BASE64_ENCODE function has two signatures:

BASE64_ENCODE(<r>  IN  RAW)

and

BASE64_ENCODE(<loid> IN  OID)

This function returns a RAW value or an OID.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>specifies the RAW string to translate to Base64.</td>
</tr>
<tr>
<td>loid</td>
<td>specifies the object ID of a large object to translate to Base64.</td>
</tr>
</tbody>
</table>

Examples

Before executing the example, invoke the command:

```
SET bytea_output = escape;
```

This command instructs the server to escape any nonprintable characters and to display BYTEA or RAW values onscreen in readable form. For more information, see the Postgres core documentation.

This example first encodes a string that contains the text `abc` using BASE64_ENCODE and then decodes the string using BASE64_DECODE:

```
edb# SELECT UTL_ENCODE.BASE64_ENCODE(CAST ('abc' AS RAW));

base64_encode
-----------
YWJj
(1 row)
```

```
edb# SELECT UTL_ENCODE.BASE64_DECODE(CAST ('YWJj' AS RAW));

base64_decode
-----------
abc
(1 row)
```

14.3.3.1.20.3  MIMEHEADER_DECODE

Use the MIMEHEADER_DECODE function to decode values that are encoded by the MIMEHEADER_ENCODE function. The signature is:

MIMEHEADER_DECODE(<buf>  IN  VARCHAR2)

This function returns a VARCHAR2 value.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td></td>
</tr>
</tbody>
</table>
buf contains the value encoded by MIMEHEADER_ENCODE to decode.

Examples

These examples use the MIMEHEADER_ENCODE and MIMEHEADER_DECODE functions to first encode and then decode a string:

```
# Example 1
SELECT UTL_ENCODE.MIMEHEADER_ENCODE('What is the date?') FROM DUAL;
```

```
+-----------------------------------------+
| =?UTF8?Q?What is the date??             |
+-----------------------------------------+
(1 row)
```

```
# Example 2
SELECT UTL_ENCODE.MIMEHEADER_DECODE('=?UTF8?Q?What is the date??') FROM DUAL;
```

```
+-----------------------------------------+
| What is the date?                       |
+-----------------------------------------+
(1 row)
```

14.3.3.1.20.4 MIMEHEADER_ENCODE

Use the MIMEHEADER_ENCODE function to convert a string into mime header format and then encode the string. The signature is:

```
MIMEHEADER_ENCODE(<buf> IN VARCHAR2, <encode_charset> IN VARCHAR2 DEFAULT NULL, <encoding> IN INTEGER DEFAULT NULL)
```

This function returns a VARCHAR2 value.

Parameters

buf

buf contains the string to format and encode. The string is a VARCHAR2 value.

encode_charset

encode_charset specifies the character set to which to convert the string before formatting and encoding it. The default value is NULL.

encoding

encoding specifies the encoding type used when encoding the string. You can specify:

- **Q** to enable quoted-printable encoding. If you don’t specify a value, MIMEHEADER_ENCODE uses quoted-printable encoding.
- **B** to enable base-64 encoding.

Examples

These example use the MIMEHEADER_ENCODE and MIMEHEADER_DECODE functions to first encode and then decode a string:

```
# Example 1
SELECT UTL_ENCODE.MIMEHEADER_ENCODE('What is the date?') FROM DUAL;
```

```
+-----------------------------------------+
| =?UTF8?Q?What is the date??             |
+-----------------------------------------+
(1 row)
```
14.3.3.1.20.5  QUOTED_PRINTABLE_DECODE

Use the QUOTED_PRINTABLE_DECODE function to translate an encoded quoted-printable string into a decoded RAW string.

The signature is:

```
QUOTED_PRINTABLE_DECODE(<r> IN RAW)
```

This function returns a RAW value.

Parameters

- `r`

  `r` contains the encoded string to decode. The string is a RAW value encoded by QUOTED_PRINTABLE_ENCODE.

Examples

Before executing the example, invoke the command:

```
SET bytea_output = escape;
```

This command escapes any nonprintable characters and displays BYTEA or RAW values onscreen in readable form. For more information, see the Postgres core documentation.

This example first encodes and then decodes a string:

```sql
edb# SELECT UTL_ENCODE.MIMEHEADER_DECODE('=?UTF8?Q?What is the date??
FROM DUAL;
```

```
mimeheader_decode
-------------------
What is the date?
1 row
```

```sql
edb# SELECT UTL_ENCODE.QUOTED_PRINTABLE_ENCODE('E=mc2') FROM DUAL;
```

```
quoted_printable_encode
-------------------------
E=3Dmc2
1 row
```

```sql
edb# SELECT UTL_ENCODE.QUOTED_PRINTABLE_DECODE('E=3Dmc2') FROM DUAL;
```

```
quoted_printable_decode
-------------------------
E=mc2
1 row
```

14.3.3.1.20.6  QUOTED_PRINTABLE_ENCODE

Use the QUOTED_PRINTABLE_ENCODE function to translate and encode a string in quoted-printable format. The signature is:

```
QUOTED_PRINTABLE_ENCODE(<r> IN RAW)
```

This function returns a RAW value.

Parameters

- `r`

  `r` contains the encoded string to decode. The string is a RAW value encoded by QUOTED_PRINTABLE_ENCODE.
contains the string (a RAW value) to encode in a quoted-printable format.

**Examples**

Before executing the example, invoke the command:

```
SET bytea_output = escape;
```

This command escapes any nonprintable characters and displays BYTEA or RAW values onscreen in readable form. For more information, see the Postgres core documentation.

This example first encodes and then decodes a string:

```
edb# SELECT UTL_ENCODE.QUOTED_PRINTABLE_ENCODE('E=mc2') FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>quoted_printable_encode</th>
</tr>
</thead>
<tbody>
<tr>
<td>E=mc2</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

```
edb# SELECT UTL_ENCODE.QUOTED_PRINTABLE_DECODE('E=3Dmc2') FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>quoted_printable_decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>E=mc2</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

**14.3.3.1.20.7 TEXT_DECODE**

Use the TEXT_DECODE function to translate and decode an encoded string to the VARCHAR2 value that was originally encoded by the TEXT_ENCODE function. The signature is:

```
TEXT_DECODE(<buf> IN VARCHAR2, <encode_charset> IN VARCHAR2 DEFAULT NULL, <encoding> IN PLS_INTEGER DEFAULT NULL)
```

This function returns a VARCHAR2 value.

**Parameters**

- **buf**
  - contains the encoded string to translate to the original value encoded by TEXT_ENCODE.

- **encode_charset**
  - specifies the character set to which to translate the string before encoding. The default value is NULL.

- **encoding**
  - specifies the encoding type used by TEXT_DECODE. Specify:
    - UTL_ENCODE.BASE64 to specify base-64 encoding.
    - UTL_ENCODE.QUOTED_PRINTABLE to specify quoted printable encoding. This is the default.

**Examples**

This example uses the TEXT_ENCODE and TEXT_DECODE functions to first encode and then decode a string:

```
edb# SELECT UTL_ENCODE.TEXT_ENCODE('What is the date?', 'BIG5', UTL_ENCODE.BASE64) FROM DUAL;
```

```
EDB Postgres Advanced Server (EPAS)  Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.  929
```
### 14.3.3.120.8 TEXT_ENCODE

Use the `TEXT_ENCODE` function to translate a string to a user-specified character set and then encode the string. The signature is:

```
TEXT_ENCODE(<buf> IN VARCHAR2, <encode_charset> IN VARCHAR2 DEFAULT NULL, <encoding> IN PLS_INTEGER DEFAULT NULL)
```

This function returns a `VARCHAR2` value.

**Parameters**

- **buf**: contains the encoded string to translate to the specified character set and encode with `TEXT_ENCODE`.
- **encode_charset**: specifies the character set to which to translate the value before encoding. The default value is `NULL`.
- **encoding**: specifies the encoding type used by `TEXT_ENCODE`. Specify:
  - `UTL_ENCODE.BASE64` to specify base-64 encoding.
  - `UTL_ENCODE.QUOTED_PRINTABLE` to specify quoted printable encoding. This is the default.

**Examples**

This example uses the `TEXT_ENCODE` and `TEXT_DECODE` functions to first encode and then decode a string:

```sql
edb# SELECT UTL_ENCODE.TEXT_ENCODE('What is the date?', 'BIG5', UTL_ENCODE.BASE64) FROM DUAL;
```

```sql
edb# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB6aG9gZGF0ZT8=', 'BIG5', UTL_ENCODE.BASE64) FROM DUAL;
```

### 14.3.3.120.9 UUDECODE

```sql
edb# SELECT UTL_ENCODE.TEXT_ENCODE('What is the date?', 'BIG5', UTL_ENCODE.BASE64) FROM DUAL;
```

```sql
edb# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB6aG9gZGF0ZT8=', 'BIG5', UTL_ENCODE.BASE64) FROM DUAL;
```
Use the `UUDECODE` function to translate and decode a uuencode encoded string to the `RAW` value that was originally encoded by the `UUENCODE` function. The signature is:

```sql
UUDECODE(<r IN RAW>)
```

This function returns a `RAW` value.

If you’re using the EDB Postgres Advanced Server `UUDECODE` function to decode uuencoded data that was created by the Oracle implementation of the `UTL_ENCODE.UUENCODE` function, then you must first set the EDB Postgres Advanced Server configuration parameter `utl_encode.uudecode_redwood` to `TRUE` before invoking the EDB Postgres Advanced Server `UUDECODE` function on the Oracle-created data. For example, this situation might occur if you migrated Oracle tables containing uuencoded data to an EDB Postgres Advanced Server database.

The uuencoded data created by the Oracle version of the `UUENCODE` function results in a format that differs from the uuencoded data created by the EDB Postgres Advanced Server `UUENCODE` function. As a result, attempting to use the EDB Postgres Advanced Server `UUDECODE` function on the Oracle uuencoded data results in an error unless the configuration parameter `utl_encode.uudecode_redwood` is set to `TRUE`.

However, if you’re using the EDB Postgres Advanced Server `UUDECODE` function on uuencoded data created by the EDB Postgres Advanced Server `UUENCODE` function, then `utl_encode.uudecode_redwood` must be set to `FALSE`, which is the default setting.

### Parameters

- `r`

  - `r` contains the uuencoded string to translate to `RAW`.

### Examples

Before executing the example, invoke the command:

```sql
SET bytea_output = escape;
```

This command escapes any nonprintable characters and to displays `BYTEA` or `RAW` values onscreen in readable form. For more information, see the Postgres core documentation.

This example uses `UUENCODE` and `UUDECODE` to first encode and then decode a string:

```sql
edb=# SET bytea_output = escape;
edb=# SELECT UTL_ENCODE.UUENCODE('What is the date?') FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>uudecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin 0 uuencode.txt\01215VAA=&quot;!I&lt;R!T:&amp;4@9&amp;%T93<code>\012</code>\012end\012</td>
</tr>
<tr>
<td>1 row</td>
</tr>
</tbody>
</table>
```

```sql
edb=# SELECT UTL_ENCODE.UUDECODE (
     'begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93`\012`\012end\012'
   )
   FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>uudecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the date?</td>
</tr>
<tr>
<td>1 row</td>
</tr>
</tbody>
</table>
```

### 14.3.3.1.20.10 UUENCODE

Use the `UUENCODE` function to translate `RAW` data into a uuencode formatted encoded string. The signature is:

```sql
UUENCODE(<r IN RAW, <type> IN INTEGER DEFAULT 1, <filename> IN VARCHAR2 DEFAULT NULL, <permission> IN VARCHAR2 DEFAULT NULL)
```

This function returns a `RAW` value.

### Parameters
contains the RAW string to translate to uuencode format.

**type**

*type* is an INTEGER value or constant that specifies the type of uuencoded string that to return. The default value is 1. The possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>complete</td>
</tr>
<tr>
<td>2</td>
<td>header_piece</td>
</tr>
<tr>
<td>3</td>
<td>middle_piece</td>
</tr>
<tr>
<td>4</td>
<td>end_piece</td>
</tr>
</tbody>
</table>

**filename**

*filename* is a VARCHAR2 value that specifies the file name that you want to embed in the encoded form. If you don't specify a file name, UUENCODE includes a file name of uuencode.txt in the encoded form.

**permission**

*permission* is a VARCHAR2 that specifies the permission mode. The default value is NULL.

### Examples

Before executing the example, invoke the command:

```sql
SET bytea_output = escape;
```

This command escapes any nonprintable characters and displays BYTEA or RAW values onscreen in readable form. For more information, see the Postgres core documentation.

This example uses UUENCODE and UUDECODE to first encode and then decode a string:

```sql
edb=# SET bytea_output = escape;
SET bytea_output = escape;
edb=# SELECT UTL_ENCODE.UUENCODE('What is the date?') FROM DUAL;

uuencode
begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93`\012`\012end\012
(1 row)

edb=# SELECT
    UTL_ENCODE.UUDECODE
  FROM DUAL;

uuencode
What is the date?
(1 row)
```

### 14.3.3.21 UTL_FILE

The UTL_FILE package reads from and writes to files on the operating system's file system. A superuser must grant non-superusers EXECUTE privilege on the UTL_FILE package before they can use any of the functions or procedures in the package. For example, the following command grants the privilege to user mary:

```sql
GRANT EXECUTE ON PACKAGE SYS.UTL_FILE TO mary;
```

Also, the operating system username enterprisedb must have the appropriate read/write permissions on the directories and files that the UTL_FILE functions and procedures access. If the required file permissions aren't in place, an exception is thrown in the UTL_FILE function or procedure.

A handle to the file to write to or read from is used to reference the file. The file handle is defined by a public variable UTL_FILE.FILE_TYPE in the UTL_FILE package. A variable of type
FILE_TYPE must be declared to receive the file handle returned by calling the FOPEN function. The file handle is then used for all subsequent operations on the file.

References to directories on the file system are done using the directory name or alias that’s assigned to the directory using the CREATE DIRECTORY command.

The procedures and functions available in the UTL_FILE package are listed in the following table.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCLOSE(file IN OUT)</td>
<td>n/a</td>
<td>Closes the specified file identified by file.</td>
</tr>
<tr>
<td>FCLOSE_ALL</td>
<td>n/a</td>
<td>Closes all open files.</td>
</tr>
<tr>
<td>FCOPY(location, filename, dest_dir[, start_line, end_line])</td>
<td>n/a</td>
<td>Copies filename in the directory identified by location to file dest_file in directory dest_dir, from line start_line to line end_line.</td>
</tr>
<tr>
<td>FFLUSH(file)</td>
<td>n/a</td>
<td>Forces data in the buffer to be written to disk in the file identified by file.</td>
</tr>
<tr>
<td>FGETATTR(location, filename, exists, file_length, block_size)</td>
<td>n/a</td>
<td>Retrieves the attributes for a file when its filename and location are given.</td>
</tr>
<tr>
<td>FGETPOS(file)</td>
<td>INTEGER</td>
<td>Returns the relative offset position in the opened file as an integer, in bytes.</td>
</tr>
<tr>
<td>FOPEN(location, filename, open_mode[, max_linesize])</td>
<td>FILE_TYPE</td>
<td>Opens file filename in the directory identified by location.</td>
</tr>
<tr>
<td>FOPEN_NCHAR(location, filename, open_mode[, max_linesize])</td>
<td>FILE_TYPE</td>
<td>Opens file filename in the directory identified by location to read and/or write Unicode character set data.</td>
</tr>
<tr>
<td>FREMOVE(location, filename)</td>
<td>n/a</td>
<td>Removes the specified file from the file system.</td>
</tr>
<tr>
<td>FRENAMEN(location, filename, dest_dir, dest_file[, overwrite])</td>
<td>n/a</td>
<td>Renames the specified file.</td>
</tr>
<tr>
<td>FSEEK(file, absolute_offset, relative_offset)</td>
<td>n/a</td>
<td>Moves the file pointer within a given file by the number of bytes specified.</td>
</tr>
<tr>
<td>GET_LINE(file, buffer OUT)</td>
<td>n/a</td>
<td>Reads a line of text into the variable buffer from the file identified by file.</td>
</tr>
<tr>
<td>GET_RAW(file, buffer OUT[, len])</td>
<td>BYTEA</td>
<td>Reads a RAW string value from a file, keeps those into read buffer, and adjusts the file pointer accordingly by the number of bytes read, ignoring the end-of-file terminator.</td>
</tr>
<tr>
<td>IS_OPEN(file)</td>
<td>BOOLEAN</td>
<td>Determines whether the given file is open.</td>
</tr>
<tr>
<td>NEW_LINE(file[, lines])</td>
<td>n/a</td>
<td>Writes an end-of-line character sequence into the file.</td>
</tr>
<tr>
<td>PUT(file, buffer)</td>
<td>n/a</td>
<td>Writes buffer to the given file. PUT doesn't write an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT_LINE(file, buffer)</td>
<td>n/a</td>
<td>Writes buffer to the given file. An end-of-line character sequence is added by the PUT_LINE procedure.</td>
</tr>
<tr>
<td>PUTF(file, format[, arg1][,...])</td>
<td>n/a</td>
<td>Writes a formatted string to the given file. You can specify up to five substitution parameters, arg1,...,arg5, for replacement in format.</td>
</tr>
<tr>
<td>PUT_NCHAR(file, buffer)</td>
<td>n/a</td>
<td>Writes buffer to the given file and converts data to Unicode. PUT doesn't write an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT_RAW(file, buffer[, autoflush])</td>
<td>BOOLEAN</td>
<td>Accepts a RAW data value as input and writes those values to the output buffer.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server's implementation of UTL_FILE is a partial implementation when compared to Oracle's version. Only the functions and procedures listed in the table are supported.

UTL_FILE exception codes

If a call to a UTL_FILE procedure or function raises an exception, you can use the condition name to catch the exception. The UTL_FILE package reports the following exception codes compatible with Oracle databases.

<table>
<thead>
<tr>
<th>Exception code</th>
<th>Condition name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-29283</td>
<td>invalid_operation</td>
</tr>
<tr>
<td>-29285</td>
<td>write_error</td>
</tr>
<tr>
<td>-29284</td>
<td>read_error</td>
</tr>
<tr>
<td>-29282</td>
<td>invalid_filehandle</td>
</tr>
<tr>
<td>-29287</td>
<td>invalid_maxlinesize</td>
</tr>
<tr>
<td>-29281</td>
<td>invalid_mode</td>
</tr>
<tr>
<td>-29280</td>
<td>invalid_path</td>
</tr>
</tbody>
</table>

Setting file permissions with utl_file.umask
When a UTL_FILE function or procedure creates a file, the following are the default file permissions:

-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile

All permissions are denied on users belonging to the enterprisedb group as well as all other users. Only the enterprisedb user has read and write permissions on the created file.

If you want to have a different set of file permissions on files created by the UTL_FILE functions and procedures, set the utl_file.umask configuration parameter.

The utl_file.umask parameter sets the file mode creation mask (or simply the mask) in a manner similar to the Linux umask command. This parameter is for use only in the EDB Postgres Advanced Server UTL_FILE package.

Note

The utl_file.umask parameter isn't supported on Windows systems.

The value specified for utl_file.umask is a 3- or 4-character octal string that's valid for the Linux umask command. The setting determines the permissions on files created by the UTL_FILE functions and procedures. (Refer to any information source regarding Linux or Unix systems for information on file permissions and the use of the umask command.)

Example

This example sets the file permissions with utl_file.umask.

First, set up the directory in the file system for the UTL_FILE package to use. Be sure the applicable operating system account enterprisedb or postgres can read and write in the directory.

```
mkdir /tmp/utldir
chmod 777 /tmp/utldir
```

The CREATE DIRECTORY command is issued in psql to create the directory database object using the file system directory you created:

```
CREATE DIRECTORY utldir AS '/tmp/utldir';
```

Set the utl_file.umask configuration parameter. The following setting allows the file owner any permission. Group users and other users are permitted any permission except for the execute permission.

```
SET utl_file.umask TO '0011';
```

In the same session during which the utl_file.umask parameter is set to the desired value, run the UTL_FILE functions and procedures.

```
DECLARE
  v_utlfile UTL_FILE.FILE_TYPE;
  v_directory VARCHAR2(50) := 'utldir';
  v_filename VARCHAR2(20) := 'utlfile';
BEGIN
  v_utlfile := UTL_FILE.FOPEN(v_directory, v_filename, 'w');
  UTL_FILE.PUT_LINE(v_utlfile, 'Simple one-line file');
  DBMS_OUTPUT.PUT_LINE( 'Created file: ' || v_filename);
  UTL_FILE.FCLOSE(v_utlfile);
END;
```

The permission settings on the resulting file show that, in addition to the file owner, group users and other users have read and write permissions on the file.

```
$ pwd
/tmp/utldir
$ ls -l
total 4
-rw-rw-r-- 1 enterprisedb enterprisedb 21 Jul 24 16:04 utlfile
```

You can also set this parameter on a per-role basis with the ALTER ROLE command. You can set it for a single database with the ALTER DATABASE command or for the entire database server instance by setting it in the postgresql.conf file.

The FCLOSE procedure closes an open file.
FCLOSE\(<\text{file}\) IN OUT FILE_TYPE\)\)

**Parameters**

**file**

Variable of type `FILE_TYPE` containing a file handle of the file to close.

**FCLOSE_ALL**

The `FCLOSE_ALL` procedure closes all open files. The procedure executes successfully even if there are no open files to close.

**FCOPY**

The `FCOPY` procedure copies text from one file to another.

`FCOPY(\langle\text{location}\rangle \text{VARCHAR2}, \langle\text{filename}\rangle \text{VARCHAR2}, \langle\text{dest_dir}\rangle \text{VARCHAR2}, \langle\text{dest_file}\rangle \text{VARCHAR2}
\,[, \langle\text{start_line}\rangle \text{PLS_INTEGER} [, \langle\text{end_line}\rangle \text{PLS_INTEGER}] )\)

**Parameters**

**location**

Directory name of the directory containing the file to copy, as stored in `pg_catalog.edb_dir.dirname`.

**filename**

Name of the source file to copy.

**dest_dir**

Directory name of the directory to which to copy the file, as stored in `pg_catalog.edb_dir.dirname`.

**dest_file**

Name of the destination file.

**start_line**

Line number in the source file from which copying begins. The default is 1.

**end_line**

Line number of the last line in the source file to copy. If omitted or null, copying goes to the last line of the file.

**Examples**

This example makes a copy of a file `C:\TEMP\EMPDIR\empfile.csv`, which contains a comma-delimited list of employees from the `emp` table. The copy, `empcopy.csv`, is then listed.

```sql
CREATE DIRECTORY empdir AS 'C:\TEMP\EMPDIR';
DECLARE
v_empfile UTL_FILE.FILE_TYPE;
  v_src_dir VARCHAR2(50) := 'empdir';
  v_src_file VARCHAR2(20) := 'empfile.csv';
  v_dest_dir VARCHAR2(50) := 'empdir';
```
The `FFLUSH` procedure flushes unwritten data from the write buffer to the file.

\[ FFLUSH(file) \]

**Parameters**

- **file**
  
  Variable of type `FILE_TYPE` containing a file handle.

**Examples**

Each line is flushed after the `NEW_LINE` procedure is called.

```sql
DECLARE
  v_empfile UTL_FILE.FILE_TYPE;
  v_directory VARCHAR2(58) := 'empdir';
  v_filename VARCHAR2(28) := 'empfile.csv';
END;
CURSOR emp_cur IS
  SELECT * FROM emp ORDER BY empno;
```

```
7369, SMITH, CLERK, 7082, 17-DEC-88, 800, 20
7499, ALLEN, SALESMAN, 7698, 28-FEB-81, 1600, 30
7521, WARD, SALESMAN, 7698, 22-FEB-81, 1250, 30
7566, JONES, MANAGER, 7839, 02-APR-81, 2975, 0
7572, CLARK, MANAGER, 7839, 09-SEP-81, 2450, 20
7580, SCOTT, ANALYST, 7698, 19-APR-87, 3000, 0
7654, MARTIN, SALESMAN, 7698, 28-SEP-81, 1250, 30
7698, BLAKE, MANAGER, 7839, 01-MAY-81, 2850, 20
7782, CLARK, MANAGER, 7839, 09-JUN-81, 2450, 10
7839, KING, PRESIDENT, 17-NOV-81, 5000, 10
7844, TURNER, SALESMAN, 7698, 08-SEP-81, 1500, 0
7876, ADAMS, CLERK, 7788, 23-MAY-87, 1100, 20
7900, JAMES, CLERK, 7698, 03-DEC-81, 950, 30
7902, FORD, ANALYST, 7566, 83-DEC-81, 3008, 20
7934, MILLER, CLERK, 7782, 23-JAN-82, 1308, 10
```

```
14 records retrieved
```

**The following is the destination file,**

'empcopy.csv'

```sql
v_dest_file VARCHAR2(20) := 'empcopy.csv';
v_emprec VARCHAR2(120);
v_count INTEGER := 0;
BEGIN
  UTL_FILE.FCOPY(v_src_dir,v_src_file,v_dest_dir,v_dest_file);
  v_empfile := UTL_FILE.FOPEN(v_dest_dir,v_dest_file,'r');
  DBMS_OUTPUT.PUT_LINE('The following is the destination file, ''
                        'empcopy.csv'' ;
  v_emprec := UTL_FILE.GET_LINE(v_empfile,v_emprec);
  DBMS_OUTPUT.PUT_LINE(v_emprec);
  v_count := v_count + 1;
END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    UTL_FILE.FCLOSE(v_empfile);
    DBMS_OUTPUT.PUT_LINE(v_count || ' records retrieved');
  WHEN OTHERS
    DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
    DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;
```

```
The following is the destination file, 
'empcopy.csv'
7369, SMITH, CLERK, 7082, 17-DEC-88, 800, 20
7499, ALLEN, SALESMAN, 7698, 28-FEB-81, 1600, 30
7521, WARD, SALESMAN, 7698, 22-FEB-81, 1250, 30
7566, JONES, MANAGER, 7839, 02-APR-81, 2975, 0
7572, CLARK, MANAGER, 7839, 09-SEP-81, 2450, 20
7580, SCOTT, ANALYST, 7698, 19-APR-87, 3000, 0
7654, MARTIN, SALESMAN, 7698, 28-SEP-81, 1250, 30
7698, BLAKE, MANAGER, 7839, 01-MAY-81, 2850, 20
7782, CLARK, MANAGER, 7839, 09-JUN-81, 2450, 10
7839, KING, PRESIDENT, 17-NOV-81, 5000, 10
7844, TURNER, SALESMAN, 7698, 08-SEP-81, 1500, 0
7876, ADAMS, CLERK, 7788, 23-MAY-87, 1100, 20
7900, JAMES, CLERK, 7698, 03-DEC-81, 950, 30
7902, FORD, ANALYST, 7566, 83-DEC-81, 3008, 20
7934, MILLER, CLERK, 7782, 23-JAN-82, 1308, 10
14 records retrieved
```
The `FGETATTR` function retrieves the attributes for a given file. When you enter `filename` and `location` values, it returns whether the file exists. If the file exists, it also returns `file_length` and `block_size` values. If the file doesn't exist or an error occurs, the attributes return as `NULL`.

```
BEGIN
  v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
  FOR i IN emp_cur LOOP
    UTL_FILE.PUT(v_empfile,i.empno);
    UTL_FILE.PUT(v_empfile,',');
    UTL_FILE.PUT(v_empfile,i.ename);
    UTL_FILE.PUT(v_empfile,',');
    UTL_FILE.PUT(v_empfile,i.job);
    UTL_FILE.PUT(v_empfile,',');
    UTL_FILE.PUT(v_empfile,i.mgr);
    UTL_FILE.PUT(v_empfile,',');
    UTL_FILE.PUT(v_empfile,i.hiredate);
    UTL_FILE.PUT(v_empfile,',');
    UTL_FILE.PUT(v_empfile,i.sal);
    UTL_FILE.PUT(v_empfile,',');
    UTL_FILE.PUT(v_empfile,i.comm);
    UTL_FILE.PUT(v_empfile,',');
    UTL_FILE.PUT(v_empfile,i.deptno);
    UTL_FILE.PUT(v_empfile,'
    UTL_FILE.NEW_LINE(v_empfile);
    UTL_FILE.PPRINT(v_empfile);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
  UTL_FILE.FCLOSE(v_empfile);
END;
```

**FGETATTR**

The `FGETATTR` function retrieves the attributes for a given file. When you enter `filename` and `location` values, it returns whether the file exists. If the file exists, it also returns `file_length` and `block_size` values. If the file doesn't exist or an error occurs, the attributes return as `NULL`.

```
FGETATTR(<location> VARCHAR2, <filename> VARCHAR2, <exists> BOOLEAN, <file_length> NUMBER, <block_size> BINARY_INTEGER)
```

**Parameters**

- `location`  
  Name of the directory containing the file whose attributes are being retrieved, stored in `pg_catalog.edb_dir.dirname`.

- `filename`  
  Name of the file whose attributes are being retrieved.

- `exists`  
  Report of whether the file exists.

- `file_length`  
  Size of the file in bytes.

- `block_size`  
  System block size of the file in bytes.

**Examples**

This example retrieves the attributes for the file `empfile.csv`:

```
DECLARE
  l_file_exists BOOLEAN;
  l_file_len NUMBER;
  l_blocksize BINARY_INTEGER;
  f utl_file.file_type;
```
The **FGETPOS** function returns the relative offset position in the opened file as an integer, in bytes.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>file</strong></td>
<td>Variable of type <strong>FILE_TYPE</strong></td>
<td>containing the file handle of the opened file</td>
</tr>
</tbody>
</table>

### Examples

Thus example finds the position of the file `empfile.csv`:

```sql
DECLARE
  f utl_file.file_type;
  v_empfile UTILITY.FILE_TYPE;
  v_directory VARCHAR2(50) := 'empdir';
  v_filename VARCHAR2(28) := 'empfile.csv';
BEGIN
  f := utl_file.fopen(v_directory, v_filename, 'r');
  utl_file.fseek(f, 15);
  pos := utl_file.fgetpos(f);
  dbms_output.put_line('1.position: ' || pos);
  utl_file.fseek(f, NULL, -5);
  pos := utl_file.fgetpos(f);
  dbms_output.put_line('2.position: ' || pos);
  utl_file.fcose(f);
END;
```

1. position: 15
2. position: 10
FOPEN

The FOPEN function opens a file for I/O.

```
FILE_TYPE FOPEN(location VARCHAR2, filename VARCHAR2, open_mode VARCHAR2 [, max_linesize BINARY_INTEGER ])"'
```

Parameters

**location**

Directory name of the directory containing the file to open, as stored in `pg_catalog.edb_dir.dirname`.

**filename**

Name of the file to open.

**open_mode**

Mode in which to open the file. Modes are:

- `'a'` &mdash; Append to file.
- `'r'` &mdash; Read from file.
- `'w'` &mdash; Write to file.

**max_linesize**

Maximum size of a line in characters. In read mode, an exception is thrown if you try to read a line exceeding `max_linesize`. In write and append modes, an exception is thrown if you try to write a line exceeding `max_linesize`. The end-of-line characters aren't included in determining if the maximum line size is exceeded. This behavior isn't compatible with Oracle databases. Oracle counts the end-of-line characters.

**filetype**

Variable of type `FILE_TYPE` containing the file handle of the opened file.

FOPEN_NCHAR

The FOPEN_NCHAR procedure opens the file `filename` in the directory identified by `location` to read and/or write Unicode character set data.

```
FOPEN_NCHAR(location VARCHAR2, filename VARCHAR2, open_mode VARCHAR2, max_linesize INTEGER)
```

Parameters

**location**

Name of the directory containing the file to open, stored in `pg_catalog.edb_dir.dirname`.

**filename**

Name of the file to open.

**open_mode**

Mode in which to open the file. Modes are:

- `'a'` &mdash; Append to file.
- `'r'` &mdash; Read from file.
- `'w'` &mdash; Write to file.

**max_linesize**

Maximum size of a line in characters. In read mode, an exception is thrown if you try to read a line exceeding `max_linesize`. In write and append modes, an exception is thrown if you try to write a line exceeding `max_linesize`. The end-of-line characters aren't included in determining if the maximum line size is exceeded. This behavior isn't compatible with Oracle databases. Oracle counts the end-of-line characters.
The end-of-line characters aren’t included in determining if the maximum line size is exceeded. This behavior isn’t compatible with Oracle databases. Oracle counts the end-of-line characters.

Examples

Thus example opens the file `empfile1.csv`:

```sql
declare
    f utl_file.file_type;
    t text;
    v_empfile UTL_FILE.FILE_TYPE;
    v_directory VARCHAR2(58) := 'empdir';
    v_filename VARCHAR2(28) := 'empfile1.csv';
    txt1 text;
begin
    f := utl_file.fopen_nchar(v_directory, v_filename, 'w');
    utl_file.put_nchar(f, 'Hello - 1' :: text);
    utl_file.put_nchar(f, 100 :: numeric);
    utl_file.put_nchar(f, '2006-08-13 12:34:56' :: timestamp);
    utl_file.put_nchar(f, '2001-12-27 04:05:06.789-08' :: time);
    utl_file.fclose(f);

    f := utl_file.fopen_nchar(v_directory, v_filename, 'r');
    loop
        utl_file.get_line_nchar(f, txt1);
        raise notice '%', txt1;
    end loop;
    exception
    when no_data_found then
        raise notice 'finish % ', sqlerrm;
    utl_file.fclose(f);
end;
```

FREMOVE

The `FREMOVE` procedure removes a file from the system.

```
FREMOVE(<location> VARCHAR2, <filename> VARCHAR2)
```

An exception is thrown if the file doesn’t exist.

Parameters

- **location**
  
  Directory name of the directory containing the file to remove, as stored in `pg_catalog.edb_dir.dirname`.

- **filename**
  
  Name of the file to remove.

Examples
This example removes the file `empfile.csv`:

```sql
DECLARE
    v_directory VARCHAR2(50) := 'empdir';
    v_filename VARCHAR2(20) := 'empfile.csv';
BEGIN
    UTL_FILE.FREMOVE(v_directory,v_filename);
    DBMS_OUTPUT.PUT_LINE('Removed file: ' || v_filename);
EXCEPTION
    WHEN OTHERS
    THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;
```

Removed file: `empfile.csv`

---

**FRENAME**

The `FRENAME` procedure renames a file, effectively moving a file from one location to another.

```sql
FRENAME(location> VARCHAR2, <filename> VARCHAR2, <dest_dir> VARCHAR2, <dest_file> VARCHAR2, [ <overwrite> BOOLEAN ]
```

**Parameters**

- **location**
  - Directory name of the directory containing the file to rename, as stored in `pg_catalog.edb_dir.dirname`.

- **filename**
  - Name of the source file to rename.

- **dest_dir**
  - Directory name of the directory to which to locate the renamed file, as stored in `pg_catalog.edb_dir.dirname`.

- **dest_file**
  - New name of the file.

- **overwrite**
  - Replaces any existing file named `dest_file` in `dest_dir` if set to `TRUE`. An exception is thrown if set to `FALSE` (the default).

**Examples**

This example renames a file, `C:/TEMP/EMPDIR/empfile.csv`, containing a comma-delimited list of employees from the `emp` table. The renamed file, `C:/TEMP/NEWDIR/newemp.csv`, is then listed.

```sql
CREATE DIRECTORY "newdir" AS 'C:/TEMP/NEWDIR';
DECLARE
    v_empfile UTL_FILE.FILE_TYPE;
    v_src_dir VARCHAR2(50) := 'empdir';
    v_src_file VARCHAR2(20) := 'empfile.csv';
    v_dest_dir VARCHAR2(50) := 'newdir';
    v_dest_file VARCHAR2(50) := 'newemp.csv';
    v_replace BOOLEAN := FALSE;
    v_emprec VARCHAR2(120);
```

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The **FSEEK** procedure moves the file pointer within a given file by the number of bytes specified.

**FSEEK**(`<file>` FILETYPE, `<absolute_offset>` INTEGER, `<relative_offset>` INTEGER)

**Parameters**

- **file**
  Variable of type FILETYPE containing the file handle of the file where the file pointer is being moved.

- **absolute_offset**
  The absolute number of bytes to move the file pointer.

- **relative_offset**
  The relative number of bytes to move the file pointer.

**Examples**

This example moves the file pointer in the file `empfile.csv`:

```sql
declare
v_count INTEGER := 0;
BEGIN
UTL_FILE.FRENAME(v_src_dir,v_src_file,v_dest_dir,v_dest_file,v_replace);
v_empfile := UTL_FILE.FOPEN(v_dest_dir,v_dest_file,'r');
DBMS_OUTPUT.PUT_LINE('The following is the renamed file, ...');
BEGIN
v_dest_file := v_dest_dir || v_dest_file;
LOOP
UTL_FILE.GET_LINE(v_empfile,v_emprec);
DBMS_OUTPUT.PUT_LINE(v_emprec);
v_count := v_count + 1;
END LOOP;
DBMS_OUTPUT.PUT_LINE(v_count || ' records retrieved');
EXCEPTION
WHEN NO_DATA_FOUND THEN
UTL_FILE.FCLOSE(v_empfile);
DBMS_OUTPUT.PUT_LINE(v_count || ' records retrieved');
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;
END;
The following is the renamed file, 'newemp.csv'
7369,SMITH,CLERK,7082,17-DEC-88 00:00:00,800.00,20
7499,ALLEN,SALESMAN,7698,28-FEB-81 00:00:00,1600.00,300.00,30
7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
7566,JONES,MANAGER,7839,2-APR-81 00:00:00,2975.00,30
7584,MARTIN,SALESMAN,7698,26-SEP-81 00:00:00,1400.00,30
7695,BLAKE,MANAGER,7839,8-MAY-81 00:00:00,2850.00,30
7782,CLARK,MANAGER,7839,9-JUN-81 00:00:00,2450.00,10
7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,20
7839,KING,PRESIDENT,17-NOV-81 00:00:00,5000.00,10
7844,TURNER,SALESMAN,7698,8-SEP-81 00:00:00,1500.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,20
7900,JAMES,CLERK,7698,15-DEC-81 00:00:00,950.00,30
7922,FORD,ANALYST,7566,8-DEC-81 00:00:00,3000.00,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,10
14 records retrieved
The `GET_LINE` procedure reads a line of text from a given file up to but not including the end-of-line terminator. A `NO_DATA_FOUND` exception is thrown when there are no more lines to read.

### Parameters

- **file**
  
  Variable of type `FILE_TYPE` containing the file handle of the opened file.

- **buffer**
  
  Variable to receive a line from the file.

### Examples

The following anonymous block reads through and displays the records in file `empfile.csv`.

```sql
DECLARE
  v_empfile UTL_FILE.FILE_TYPE;
  v_directory VARCHAR2(50) := 'empdir';
  v_filename VARCHAR2(20) := 'empfile.csv';
  v_emprec VARCHAR2(128);
  v_count INTEGER := 0;
BEGIN
  v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'r');
  LOOP
    UTL_FILE.GET_LINE(v_empfile,v_emprec);
    DBMS_OUTPUT.PUT_LINE(v_emprec);
    v_count := v_count + 1;
  END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    UTL_FILE.FCLOSE(v_empfile);
    DBMS_OUTPUT.PUT_LINE('End of file ' || v_filename || ' - ' || v_count || ' records retrieved');
  WHEN OTHERS
  THEN
```

---

**Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.**
The `GET_RAW` procedure reads a `RAW` string value from a file, keeps those into read buffer, and adjusts the file pointer accordingly by the number of bytes read. `GET_RAW` ignores the end-of-file terminator. `INVALID_FILEHANDLE`, `INVALID_OPERATION`, and `READ_ERROR` exceptions are thrown when there are no more lines to read.

```
GET_RAW(<file> FILE_TYPE, <buffer> OUT BYTEA [, <len> INTEGER ])
```

**Parameters**

- **file**
  
  Variable of type `FILE_TYPE` containing the file handle of the opened file.

- **buffer**
  
  Assign `RAW` data from the file to the read buffer.

- **len**
  
  The number of bytes read from a file. Default is `NULL`. If `NULL`, `len` tries to read a maximum of 32767 `RAW` bytes.

**Examples**

This example attempts to read a `RAW` string value from the file.

```
CREATE DIRECTORY empdir AS '/TMP/EMPDIR';

CREATE or REPLACE FUNCTION read_bin_file() RETURN void AS

DECLARE
  v_tempfile UTL_FILE.FILE_TYPE;
  v_filename VARCHAR2(28) := 'sample.png';
  v_temprec BYTEA;
  v_count INTEGER := 0;
BEGIN
  v_tempfile := UTL_FILE.FOPEN('empdir', v_filename, 'rb');
  UTL_FILE.GET_RAW(v_tempfile, v_temprec);
  INSERT INTO emp VALUES (1, v_temprec);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE NOTICE 'Finish %', SQLERRM;
END;
```

End of file empfile.csv - 14 records retrieved
**IS_OPEN**

The **IS_OPEN** function determines whether a file is open.

```sql
BOOLEAN IS_OPEN(<file> FILE_TYPE)
```

**Parameters**

- **file**
  Variable of type **FILE_TYPE** containing the file handle of the file to test.

- **status**
  **TRUE** if the file is open, **FALSE** otherwise.

**NEW_LINE**

The **NEW_LINE** procedure writes an end-of-line character sequence in the file.

```sql
NEW_LINE(<file> FILE_TYPE [, <lines> INTEGER ])
```

**Parameters**

- **file**
  Variable of type **FILE_TYPE** containing the file handle of the file to which to write end-of-line character sequences.

- **lines**
  Number of end-of-line character sequences to write. The default is 1.

**Examples**

This example writes a file containing a double-spaced list of employee records.

```sql
DECLARE
  v_empfile UTL_FILE.FILE_TYPE;
  v_directory VARCHAR2(50) := 'empdir';
  v_filename VARCHAR2(20) := 'empfile.csv';
CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
  v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
  FOR i IN emp_cur LOOP
    UTL_FILE.PUT(v_empfile,i.empno);
    UTL_FILE.PUT(v_empfile,'\n');
    UTL_FILE.PUT(v_empfile,i.ename);
    UTL_FILE.PUT(v_empfile,'\n');
    UTL_FILE.PUT(v_empfile,i.job);
END LOOP;
  UTL_FILE.FCLOSE(v_empfile);
END;
```
The PUT procedure writes a string to the given file. No end-of-line character sequence is written at the end of the string. Use the NEW_LINE procedure to add an end-of-line character sequence.

**Parameters**

$file$

Variable of type FILE_TYPE containing the file handle of the file to which to write the given string.

$buffer$

Text to write to the specified file.

**Examples**

```sql
UTL_FILE.PUT(v_empfile,'C:\TEMP\EMPDIR');
UTL_FILE.PUT(v_empfile,'empfile.csv');
UTL_FILE.PUT(v_empfile,'7369,SMITH,CLERK,7902,17-DEC-88 00:00:00,800.00,.,28');
UTL_FILE.PUT(v_empfile,'7499,ALLEN,SALESMAN,7698,28-FEB-81 00:00:00,1600.00,300.00,30');
UTL_FILE.PUT(v_empfile,'7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.80,500.80,30');
UTL_FILE.PUT(v_empfile,'7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.80,.,20');
UTL_FILE.PUT(v_empfile,'7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30');
UTL_FILE.PUT(v_empfile,'7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.80,.,30');
UTL_FILE.PUT(v_empfile,'7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.80,.,10');
UTL_FILE.PUT(v_empfile,'7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,.,20');
UTL_FILE.PUT(v_empfile,'7839,KING,PRESIDENT,17-NOV-81 00:00:00,5000.00,.,10');
UTL_FILE.PUT(v_empfile,'7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,8.00,30');
UTL_FILE.PUT(v_empfile,'7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,.,20');
UTL_FILE.PUT(v_empfile,'7980,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,.,30');
UTL_FILE.PUT(v_empfile,'7982,FORD,ANALYST,7566,03-DEC-81 00:00:00,3080.00,.,20');
UTL_FILE.PUT(v_empfile,'7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,.,10');
UTL_FILE.NEW_LINE(v_empfile,2);
```

This file is then displayed:

```
C:\TEMP\EMPDIR\TYPE
empfile.csv
7369,SMITH,CLERK,7902,17-DEC-88 00:00:00,800.00,.,28
7499,ALLEN,SALESMAN,7698,28-FEB-81 00:00:00,1600.00,300.00,30
7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.80,500.80,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.80,.,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.80,.,30
7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.80,.,10
7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,.,20
7839,KING,PRESIDENT,17-NOV-81 00:00:00,5000.00,.,10
7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,8.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,.,20
7980,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,.,30
7982,FORD,ANALYST,7566,03-DEC-81 00:00:00,3080.00,.,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,.,10
```
This example uses the **PUT** procedure to create a comma-delimited file of employees from the **emp** table.

```sql
DECLARE
  v_empfile   UTL_FILE.FILE_TYPE;
  v_directory VARCHAR2(50) := 'empdir';
  v_filename  VARCHAR2(20) := 'empfile.csv';
CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
  v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
  FOR i IN emp_cur LOOP
    UTL_FILE.PUT(v_empfile,i.empno);
    UTL_FILE.PUT(v_empfile,i.ename);
    UTL_FILE.PUT(v_empfile,i.job);
    UTL_FILE.PUT(v_empfile,i.comm);
    UTL_FILE.PUT(v_empfile,i.hiredate);
    UTL_FILE.PUT(v_empfile,i.sal);
    UTL_FILE.PUT(v_empfile,i.deptno);
  END LOOP;
  UTL_FILE.FCLOSE(v_empfile);
END;
```

The following are the contents of **empfile.csv**:

```sql
C:\TEMP\EMPDIR\TYPE
empfile.csv

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>comm</th>
<th>hiredate</th>
<th>sal</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td></td>
<td>00-00-00</td>
<td>1300.00</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>1600.00</td>
<td>00-00-00</td>
<td>3000.00</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>500.00</td>
<td>00-00-00</td>
<td>2000.00</td>
</tr>
<tr>
<td>7556</td>
<td>JONES</td>
<td>MANAGER</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>2500.00</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>1600.00</td>
<td>00-00-00</td>
<td>2300.00</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>3000.00</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>2500.00</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>3000.00</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>5000.00</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>2300.00</td>
</tr>
<tr>
<td>8768</td>
<td>ADAMS</td>
<td>MANAGER</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>3000.00</td>
</tr>
<tr>
<td>8790</td>
<td>JAMES</td>
<td>CLERK</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>2000.00</td>
</tr>
<tr>
<td>7982</td>
<td>FORD</td>
<td>ANALYST</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>2000.00</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>1500.00</td>
<td>00-00-00</td>
<td>1300.00</td>
</tr>
</tbody>
</table>

**PUT_LINE**

The **PUT_LINE** procedure writes a single line to a file, including an end-of-line character sequence.

```sql
PUT_LINE(file TYPE, <buffer> { DATE | NUMBER | TIMESTAMP | VARCHAR2 })
```

Parameters

- **file**
  - Variable of type **FILE_TYPE** containing the file handle of the file to which write the line.
Examples

This example uses the `PUT_LINE` procedure to create a comma-delimited file of employees from the `emp` table.

```sql
DECLARE
  v_empfile UTL_FILE.FILE_TYPE;
  v_directory VARCHAR2(80) := 'empdir';
  v_filename VARCHAR2(28) := 'empfile.csv';
  v_emprec VARCHAR2(128);
BEGIN
  CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
  LOOP
    v_empfile := UTL_FILE.FOPEN(v_directory, v_filename, 'w');
    FOR i IN emp_cur LOOP
      i.empno || ',', i.ename || ',', i.job || ',', NVL(LTRIM(TO_CHAR(i.hiredate, 'DD-MON-RRR9999')),'') || ',', NVL(LTRIM(TO_CHAR(i.sal, '9999.99')),'') || ',', NVL(LTRIM(TO_CHAR(i.comm, '9999.99')),'') || ',', i.deptno;
      UTL_FILE.PUT_LINE(v_empfile, v_emprec);
    END LOOP;
  END LOOP;
  UTL_FILE.FCLOSE(v_empfile);
END;
```

The following are the contents of `empfile.csv`:

```
C:\TEMP\EMPDIR\TYPE
empfile.csv
7369,SMITH,CLERK,7082,17-DEC-88 00:00:00,3000.00,10,7839,MANAGER,7082,01-MAY-81 00:00:00,2450.00,10,7782,CLARK,MANAGER,7082,09-JUN-81 00:00:00,2450.00,10,7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,20,7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,10,7844,TURNER,SALESMAN,7604,08-SEP-81 00:00:00,1500.00,5,7786,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1300.00,20,7908,JAMES,CLERK,7604,01-DEC-81 00:00:00,950.00,30,7982,FORD,ANALYST,7566,83-DEC-81 00:00:00,3000.00,20,7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,10
```

The `PUTF` procedure writes a formatted string to a file.

```sql
PUTF(file FILE_TYPE, format VARCHAR2 [, arg1 VARCHAR2] [, ...])
```

**Parameters**

- **file**

  Variable of type `FILE_TYPE` containing the file handle of the file to which to write the formatted line.

- **format**

  String to format the text written to the file. The special character sequence `\n` is substituted by the value of `arg`. The special character sequence `\n` indicates a new line. In EDB Postgres Advanced Server, specify a new line character with two consecutive backslashes instead of one: `\n`. This characteristic isn't compatible with Oracle databases.
Up to five arguments, \texttt{arg1,...,arg5}, to substitute in the format string for each occurrence of \texttt{%s}. The first arg is substituted for the first occurrence of \texttt{%s}, the second arg is substituted for the second occurrence of \texttt{%s}, and so on.

Examples

The following anonymous block produces formatted output containing data from the \texttt{emp} table.

Note

The E literal syntax and double backslashes for the new-line character sequence in the format string aren't compatible with Oracle databases.

```sql
DECLARE
  v_empfile UTL_FILE.FILE_TYPE;
  v_directory VARCHAR2(50) := 'empdir';
  v_filename VARCHAR2(20) := 'empfile.csv';
  v_format VARCHAR2(200);
  CURSOR emp_cur IS
    SELECT *
    FROM emp
    ORDER BY empno;
BEGIN
  v_format := E'%s %s, %s
Salary: $%s Commission: $%s
' ||
  FOR i IN emp_cur LOOP
    UTL_FILE.PUTF(v_empfile,v_format,i.empno,i.ename,i.job,i.sal,
    NVL(i.comm,0));
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Created file: ' ||
    v_filename);
  UTL_FILE.FCLOSE(v_empfile);
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE('SQLERRM: ' ||
      SQLERRM);
    DBMS_OUTPUT.PUT_LINE('SQLCODE: ' ||
      SQLCODE);
  END;
END;
```

The following are the contents of \texttt{empfile.csv}:

```
C:\TEMP\EMPDIR\TYPE
empfile.csv
7369 SMITH, CLERK
Salary: $800.00 Commission: $0
7499 ALLEN, SALESMAN
Salary: $1600.00 Commission: $300.00
7521 WARD, SALESMAN
Salary: $1250.00 Commission: $500.00
7566 JONES, MANAGER
Salary: $2975.00 Commission: $0
7654 MARTIN, SALESMAN
Salary: $1250.00 Commission: $1400.00
7698 BLAKE, MANAGER
Salary: $2850.00 Commission: $0
7782 CLARK, MANAGER
Salary: $2450.00 Commission: $0
7708 SCOTT, ANALYST
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
PUT_NCHAR

The PUT_NCHAR procedure writes buffer to the given file and converts data to Unicode. PUT doesn't write an end-of-line character sequence.

PUT_NCHAR(<file> FILE_TYPE, <buffer> NVARCHAR2)

Parameters

file

Variable of type FILE_TYPE containing the file handle of the file to which to write the line.

buffer

Variable to write RAW data to the output buffer.

Examples

This example writes buffer to the file empfile.csv and converts data to Unicode:

```sql
DECLARE
  f utl_file.file_type;
  text;
  v_empfile utl_file.FILE_TYPE;
    v_directory VARCHAR2(50) := 'empdir';
    v_filename VARCHAR2(20) := 'empfile1.csv';
  txt1 text;
BEGIN
  f := utl_file.fopen_nchar(v_directory, v_filename, 'w');
  utl_file.put_nchar(f, 'Hello - 1':text);
  utl_file.put_nchar(f, 100::numeric);
  utl_file.put_nchar(f, '2006-08-13 12:34:56':timestamp);
  utl_file.put_nchar(f, '2001-12-27 04:05:06.789-08':pg_catalog.date);
  utl_file.put_nchar(f, '2001-12-27 04:05:06.789-08':time);
  utl_file.fclose(f);
  f := utl_file.fopen_nchar(v_directory, v_filename, 'r');
  loop
    -- Read data
  end loop;
  utl_file.fclose(f);
END;
```

Salary: $3000.00 Commission: 
7839 KING, PRESIDENT
Salary: $5000.00 Commission: 
7844 TURNER, SALESMAN
Salary: $1500.00 Commission: 
7876 ADAMS, CLERK
Salary: $1100.00 Commission: 
7900 JAMES, CLERK
Salary: $950.00 Commission: 
7982 FORD, ANALYST
Salary: $3000.00 Commission: 
7934 MILLER, CLERK
Salary: $1300.00 Commission:
The **PUT_RAW** procedure accepts a RAW data value and writes those values to the output buffer. **INVALID_FILEHANDLE**, **INVALID_OPERATION**, **WRITE_ERROR**, and **VALUE_ERROR** exceptions are thrown when there are no more lines to read.

**PUT_RAW**

The **PUT_RAW** procedure accepts a RAW data value and writes those values to the output buffer. **INVALID_FILEHANDLE**, **INVALID_OPERATION**, **WRITE_ERROR**, and **VALUE_ERROR** exceptions are thrown when there are no more lines to read.

**PUT_RAW]**<file> FILE_TYPE, <buffer> BYTEA [, <autoflush> BOOLEAN ]

**Parameters**

**file**

Variable of type **FILE_TYPE** containing the file handle of the file to which to write the line.

**buffer**

Variable to write RAW data to the output buffer.

**autoflush**

If **TRUE**, performs a flush after writing the value to the output buffer. By default, **autoflush** is **FALSE**.

**Examples**

This example writes the RAW data value from the **emp** table.

```sql
CREATE or REPLACE FUNCTION write_bin_file() RETURN void AS
DEKLARATION
v_tempfile
UTL_FILE.FILE_TYPE;
  v_filename
VARCHAR2(20) := 'sample.png';
  v_temprec
BYTEA;
BEGIN
SELECT imagerawdata INTO v_temprec from emp;
  v_tempfile := UTL_FILE.FOPEN('empdir', v_filename,
    'wb');</n>
  UTL_FILE.PUT_RAW(v_tempfile,v_temprec, TRUE);
  UTL_FILE.FCLOSE(v_tempfile);
END;
edb=# SELECT write_bin_file();

write_bin_file
-------------
(1 row)

**14.3.3.22** **UTL_HTTP**

The **UTL_HTTP** package provides a way to use the HTTP or HTTPS protocol to retrieve information found at a URL. EDB Postgres Advanced Server supports the following functions and procedures.
### UTL_HTTP Functions and Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_REQUEST(url, method, http_version)</td>
<td>UTL_HTTP.REQ</td>
<td>Initiates a new HTTP request.</td>
</tr>
<tr>
<td>END_REQUEST(r IN OUT)</td>
<td>n/a</td>
<td>Ends an HTTP request before allowing it to complete.</td>
</tr>
<tr>
<td>END_RESPONSE(r IN OUT)</td>
<td>n/a</td>
<td>Ends the HTTP response.</td>
</tr>
<tr>
<td>END_OF_BODY(r IN OUT)</td>
<td>n/a</td>
<td>Ends package body.</td>
</tr>
<tr>
<td>GET_BODY_CHARSET</td>
<td>VARCHAR2</td>
<td>Returns the default character set of the body of future HTTP requests.</td>
</tr>
<tr>
<td>GET_BODY_CHARSET(charset OUT)</td>
<td>n/a</td>
<td>Returns the default character set of the body of future HTTP requests.</td>
</tr>
<tr>
<td>GET_FOLLOW_REDIRECT(max_redirects OUT)</td>
<td>n/a</td>
<td>Current setting for the maximum number of redirections allowed.</td>
</tr>
<tr>
<td>GET_HEADER(r IN OUT, n, name OUT, value OUT)</td>
<td>n/a</td>
<td>Returns the nth header of the HTTP response.</td>
</tr>
<tr>
<td>GET_HEADER_BY_NAME(r IN OUT, name, value OUT, n)</td>
<td>n/a</td>
<td>Returns the HTTP response header for the specified name.</td>
</tr>
<tr>
<td>GET_HEADER_COUNT(r IN OUT)</td>
<td>INTEGER</td>
<td>Returns the number of HTTP response headers.</td>
</tr>
<tr>
<td>GET_RESPONSE(r IN OUT)</td>
<td>UTL_HTTP.RESP</td>
<td>Returns the HTTP response.</td>
</tr>
<tr>
<td>GET_RESPONSE_ERROR_CHECK(enable OUT)</td>
<td>n/a</td>
<td>Returns whether or not response error check is set.</td>
</tr>
<tr>
<td>GET_TRANSFER_TIMEOUT(timeout OUT)</td>
<td>n/a</td>
<td>Returns the transfer timeout setting for HTTP requests.</td>
</tr>
<tr>
<td>READ_LINE(r IN OUT, data OUT, remove_crlf)</td>
<td>n/a</td>
<td>Returns the HTTP response body in text form until the end of line.</td>
</tr>
<tr>
<td>READ_RAW(r IN OUT, data OUT, len)</td>
<td>n/a</td>
<td>Returns the HTTP response body in binary form for a specified number of bytes.</td>
</tr>
<tr>
<td>READ_TEXT(r IN OUT, data OUT, len)</td>
<td>n/a</td>
<td>Returns the HTTP response body in text form for a specified number of characters.</td>
</tr>
<tr>
<td>REQUEST(url)</td>
<td>VARCHAR2</td>
<td>Returns the content of a web page.</td>
</tr>
<tr>
<td>REQUEST_PIECES(url, max_pieces)</td>
<td>UTL_HTTP.HTML_PIECES</td>
<td>Returns a table of 2000-byte segments retrieved from an URL.</td>
</tr>
<tr>
<td>SET_BODY_CHARSET(charset)</td>
<td>n/a</td>
<td>Sets the default character set of the body of future HTTP requests.</td>
</tr>
<tr>
<td>SET_FOLLOW_REDIRECT(max_redirects)</td>
<td>n/a</td>
<td>Sets the maximum number of times to follow the redirect instruction.</td>
</tr>
<tr>
<td>SET_FOLLOW_REDIRECT(r IN OUT, max_redirects)</td>
<td>n/a</td>
<td>Sets the maximum number of times to follow the redirect instruction for an individual request.</td>
</tr>
<tr>
<td>SET_HEADER(r IN OUT, name, value)</td>
<td>n/a</td>
<td>Sets the HTTP request header.</td>
</tr>
<tr>
<td>SET_RESPONSE_ERROR_CHECK(enable)</td>
<td>n/a</td>
<td>Determines whether to treat HTTP 4xx and 5xx status codes as errors.</td>
</tr>
<tr>
<td>SET_TRANSFER_TIMEOUT(timeout)</td>
<td>n/a</td>
<td>Sets the default transfer timeout value for HTTP requests.</td>
</tr>
<tr>
<td>SET_TRANSFER_TIMEOUT(r IN OUT, timeout)</td>
<td>n/a</td>
<td>Sets the transfer timeout value for an individual HTTP request.</td>
</tr>
<tr>
<td>WRITE_LINE(r IN OUT, data)</td>
<td>n/a</td>
<td>Writes CRLF-terminated data to the HTTP request body in TEXT form.</td>
</tr>
<tr>
<td>WRITE_RAW(r IN OUT, data)</td>
<td>n/a</td>
<td>Writes data to the HTTP request body in BINARY form.</td>
</tr>
<tr>
<td>WRITE_TEXT(r IN OUT, data)</td>
<td>n/a</td>
<td>Writes data to the HTTP request body in TEXT form.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server’s implementation of **UTL_HTTP** is a partial implementation when compared to Oracle’s version. Only the functions and procedures listed in the table are supported.

**Note**

In EDB Postgres Advanced Server, an HTTP 4xx or HTTP 5xx response produces a database error. In Oracle, you can configure this option. It is **FALSE** by default.

In EDB Postgres Advanced Server, the **UTL_HTTP** text interfaces expect the downloaded data to be in the database encoding. All interfaces that are currently available are text interfaces. In Oracle, the encoding is detected from HTTP headers. Without the header, the default is configurable and defaults to **ISO-8859-1**.

EDB Postgres Advanced Server ignores all cookies it receives.

The **UTL_HTTP** exceptions that can be raised in Oracle aren't recognized by EDB Postgres Advanced Server. In addition, the error codes returned by EDB Postgres Advanced Server aren't the same as those returned by Oracle.

### UTL_HTTP Exception Codes

If a call to a **UTL_HTTP** procedure or function raises an exception, you can use the condition name to catch the exception. The **UTL_HTTP** package reports the following exception codes compatible with Oracle databases.

<table>
<thead>
<tr>
<th>Exception code</th>
<th>Condition name</th>
<th>Description</th>
<th>Raised where</th>
</tr>
</thead>
<tbody>
<tr>
<td>-29266</td>
<td>END_OF_BODY</td>
<td>The end of HTTP response body is reached</td>
<td>READ_LINE, READ_RAW, and READ_TEXT</td>
</tr>
</tbody>
</table>

To use the **UTL_HTTP.END_OF_BODY** exception, first you need to run the **utl_http_public.sql** file from the **contrib/utl_http** directory of your installation directory.

Various public constants are available with **UTL_HTTP**. These are listed in the following tables.

---

**Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.**
The following table contains UTL_HTTP public constants defining HTTP versions and port assignments.

### HTTP versions

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_VERSION_1_0</td>
<td>'HTTP/1.0';</td>
</tr>
<tr>
<td>HTTP_VERSION_1_1</td>
<td>'HTTP/1.1';</td>
</tr>
</tbody>
</table>

### Standard port assignments

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT_HTTP_PORT</td>
<td>88</td>
</tr>
<tr>
<td>DEFAULT_HTTPS_PORT</td>
<td>443</td>
</tr>
</tbody>
</table>

The following table contains UTL_HTTP public status code constants.

### 1XX informational

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_CONTINUE</td>
<td>100</td>
</tr>
<tr>
<td>HTTP_SWITCHING_PROTOCOLS</td>
<td>101</td>
</tr>
<tr>
<td>HTTP_PROCESSING</td>
<td>102</td>
</tr>
</tbody>
</table>

### 2XX success

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_OK</td>
<td>200</td>
</tr>
<tr>
<td>HTTP_CREATED</td>
<td>201</td>
</tr>
<tr>
<td>HTTP_ACCEPTED</td>
<td>202</td>
</tr>
<tr>
<td>HTTP_NOW_AUTHORITATIVE_INFO</td>
<td>203</td>
</tr>
<tr>
<td>HTTP_NO_CONTENT</td>
<td>204</td>
</tr>
<tr>
<td>HTTP_RESET_CONTENT</td>
<td>205</td>
</tr>
<tr>
<td>HTTP_PARTIAL_CONTENT</td>
<td>206</td>
</tr>
<tr>
<td>HTTP_MULTI_STATUS</td>
<td>207</td>
</tr>
<tr>
<td>HTTP_ALREADY_REPORTED</td>
<td>208</td>
</tr>
<tr>
<td>HTTP_IM_USED</td>
<td>226</td>
</tr>
</tbody>
</table>

### 3XX redirection

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_MULTIPLE_CHOICES</td>
<td>300</td>
</tr>
<tr>
<td>HTTP_MOVED_PERMANENTLY</td>
<td>301</td>
</tr>
<tr>
<td>HTTPFOUND</td>
<td>302</td>
</tr>
<tr>
<td>HTTP_SEE_OTHER</td>
<td>303</td>
</tr>
<tr>
<td>HTTP_NOT_MODIFIED</td>
<td>304</td>
</tr>
<tr>
<td>HTTP_USE_PROXY</td>
<td>305</td>
</tr>
<tr>
<td>HTTP_SWITCH_PROXY</td>
<td>306</td>
</tr>
<tr>
<td>HTTP_TEMPORARY_REDIRECT</td>
<td>307</td>
</tr>
<tr>
<td>HTTP_PERMANENT_REDIRECT</td>
<td>308</td>
</tr>
</tbody>
</table>

### 4XX client error

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_BAD_REQUEST</td>
<td>400</td>
</tr>
<tr>
<td>HTTP_UNAUTHORIZED</td>
<td>401</td>
</tr>
<tr>
<td>HTTP_PAYMENT_REQUIRED</td>
<td>402</td>
</tr>
<tr>
<td>HTTP_FORBIDDEN</td>
<td>403</td>
</tr>
<tr>
<td>HTTP_NOT_FOUND</td>
<td>404</td>
</tr>
<tr>
<td>HTTP_METHOD_NOT_ALLOWED</td>
<td>405</td>
</tr>
<tr>
<td>HTTP_NOT_ACCEPTABLE</td>
<td>406</td>
</tr>
<tr>
<td>HTTP_PROXY_AUTH_REQUIRED</td>
<td>407</td>
</tr>
<tr>
<td>HTTP_REQUEST_TIME_OUT</td>
<td>408</td>
</tr>
<tr>
<td>HTTP_CONFLICT</td>
<td>409</td>
</tr>
<tr>
<td>HTTP_LENGTH_REQUIRED</td>
<td>410</td>
</tr>
<tr>
<td>HTTP_REQUEST_ENTITY_TOO_LARGE</td>
<td>411</td>
</tr>
<tr>
<td>HTTP_PRECONDITION_FAILED</td>
<td>412</td>
</tr>
<tr>
<td>HTTP_REQUEST_URI_TOO_LARGE</td>
<td>413</td>
</tr>
</tbody>
</table>

HTTP_REQUEST_URI_TOO_LARGE | 414   |
### 4XX client error

- **HTTP_UNSUPPORTED_MEDIA_TYPE**  
  CONSTANT INTEGER := 415;
- **HTTP_REQ_RANGE_NOT_SATISFIABLE**  
  CONSTANT INTEGER := 416;
- **HTTP_EXPECTATION_FAILED**  
  CONSTANT INTEGER := 417;
- **HTTP_I_AM_A_TEAPOT**  
  CONSTANT INTEGER := 418;
- **HTTP_AUTHENTICATION_TIME_OUT**  
  CONSTANT INTEGER := 419;
- **HTTP_ENHANCE_YOUR_CALM**  
  CONSTANT INTEGER := 420;
- **HTTP_UNPROCESSABLE_ENTITY**  
  CONSTANT INTEGER := 422;
- **HTTP_LOCKED**  
  CONSTANT INTEGER := 423;
- **HTTP_FAILED_DEPENDENCY**  
  CONSTANT INTEGER := 424;
- **HTTP_UNORDERED_COLLECTION**  
  CONSTANT INTEGER := 425;
- **HTTP_UPGRADE_REQUIRED**  
  CONSTANT INTEGER := 426;
- **HTTP_REQUEST_HEADER_FIELDS_TOO_LARGE**  
  CONSTANT INTEGER := 428;
- **HTTP_NO_RESPONSE**  
  CONSTANT INTEGER := 431;
- **HTTP_RETRY_WITH**  
  CONSTANT INTEGER := 449;
- **HTTP_BLOCKED_BY_WINDOWS_PARENTAL_CONTROLS**  
  CONSTANT INTEGER := 450;
- **HTTP_REDIRECT**  
  CONSTANT INTEGER := 451;
- **HTTP_REQUEST_HEADER_TOO_LARGE**  
  CONSTANT INTEGER := 452;
- **HTTP_CERT_ERROR**  
  CONSTANT INTEGER := 453;
- **HTTP_NO_CERT**  
  CONSTANT INTEGER := 454;
- **HTTP_HTTP_TO_HTTPS**  
  CONSTANT INTEGER := 455;
- **HTTP_CLIENT_CLOSED_REQUEST**  
  CONSTANT INTEGER := 499;

### 5XX server error

- **HTTP_INTERNAL_SERVER_ERROR**  
  CONSTANT INTEGER := 500;
- **HTTP_NOT_IMPLEMENTED**  
  CONSTANT INTEGER := 501;
- **HTTP_BAD_GATEWAY**  
  CONSTANT INTEGER := 502;
- **HTTP_SERVICE_UNAVAILABLE**  
  CONSTANT INTEGER := 503;
- **HTTP_GATEWAY_TIME_OUT**  
  CONSTANT INTEGER := 504;
- **HTTP_VERSION_NOT_SUPPORTED**  
  CONSTANT INTEGER := 505;
- **HTTP_VARIANT_ALSO_NEGOTIATES**  
  CONSTANT INTEGER := 506;
- **HTTP_INSUFFICIENT_STORAGE**  
  CONSTANT INTEGER := 507;
- **HTTP_LOOP_DETECTED**  
  CONSTANT INTEGER := 508;
- **HTTP_BANDWIDTH_LIMIT_EXCEEDED**  
  CONSTANT INTEGER := 509;
- **HTTP_NOT_EXTENDED**  
  CONSTANT INTEGER := 510;
- **HTTP_NETWORK_AUTHENTICATION_REQUIRED**  
  CONSTANT INTEGER := 511;
- **HTTP_NETWORK_READ_TIME_OUT_ERROR**  
  CONSTANT INTEGER := 598;
- **HTTP_NETWORK_CONNECT_TIME_OUT_ERROR**  
  CONSTANT INTEGER := 599;

### HTML_PIECES

The **UTL_HTTP** package declares a type named **HTML_PIECES**, which is a table of type **VARCHAR2 (2000)** indexed by **BINARY_INTEGER**. A value of this type is returned by the **REQUEST_PIECES** function.

```sql
TYPE html_pieces IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
```

### REQ

The **REQ** record type holds information about each HTTP request.

```sql
TYPE req IS RECORD
```

---

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved. 954
RESP

The RESP record type holds information about the response from each HTTP request.

```
url     VARCHAR2(32767), -- URL to be accessed
method  VARCHAR2(64),  -- HTTP method
http_version VARCHAR2(64), -- HTTP protocol version
private_hndl INTEGER, -- Holds handle for this request
);
```

BEGIN_REQUEST

The BEGIN_REQUEST function initiates a new HTTP request. A network connection is established to the web server with the specified URL. The signature is:

```
BEGIN_REQUEST(<url> IN VARCHAR2, <method> IN VARCHAR2 DEFAULT 'GET', <http_version> IN VARCHAR2 DEFAULT NULL)
RETURN UTL_HTTP.REQ
```

Parameters

- **url**
  - The Uniform Resource Locator from which UTL_HTTP returns content.

- **method**
  - The HTTP method to use. The default is GET.

- **http_version**
  - The HTTP protocol version sending the request. Specify either HTTP/1.0 or HTTP/1.1. The default is null. In that case, the latest HTTP protocol version supported by the UTL_HTTP package is used, which is 1.1.

END_REQUEST

The END_REQUEST procedure terminates an HTTP request. Use the END_REQUEST procedure to terminate an HTTP request without completing it and waiting for the response. The normal process is to begin the request, get the response, and then close the response. The signature is:

```
END_REQUEST(<r> IN OUT UTL_HTTP.REQ)
```

Parameters

- **r**
  - The HTTP request record.
The `END_RESPONSE` procedure terminates the HTTP response. The `END_RESPONSE` procedure completes the HTTP request and response. This is the normal method to end the request and response process. The signature is:

```sql
END_RESPONSE(<r> IN OUT
UTL_HTTP.RESP)
```

Parameters

- `r` is the HTTP response record.

**GET_BODY_CHARSET**

The `GET_BODY_CHARSET` program is available in the form of both a procedure and a function. A call to `GET_BODY_CHARSET` returns the default character set of the body of future HTTP requests.

The procedure signature is:

```sql
GET_BODY_CHARSET(<charset> OUT VARCHAR2)
```

The function signature is:

```sql
GET_BODY_CHARSET() RETURN VARCHAR2
```

This function returns a `VARCHAR2` value.

Parameters

- `charset` is the character set of the body.

Examples

This example shows the use of the `GET_BODY_CHARSET` function.

```sql
edb> SELECT UTL_HTTP.GET_BODY_CHARSET() FROM DUAL;
get_body_charset
---------------
ISO-8859-1
(1 row)
```

**GET_FOLLOW_REDIRECT**

The `GET_FOLLOW_REDIRECT` procedure returns the current setting for the maximum number of redirections allowed. The signature is:

```sql
GET_FOLLOW_REDIRECT(<max_redirects> OUT INTEGER)
```

Parameters

- `max_redirects` is the maximum number of redirections allowed.
GET_HEADER

The `GET_HEADER` procedure returns the \textit{nth} header of the HTTP response. The signature is:

\begin{verbatim}
GET_HEADER(r IN OUT UTL_HTTP.RESP, n INTEGER, name OUT VARCHAR2, value OUT VARCHAR2)
\end{verbatim}

Parameters

\textit{r}

\textit{r} is the HTTP response record.

\textit{n}

\textit{n} is the \textit{nth} header of the HTTP response record to retrieve.

\textit{name}

\textit{name} is the name of the response header.

\textit{value}

\textit{value} is the value of the response header.

Examples

This example retrieves the header count and then the headers.

\begin{verbatim}
DECLARE
  v_req           UTL_HTTP.REQ;
  v_resp          UTL_HTTP.RESP;
  v_name          VARCHAR2(30);
  v_value         VARCHAR2(200);
  v_header_cnt    INTEGER;
BEGIN
  -- Initiate request and get response
  v_req := UTL_HTTP.BEGIN_REQUEST('www.enterprisedb.com');
  v_resp := UTL_HTTP.GET_RESPONSE(v_req);

  -- Get header count
  v_header_cnt := UTL_HTTP.GET_HEADER_COUNT(v_resp);
  DBMS_OUTPUT.PUT_LINE('Header Count: ' || v_header_cnt);

  -- Get all headers
  FOR i IN 1 .. v_header_cnt LOOP
    UTL_HTTP.GET_HEADER(v_resp, i, v_name, v_value);
    DBMS_OUTPUT.PUT_LINE(v_name || ': ' || v_value);
  END LOOP;

  -- Terminate request
  UTL_HTTP.END_RESPONSE(v_resp);
END;
\end{verbatim}

The following is the output from the example.

```
Header Count: 23
Age: 379
Cache-Control: must-revalidate
Content-Type: text/html; charset=utf-8
Date: Wed, 30 Apr 2015 14:57:52 GMT
ETag: "aab0f72bd6d99edd817ca89e411dad"```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The `GET_HEADER_BY_NAME` procedure returns the header of the HTTP response according to the specified name. The signature is:

```
GET_HEADER_BY_NAME(r IN OUT UTL_HTTP.RESP, name VARCHAR2, value OUT VARCHAR2, n INTEGER DEFAULT 1)
```

### Parameters

- `r`
  - is the HTTP response record.

- `name`
  - is the name of the response header to retrieve.

- `value`
  - is the value of the response header.

- `n`
  - is the `n`th header of the HTTP response record to retrieve according to the values specified by `name`. The default is `1`.

### Examples

The following example retrieves the header for Content-Type.

```sql
DECLARE
    v_req UTL_HTTP.REQ;
    v_resp UTL_HTTP.RESP;
    v_name VARCHAR2(38) := 'Content-Type';
    v_value VARCHAR2(280);
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);    
    UTL_HTTP.GET_HEADER_BY_NAME(v_resp, v_name, v_value);
    DBMS_OUTPUT.PUT_LINE(v_name || ' : ' || v_value);
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```
The `GET_HEADER_COUNT` function returns the number of HTTP response headers. The signature is:

```
GET_HEADER_COUNT(<r IN OUT UTL_HTTP.RESP) RETURN INTEGER
```

This function returns an `INTEGER` value.

**Parameters**

- `r`

  `r` is the HTTP response record.

---

The `GET_RESPONSE` function sends the network request and returns any HTTP response. The signature is:

```
GET_RESPONSE(<r IN OUT UTL_HTTP.REQ) RETURN UTL_HTTP.RESP
```

This function returns a `UTL_HTTP.RESP` record.

**Parameters**

- `r`

  `r` is the HTTP request record.

---

The `GET_RESPONSE_ERROR_CHECK` procedure returns whether response error check is set. The signature is:

```
GET_RESPONSE_ERROR_CHECK(<enable> OUT BOOLEAN)
```

**Parameters**

- `enable`

  `enable` returns `TRUE` if response error check is set. Otherwise it returns `FALSE`.

---

The `GET_TRANSFER_TIMEOUT` procedure returns the current default transfer timeout setting for HTTP requests. The signature is:

```
GET_TRANSFER_TIMEOUT(<timeout> OUT INTEGER)
```

**Parameters**

- `timeout`

  `timeout` is the transfer timeout setting in seconds.
The `READ_LINE` procedure returns the data from the HTTP response body in text form until the end of line is reached. A CR character, a LF character, a CR LF sequence, or the end of the response body constitutes the end of line. The signature is:

```sql
READ_LINE(<r> IN OUT UTL_HTTP.RESP, <data> OUT VARCHAR2,
<remove_crlf> BOOLEAN DEFAULT FALSE)
```

### Parameters

- `<r>` is the HTTP response record.
- `<data>` is the response body in text form.
- `<remove_crlf>`

Set `<remove_crlf>` to `TRUE` to remove new line characters. The default is `FALSE`.

### Examples

This example retrieves and displays the body of the specified website.

```sql
DECLARE
  v_req           UTL_HTTP.REQ;
  v_resp          UTL_HTTP.RESP;
  v_value         VARCHAR2(1024);
BEGIN
  v_req := UTL_HTTP.BEGIN_REQUEST('http://www.enterprisedb.com');
  v_resp := UTL_HTTP.GET_RESPONSE(v_req);
  LOOP
    UTL_HTTP.READ_LINE(v_resp, v_value, TRUE);
    DBMS_OUTPUT.PUT_LINE(v_value);
  END LOOP;
EXCEPTION
  WHEN OTHERS
    THEN
      UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The following is the output.

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">

<title>EnterpriseDB | The Postgres Database Company</title>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta name="keywords" content="postgres, postgresql, postgresql installer, mysql migration, open source database, training, replication" />
<meta name="description" content="The leader in open source database" />
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
products, services, support, training and expertise based on PostgreSQL.

Free downloads, documentation, and tutorials.

The **READ_RAW** procedure returns the data from the HTTP response body in binary form. The number of bytes returned is specified by the `len` parameter. The signature is:

```sql
READ_RAW(<<!-- IN OUT UTL_HTTP.RESP, <data> OUT RAW, <len> INTEGER) -->
```

**Parameters**

- `r` is the HTTP response record.
- `data` is the response body in binary form.
- `len` to the number of bytes of data to return.

**Examples**

This example retrieves and displays the first 150 bytes in binary form.

```sql
DECLARE
  v_req           UTL_HTTP.REQ;
  v_resp
  UTL_HTTP.RESP;
  v_data
  RAW;
BEGIN
  v_req := UTL_HTTP.BEGIN_REQUEST('http://www.enterprisedb.com');
  v_resp :=
  UTL_HTTP.GET_RESPONSE(v_req);
  UTL_HTTP.READ_RAW(v_resp, v_data, 150);
```
The following is the output from the example.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/">
  <body>
    <p>The following is the output from the example.</p>
  </body>
</html>
```

**READ_TEXT**

The **READ_TEXT** procedure returns the data from the HTTP response body in text form. The maximum number of characters returned is specified by the `len` parameter. The signature is:

```
READ_TEXT(<r> IN OUT UTL_HTTP.RESP, <data> OUT VARCHAR2, <len> INTEGER)
```

**Parameters**

- **r**
  - `r` is the HTTP response record.

- **data**
  - `data` is the response body in text form.

- **len**
  - Set `len` to the maximum number of characters to return.

**Examples**

This example retrieves the first 150 characters:

```
DECLARE
  v_req UTL_HTTP.REQ;
  v_res UTL_HTTP.RESP;
  v_data VARCHAR2(150);
BEGIN
  v_req := UTL_HTTP.BEGIN_REQUEST('http://www.enterprisedb.com');
  v_res := UTL_HTTP.GET_RESPONSE(v_req);
  UTL_HTTP.READ_TEXT(v_res, v_data, 150);
  DBMS_OUTPUT.PUT_LINE(v_data);
  UTL_HTTP.END_RESPONSE(v_res);
END;
```

The following is the output:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/">
  <body>
    <p>The following is the output from the example.</p>
  </body>
</html>
```

**REQUEST**

The **REQUEST** function returns the first 2000 bytes retrieved from a URL. The signature is:
If the data found at the URL is longer than 2000 bytes, the remainder is discarded. If the data found at the given URL is shorter than 2000 bytes, the result is shorter than 2000 bytes.

Parameters

url

url is the Uniform Resource Locator from which UTL_HTTP returns content.

Example

This command returns the first 2000 bytes retrieved from the EDB website:

```sql
SELECT UTL_HTTP.REQUEST('http://www.enterprisedb.com/') FROM DUAL;
```

REQUEST_PIECES

The REQUEST_PIECES function returns a table of 2000-byte segments retrieved from a URL. The signature is:

```sql
REQUEST_PIECES(url IN VARCHAR2, max_pieces NUMBER IN DEFAULT 32767) RETURN UTL_HTTP.HTML_PIECES
```

Parameters

url

url is the Uniform Resource Locator from which UTL_HTTP returns content.

max_pieces

max_pieces specifies the maximum number of 2000-byte segments that the REQUEST_PIECES function returns. If max_pieces specifies more units than are available at the specified URL, the final unit contains fewer bytes.

Example

This example returns the first four 2000-byte segments retrieved from the EDB website:

```sql
DECLARE
  result UTL_HTTP.HTML_PIECES;
BEGIN
  result := UTL_HTTP.REQUEST_PIECES('http://www.enterprisedb.com/', 4);
END;
```

SET_BODY_CHARSET

The SET_BODY_CHARSET procedure sets the default character set of the body of future HTTP requests. The signature is:

```sql
SET_BODY_CHARSET(charset IN VARCHAR2 DEFAULT NULL)
```

Parameters

charset

charset is the character set of the body of future requests. The default is null, meaning the database character set is assumed.
SET_FOLLOW_REDIRECT

The `SET_FOLLOW_REDIRECT` procedure sets the maximum number of times to follow the HTTP redirect instruction in the response to this request or future requests. This procedure has two signatures:

```sql
SET_FOLLOW_REDIRECT(<max_redirects> IN INTEGER DEFAULT 3)
```

and

```sql
SET_FOLLOW_REDIRECT(<r> IN OUT UTL_HTTP.REQ, <max_redirects>
IN INTEGER DEFAULT 3)
```

Use the second form to change the maximum number of redirections for an individual request that a request inherits from the session default settings.

**Parameters**

- `<r>` is the HTTP request record.
- `<max_redirects>` is the maximum number of redirections allowed. Set to 0 to disable redirections. The default is 3.

SET_HEADER

The `SET_HEADER` procedure sets the HTTP request header. The signature is:

```sql
SET_HEADER(<r> IN OUT UTL_HTTP.REQ, <name> IN VARCHAR2, <value>
IN VARCHAR2 DEFAULT NULL)
```

**Parameters**

- `<r>` is the HTTP request record.
- `<name>` is the name of the request header.
- `<value>` is the value of the request header. The default is null.

SET_RESPONSE_ERROR_CHECK

The `SET_RESPONSE_ERROR_CHECK` procedure determines whether to interpret HTTP 4xx and 5xx status codes returned by the `GET_RESPONSE` function as errors. The signature is:

```sql
SET_RESPONSE_ERROR_CHECK(<enable> IN BOOLEAN DEFAULT FALSE)
```

**Parameters**

- `<enable>` Set to TRUE if you want to treat HTTP 4xx and 5xx status codes as errors. The default is FALSE.
The `SET_TRANSFER_TIMEOUT` procedure sets the default transfer timeout setting for waiting for a response from an HTTP request. This procedure has two signatures:

```
SET_TRANSFER_TIMEOUT(<timeout> IN INTEGER DEFAULT 60)
```

and

```
SET_TRANSFER_TIMEOUT(<r> IN OUT UTL_HTTP.REQ, <timeout> IN INTEGER DEFAULT 60)
```

Use the second form to change the transfer timeout setting for an individual request that a request inherits from the session default settings.

**Parameters**

- `<timeout>`
  - `r` is the HTTP request record.
  - `timeout` is the transfer timeout setting in seconds for HTTP requests. The default is 60 seconds.

**WRITE_LINE**

The `WRITE_LINE` procedure writes data to the HTTP request body in text form. The text is terminated with a CRLF character pair. The signature is:

```
WRITE_LINE(<r> IN OUT UTL_HTTP.REQ, <data> IN VARCHAR2)
```

**Parameters**

- `<r>`
  - `r` is the HTTP request record.
- `<data>`
  - `data` is the request body in `TEXT` form.

**Example**

This example writes data (`Account balance $500.00`) in text form to the request body to send using the HTTP `POST` method. The data is sent to a web application that accepts and processes data (`post.php`).

```sql
DECLARE
  v_req UTL_HTTP.REQ;
  v_resp UTL_HTTP.RESP;
BEGIN
  v_req := UTL_HTTP.BEGIN_REQUEST('http://www.example.com/post.php', 'POST');
  UTL_HTTP.SET_HEADER(v_req, 'Content-Length', '23');
  UTL_HTTP.WRITE_LINE(v_req, 'Account balance $500.00');
  vresp := UTL_HTTP.GET_RESPONSE(v_req);
  DBMS_OUTPUT.PUT_LINE('Status Code: ' || v_resp.status_code);
  DBMS_OUTPUT.PUT_LINE('Reason Phrase: ' || vresp.reason_phrase);
  UTL_HTTP.END_RESPONSE(vresp);
END;
```

Assuming the web application successfully processed the `POST` method, the following output is displayed:

```
__OUTPUT__
Status Code: 200
Reason Phrase: OK
```
The **WRITE_RAW** procedure writes data to the HTTP request body in binary form. The signature is:

```
WRITE_RAW(<r> IN OUT UTL_HTTP.REQ, <data> IN RAW)
```

### Parameters

- **r**
  - is the HTTP request record.

- **data**
  - is the request body in binary form.

### Example

This example writes data in binary form to the request body to send using the HTTP **POST** method to a web application that accepts and processes such data.

```sql
DECLARE
  v_req           UTL_HTTP.REQ;
  v_resp          UTL_HTTP.RESP;
BEGIN
  v_req := UTL_HTTP.BEGIN_REQUEST('http://www.example.com/post.php', 'POST');
  UTL_HTTP.SET_HEADER(v_req, 'Content-Length', '23');
  UTL_HTTP.WRITE_RAW(v_req, HEXTORAW('54657374696e6720504f53545020696e20485454502072657175657374'));
  v_resp := UTL_HTTP.GET_RESPONSE(v_req);
  DBMS_OUTPUT.PUT_LINE('Status Code: ' || v_resp.status_code);
  DBMS_OUTPUT.PUT_LINE('Reason Phrase: ' || v_resp.reason_phrase);
  UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The text string shown in the **HEXTORAW** function is the hexadecimal translation of the text **Testing POST method in HTTP request**.

When the web application successfully processes the **POST** method, the following output is displayed:

```
| Status Code: 200 |
| Reason Phrase: OK |
```

### WRITE_TEXT

The **WRITE_TEXT** procedure writes data to the HTTP request body in text form. The signature is:

```
WRITE_TEXT(<r> IN OUT UTL_HTTP.REQ, <data> IN VARCHAR2)
```

### Parameters

- **r**
  - is the HTTP request record.

- **data**
  - is the request body in text form.
This example writes data (Account balance $500.00) in text form to the request body to send using the HTTP POST method. The data is sent to a web application that accepts and processes data (post.php).

```sql
DECLARE
  v_req UTL_HTTP.REQ;
  v_resp UTL_HTTP.RESP;
BEGIN
  v_req := UTL_HTTP.BEGIN_REQUEST('http://www.example.com/post.php', 'POST');
  UTL_HTTP.SET_HEADER(v_req, 'Content-Length', '23');
  UTL_HTTP.WRITE_TEXT(v_req, 'Account balance $500.00');
  v_resp := UTL_HTTP.GET_RESPONSE(v_req);
  DBMS_OUTPUT.PUT_LINE('Status Code: ' || v_resp.status_code);
  DBMS_OUTPUT.PUT_LINE('Reason Phrase: ' || v_resp.reason_phrase);
  UTL_HTTP.END_RESPONSE(v_resp);
END;
```

When the web application successfully processes the POST method, the following output is displayed:

```
__OUTPUT__
Status Code: 200
Reason Phrase: OK
```

END_OF_BODY

The END_OF_BODY exception is raised when it reaches the end of the HTTP response body.

Example

This example handles the exception and writes Exception caught in text form to the request body to send using the HTTP POST method. The data is sent to a web application that accepts and processes data (post.php).

```sql
DECLARE
  req UTL_HTTP.REQ;
  resp UTL_HTTP.RESP;
  value VARCHAR2(32768);
BEGIN
  req := UTL_HTTP.BEGIN_REQUEST('https://www.google.com/');
  resp := UTL_HTTP.GET_RESPONSE(req);
  LOOP
    UTL_HTTP.READ_LINE(resp, value, TRUE);
  END LOOP;
  UTL_HTTP.END_RESPONSE(resp);
EXCEPTION
  WHEN UTL_HTTP.END_OF_BODY THEN
    DBMS_OUTPUT.PUT_LINE('Exception caught');
    UTL_HTTP.END_RESPONSE(resp);
END;
```

When the web application successfully processes the POST method, the following output is displayed:

```
__OUTPUT__
Exception caught
```

14.3.3.123 UTL_MAIL
The **UTL_MAIL** package manages email. EDB Postgres Advanced Server supports the following procedures.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND(sender, recipients, cc, bcc, subject, message [, mime_type [, priority ]])</td>
<td>n/a</td>
<td>Packages and sends an email to an SMTP server.</td>
</tr>
<tr>
<td>SEND_ATTACH_RAW(sender, recipients, cc, bcc, subject, message, mime_type, priority, attachment [, att_inline [, att_mime_type [, att_filename ]]])</td>
<td>n/a</td>
<td>Same as the <strong>SEND</strong> procedure but with BYTEA or large object attachments.</td>
</tr>
<tr>
<td>SEND_ATTACH_VARCHAR2(sender, recipients, cc, bcc, subject, message, mime_type, priority, attachment [, att_inline [, att_mime_type [, att_filename ]]])</td>
<td>n/a</td>
<td>Same as the <strong>SEND</strong> procedure but with VARCHAR2 attachments.</td>
</tr>
</tbody>
</table>

**Note**

An administrator must grant execute privileges to each user or group before they can use this package.

### SEND

The **SEND** procedure sends an email to an SMTP server.

```sql
SEND(<sender> VARCHAR2, <recipients> VARCHAR2, <cc> VARCHAR2,
     <bcc> VARCHAR2, <subject> VARCHAR2, <message> VARCHAR2
     [, <mime_type> VARCHAR2 [, <priority> PLS_INTEGER ]])
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sender</td>
<td>Email address of the sender.</td>
</tr>
<tr>
<td>recipients</td>
<td>Comma-separated email addresses of the recipients.</td>
</tr>
<tr>
<td>cc</td>
<td>Comma-separated email addresses of copy recipients.</td>
</tr>
<tr>
<td>bcc</td>
<td>Comma-separated email addresses of blind copy recipients.</td>
</tr>
<tr>
<td>subject</td>
<td>Subject line of the email.</td>
</tr>
<tr>
<td>message</td>
<td>Body of the email.</td>
</tr>
<tr>
<td>mime_type</td>
<td>Mime type of the message. The default is <code>text/plain; charset=us-ascii</code>.</td>
</tr>
<tr>
<td>priority</td>
<td>Priority of the email. The default is 3.</td>
</tr>
</tbody>
</table>

#### Examples

The following anonymous block sends a simple email message.

```sql
DECLARE
```

---

**EDB Postgres Advanced Server (EPAS)**

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
SEND_ATTACH_RAW

The **SEND_ATTACH_RAW** procedure sends an email to an SMTP server with an attachment containing either BYTEA data or a large object (identified by the large object’s OID). You can write the call to **SEND_ATTACH_RAW** in two ways:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sender</code></td>
<td>Email address of the sender.</td>
</tr>
<tr>
<td><code>recipients</code></td>
<td>Comma-separated email addresses of the recipients.</td>
</tr>
<tr>
<td><code>cc</code></td>
<td>Comma-separated email addresses of copy recipients.</td>
</tr>
<tr>
<td><code>bcc</code></td>
<td>Comma-separated email addresses of blind copy recipients.</td>
</tr>
<tr>
<td><code>subject</code></td>
<td>Subject line of the email.</td>
</tr>
<tr>
<td><code>message</code></td>
<td>Body of the email.</td>
</tr>
<tr>
<td><code>mime_type</code></td>
<td>Mime type of the message. The default is <code>text/plain; charset=us-ascii</code>.</td>
</tr>
<tr>
<td><code>priority</code></td>
<td></td>
</tr>
</tbody>
</table>

```sql
BEGIN
  v_sender := 'jsmith@enterprisedb.com';
  v_recipients := 'ajones@enterprisedb.com,rrogers@enterprisedb.com';
  v_subj := 'Holiday Party';
  v_msg := 'This year’s party is scheduled for Friday, Dec. 21 at ' ||
           '6:00 PM. Please RSVP by Dec. 15th.';
  UTL_MAIL.SEND(v_sender, v_recipients, NULL, NULL, v_subj, v_msg);
END;
```

EDB Postgres Advanced Server (EPAS) 
Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
Priority of the email. The default is 3.

The attachment.

If set to TRUE, then the attachment is viewable inline. The default is TRUE.

Mime type of the attachment. The default is application/octet.

The file name containing the attachment. The default is NULL.

SEND_ATTACH_VARCHAR2

The SEND_ATTACH_VARCHAR2 procedure sends an email to an SMTP server with a text attachment.

SEND_ATTACH_VARCHAR2(<sender> VARCHAR2, <recipients> VARCHAR2, <cc> VARCHAR2, <bcc> VARCHAR2, <subject> VARCHAR2, <message> VARCHAR2, <mime_type> VARCHAR2, <priority> PLS_INTEGER, <attachment> VARCHAR2 [, <att_inline> BOOLEAN [, <att_mime_type> VARCHAR2 [, <att_filename> VARCHAR2 ]]])

Parameters

sender

Email address of the sender.

recipients

Comma-separated email addresses of the recipients.

cc

Comma-separated email addresses of copy recipients.

bcc

Comma-separated email addresses of blind copy recipients.

subject

Subject line of the email.

message

Body of the email.

mime_type

Mime type of the message. The default is text/plain; charset=utf-8.

priority

Priority of the email. The default is 3.
If set to `TRUE`, then the attachment is viewable inline. The default is `TRUE`.

Mime type of the attachment. The default is `text/plain; charset=us-ascii`.

The file name containing the attachment. The default is `NULL`.

### 14.3.3.1.24 UTL_RAW

The `UTL_RAW` package allows you to manipulate or retrieve the length of raw data types.

**Note**

An administrator must grant execute privileges to each user or group before they can use this package.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Function or procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST_TO_RAW(c IN VARCHAR2)</td>
<td>Function</td>
<td>RAW</td>
<td>Converts a <code>VARCHAR2</code> string to a <code>RAW</code> value.</td>
</tr>
<tr>
<td>CAST_TO_VARCHAR2(r IN RAW)</td>
<td>Function</td>
<td>VARCHAR2</td>
<td>Converts a <code>RAW</code> value to a <code>VARCHAR2</code> string.</td>
</tr>
<tr>
<td>CONCAT(r1 IN RAW, r2 IN RAW, r3 IN RAW,...)</td>
<td>Function</td>
<td>RAW</td>
<td>Concatenates multiple <code>RAW</code> values into a single <code>RAW</code> value.</td>
</tr>
<tr>
<td>CONVERT(r IN RAW, to_charset IN VARCHAR2, from_charset IN VARCHAR2)</td>
<td>Function</td>
<td>RAW</td>
<td>Converts encoded data from one encoding to another and returns the result as a <code>RAW</code> value.</td>
</tr>
<tr>
<td>LENGTH(r IN RAW)</td>
<td>Function</td>
<td>NUMBER</td>
<td>Returns the length of a <code>RAW</code> value.</td>
</tr>
<tr>
<td>SUBSTR(r IN RAW, pos IN INTEGER, len IN INTEGER)</td>
<td>Function</td>
<td>RAW</td>
<td>Returns a portion of a <code>RAW</code> value.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server's implementation of `UTL_RAW` is a partial implementation when compared to Oracle's version. Only the functions and procedures listed in the table are supported.

**CAST_TO_RAW**

The `CAST_TO_RAW` function converts a `VARCHAR2` string to a `RAW` value. The signature is:

```
CAST_TO_RAW(c IN VARCHAR2)
```

The function returns a `RAW` value if you pass a non-`NULL` value. If you pass a `NULL` value, the function returns `NULL`.

**Parameters**

- `c`

  The `VARCHAR2` value to convert to `RAW`.

**Example**

This example uses the `CAST_TO_RAW` function to convert a `VARCHAR2` string to a `RAW` value:

```
DECLARE
    v VARCHAR2;
    r RAW;
BEGIN
    v := 'Accounts';
    dbms_output.put_line(v);
```

**Note**

EDB Postgres Advanced Server (EPAS)

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The result set includes the content of the original string and the converted RAW value:

```
Accounts
\x4163636f756e7473
```

**CAST_TO_VARCHAR2**

The `CAST_TO_VARCHAR2` function converts RAW data to VARCHAR2 data. The signature is:

```
CAST_TO_VARCHAR2(< RAW>
```

The function returns a VARCHAR2 value if you pass a non-NULL value. If you pass a NULL value, the function returns NULL.

**Parameters**

`r`

The RAW value to convert to a VARCHAR2 value.

**Example**

This example uses the `CAST_TO_VARCHAR2` function to convert a RAW value to a VARCHAR2 string:

```
DECLARE
  r RAW;
  v VARCHAR2;
BEGIN
  r := '\x4163636f756e7473';
  dbms_output.put_line(r);
  v := UTL_RAW.CAST_TO_VARCHAR2(r);
  dbms_output.put_line(r);
END;
```

The result set includes the content of the original string and the converted RAW value:

```
\x4163636f756e7473
```

**CONCAT**

The `CONCAT` function concatenates multiple RAW values into a single RAW value. The signature is:

```
CONCAT(<r1> RAW, <r2> RAW, <r3> RAW,...)
```

The function returns a RAW value. Unlike the Oracle implementation, the EDB Postgres Advanced Server implementation is a variadic function and doesn't place a restriction on the number of values that can be concatenated.

**Parameters**

`r1, r2, r3,...`

The RAW values for CONCAT to concatenate.
Example

This example uses the `CONCAT` function to concatenate multiple `RAW` values into a single `RAW` value:

```sql
SELECT UTL_RAW.CAST_TO_VARCHAR2(UTL_RAW.CONCAT('\x61', '\x62', '\x63')) FROM DUAL;
```

<table>
<thead>
<tr>
<th>concat</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
</tr>
</tbody>
</table>

(1 row)

The result (the concatenated values) is then converted to `VARCHAR2` format by the `CAST_TO_VARCHAR2` function.

CONVERT

The `CONVERT` function converts a string from one encoding to another encoding and returns the result as a `RAW` value. The signature is:

```sql
CONVERT(<r> RAW, <to_charset> VARCHAR2, <from_charset> VARCHAR2)
```

The function returns a `RAW` value.

Parameters

- `r`
  The `RAW` value to convert.
- `to_charset`
  The name of the encoding to which `r` is converted.
- `from_charset`
  The name of the encoding from which `r` is converted.

Example

This example uses the `UTL_RAW.CAST_TO_RAW` function to convert a `VARCHAR2` string (`Accounts`) to a raw value. It then converts the value from `UTF8` to `LATIN7` and then from `LATIN7` to `UTF8`:

```sql
DECLARE
  r RAW;
  v VARCHAR2;
BEGIN
  v := 'Accounts';
  dbms_output.put_line(v);
  r := UTL_RAW.CAST_TO_RAW(v);
  dbms_output.put_line(r);
  r := UTL_RAW.CONVERT(r, 'UTF8', 'LATIN7');
  dbms_output.put_line(r);
  r := UTL_RAW.CONVERT(r, 'LATIN7', 'UTF8');
  dbms_output.put_line(r);
END;
```

The example returns the `VARCHAR2` value, the `RAW` value, and the converted values:

```
Accounts
\x4163636f756e7473
\x4163636f756e7473
\x4163636f756e7473
```
LENGTH

The LENGTH function returns the length of a RAW value. The signature is:

\[ \text{LENGTH}(r \to \text{RAW}) \]

The function returns a RAW value.

Parameters

- \( r \)
  - The RAW value for LENGTH to evaluate.

Example

This example uses the LENGTH function to return the length of a RAW value:

```
SELECT UTL_RAW.LENGTH(UTL_RAW.CAST_TO_RAW('Accounts')) FROM DUAL;
```

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

(1 row)

This example uses the LENGTH function to return the length of a RAW value that includes multi-byte characters:

```
SELECT UTL_RAW.LENGTH(UTL_RAW.CAST_TO_RAW(' обы '));
```

<table>
<thead>
<tr>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

(1 row)

SUBSTR

The SUBSTR function returns a substring of a RAW value. The signature is:

\[ \text{SUBSTR}(r \to \text{RAW}, \langle pos \rangle \to \text{INTEGER}, \langle len \rangle \to \text{INTEGER}) \]

This function returns a RAW value.

Parameters

- \( r \)
  - The RAW value from which the substring is returned.
- \( \text{pos} \)
  - The position of the first byte of the returned substring in the RAW value.
- \( \text{len} \)
  - The maximum number of bytes to return.

  - If \( \text{pos} \) is \( 0 \) or \( 1 \), the substring begins at the first byte of the RAW value.
  - If \( \text{pos} \) is greater than one, the substring begins at the first byte specified by \( \text{pos} \). For example, if \( \text{pos} \) is \( 3 \), the substring begins at the third byte of the value.
  - If \( \text{pos} \) is negative, the substring begins at \( \text{pos} \) bytes from the end of the source value. For example, if \( \text{pos} \) is \( -3 \), the substring begins at the third byte from the end of the value.
Example

This example uses the `SUBSTR` function to select a substring that begins 3 bytes from the start of a `RAW` value:

```sql
SELECT UTL_RAW.SUBSTR(UTL_RAW.CAST_TO_RAW('Accounts'), 3, 5) FROM DUAL;
```

<table>
<thead>
<tr>
<th>substr</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>own</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>

This example uses the `SUBSTR` function to select a substring that starts 5 bytes from the end of a `RAW` value:

```sql
SELECT UTL_RAW.SUBSTR(UTL_RAW.CAST_TO_RAW('Accounts'), -5, 3) FROM DUAL;
```

<table>
<thead>
<tr>
<th>substr</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>own</td>
<td>(1 row)</td>
</tr>
</tbody>
</table>

### 14.3.3.1.25 UTL_SMTP

The `UTL_SMTP` package sends emails over the Simple Mail Transfer Protocol (SMTP).

**Note**

An administrator must grant execute privileges to each user or group before they can use this package.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Function or procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE_DATA(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>End an email message.</td>
</tr>
<tr>
<td>COMMAND(c IN OUT, cmd [, arg ])</td>
<td>Both</td>
<td>REPLY</td>
<td>Execute an SMTP command.</td>
</tr>
<tr>
<td>COMMAND_REPLIES(c IN OUT, cmd [, arg ])</td>
<td>Function</td>
<td>REPLY</td>
<td>Execute an SMTP command where multiple reply lines are expected.</td>
</tr>
<tr>
<td>DATA(c IN OUT, body VARCHAR2)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Specify the body of an email message.</td>
</tr>
<tr>
<td>EHLO(c IN OUT, domain)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Perform initial handshaking with an SMTP server and return extended information.</td>
</tr>
<tr>
<td>HELP(c IN OUT, domain)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Perform initial handshaking with an SMTP server.</td>
</tr>
<tr>
<td>HELP(c IN OUT [, command ])</td>
<td>Function</td>
<td>REPLY</td>
<td>Send the <code>HELP</code> command.</td>
</tr>
<tr>
<td>MAIL(c IN OUT, sender [, parameters ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Start a mail transaction.</td>
</tr>
<tr>
<td>NOOP(c IN OUT)</td>
<td>Both</td>
<td>REPLY</td>
<td>Send the null command.</td>
</tr>
<tr>
<td>OPEN_CONNECTION(host [, port [, tx_timeout ]])</td>
<td>Function</td>
<td>CONNECTION</td>
<td>Open a connection.</td>
</tr>
<tr>
<td>OPEN_DATA(c IN OUT)</td>
<td>Both</td>
<td>n/a</td>
<td>Send the <code>DATA</code> command.</td>
</tr>
<tr>
<td>QUIT(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Terminate the SMTP session and disconnect.</td>
</tr>
<tr>
<td>RCPT(c IN OUT, recipient [, parameters ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Specify the recipient of an email message.</td>
</tr>
<tr>
<td>RSET(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Terminate the current mail transaction.</td>
</tr>
<tr>
<td>VRFY(c IN OUT, recipient)</td>
<td>Function</td>
<td>REPLY</td>
<td>Validate an email address.</td>
</tr>
<tr>
<td>WRITE_DATA(c IN OUT, data)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Write a portion of the email message.</td>
</tr>
</tbody>
</table>

EDB Postgres Advanced Server’s implementation of `UTL_SMTP` is a partial implementation when compared to Oracle’s version. Only the functions and procedures listed in the table are supported.

The following table lists the public variables available in the `UTL_SMTP` package.

<table>
<thead>
<tr>
<th>Public variables</th>
<th>Data type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>RECORD</td>
<td></td>
<td>Description of an SMTP connection.</td>
</tr>
<tr>
<td>reply</td>
<td>RECORD</td>
<td></td>
<td>SMTP reply line.</td>
</tr>
</tbody>
</table>

**EDB Postgres Advanced Server**
The **CONNECTION** record type provides a description of an SMTP connection.

```plaintext
TYPE connection IS RECORD
(
    host VARCHAR2(255),
    port PLS_INTEGER,
    tx_timeout PLS_INTEGER
);
```

**REPLY/REPLIES**

The **REPLY** record type provides a description of an SMTP reply line. **REPLIES** is a table of multiple SMTP reply lines.

```plaintext
TYPE reply IS RECORD
(
    code INTEGER,
    text VARCHAR2(508)
);
TYPE replies IS TABLE OF reply INDEX BY BINARY_INTEGER;
```

**CLOSE DATA**

The **CLOSE_DATA** procedure terminates an email message by sending the following sequence:

```
<CR><LF>.<CR><LF>
```

This is a single period at the beginning of a line.

```plaintext
CLOSE_DATA(<< IN OUT CONNECTION)
```

**Parameters**

- `<c>`
  The SMTP connection to close.

**COMMAND**

The **COMMAND** procedure executes an SMTP command. If you're expecting multiple reply lines, use **COMMAND_REPLIES**.

```plaintext
<reply> REPLY COMMAND(<< IN OUT CONNECTION, <cmd> VARCHAR2 [, <arg> VARCHAR2 ])
COMMAND(<< IN OUT CONNECTION, <cmd> VARCHAR2 [, <arg> VARCHAR2 ])
```

**Parameters**

- `<c>`
  The SMTP connection to which to send the command.

- `<cmd>`
  The SMTP command to process.

- `<arg>`
  An argument to the SMTP command. The default is null.
reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in reply.

See Reply/Replies for a description of REPLY and REPLIES.

COMMAND_REPLIES

The COMMAND_REPLIES function processes an SMTP command that returns multiple reply lines. Use COMMAND if you expect only a single reply line.

```
<replies> REPLIES COMMAND(<c> IN OUT CONNECTION, <cmd>
VARCHAR2
 [, <arg> VARCHAR2 ])
```

Parameters

- \( c \):
The SMTP connection to which to send the command.
- \( cmd \):
The SMTP command to process.
- \( arg \):
An argument to the SMTP command. The default is null.
- \( replies \):
SMTP reply lines to the command. See Reply/Replies for a description of REPLY and REPLIES.

DATA

The DATA procedure specifies the body of the email message. The message is terminated with a \(<\text{CR}><\text{LF}><\text{CR}><\text{LF}>\) sequence.

```
DATA(<c> IN OUT CONNECTION, <body>
VARCHAR2)
```

Parameters

- \( c \):
The SMTP connection to which to send the command.
- \( body \):
Body of the email message to send.

EHLO

The EHLO procedure performs initial handshaking with the SMTP server after establishing the connection. The EHLO procedure allows the client to identify itself to the SMTP server according to RFC 821. RFC 1869 specifies the format of the information returned in the server's reply. The HELO procedure performs the equivalent functionality but returns less information about the server.

```
EHLO(<c> IN OUT CONNECTION, <domain>
VARCHAR2)
```

Parameters
The connection to the SMTP server over which to perform handshaking.

domain

Domain name of the sending host.

HELO

The HELO procedure performs initial handshaking with the SMTP server after establishing the connection. The HELO procedure allows the client to identify itself to the SMTP server according to RFC 821. The EHLO procedure performs the equivalent functionality but returns more information about the server.

HELO(<c> IN OUT, <domain> VARCHAR2)

Parameters

c

The connection to the SMTP server over which to perform handshaking.

domain

Domain name of the sending host.

HELP

The HELP function sends the HELP command to the SMTP server.

<replies> REPLIES HELP(<c> IN OUT CONNECTION [, <command> VARCHAR2 ])

Parameters

c

The SMTP connection to which to send the command.

command

Command for which you want help.

replies

SMTP reply lines to the command. See Reply/Replies for a description of REPLY and REPLIES.

MAIL

The MAIL procedure initiates a mail transaction.

MAIL(<c> IN OUT CONNECTION, <sender> VARCHAR2 [, <parameters> VARCHAR2 ])

Parameters

c

Connection to SMTP server on which to start a mail transaction.
**sender**
The sender's email address.

**parameters**
Mail command parameters in the format `key=value` as defined in RFC 1869.

**NOOP**
The `NOOP` function/procedure sends the null command to the SMTP server. The `NOOP` has no effect on the server except to obtain a successful response.

```sql
<reply> REPLY NOOP(<c> IN OUT CONNECTION)
NOOP(<c> IN OUT CONNECTION)
```

**Parameters**

- **c**
The SMTP connection on which to send the command.

- **reply**
SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in `reply`. See `Reply/Replies` for a description of `REPLY` and `REPLIES`.

**OPEN_CONNECTION**
The `OPEN_CONNECTION` functions open a connection to an SMTP server.

```sql
<OUT CONNECTION OPEN_CONNECTION(<host> VARCHAR2 [, <port> PLS_INTEGER [, <tx_timeout> PLS_INTEGER DEFAULT NULL]])
```

**Parameters**

- **host**
Name of the SMTP server.

- **port**
Port number on which the SMTP server is listening. The default is 25.

- **tx_timeout**
Timeout value in seconds. Specify 0 to indicate not to wait. Set this value to null to wait indefinitely. The default is null.

- **c**
Connection handle returned by the SMTP server.

**OPEN_DATA**
The `OPEN_DATA` procedure sends the `DATA` command to the SMTP server.

```sql
OPEN_DATA(<c> IN OUT CONNECTION)
```
QUIT

The `QUIT` procedure closes the session with an SMTP server.

```
QUIT(<c> IN OUT CONNECTION)
```

RCPT

The `RCPT` procedure provides the email address of the recipient. To schedule multiple recipients, invoke `RCPT` multiple times.

```
RCPT(<c> IN CONNECTION, <recipient> VARCHAR2 [, <parameters> VARCHAR2 ])
```

RSET

The `RSET` procedure terminates the current mail transaction.

```
RSET(<c> IN OUT CONNECTION)
```

VRFY

The `VRFY` function validates and verifies the recipient's email address. If valid, the recipient's full name and fully qualified mailbox are returned.
Parameters

**c**

The SMTP connection on which to verify the email address.

**recipient**

The recipient's email address to verify.

**reply**

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in `reply`. See **Reply/Replies** for a description of **REPLY** and **REPLIES**.

WRITE_DATA

The **WRITE_DATA** procedure adds **VARCHAR2** data to an email message. You can call the **WRITE_DATA** procedure repeatedly to add data.

Parameters

**c**

The SMTP connection on which to add data.

**data**

Data to add to the email message. The data must conform to the RFC 822 specification.

Comprehensive example

This procedure constructs and sends a text email message using the **UTL_SMTP** package.

```sql
CREATE OR REPLACE PROCEDURE send_mail
(p_sender VARCHAR2,
p_recipient VARCHAR2,
p_subj VARCHAR2,
p_msg VARCHAR2,
p_mailhost VARCHAR2)
IS
v_conn UTL_SMTP.CONNECTION;
v_crlf CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);
v_port CONSTANT PLS_INTEGER := 25;
BEGIN
  v_conn := UTL_SMTP.OPEN_CONNECTION(p_mailhost, v_port);
  UTL_SMTP.HELO(v_conn, p_mailhost);
  UTL_SMTP.MAIL(v_conn, p_sender);
  UTL_SMTP.RCPT(v_conn, p_recipient);
  UTL_SMTP.DATA(v_conn, SUBSTR('Date: ' || TO_CHAR(SYSDATE),
```
This example uses the OPEN_DATA, WRITE_DATA, and CLOSE_DATA procedures instead of the DATA procedure.

14.3.3.1.26 UTL_URL

The UTL_URL package provides a way to escape illegal and reserved characters in a URL.

<table>
<thead>
<tr>
<th>Function/procedure</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESCAPE(url, escape_reserved_chars, url_charset)</td>
<td>VARCHAR2</td>
<td>Use the ESCAPE function to escape any illegal and reserved characters in a URL.</td>
</tr>
<tr>
<td>UNESCAPE(url, url_charset)</td>
<td>VARCHAR2</td>
<td>Use the UNESCAPE function to convert a URL to its original form.</td>
</tr>
</tbody>
</table>

The UTL_URL package returns the BAD_URL exception if the call to a function includes a URL formed incorrectly.
Use the `ESCAPE` function to escape illegal and reserved characters in a URL. The signature is:

```sql
ESCAPE(<url> VARCHAR2, <escape_reserved_chars> BOOLEAN, <url_charset> VARCHAR2)
```

Reserved characters are replaced with a percent sign followed by the two-digit hex code of the ASCII value for the escaped character.

**Parameters**

- **url**
  - Specifies the Uniform Resource Locator that `URL_URL` escapes.

- **escape_reserved_chars**
  - A Boolean value that instructs the `ESCAPE` function to escape reserved characters as well as illegal characters:
    - If `escape_reserved_chars` is `FALSE`, `ESCAPE` escapes only the illegal characters in the specified URL.
    - If `escape_reserved_chars` is `TRUE`, `ESCAPE` escapes both the illegal characters and the reserved characters in the specified URL.

By default, `escape_reserved_chars` is `FALSE`.

In a URL, legal characters are:

- Uppercase A through Z
- Lowercase a through z
- 0 through 9
- Asterisk (*)
- Exclamation point (!)
- Hyphen (-)
- Left parenthesis ( ()
- Period (.)
- Right parenthesis ( )
- Single-quote (’)
- Tilde (~)
- Underscore (_)

Some characters are legal in some parts of a URL while illegal in others. To review comprehensive rules about illegal characters, refer to RFC 2396. Some examples of characters that are considered illegal in any part of a URL are:

<table>
<thead>
<tr>
<th>Illegal character</th>
<th>Escape sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>a blank space ()</td>
<td>%20</td>
</tr>
<tr>
<td>curly braces {{ or }}</td>
<td>%7b and %7d</td>
</tr>
<tr>
<td>hash mark (#)</td>
<td>%23</td>
</tr>
</tbody>
</table>

The `ESCAPE` function treats the following characters as reserved and escapes them if `escape_reserved_chars` is set to `TRUE`:

<table>
<thead>
<tr>
<th>Reserved character</th>
<th>Escape sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ampersand (&amp;)</td>
<td>%5C</td>
</tr>
<tr>
<td>at sign (@)</td>
<td>%25</td>
</tr>
<tr>
<td>colon (;)</td>
<td>%3a</td>
</tr>
<tr>
<td>comma (,)</td>
<td>%2c</td>
</tr>
<tr>
<td>dollar sign ($)</td>
<td>%24</td>
</tr>
<tr>
<td>equal sign (=)</td>
<td>%3d</td>
</tr>
<tr>
<td>plus sign (+)</td>
<td>%2b</td>
</tr>
<tr>
<td>question mark (?)</td>
<td>%3f</td>
</tr>
<tr>
<td>semi-colon (:)</td>
<td>%3b</td>
</tr>
<tr>
<td>slash (/)</td>
<td>%2f</td>
</tr>
</tbody>
</table>

**url_charset**

- Specifies a character set to which to convert a given character before escaping it. If `url_charset` is `NULL`, the character isn’t converted. The default value of `url_charset` is `ISO-8859-1`.

**Examples**
This anonymous block uses the `ESCAPE` function to escape the blank spaces in the URL:

```sql
DECLARE
    result varchar2(400);
BEGIN
    result := UTL_URL.ESCAPE('http://www.example.com/Using the ESCAPE function.html');
    DBMS_OUTPUT.PUT_LINE(result);
END;
```

The resulting escaped URL is:

http://www.example.com/Using%20the%20ESCAPE%20function.html

Suppose you include a value of `TRUE` for the `escape_reserved_chars` parameter when invoking the function:

```sql
DECLARE
    result varchar2(400);
BEGIN
    result := UTL_URL.ESCAPE('http://www.example.com/Using the ESCAPE function.html', TRUE);
    DBMS_OUTPUT.PUT_LINE(result);
END;
```

The `ESCAPE` function escapes the reserved characters as well as the illegal characters in the URL:

http%3A%2Fwww.example.com%2FUsing%20the%20ESCAPE%20function.html

**UNESCAPE**

The `UNESCAPE` function removes escape characters added to an URL by the `ESCAPE` function, converting the URL to its original form.

The signature is:

```
UNESCAPE(<url> VARCHAR2, <url_charset> VARCHAR2)
```

**Parameters**

- `url`
  - Specifies the Uniform Resource Locator that `UTL_URL` unescapes.

- `url_charset`
  - After unescaping a character, the character is assumed to be in `url_charset` encoding and is converted from that encoding to database encoding before being returned. If `url_charset` is `NULL`, the character isn’t converted. The default value of `url_charset` is `ISO-8859-1`.

**Examples**

This anonymous block uses the `ESCAPE` function to escape the blank spaces in the URL:

```sql
DECLARE
    result varchar2(400);
BEGIN
    result := UTL_URL.UNESCAPE('http://www.example.com/Using%20the%20UNESCAPE%20function.html');
    DBMS_OUTPUT.PUT_LINE(result);
END;
```

The resulting (unescape) URL is:
14.3.3.1.27  HTP and HTF

The hypertext procedures (HTP) and hypertext functions (HTF) Oracle packages generate HTML tags. Each HTP has a corresponding HTF. They generate the same HTML tags and use the same parameters, but their syntaxes differ. For example, `htp.address` and `htf.address` both generate the `<ADDRESS>` and `</ADDRESS>` tags and use the same parameters. However, they have different syntaxes:

```
HTP.ADDRESS (cvalue         IN       VARCHAR2
            cnnowrap        IN       VARCHAR2   DEFAULT NULL
            cclear         IN       VARCHAR2   DEFAULT NULL
            cattributes    IN       VARCHAR2   DEFAULT NULL);
```

```
HTF.ADDRESS (cvalue, cnnowrap, cclear, cattributes) RETURN VARCHAR2;
```

### 14.3.3.1.27.1  ADDRESS

This function and procedure generates the `<ADDRESS>` and `</ADDRESS>` tags, which specify the address, author, and signature of a document.

**Syntax**

The following is the syntax for HTP:

```
HTP.ADDRESS (cvalue         IN       VARCHAR2
            cnnowrap        IN       VARCHAR2   DEFAULT NULL
            cclear         IN       VARCHAR2   DEFAULT NULL
            cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.ADDRESS (cvalue, cnnowrap, cclear, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvalue</td>
<td>The string between the <code>&lt;ADDRESS&gt;</code> and <code>&lt;/ADDRESS&gt;</code> tags</td>
</tr>
<tr>
<td>cnnowrap</td>
<td>If the value for this parameter isn’t <code>NULL</code>, adds the NOWRAP attribute in the tag</td>
</tr>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.2  ANCHOR and ANCHOR2

These functions and procedures generate the `<A>` and `</A>` HTML tags, which specify the source or destination of a hypertext link.

**Syntax**

The following is the syntax for HTP:

```
HTP.ANCHOR (curl           IN       VARCHAR2
            ctext          IN       VARCHAR2
            cclear         IN       VARCHAR2   DEFAULT NULL
            cattributes    IN       VARCHAR2   DEFAULT NULL);
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The following is the syntax for HTF:

```
HTF.ANCHOR (curl, ctext, cname, cattributes) RETURN VARCHAR2;
HTF.ANCHOR2 (curl, ctext, cname, ctarget, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>curl</td>
<td>The value for the HREF attribute</td>
</tr>
<tr>
<td>ctext</td>
<td>The string that goes between the <code>&lt;A&gt;</code> and <code>&lt;/A&gt;</code> tags</td>
</tr>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>ctarget</td>
<td>The value for the TARGET attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

#### 14.3.3.1.27.3 APPLETOPEN and APPLETCLOSE

`htp.appletopen` generates the `<APPLET>` HTML tag, which invokes a Java applet. Close the applet with `htp.appletclose`, which generates the `</APPLET>` HTML tag.

### Syntax

The following is the syntax for HTP:

```
HTP.APPLETOPEN(
  ccode          IN       VARCHAR2
  cheight        IN       NUMBER
  cwidth         IN       NUMBER
  cattributes    IN       VARCHAR2   DEFAULT NULL);
```

```
HTP.APPLETCLOSE;
```

The following is the syntax for HTF:

```
HTF.APPLETOPEN(ccode, cheight, cwidth, cattributes) RETURN VARCHAR2;
```

```
HTF.APPLETCLOSE RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccode</td>
<td>The value for the CODE attribute, which specifies the name of the applet class</td>
</tr>
<tr>
<td>cheight</td>
<td>The value for the HEIGHT attribute</td>
</tr>
<tr>
<td>cwidth</td>
<td>The value for the WIDTH attribute</td>
</tr>
</tbody>
</table>

#### 14.3.3.1.27.4 AREA
This function and procedure generates the `<AREA>` HTML tag, which defines a client-side image map.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.AREA(
    ccoords        IN       VARCHAR2
,    cshape         IN       VARCHAR2   DEFAULT NULL
,    chref          IN       VARCHAR2   DEFAULT NULL
,    cnohref        IN       VARCHAR2   DEFAULT NULL
,    ctarget        IN       VARCHAR2   DEFAULT NULL
,    cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.AREA(ccords, cshape, chref, cnohref, ctarget, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccoords</td>
<td>The value for the COORDS attribute</td>
</tr>
<tr>
<td>cshape</td>
<td>The value for the SHAPE attribute</td>
</tr>
<tr>
<td>chref</td>
<td>The value for the HREF attribute</td>
</tr>
<tr>
<td>cnohref</td>
<td>If the value for this parameter isn’t NULL, adds the NOHREF attribute to the tag</td>
</tr>
<tr>
<td>ctarget</td>
<td>The value for the TARGET attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.5 BASE

This function and procedure generates the `<BASE>` HTML tag, which records the URL of the document.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.BASE(ctarget, cattributes) RETURN VARCHAR2;
```

The following is the syntax for HTF:

```sql
HTF.BASE(ctarget, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctarget</td>
<td>The value for the TARGET attribute, which establishes a window name to which all links in this document are targeted</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.6 BASEFONT

This function and procedure generates the `<BASEFONT>` HTML tag, which specifies the base font size for a web page.
Syntax

The following is the syntax for HTP:

```plaintext
HTP.BASEFONT(nsize IN INTEGER);
```

The following is the syntax for HTF:

```plaintext
HTF.BASEFONT(nsize) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsize</td>
<td>The value for the SIZE attribute</td>
</tr>
</tbody>
</table>

**14.3.3.1.27.7 BGSOUND**

This function and procedure generates the `<BGSOUND>` HTML tag, which includes audio for a web page.

Syntax

The following is the syntax for HTP:

```plaintext
HTP.BGSOUND(
  csrc           IN       VARCHAR2
  cloop          IN       VARCHAR2   DEFAULT NULL
  cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```plaintext
HTF.BGSOUND(csrc, cloop, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrc</td>
<td>The value for the SRC attribute</td>
</tr>
<tr>
<td>cloop</td>
<td>The value for the LOOP attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

**14.3.3.1.27.8 BIG**

This function and procedure generates the `<BIG>` and `</BIG>` tags, which increase font size.

Syntax

The following is the syntax for HTP:

```plaintext
HTP.BIG(
  ctext       IN VARCHAR2 CHARACTER SET ANY_CS,
  cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:
14.3.3.1.27.9  BLOCKQUOTEOPEN and BLOCKQUOTECLOSE

These functions and procedures generate the `<BLOCKQUOTE>` and `</BLOCKQUOTE>` tags, which mark a block of quoted text.

### Syntax

The following is the syntax for HTP:

```sql
HTP.BLOCKQUOTEOPEN (cnowrap, cclear, cattributes) RETURN VARCHAR2;
```

```
HTP.blockquoteClose;
```

The following is the syntax for HTF:

```sql
HTF.BLOCKQUOTEOPEN (cnowrap, cclear, cattributes) RETURN VARCHAR2;
```

```sql
HTF.BLOCKQUOTECLOSE RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnowrap</td>
<td>If the value for this parameter isn't <code>NULL</code>, adds the NOWRAP attribute to the tag</td>
</tr>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.10  BODYOPEN and BODYCLOSE

These functions and procedures generate the `<BODY>` and `</BODY>` tags, which mark the body section of an HTML document.

### Syntax

The following is the syntax for HTP:

```sql
HTP.BODYOPEN (cbackground, cattributes) RETURN VARCHAR2;
```

```sql
htp.bodyClose;
```

The following is the syntax for HTF:

```sql
HTF.BODYOPEN (cbackground, cattributes) RETURN VARCHAR2;
```

```sql
HTF.BODYCLOSE RETURN VARCHAR2;
```
HTF.BODYCLOSE RETURN VARCHAR2;

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbackground</td>
<td>The value for the BACKGROUND attribute, which specifies a graphic file to use for the background of the document</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.11 BOLD

This function and procedure generates the `<B>` and `</B>` tags, which display the text in bold.

Syntax

The following is the syntax for HTP:

```sql
HTP.BOLD (ctext          IN       VARCHAR2,
          cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.BOLD (ctext, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text that goes between the tags</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.12 BR and NL

These functions and procedures generate the `<BR>` tag, which begins a new line of text.

Syntax

The following is the syntax for HTP:

```sql
HTP.NL (cclear         IN       VARCHAR2   DEFAULT NULL,
        cattributes    IN       VARCHAR2   DEFAULT NULL);
htp.br (cclear         IN       VARCHAR2   DEFAULT NULL,
        cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.NL (cclear, cattributes) RETURN VARCHAR2;
HTF.BR (cclear, cattributes) RETURN VARCHAR2;
```
**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.13 TABLECAPTION

This procedure and function generates the `<CAPTION>` and `</CAPTION>` tags, which insert a caption in an HTML table.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.TABLECAPTION (ccaption IN VARCHAR2, calign IN VARCHAR2 DEFAULT NULL, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.TABLECAPTION (ccaption, calign, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccaption</td>
<td>The text for the caption</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.14 CENTER

This function and procedure generates the `<CENTER>` and `</CENTER>` tags, which center text in a web page.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.CENTER(ctext IN VARCHAR2);
```

The following is the syntax for HTF:

```sql
HTF.CENTER(ctext IN VARCHAR2) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>Centers the text</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.15 CENTEROPEN and CENTERCLOSE
These functions and procedures generate the `<CENTER>` and `<CENTER>` tags, which mark the text to center.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.CENTEROPEN;
HTP.CENTERCLOSE;
```

The following is the syntax for HTF:

```sql
HTF.CENTEROPEN RETURN VARCHAR2;
HTF.CENTERCLOSE RETURN VARCHAR2;
```

**14.3.3.1.27.16 CITE**

This function and procedure generates the `<CITE>` and `</CITE>` tags, which render text as a citation.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.CITE (ctext IN VARCHAR2, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.CITE (ctext, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render as a citation</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

**14.3.3.1.27.17 CODE**

This function and procedure generates the `<CODE>` and `</CODE>` tags, which render text in a monospace font.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.CODE (ctext IN VARCHAR2, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.CODE (ctext, cattributes) RETURN VARCHAR2;
```
### 14.3.3.1.27.18  COMMENT

This function and procedure generates the comment tags `<-- ctext -->`.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.COMMENT (ctext IN VARCHAR2);
```

The following is the syntax for HTF:

```sql
HTF.COMMENT (ctext IN VARCHAR2) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render as code</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.19  DFN

This function and procedure generates the `<DFN>` and `</DFN>` tags, which render text in italics.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.DFN(ctext IN VARCHAR2);
```

The following is the syntax for HTF:

```sql
HTF.DFN(ctext IN VARCHAR2) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in italics</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.20  DIRLIST

This function and procedure generates the `<DIR>` and `</DIR>` tags, which create a directory list section.
Syntax

The following is the syntax for HTP:

```
HTP.DIRLISTOPEN;
HTP.DIRLISTCLOSE;
```

The following is the syntax for HTF:

```
HTF.DIRLISTOPEN RETURN VARCHAR2;
HTF.DIRLISTCLOSE RETURN VARCHAR2;
```

**14.3.3.1.27.21 DIV**

This function and procedure generates the `<DIV>` tag, which creates document divisions.

Syntax

The following is the syntax for HTP:

```
HTP.DIV(calign IN VARCHAR2 DEFAULT NULL, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.DIV(calign, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

**14.3.3.1.27.22 DLISTDEF**

This function and procedure generates the `<DD>` tag, which inserts a definition of terms. Use this tag for a definition list `<DL>`. Terms are tagged `<DT>`, and definitions are tagged `<DD>`.

Syntax

The following is the syntax for HTP:

```
htp.DLISTDEF(ctext IN VARCHAR2 DEFAULT NULL, cclear IN VARCHAR2 DEFAULT NULL, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.DLISTDEF(ctext, cclear, cattributes) RETURN VARCHAR2;
```

**Parameters**
### 14.3.3.1.27.23 DLISTOPEN and DLISTCLOSE

These functions and procedures generate the `<DL>` and `</DL>` tags, which create a definition list.

#### Syntax

The following is the syntax for HTP:

```sql
HTP.DLISTOPEN (cclear IN VARCHAR2 DEFAULT NULL,
cattributes IN VARCHAR2 DEFAULT NULL);
```

```sql
htp.dlistclose;
```

The following is the syntax for HTF:

```sql
HTF.DLISTOPEN (cclear, cattributes) RETURN VARCHAR2;
```

```sql
HTF.DLISTCLOSE RETURN VARCHAR2;
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.24 DLISTTERM

This function and procedure generates the `<DT>` tag, which defines a term in a definition list `<DL>`.

#### Syntax

The following is the syntax for HTP:

```sql
HTP.DLISTTERM (ctext IN VARCHAR2 DEFAULT NULL,
cclear IN VARCHAR2 DEFAULT NULL,
cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.DLISTTERM (ctext, cclear, cattributes) RETURN VARCHAR2;
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The term</td>
</tr>
</tbody>
</table>
14.3.3.1.27.25  EM and EMPHASIS

This function and procedure generates the `<EM>` and `</EM>` tags, which define text to emphasize.

Syntax

The following is the syntax for HTP:

```sql
HTP.EM (ctext IN VARCHAR2, cattributes IN VARCHAR2 DEFAULT NULL);
htp.emphasis (ctext IN VARCHAR2, cattributes IN VARCHAR2 DEFAULT NULL);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to emphasize</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.26  ESCAPE_SC

This function and procedure replaces characters that have special meaning in HTML with their escape sequences. The following characters are converted:

- `&` to `&amp;`
- `'` to `&quot;`
- `<` to `&lt;`
- `>` to `&gt;`

Syntax

The following is the syntax for HTP:

```sql
HTP.ESCAPE_SC(ctext IN VARCHAR2);
```

The following is the syntax for HTF:

```sql
HTF.ESCAPE_SC(ctext in VARCHAR2) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to emphasize</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>
### 14.3.3.1.27.27  FONT

This function and procedure generates the `<FONT>` and `</FONT>` tags, which mark text with the specified font characteristics.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.FONTOPEN(
  ccolor         IN       VARCHAR2   DEFAULT NULL,
  cface          IN       VARCHAR2   DEFAULT NULL,
  csize          IN       VARCHAR2   DEFAULT NULL,
  cattributes    IN       VARCHAR2   DEFAULT NULL);
HTP.FontClose;
```

The following is the syntax for HTF:

```sql
HTF.FONTOPEN(ccolor, cface, csize, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccolor</td>
<td>The value for the COLOR attribute</td>
</tr>
<tr>
<td>cface</td>
<td>The value for the FACE attribute</td>
</tr>
<tr>
<td>csize</td>
<td>The value for the SIZE attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.28  FORMCHECKBOX

This function and procedure generates the `<INPUT>` tag with `TYPE="checkbox"`. This tag inserts a check box element in a form.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.FORMCHECKBOX (cname, cvalue, cchecked, cattributes)
```

The following is the syntax for HTF:

```sql
HTF.FORMCHECKBOX (cname, cvalue, cchecked, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>cvalue</td>
<td>The value for the VALUE attribute</td>
</tr>
<tr>
<td>cchecked</td>
<td>The value for the CHECKED attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>
### 14.3.3.1.27.29 FORMFILE

This function and procedure generates the `<INPUT>` tag with `TYPE="file"`. This tag allows the user to select files whose contents can be submitted with a form.

#### Syntax

The following is the syntax for HTP:

```sql
HTP.FORMFILE(
    cname          IN       varchar2,
    caccept        IN       varchar2 DEFAULT NULL,
    cattributes    IN       varchar2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.FORMFILE (cname, caccept, cattributes) RETURN VARCHAR2;
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>caccept</td>
<td>A comma-separated list of MIME types for upload</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.30 FORMHIDDEN

This function and procedure generates the `<INPUT>` tag with `TYPE="hidden"`. This tag inserts a hidden form element that the user doesn’t see. It submits additional values to the script.

#### Syntax

The following is the syntax for HTP:

```sql
HTP.FORMHIDDEN (
    cname          IN       VARCHAR2,
    cvalue         IN       VARCHAR2 DEFAULT NULL,
    cattributes    IN       VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.FORMHIDDEN (cname, cvalue, cattributes) RETURN VARCHAR2;
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
</tbody>
</table>
### 14.3.3.1.27.31 FORMIMAGE

This function and procedure generates the `<INPUT>` tag with `TYPE="image"`. This tag creates an image field that the user clicks to submit the form.

#### Syntax

The following is the syntax for HTP:

```sql
HTP.FORMIMAGE (cname          IN       VARCHAR2
csrc           IN       VARCHAR2
calign         IN       VARCHAR2   DEFAULT NULL
cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.FORMIMAGE (cname, csrc, calign, cattributes) RETURN VARCHAR2;
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cname</code></td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td><code>csrc</code></td>
<td>The value for the SRC attribute, which specifies the image file</td>
</tr>
<tr>
<td><code>calign</code></td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td><code>cattributes</code></td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.32 FORMOPEN and FORMCLOSE

This function and procedure generates the `<FORM>` and `</FORM>` tags, which create a form section in an HTML document.

#### Syntax

The following is the syntax for HTP:

```sql
HTP.FORMOPEN (curl           IN       VARCHAR2
cmethod        IN       VARCHAR2   DEFAULT 'POST'
ctarget        IN       VARCHAR2   DEFAULT NULL
cenctype       IN       VARCHAR2   DEFAULT NULL
cattributes    IN       VARCHAR2   DEFAULT NULL);
htp.FORMCLOSE;
```

The following is the syntax for HTF:

```sql
HTF.FORMOPEN (curl, cmethod, ctarget, cenctype, cattributes) RETURN VARCHAR2;
HTF.FORMCLOSE RETURN VARCHAR2;
```

#### Parameters
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>curl</td>
<td>The URL of the WRB or CGI script where the contents of the form are sent (required)</td>
</tr>
<tr>
<td>cmethod</td>
<td>The value for the METHOD attribute, either 'GET' or 'POST'</td>
</tr>
<tr>
<td>ctarget</td>
<td>The value for the TARGET attribute</td>
</tr>
<tr>
<td>cencetype</td>
<td>The value for the ENCTYPE attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.33 FORMPASSWORD

This function and procedure generates the `<INPUT>` tag with `TYPE="password"`. This tag creates a single-line text entry field.

**Syntax**

The following is the syntax for HTP:

```
HTP.FORMPASSWORD (cname IN VARCHAR2, csize IN VARCHAR2, cmaxlength IN VARCHAR2 DEFAULT NULL, cvalue IN VARCHAR2 DEFAULT NULL, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.FORMPASSWORD (cname, csize, cmaxlength, cvalue, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>csize</td>
<td>The value for the SIZE attribute</td>
</tr>
<tr>
<td>cmaxlength</td>
<td>The value for the MAXLENGTH attribute</td>
</tr>
<tr>
<td>cvalue</td>
<td>The value for the VALUE attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.34 FORMRADIO

This function and procedure generates the `<INPUT>` tag with `TYPE="radio"`. This tag creates a radio button (single-selection option) on the HTML form.

**Syntax**

The following is the syntax for HTP:

```
HTP.FORMRADIO (cname IN VARCHAR2, cvalue IN VARCHAR2, cchecked IN VARCHAR2 DEFAULT NULL, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.FORMRADIO (cname, cvalue, cchecked, cattributes) RETURN VARCHAR2;
```
Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>cvalue</td>
<td>The value for the VALUE attribute</td>
</tr>
<tr>
<td>cchecked</td>
<td>If the value for this parameter isn't NULL, adds the CHECKED attribute to the tag</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.35 FORMRESET

This function and procedure generates the `<INPUT>` tag with TYPE=”reset”. This tag creates a button that resets the form fields to their initial values.

Syntax

The following is the syntax for HTP:

```
HTP.FORMRESET (cvalue IN VARCHAR2 DEFAULT 'Reset', cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.FORMRESET (cvalue, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvalue</td>
<td>The value for the VALUE attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.36 FROMSELECTOPEN and FORMSELECTCLOSE

This function and procedure generates the `<SELECT>` and `</SELECT>` tags, which create a select form element. A select form element is a list box from which the user selects one or more values.

Syntax

The following is the syntax for HTP:

```
Htp.FORMSELECTOPEN (cname IN VARCHAR2, cprompt IN VARCHAR2 DEFAULT NULL, nsize IN INTEGER DEFAULT NULL, cattributes IN VARCHAR2 DEFAULT NULL);

htp.formSelectClose;
```

The following is the syntax for HTF:

```
HTF.FORMSELECTOPEN (cname, cprompt, nsize, cattributes) RETURN VARCHAR2;

HTF.FORMSELECTCLOSE RETURN VARCHAR2;
```

Parameters
### 14.3.3.1.27.37 FORMSELECTOPTION

This function and procedure generates the `<OPTION>` tag, which represents one choice in a `<SELECT>` element.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.FORMSELECTOPTION (cvalue         IN       VARCHAR2
                     cselected      IN       VARCHAR2   DEFAULT NULL
                     cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.FORMSELECTOPTION (cvalue, cselected, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvalue</td>
<td>The text for the option</td>
</tr>
<tr>
<td>cselected</td>
<td>If the value for this parameter isn't NULL, adds the SELECTED attribute to the tag</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.38 FORMSUBMIT

This function and procedure generates the `<INPUT>` tag with `TYPE="submit"`. This tag creates a button that submits the form.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.FORMSUBMIT (cname          IN       VARCHAR2   DEFAULT NULL
                cvalue         IN       VARCHAR2   DEFAULT 'Submit'
                cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.FORMSUBMIT (cname, cvalue, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
</tbody>
</table>
The value for the VALUE attribute
Other attributes to include as is in the tag

**14.3.3.1.27.39 FORMTEXT**

This function and procedure generates the `<INPUT>` tag with `TYPE="text"`. This tag creates a field for a single line of text.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.FORMTEXT (cname IN VARCHAR2,
csize IN VARCHAR2 DEFAULT NULL,
cmaxlength IN VARCHAR2 DEFAULT NULL,
cvalue IN VARCHAR2 DEFAULT NULL,
cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTP.FORMTEXT (cname, csize, cmaxlength, cvalue, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>csize</td>
<td>The value for the SIZE attribute</td>
</tr>
<tr>
<td>cmaxlength</td>
<td>The value for the MAXLENGTH attribute</td>
</tr>
<tr>
<td>cvalue</td>
<td>The value for the VALUE attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

**14.3.3.1.27.40 FORMTEXTAREAOPEN, FORMTEXTAREAOPEN2, and FORMTEXTAREACLOSE**

These functions and procedures generate the `<TEXTAREA>` and `</TEXTAREA>` tags, which create a text area form element. The difference between the two open functions and procedures is that `http.formTextareaOpen2` has the `cwrap` parameter, which specifies a wrap style.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.FORMTEXTAREAOPEN (cname IN VARCHAR2,
nrows IN INTEGER,
columns IN INTEGER,
calign IN VARCHAR2 DEFAULT NULL,
cattributes IN VARCHAR2 DEFAULT NULL);
```

```sql
HTP.FORMTEXTAREAOPEN2 (cname IN VARCHAR2,
nrows IN INTEGER,
columns IN INTEGER,
calign IN VARCHAR2 DEFAULT NULL,
cwrap IN VARCHAR2 DEFAULT NULL,
cattributes IN VARCHAR2 DEFAULT NULL);
```

```sql
HTP.FORMTEXTAREACLOSE;
```
The following is the syntax for HTF:

```
HTF.FORMTEXTAREAOPEN (cname, nrows, ncolumns, calign, cattributes) RETURN VARCHAR2;
HTF.FORMTEXTAREAOPEN2(cname, nrows, ncolumns, calign, cwrap, cattributes) RETURN VARCHAR2;
HTF.FORMTEXTAREACLOSE RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>nrows</td>
<td>The integer value for the ROWS attribute</td>
</tr>
<tr>
<td>ncolumns</td>
<td>The integer value for the COLS attribute</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cwrap</td>
<td>The value for the WRAP attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.41 FORMTEXTAREA and FORMTEXTAREA2

These functions and procedures generate the `<TEXTAREA>` tag, which creates a text field that has no predefined text in the text area. This field allows the user to enter several lines of text.

### Syntax

The following is the syntax for HTP:

```
HTP.FORMTEXTAREA (
    cname          IN       VARCHAR2
    nrows          IN       INTEGER
    ncolumns       IN       INTEGER
    calign         IN       VARCHAR2   DEFAULT NULL
    cattributes    IN       VARCHAR2   DEFAULT NULL);
```

```
HTP.FORMTEXTAREA2 (
    cname          IN       VARCHAR2
    nrows          IN       INTEGER
    ncolumns       IN       INTEGER
    calign         IN       VARCHAR2   DEFAULT NULL
    cwrap          IN       VARCHAR2   DEFAULT NULL
    cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.FORMTEXTAREA (cname, nrows, ncolumns, calign, cattributes) RETURN VARCHAR2;
HTF.FORMTEXTAREA2 (cname, nrows, ncolumns, calign, cwrap, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>nrows</td>
<td>The integer value for the ROWS attribute</td>
</tr>
<tr>
<td>ncolumns</td>
<td>The integer value for the COLS attribute</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cwrap</td>
<td>The value for the WRAP attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.42 FRAME
This function and procedure generates the `<FRAME>` tag, which defines the characteristics of a frame created by a `<FRAMESET>` tag.

### Syntax

The following is the syntax for HTP:

```
HTP.FRAME(
  csrc           IN       VARCHAR2
, cname          IN       VARCHAR2   DEFAULT NULL
, cmarginwidth   IN       VARCHAR2   DEFAULT NULL
, cmarginheight  IN       VARCHAR2   DEFAULT NULL
, cscrolling     IN       VARCHAR2   DEFAULT NULL
, cnoresize      IN       VARCHAR2   DEFAULT NULL
, cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.FRAME (csrc, cname, cmarginwidth, cmarginheight, cscrolling, cnoresize, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>csrc</td>
<td>The URL to display in the frame</td>
</tr>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>cmarginwidth</td>
<td>The value for the MARGINWIDTH attribute</td>
</tr>
<tr>
<td>cmarginheight</td>
<td>The value for the MARGINHEIGHT attribute</td>
</tr>
<tr>
<td>cscrolling</td>
<td>The value for the SCROLLING attribute</td>
</tr>
<tr>
<td>cnoresize</td>
<td>If the value for this parameter isn't NULL, adds the NORESIZE attribute to the tag</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.43 FRAMESETOPEN and FRAMESETCLOSE

These functions and procedures generate the `<FRAMESET>` and `</FRAMESET>` tags, which define a frame set section.

### Syntax

The following is the syntax for HTP:

```
HTP.FRAMESETOPEN(
  crows          IN       VARCHAR2   DEFAULT NULL
, ccols          IN       VARCHAR2   DEFAULT NULL
, cattributes    IN       VARCHAR2   DEFAULT NULL);

HTP.FRAMESETCLOSE;
```

The following is the syntax for HTF:

```
HTF.FRAMESETOPEN (crows, ccols, cattributes) RETURN VARCHAR2;

HTF.FRAMESETCLOSE RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>crows</td>
<td>The value for the ROWS attribute</td>
</tr>
<tr>
<td>ccols</td>
<td>The value for the COLS attribute</td>
</tr>
</tbody>
</table>
**14.3.3.1.27.44   HEADER**

This function and procedure generates heading tags (<H1> to <H6>) and their corresponding closing tags (<H1> to <H6>).

**Syntax**

The following is the syntax for HTP:

```sql
HTP.HEADER (nsize, cheader, calign, cnowrap, cclear, cattributes);
```

The following is the syntax for HTF:

```sql
HTF.HEADER (nsize, cheader, calign, cnowrap, cclear, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsize</td>
<td>The heading level, set as an integer between 1 and 6</td>
</tr>
<tr>
<td>cheader</td>
<td>The text to display in the heading</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cnowrap</td>
<td>The value for the NOWRAP attribute</td>
</tr>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

**14.3.3.1.27.45   HEADOPEN and HEADCLOSE**

This function and procedure generates the `<HEAD>` and `</HEAD>` tags, which mark the HTML document head section.

**Syntax**

The following is the syntax for HTP:

```sql
HTP.HEADOOPEN;
HTP.HEADCLOSE;
```

The following is the syntax for HTF:

```sql
HTF.HEADOOPEN RETURN VARCHAR2;
HTF.HEADCLOSE RETURN VARCHAR2;
```

**14.3.3.1.27.46   HR**
This function and procedure generates the `<HR>` tag, which generates a line in the HTML document.

### Syntax

The following is the syntax for HTP:

```sql
HTP.HR (cclear IN VARCHAR2 DEFAULT NULL,
csrc IN VARCHAR2 DEFAULT NULL,
cattributes IN VARCHAR2 DEFAULT NULL);

HTP.LINE (cclear IN VARCHAR2 DEFAULT NULL,
csrc IN VARCHAR2 DEFAULT NULL,
cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.HR (cclear, csrc, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>csrc</td>
<td>The value for the SRC attribute, which specifies a custom image as the source of the line</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.47 HTMLOPEN and HTMLCLOSE

These functions and procedures generate the `<HTML>` and `</HTML>` tags, which mark the beginning and the end of an HTML document.

### Syntax

The following is the syntax for HTP:

```sql
HTP.HTMLOPEN;

HTP.HTMLCLOSE;
```

The following is the syntax for HTF:

```sql
HTF.HTMLOPEN RETURN VARCHAR2;

HTF.HTMLCLOSE RETURN VARCHAR2;
```

### 14.3.3.1.27.48 IMG and IMG2

These functions and procedures generate the `<IMG>` tag, which loads an image onto the HTML page. The difference between these syntaxes is that `http.img2` uses the `cusemap` parameter.

### Syntax

The following is the syntax for HTP:

```sql
```
The following is the syntax for HTP:

```sql
HTP.IMG (curl, calign, calt, cismap, cattributes) RETURN VARCHAR2;
```

```sql
HTF.IMG (curl, calign, calt, cismap, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>curl</td>
<td>The value for the SRC attribute.</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute.</td>
</tr>
<tr>
<td>calt</td>
<td>The value for the ALT attribute, which specifies alternative text to display if the browser doesn't support images.</td>
</tr>
<tr>
<td>cismap</td>
<td>If the value for this parameter isn’t NULL, adds the ISMAP attribute to the tag. The attribute indicates that the image is an imagemap.</td>
</tr>
<tr>
<td>cusemap</td>
<td>The value for the USEMAP attribute, which specifies a client-side image map.</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag.</td>
</tr>
</tbody>
</table>

14.3.3.1.27.49  ISINDEX

This function and procedure creates a single entry field with a text prompt, such as "Enter value!", and then sends that value to the URL of the page or program.

Syntax

The following is the syntax for HTP:

```sql
HTP.ISINDEX (cprompt, curl) RETURN VARCHAR2;
```

The following is the syntax for HTF:

```sql
HTF.ISINDEX (cprompt, curl) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cprompt</td>
<td>The value for the PROMPT attribute.</td>
</tr>
<tr>
<td>curl</td>
<td>The value for the HREF attribute.</td>
</tr>
</tbody>
</table>

14.3.3.1.27.50  ITALIC

This function and procedure generates the `<i>` and `</i>` tags, which render the text in italics.
Syntax

The following is the syntax for HTP:

```sql
HTP.ITALIC (ctext          IN       VARCHAR2
          , cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.ITALIC (ctext, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to be rendered in italics</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.51 KBD and KEYBOARD

These functions and procedures generate the `<KBD>` and `</KBD>` tags, which render the text in a monospace font.

Syntax

The following is the syntax for HTP:

```sql
HTP.KEYBOARD (ctext          IN       VARCHAR2
            , cattributes    IN       VARCHAR2   DEFAULT NULL);
```

HTP.KBD (ctext          IN       VARCHAR2
         , cattributes    IN       VARCHAR2   DEFAULT NULL);

The following is the syntax for HTF:

```sql
HTF.KEYBOARD (ctext, cattributes) RETURN VARCHAR2;
```

HTF.KBD (ctext, cattributes) RETURN VARCHAR2;

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in monospace</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.52 LINKREL

This function and procedure generates the `<LINK>` tag with the REL attribute. The REL attribute is the relationship described by the hypertext link from the anchor to the target.

Syntax

The following is the syntax for HTP:

```sql
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The following is the syntax for HTP:

```
HTP.LINKREL (crel, curl, ctitle) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>crel</td>
<td>The value for the REL attribute</td>
</tr>
<tr>
<td>curl</td>
<td>The value for the HREF attribute</td>
</tr>
<tr>
<td>ctitle</td>
<td>The value for the TITLE attribute</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.53 LINKREV

This function and procedure generates the `<LINK>` tag with the REV attribute. The REV attribute is the relationship described by the hypertext link from the target to the anchor.

#### Syntax

The following is the syntax for HTP:

```
HTP.LINKREV (crev, curl, ctitle) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>crev</td>
<td>The value for the REV attribute</td>
</tr>
<tr>
<td>curl</td>
<td>The value for the HREF attribute</td>
</tr>
<tr>
<td>ctitle</td>
<td>The value for the TITLE attribute</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.54 LISTHEADER

This function and procedure generates the `<LH>` and `</LH>` tags, which print an HTML tag at the beginning of the list.

#### Syntax

The following is the syntax for HTP:

```
HTP.LISTHEADER (ctext, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The value for the LISTS attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>The value for the ATTRIBUTES attribute</td>
</tr>
</tbody>
</table>
14.3.3.1.27.55 LISTINGOPEN and LISTINGCLOSE

These functions and procedures generate the `<LISTING>` and `</LISTING>` tags, which mark a section of fixed-width text in the body of an HTML page.

Syntax

The following is the syntax for HTP:

```sql
HTP.LISTINGOPEN;
HTP.LISTINGCLOSE;
```

The following is the syntax for HTF:

```sql
HTF.LISTINGOPEN RETURN VARCHAR2;
HTF.LISTINGCLOSE RETURN VARCHAR2;
```

14.3.3.1.27.56 LISTITEM

This function and procedure generates the `<LI>` tag, which creates a list item.

Syntax

The following is the syntax for HTP:

```sql
HTP.LISTITEM (ctext IN VARCHAR2 DEFAULT NULL,
cclear IN VARCHAR2 DEFAULT NULL,
cdingbat IN VARCHAR2 DEFAULT NULL,
csrc IN VARCHAR2 DEFAULT NULL,
cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF.LISTITEM (ctext, cclear, cdingbat, csrc, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text for the list item</td>
</tr>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cdingbat</td>
<td>The value for the DINGBAT attribute</td>
</tr>
<tr>
<td>csrc</td>
<td>The value for the SRC attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>
14.3.3.1.27.57  MAILTO

This function and procedure generates the `<A>` tag with HREF set to `mailto` prepended to the mail address argument.

Syntax

The following is the syntax for HTP:

```
HTP.MAILTO (caddress       IN       VARCHAR2
             ctext          IN       VARCHAR2
             cname          IN       VARCHAR2
             cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.MAILTO (caddress, ctext, cname, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>caddress</td>
<td>The email address of the recipient</td>
</tr>
<tr>
<td>ctext</td>
<td>The clickable portion of the link</td>
</tr>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.58  MAPOPEN and MAPCLOSE

This function and procedure generates the `<MAP>` and `</MAP>` tags, which mark a set of regions in a client-side image map.

Syntax

The following is the syntax for HTP:

```
HTP.MAPOPEN(   
cname          IN       VARCHAR2
             cattributes    IN       VARCHAR2   DEFAULT NULL);
```

```
HTP.MAPCLOSE;
```

The following is the syntax for HTF:

```
HTF.MAPOPEN(cname, cattributes) RETURN VARCHAR2;
```

```
HTF.MAPCLOSE RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.59  MENULISTOPEN and MENULISTCLOSE

1012
These function and procedure generate the `<MENU>` and `</MENU>` tags, which create a list consisting of one line for each item.

**Syntax**

The following is the syntax for HTP:

```
HTP.MENULISTOPEN;
HTP.MENULISTCLOSE;
```

The following is the syntax for HTF:

```
HTP.MENULISTOPEN RETURN VARCHAR2;
HTP.MENULISTCLOSE RETURN VARCHAR2;
```

**14.3.3.1.27.60 META**

This function and procedure generates the `<META>` tag, which embeds meta-information about the document and specifies values for HTTP headers.

**Syntax**

The following is the syntax for HTP:

```
HTP.META (chttp_equiv IN VARCHAR2,
           cname IN VARCHAR2,
           ccontent IN VARCHAR2);
```

The following is the syntax for HTF:

```
HTF.META (chttp_equiv, cname, ccontent) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>chttp_equiv</td>
<td>The value for the HTTP-EQUIV attribute</td>
</tr>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>ccontent</td>
<td>The value for the CONTENT attribute</td>
</tr>
</tbody>
</table>

**14.3.3.1.27.61 NOBR**

This function and procedure generates the `<NOBR>` and `</NOBR>` tags, which turn off line breaks for the tagged text.

**Syntax**

The following is the syntax for HTP:

```
HTP.NOBR(ctext in VARCHAR2);
```

The following is the syntax for HTF:

```
HTF.NOBR(ctext) RETURN VARCHAR2;
```
### 14.3.3.1.27.62 NOFRAMESOPEN and NOFRAMESCLOSE

This function and procedure generates the `<NOFRAMES>` and `</NOFRAMES>` tags, which mark a no-frames section.

#### Syntax

The following is the syntax for HTP:

```plsql
HTP.NOFRAMESOPEN
HTP.NOFRAMESCLOSE
```

The following is the syntax for HTF:

```plsql
HTF.NOFRAMESOPEN RETURN VARCHAR2;
HTF.NOFRAMESCLOSE RETURN VARCHAR2;
```

### 14.3.3.1.27.63 OLISTOPEN and OLISTCLOSE

These functions and procedures generate the `<OL>` and `</OL>` tags, which define an ordered (numbered) list.

#### Syntax

The following is the syntax for HTP:

```plsql
HTP.OLISTOPEN (cclear IN VARCHAR2 DEFAULT NULL,
cwrap IN VARCHAR2 DEFAULT NULL,
cattributes IN VARCHAR2 DEFAULT NULL);
HTP.OLISTCLOSE;
```

The following is the syntax for HTF:

```plsql
HTF.OLISTOPEN (cclear, cwrap, cattributes) RETURN VARCHAR2;
HTF.OLISTCLOSE RETURN VARCHAR2;
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cwrap</td>
<td>The value for the WRAP attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.64 PARA and PARAGRAPH
These functions and procedures generate the `<P>` tag, which marks the text as a paragraph. `htp.paragraph` enables you to add attributes to the tag.

### Syntax

The following is the syntax for HTP:

```sql
HTP_PARA;

HTP_PARAGRAPH (calign IN VARCHAR2 DEFAULT NULL, cnwrap IN VARCHAR2 DEFAULT NULL, cclear IN VARCHAR2 DEFAULT NULL, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTF_PARA RETURN VARCHAR2;

HTF_PARAGRAPH (calign, cnwrap, cclear, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cnwrap</td>
<td>If the value for this parameter isn’t <code>NULL</code>, adds the NOWRAP attribute to the tag</td>
</tr>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.65 PARAM

This function and procedure generates the `<PARAM>` tag, which specifies parameter values for Java applets.

### Syntax

The following is the syntax for HTP:

```sql
HTP_PARAM(
cname IN VARCHAR2,
cvalue IN VARCHAR2);
```

The following is the syntax for HTF:

```sql
HTF_PARAM(cname, cvalue) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>The value for the NAME attribute</td>
</tr>
<tr>
<td>cvalue</td>
<td>The value for the VALUE attribute</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.66 PLAINTEXT
This function and procedure generates the `<PLAINTEXT>` and `</PLAINTEXT>` tags, which render text in a fixed-width font.

**Syntax**

The following is the syntax for HTP:

```
HTP.PLAINTEXT(
    ctext          IN       VARCHAR2
    cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.PLAINTEXT(ctext, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in fixed-width font</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

**14.3.3.1.27.67 PREOPEN and PRECLOSE**

These functions and procedures generate the `<PRE>` and `</PRE>` tags, which mark preformatted text in the body of the HTML page.

**Syntax**

The following is the syntax for HTP:

```
HTP.PREOPEN (
    cclear         IN       VARCHAR2   DEFAULT NULL
    cwidth         IN       VARCHAR2   DEFAULT NULL
    cattributes    IN       VARCHAR2   DEFAULT NULL);

HTP.PRECLOSE;
```

The following is the syntax for HTF:

```
HTF.PREOPEN (cclear, cwidth, cattributes) RETURN VARCHAR2;

HTF.PRECLOSE RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cwidth</td>
<td>The value for the WIDTH attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

**14.3.3.1.27.68 PRINT and PRN**

`htp.print` generates the specified parameter as a string terminated with the `\n` newline character. The `\n` character is different from `<BR>`. `htp.prn` generates the specified parameter as a string. Unlike `print`, the string isn’t terminated with the `\n` newline character.

**Note**
PRINT and PRN operate only as HTPs.

Syntax

The following is the syntax for HTP:

```sql
HTP.PRINT (cbuf IN VARCHAR2);
HTP.PRINT (dbuf IN DATE);
HTP.PRINT (nbuf IN NUMBER);

HTP.PRN (cbuf IN VARCHAR2);
HTP.PRN (dbuf IN DATE);
HTP.PRN (nbuf IN NUMBER);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbuf, dbuf, and nbuf</td>
<td>The string to generate</td>
</tr>
</tbody>
</table>

14.3.3.1.27.69 PRINTS and PS

These procedures generate a string and replace the following characters with the corresponding escape sequence:

- `<` with `&lt;`
- `>` with `&gt;`
- `"` with `&quot;`
- `&` with `&amp;`

Note

PRINTS and PS operate only as HTPs.

Syntax

The following is the syntax for HTP:

```sql
HTP.PRINTS (ctext IN VARCHAR2);
HTP.PS (ctext IN VARCHAR2);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The string on which to perform character substitution</td>
</tr>
</tbody>
</table>

14.3.3.1.27.70 S

This function and procedure generates the `<S>` and `</S>` tags, which render the text with a strikethrough.

Syntax

The following is the syntax for HTP:
The following is the syntax for HTF:

```sql
HTP.S(ctext, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render with a strikethrough</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.71  **SAMPLE**

This function and procedure generates the `<SAMP>` and `</SAMP>` tags, which render text in a monospace font.

### Syntax

The following is the syntax for HTP:

```sql
HTP.SAMPLE (ctext, cattributes) RETURN VARCHAR2;
```

The following is the syntax for HTF:

```sql
HTP.SAMPLE (ctext, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in a monospace font</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.72  **SMALL**

This function and procedure generates the `<SMALL>` and `</SMALL>` tags, which render text in a small font.

### Syntax

The following is the syntax for HTP:

```sql
HTP.SMALL (ctext, cattributes) RETURN VARCHAR2;
```

The following is the syntax for HTF:

```sql
HTP.SMALL (ctext, cattributes) RETURN VARCHAR2;
```
14.3.3.1.27.73 STRIKE

This function and procedure generates the `<STRIKE>` and `</STRIKE>` tags, which render text with a strikethrough.

Syntax

The following is the syntax for HTP:

```sql
HTP.STRIKE (ctext IN VARCHAR2, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTP.STRIKE (ctext, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in a small font</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.74 STRONG

This function and procedure generates the `<STRONG>` and `</STRONG>` tags, which render text in bold.

Syntax

The following is the syntax for HTP:

```sql
HTP.STRONG (ctext IN VARCHAR2, cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTP.STRONG (ctext, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in bold</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>
14.3.3.1.27.75  STYLE

This function and procedure generates the `<STYLE>` and `</STYLE>` tags, which include a style sheet in the web page.

Syntax

The following is the syntax for HTP:

```
HTP.STYLE(cstyle IN VARCHAR2);
```

The following is the syntax for HTF:

```
HTF.STYLE(cstyle) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstyle</td>
<td>The style information to include</td>
</tr>
</tbody>
</table>

14.3.3.1.27.76  SUB

This function and procedure generates the `<SUB>` and `</SUB>` tags, which render text in subscript.

Syntax

The following is the syntax for HTP:

```
HTP.SUB(
  ctext          IN       VARCHAR2
, calign         IN       VARCHAR2   DEFAULT NULL
, cattributes    IN       VARCHAR2   DEFAULT NULL)
```

The following is the syntax for HTF:

```
HTP.SUB(ctext, calign, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in subscript</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.77  SUP

This function and procedure generates the `<SUP>` and `</SUP>` tags, which render text in superscript.

Syntax

```

```
The following is the syntax for HTP:

```sql
HTP.SUP(
    ctext          IN       VARCHAR2
    calign         IN       VARCHAR2   DEFAULT NULL
    cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTP.SUP(ctext, calign, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in superscript</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.78  TABLEDATA

This function and procedure generates the `<TD>` and `</TD>` tags, which insert data into a cell of an HTML table.

### Syntax

The following is the syntax for HTP:

```sql
HTP.TABLEDATA ( 
    cvalue         IN       VARCHAR2   DEFAULT NULL
    calign         IN       VARCHAR2   DEFAULT NULL
    cdp            IN       VARCHAR2   DEFAULT NULL
    cnowrap        IN       VARCHAR2   DEFAULT NULL
    crowspan       IN       VARCHAR2   DEFAULT NULL
    ccolspan       IN       VARCHAR2   DEFAULT NULL
    cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTP.TABLEDATA (cvalue, calign, cdp, cnowrap, crowspan, ccolspan, cattributes) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvalue</td>
<td>The data for the cell in the table</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cdp</td>
<td>The value for the DP attribute</td>
</tr>
<tr>
<td>cnowrap</td>
<td>If the value of this parameter isn't NULL, adds the NOWRAP attribute to the tag</td>
</tr>
<tr>
<td>crowspan</td>
<td>The value for the ROWSPAN attribute</td>
</tr>
<tr>
<td>ccolspan</td>
<td>The value for the COLSPAN attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

14.3.3.1.27.79  TABLEHEADER

This function and procedure generates the `<TH>` and `</TH>` tags, which insert a header cell in an HTML table. The `<TH>` tag is similar to the `<TD>` tag except that the text in the row is usually rendered in bold.
Syntax

The following is the syntax for HTP:

```sql
HTP.TABLEHEADER (cvalue IN VARCHAR2 DEFAULT NULL,
    calign IN VARCHAR2 DEFAULT NULL,
    cdp IN VARCHAR2 DEFAULT NULL,
    cnwrap IN VARCHAR2 DEFAULT NULL,
    crowspan IN VARCHAR2 DEFAULT NULL,
    ccolspan IN VARCHAR2 DEFAULT NULL,
    cattributes IN VARCHAR2 DEFAULT NULL);
```

The following is the syntax for HTF:

```sql
HTP.TABLEHEADER (cvalue, calign, cdp, cnwrap, crowspan, ccolspan, cattributes) RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvalue</td>
<td>The data for the cell in the table.</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute.</td>
</tr>
<tr>
<td>cdp</td>
<td>The value for the DP attribute.</td>
</tr>
<tr>
<td>cnwrap</td>
<td>If the value of this parameter is not NULL, adds the NOWRAP attribute to the tag.</td>
</tr>
<tr>
<td>crowspan</td>
<td>The value for the ROWSPAN attribute.</td>
</tr>
<tr>
<td>ccolspan</td>
<td>The value for the COLSPAN attribute.</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag.</td>
</tr>
</tbody>
</table>

14.3.1.27.80 TABLEOPEN and TABLECLOSE

This function and procedure generates the `<TABLE>` and `</TABLE>` tags, which define an HTML table.

Syntax

The following is the syntax for HTP:

```sql
HTP.TABLEOPEN (cborder IN VARCHAR2 DEFAULT NULL,
    calign IN VARCHAR2 DEFAULT NULL,
    cnwrap IN VARCHAR2 DEFAULT NULL,
    cclear IN VARCHAR2 DEFAULT NULL,
    cattributes IN VARCHAR2 DEFAULT NULL);
```

```sql
HTP.TABLECLOSE;
```

The following is the syntax for HTF:

```sql
HTF.TABLEOPEN (cborder, calign, cnwrap, cclear, cattributes) RETURN VARCHAR2;
```

```sql
HTF.TABLECLOSE RETURN VARCHAR2;
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cborder</td>
<td>The value for the BORDER attribute.</td>
</tr>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute.</td>
</tr>
</tbody>
</table>
If the value of this parameter isn't NULL, adds the NOWRAP attribute to the tag.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>nowrap</td>
<td>If the value of this parameter isn't NULL, adds the NOWRAP attribute to the tag</td>
</tr>
<tr>
<td>clear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>attributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.81 TABLEROWOPEN and TABLEROWCLOSE

This function and procedure generates the `<TR>` and `</TR>` tags, which insert a row in an HTML table.

#### Syntax

The following is the syntax for HTP:

```plaintext
HTP.TABLEROWOPEN (calign, cvalign, cdp, nowrap, attributes)
```

The following is the syntax for HTF:

```plaintext
HTF.TABLEROWOPEN (calign, cvalign, cdp, nowrap, attributes) RETURN VARCHAR2;
```

#### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>calign</td>
<td>The value for the ALIGN attribute</td>
</tr>
<tr>
<td>cvalign</td>
<td>The value for the VALIGN attribute</td>
</tr>
<tr>
<td>cdp</td>
<td>The value for the DP attribute</td>
</tr>
<tr>
<td>nowrap</td>
<td>If the value of this parameter isn't NULL, adds the NOWRAP attribute to the tag</td>
</tr>
<tr>
<td>attributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.82 TELETYPE

This function and procedure generates the `<TT>` and `</TT>` tags, which render text in a fixed-width, typewriter font such as Courier.

#### Syntax

The following is the syntax for HTP:

```plaintext
HTP.TELETYPE (ctext, attributes)
```

The following is the syntax for HTF:

```plaintext
HTF.TELETYPE (ctext, attributes) RETURN VARCHAR2;
```
### 14.3.3.1.27.83 TITLE

This function and procedure generates the `<TITLE>` and `</TITLE>` tags, which specify the text to display in the title bar of the browser window.

**Syntax**

The following is the syntax for HTP:

```
HTP.TITLE (ctitle IN VARCHAR2);
```

The following is the syntax for HTF:

```
HTF.TITLE (ctitle) RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in a fixed-width typewriter font</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.84 ULISTOPEN and ULISTCLOSE

These function and procedure generate the `<UL>` and `</UL>` tags, which define an unordered (bullet) list.

**Syntax**

The following is the syntax for HTP:

```
HTP.ULISTOPEN (cclear IN VARCHAR2 DEFAULT NULL, cwrap IN VARCHAR2 DEFAULT NULL, cdingbat IN VARCHAR2 DEFAULT NULL, csrc IN VARCHAR2 DEFAULT NULL, cattributes IN VARCHAR2 DEFAULT NULL);
HTP.ULISTCLOSE;
```

The following is the syntax for HTF:

```
HTF.ULISTOPEN (cclear, cwrap, cdingbat, csrc, cattributes) RETURN VARCHAR2;
HTF.ULISTCLOSE RETURN VARCHAR2;
```

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cclear</td>
<td>The value for the CLEAR attribute</td>
</tr>
<tr>
<td>cwrap</td>
<td>The value for the WRAP attribute</td>
</tr>
<tr>
<td>cdingbat</td>
<td>The value for the DWORD attribute</td>
</tr>
<tr>
<td>csrc</td>
<td>The value for the SRC attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
<tr>
<td>Parameter</td>
<td>Purpose</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>cdingbat</td>
<td>The value for the DINGBAT attribute</td>
</tr>
<tr>
<td>csrc</td>
<td>The value for the SRC attribute</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.85 UNDERLINE

This function and procedure generates the `<U>` and `</U>` tags, which render text with an underline.

**Syntax**

The following is the syntax for HTP:

```
HTP.UNDERLINE(
    ctext          IN       VARCHAR2
    cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.UNDERLINE(ctext, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render with an underline</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.86 VARIABLE

This function and procedure generates the `<VAR>` and `</VAR>` tags, which render text in italics.

**Syntax**

The following is the syntax for HTP:

```
HTP.VARIABLE(
    ctext          IN       VARCHAR2
    cattributes    IN       VARCHAR2   DEFAULT NULL);
```

The following is the syntax for HTF:

```
HTF.VARIABLE (ctext, cattributes) RETURN VARCHAR2;
```

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctext</td>
<td>The text to render in italics</td>
</tr>
<tr>
<td>cattributes</td>
<td>Other attributes to include as is in the tag</td>
</tr>
</tbody>
</table>

### 14.3.3.1.27.87 WBR
This function and procedure generates the `<WBR>` tag, which inserts a soft line break in text marked as NOBR.

**Syntax**

The following is the syntax for HTP:

```
HTP.WBR;
```

The following is the syntax for HTF:

```
HTF.WBR RETURN WBR;
```

### 14.3.4 Database compatibility for Oracle developers: catalog views

Catalog views provide information about database objects. There are two categories of catalog views:

- Oracle catalog views provide information about database objects in a manner compatible with the Oracle data dictionary views.
- System catalog views are present in PostgreSQL and might have extra information when accessed in EDB Postgres Advanced Server.

These features related to the catalog views support are provided by EDB Postgres Advanced Server.

#### 14.3.4.1 Oracle catalog views

Oracle catalog views provide information about database objects in a manner compatible with Oracle data dictionary views.

##### 14.3.4.1.1 ALL_ALL_TABLES

The `ALL_ALL_TABLES` view provides information about the tables the current user can access.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the table’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides. If the tablespace name isn't specified, then default tablespace <code>PG_DEFAULT</code>.</td>
</tr>
<tr>
<td>degree</td>
<td>CHARACTER VARYING(10)</td>
<td>Number of threads per instance to scan a table, or <code>DEFAULT</code>.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING (5)</td>
<td>Included only for compatibility. Always set to <code>VALID</code>.</td>
</tr>
<tr>
<td>temporary</td>
<td>TEXT</td>
<td>Y if the table is temporary, N if the table is permanent.</td>
</tr>
</tbody>
</table>

##### 14.3.4.1.2 ALL_CONS_COLUMNS

The `ALL_CONS_COLUMNS` view provides information about the columns specified in constraints placed on tables accessible by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column referenced in the constraint.</td>
</tr>
<tr>
<td>position</td>
<td>SMALLINT</td>
<td>Position of the column in the object definition.</td>
</tr>
</tbody>
</table>
14.3.4.1.3 **ALL_CONSTRAINTS**

The **ALL_CONSTRAINTS** view provides information about the constraints placed on tables accessible by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint.</td>
</tr>
<tr>
<td>constraint_type</td>
<td>TEXT</td>
<td>The constraint type. Possible values are: C – check constraint; F – foreign key constraint; P – primary key constraint; U – unique key constraint; R – referential integrity constraint; V – constraint on a view; O – with read-only, on a view</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>search_condition</td>
<td>TEXT</td>
<td>Search condition that applies to a check constraint.</td>
</tr>
<tr>
<td>r_owner</td>
<td>TEXT</td>
<td>Owner of a table referenced by a referential constraint.</td>
</tr>
<tr>
<td>r_constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint definition for a referenced table.</td>
</tr>
<tr>
<td>delete_rule</td>
<td>TEXT</td>
<td>Delete rule for a referential constraint. Possible values are: C – cascade; R – restrict; N – no action</td>
</tr>
<tr>
<td>deferrable</td>
<td>BOOL</td>
<td>Specified if the constraint is deferrable (T or F).</td>
</tr>
<tr>
<td>deferred</td>
<td>BOOL</td>
<td>Specified if the constraint was deferred (T or F).</td>
</tr>
<tr>
<td>index_owner</td>
<td>TEXT</td>
<td>User name of the index owner.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>Definition of the constraint.</td>
</tr>
</tbody>
</table>

14.3.4.1.4 **ALL_COL_PRIVS**

The **ALL_COL_PRIVS** view provides the following types of privileges:

- Column object privileges for which a current user is either an object owner, grantor, or grantee.
- Column object privileges for which PUBLIC is the grantee.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user who granted the privilege.</td>
</tr>
<tr>
<td>grantee</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user with the privilege.</td>
</tr>
<tr>
<td>table_schema</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user who owns the object.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the schema in which the object resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Object name.</td>
</tr>
<tr>
<td>column_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Column name.</td>
</tr>
<tr>
<td>privilege</td>
<td>CHARACTER VARYING(48)</td>
<td>Privilege on the column.</td>
</tr>
<tr>
<td>grantable</td>
<td>CHARACTER VARYING(3)</td>
<td>Indicates whether the privilege was granted with the grant option YES or NO. YES indicates that the GRANTEE (recipient of the privilege) can grant the privilege to others. The value can be YES if the grantee has administrator privileges.</td>
</tr>
<tr>
<td>common</td>
<td>CHARACTER VARYING(3)</td>
<td>Included for compatibility only. Always NO.</td>
</tr>
<tr>
<td>inherited</td>
<td>CHARACTER VARYING(3)</td>
<td>Included for compatibility only. Always NO.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.5 ALL_DB_LINKS

The **ALL_DB_LINKS** view provides information about the database links accessible by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the database link’s owner.</td>
</tr>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>Name of the database link.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING</td>
<td>Type of remote server. Value is REDWOOD or EDB.</td>
</tr>
<tr>
<td>username</td>
<td>TEXT</td>
<td>User name of the user logging in.</td>
</tr>
<tr>
<td>host</td>
<td>TEXT</td>
<td>Name or IP address of the remote server.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.6 ALL_DEPENDENCIES

The **ALL_DEPENDENCIES** view provides information about the dependencies between database objects that the current user can access, except for synonyms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>CHARACTER VARYING(128)</td>
<td>Owner of dependent object.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the schema in which the dependent object resides.</td>
</tr>
<tr>
<td>name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the dependent object.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING(18)</td>
<td>Type of the dependent object.</td>
</tr>
<tr>
<td>referenced_owner</td>
<td>CHARACTER VARYING(128)</td>
<td>Owner of the referenced object.</td>
</tr>
<tr>
<td>referenced_schema_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the schema in which the referenced object resides.</td>
</tr>
<tr>
<td>referenced_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the referenced object.</td>
</tr>
<tr>
<td>referenced_type</td>
<td>CHARACTER VARYING(18)</td>
<td>Type of the referenced object.</td>
</tr>
<tr>
<td>referenced_link_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Included for compatibility only. Always NULL.</td>
</tr>
<tr>
<td>dependency_type</td>
<td>CHARACTER VARYING(4)</td>
<td>Included for compatibility only. Always set to HARD.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.7 ALL_DIRECTORIES

The **ALL_DIRECTORIES** view provides information about all directories created with the `CREATE DIRECTORY` command.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>CHARACTER VARYING(30)</td>
<td>User name of the directory owner.</td>
</tr>
<tr>
<td>directory_name</td>
<td>CHARACTER VARYING(30)</td>
<td>Alias name assigned to the directory.</td>
</tr>
<tr>
<td>directory_path</td>
<td>CHARACTER VARYING(4080)</td>
<td>Path to the directory.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.8 ALL_IND_COLUMNS

The **ALL_IND_COLUMNS** view provides information about columns included in indexes on the tables accessible by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index_owner</td>
<td>TEXT</td>
<td>User name of the index owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the table owner.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table to which the index belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>column_position</td>
<td>SMALLINT</td>
<td>Position of the column in the index.</td>
</tr>
<tr>
<td>column_length</td>
<td>SMALLINT</td>
<td>Length of the column in bytes.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.9 ALL_INDEXES

The **ALL_INDEXES** view provides information about the indexes on tables that the current user can access.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the index owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>index_type</td>
<td>TEXT</td>
<td>The index type is always <strong>BTREE</strong>. Included for compatibility only.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the owner of the indexed table.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the indexed table.</td>
</tr>
<tr>
<td>table_type</td>
<td>TEXT</td>
<td>Included for compatibility only. Always set to <strong>TABLE</strong>.</td>
</tr>
<tr>
<td>uniqueness</td>
<td>TEXT</td>
<td>Indicates if the index is <strong>UNIQUE</strong> or <strong>NONUNIQUE</strong>.</td>
</tr>
<tr>
<td>compression</td>
<td>CHARACTER(1)</td>
<td>Always set to <strong>N</strong> (not compressed). Included only for compatibility.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if not the default tablespace.</td>
</tr>
<tr>
<td>degree</td>
<td>CHARACTER VARYING(10)</td>
<td>Number of threads per instance to scan the index.</td>
</tr>
<tr>
<td>logging</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to <strong>LOGGING</strong>.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to <strong>VALID</strong>.</td>
</tr>
<tr>
<td>partitioned</td>
<td>CHARACTER(3)</td>
<td>Indicates that the index is partitioned. Always set to <strong>NO</strong>.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHARACTER(1)</td>
<td>Indicates that an index is on a temporary table. Included only for compatibility. Always set to <strong>N</strong>.</td>
</tr>
<tr>
<td>secondary</td>
<td>CHARACTER(1)</td>
<td>Included only for compatibility. Always set to <strong>N</strong>.</td>
</tr>
<tr>
<td>join_index</td>
<td>CHARACTER(3)</td>
<td>Included only for compatibility. Always set to <strong>NO</strong>.</td>
</tr>
<tr>
<td>dropped</td>
<td>CHARACTER(3)</td>
<td>Included only for compatibility. Always set to <strong>NO</strong>.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.10 ALL_JOBS

The **ALL_JOBS** view provides information about all jobs that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job</td>
<td>INTEGER</td>
<td>The identifier of the job (Job ID).</td>
</tr>
<tr>
<td>log_user</td>
<td>TEXT</td>
<td>Name of the user that submitted the job.</td>
</tr>
<tr>
<td>priv_user</td>
<td>TEXT</td>
<td>Same as <strong>log_user</strong>. Included only for compatibility.</td>
</tr>
<tr>
<td>schema_user</td>
<td>TEXT</td>
<td>Name of the schema used to parse the job.</td>
</tr>
<tr>
<td>last_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>Last date that this job executed successfully.</td>
</tr>
<tr>
<td>last_sec</td>
<td>TEXT</td>
<td>Same as <strong>last_date</strong>.</td>
</tr>
<tr>
<td>this_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>Date that the job began executing.</td>
</tr>
<tr>
<td>this_sec</td>
<td>TEXT</td>
<td>Same as <strong>this_date</strong>.</td>
</tr>
<tr>
<td>next_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>Next date for this job to execute.</td>
</tr>
<tr>
<td>next_sec</td>
<td>TEXT</td>
<td>Same as <strong>next_date</strong>.</td>
</tr>
<tr>
<td>total_time</td>
<td>INTERVAL</td>
<td>Execution time of this job, in seconds.</td>
</tr>
<tr>
<td>broken</td>
<td>TEXT</td>
<td>If <strong>Y</strong>, no attempt is be made to run this job. If <strong>N</strong>, this job attempts to run.</td>
</tr>
<tr>
<td>interval</td>
<td>TEXT</td>
<td>Determines how often the job repeats.</td>
</tr>
<tr>
<td>failures</td>
<td>BIGINT</td>
<td>Number of times that the job failed to complete since its last successful execution.</td>
</tr>
<tr>
<td>what</td>
<td>TEXT</td>
<td>Job definition (PL/SQL code block) that runs when the job executes.</td>
</tr>
<tr>
<td>nls_env</td>
<td>CHARACTER VARYING(4096)</td>
<td>Always <strong>NULL</strong>. Provided only for compatibility.</td>
</tr>
<tr>
<td>misc_env</td>
<td>BYTEA</td>
<td>Always <strong>NULL</strong>. Provided only for compatibility.</td>
</tr>
</tbody>
</table>
14.3.4.1.11  ALL_OBJECTS

The **ALL_OBJECTS** view provides information about all objects that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the object's owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the object belongs.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>Name of the object.</td>
</tr>
<tr>
<td>object_type</td>
<td>TEXT</td>
<td>Type of the object. Possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.</td>
</tr>
<tr>
<td>created</td>
<td>DATE</td>
<td>Timestamp for when the object was created.</td>
</tr>
<tr>
<td>last_ddl_time</td>
<td>DATE</td>
<td>Timestamp for the last modification of an object resulting from a DDL statement, including grants and revokes.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING</td>
<td>Whether or not the state of the object is valid. Included only for compatibility. Always set to VALID.</td>
</tr>
<tr>
<td>temporary</td>
<td>TEXT</td>
<td>Y if a temporary object, N if this is a permanent object.</td>
</tr>
</tbody>
</table>

14.3.4.1.12  ALL_PART_KEY_COLUMNS

The **ALL_PART_KEY_COLUMNS** view provides information about the key columns of the partitioned tables that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Owner of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the table in which the column resides.</td>
</tr>
<tr>
<td>object_type</td>
<td>CHARACTER(5)</td>
<td>For compatibility only. Always TABLE.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column on which the key is defined.</td>
</tr>
<tr>
<td>column_position</td>
<td>INTEGER</td>
<td>1 for the first column, 2 for the second column, and so on.</td>
</tr>
</tbody>
</table>

14.3.4.1.13  ALL_PART_TABLES

The **ALL_PART_TABLES** view provides information about all of the partitioned tables that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Owner of the partitioned table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>partitioning_type</td>
<td>TEXT</td>
<td>Partitioning type used to define table partitions.</td>
</tr>
<tr>
<td>subpartitioning_type</td>
<td>TEXT</td>
<td>Subpartitioning type used to define table subpartitions.</td>
</tr>
<tr>
<td>partition_count</td>
<td>BIGINT</td>
<td>Number of partitions in the table.</td>
</tr>
<tr>
<td>def_subpartition_count</td>
<td>INTEGER</td>
<td>Number of subpartitions in the table.</td>
</tr>
<tr>
<td>partitioning_key_count</td>
<td>INTEGER</td>
<td>Number of partitioning keys specified.</td>
</tr>
<tr>
<td>subpartitioning_key_count</td>
<td>INTEGER</td>
<td>Number of subpartitioning keys specified.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING(8)</td>
<td>Provided only for compatibility. Always VALID.</td>
</tr>
<tr>
<td>def_tablespace_name</td>
<td>CHARACTER VARYING(30)</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_pct_free</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_pct_used</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_init_trans</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.14  ALL_POLICIES

The **ALL_POLICIES** view provides information on all policies in the database. This view is accessible only to superusers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_owner</td>
<td>TEXT</td>
<td>Name of the owner of the object.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the object belongs.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>Name of the object on which the policy applies.</td>
</tr>
<tr>
<td>policy_group</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to an empty string.</td>
</tr>
<tr>
<td>policy_name</td>
<td>TEXT</td>
<td>Name of the policy.</td>
</tr>
<tr>
<td>pf_owner</td>
<td>TEXT</td>
<td>Name of the schema containing the policy function or the schema containing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the package that contains the policy function.</td>
</tr>
<tr>
<td>package</td>
<td>TEXT</td>
<td>Name of the package containing the policy function if the function belongs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to a package.</td>
</tr>
<tr>
<td>function</td>
<td>TEXT</td>
<td>Name of the policy function.</td>
</tr>
<tr>
<td>sel</td>
<td>TEXT</td>
<td>Whether or not the policy applies to SELECT commands. Possible values are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES or NO.</td>
</tr>
<tr>
<td>ins</td>
<td>TEXT</td>
<td>Whether or not the policy applies to INSERT commands. Possible values are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES or NO.</td>
</tr>
<tr>
<td>upd</td>
<td>TEXT</td>
<td>Whether or not the policy applies to UPDATE commands. Possible values are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES or NO.</td>
</tr>
<tr>
<td>del</td>
<td>TEXT</td>
<td>Whether or not the policy applies to DELETE commands. Possible values are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES or NO.</td>
</tr>
<tr>
<td>idx</td>
<td>TEXT</td>
<td>Whether or not the policy applies to index maintenance. Possible values are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES or NO.</td>
</tr>
<tr>
<td>chk_option</td>
<td>TEXT</td>
<td>Whether or not the check option is in force for INSERT and UPDATE commands.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Possible values are YES or NO.</td>
</tr>
<tr>
<td>enable</td>
<td>TEXT</td>
<td>Whether or not the policy is enabled on the object. Possible values are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YES or NO.</td>
</tr>
<tr>
<td>static_policy</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to NO.</td>
</tr>
<tr>
<td>policy_type</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to UNKNOWN.</td>
</tr>
<tr>
<td>long_predicate</td>
<td>TEXT</td>
<td>Included for compatibility only. Always set to YES.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.15  ALL_QUEUES

The **ALL_QUEUES** view provides information about any currently defined queues.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the queue owner.</td>
</tr>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the queue.</td>
</tr>
<tr>
<td>queue_table</td>
<td>TEXT</td>
<td>Name of the queue table in which the queue resides.</td>
</tr>
<tr>
<td>qid</td>
<td>OID</td>
<td>The system-assigned object ID of the queue.</td>
</tr>
<tr>
<td>queue_type</td>
<td>CHARACTER VARYING</td>
<td>The queue type. Can be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.</td>
</tr>
<tr>
<td>max_retries</td>
<td>NUMERIC</td>
<td>The maximum number of dequeue attempts.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.16  ALL_QUEUE_TABLES

The `ALL_QUEUE_TABLES` view provides information about all of the queue tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>retrydelay</td>
<td>NUMERIC</td>
<td>The maximum time allowed between retries.</td>
</tr>
<tr>
<td>enqueue_enabled</td>
<td>CHARACTER VARYING</td>
<td>YES if the queue allows enqueuing, NO if the queue does not.</td>
</tr>
<tr>
<td>dequeue_enabled</td>
<td>CHARACTER VARYING</td>
<td>YES if the queue allows dequeuing, NO if the queue does not.</td>
</tr>
<tr>
<td>retention</td>
<td>CHARACTER VARYING</td>
<td>The number of seconds that a processed message is retained in the queue.</td>
</tr>
<tr>
<td>user_comment</td>
<td>CHARACTER VARYING</td>
<td>User-specified comment.</td>
</tr>
<tr>
<td>network_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the network on which the queue resides.</td>
</tr>
<tr>
<td>shared</td>
<td>CHARACTER VARYING</td>
<td>YES if the queue resides on a sharded network, NO if the queue does not.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.17  ALL_SEQUENCES

The `ALL_SEQUENCES` view provides information about all user-defined sequences on which the user has `SELECT` or `UPDATE` privileges.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence_owner</td>
<td>TEXT</td>
<td>User name of the sequence owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the sequence resides.</td>
</tr>
<tr>
<td>sequence_name</td>
<td>TEXT</td>
<td>Name of the sequence.</td>
</tr>
<tr>
<td>min_value</td>
<td>NUMERIC</td>
<td>Lowest value that the server assigns to the sequence.</td>
</tr>
<tr>
<td>max_value</td>
<td>NUMERIC</td>
<td>Highest value that the server assigns to the sequence.</td>
</tr>
<tr>
<td>increment_by</td>
<td>NUMERIC</td>
<td>Value added to the current sequence number to create the next sequent number.</td>
</tr>
<tr>
<td>cycle_flag</td>
<td>CHARACTER VARYING</td>
<td>Specifies whether the sequence wraps when it reaches <code>min_value</code> or <code>max_value</code>.</td>
</tr>
<tr>
<td>order_flag</td>
<td>CHARACTER VARYING</td>
<td>Always returns Y.</td>
</tr>
<tr>
<td>cache_size</td>
<td>NUMERIC</td>
<td>Number of preallocated sequence numbers stored in memory.</td>
</tr>
<tr>
<td>last_number</td>
<td>NUMERIC</td>
<td>Value of the last sequence number saved to disk.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.18  ALL_SOURCE

The `ALL_SOURCE` view provides a source code listing of the following program types: functions, procedures, triggers, package specifications, and package bodies.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
### 14.3.4.19 ALL_SUBPART_KEY_COLUMNS

The `ALL_SUBPART_KEY_COLUMNS` view provides information about the key columns of those partitioned tables that are subpartitioned that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Owner of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the table in which the column resides.</td>
</tr>
<tr>
<td>object_type</td>
<td>CHARACTER(5)</td>
<td>Only for compatibility. Always TABLE.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column on which the key is defined.</td>
</tr>
<tr>
<td>column_position</td>
<td>INTEGER</td>
<td>1 for the first column, 2 for the second column, and so on.</td>
</tr>
</tbody>
</table>

### 14.3.4.20 ALL_SYNONYMS

The `ALL_SYNONYMS` view provides information on all synonyms that the current user can reference.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the synonym owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the synonym resides.</td>
</tr>
<tr>
<td>synonym_name</td>
<td>TEXT</td>
<td>Name of the synonym.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the object owner.</td>
</tr>
<tr>
<td>table_schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the object that the synonym refers to.</td>
</tr>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>Name of any associated database link.</td>
</tr>
</tbody>
</table>

### 14.3.4.21 ALL_TAB_COLUMNS

The `ALL_TAB_COLUMNS` view provides information on all columns in all user-defined tables and views.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>CHARACTER VARYING</td>
<td>User name of the owner of the table or view in which the column resides.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the schema in which the table or view resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the table or view.</td>
</tr>
<tr>
<td>column_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>data_type</td>
<td>CHARACTER VARYING</td>
<td>Data type of the column.</td>
</tr>
<tr>
<td>data_length</td>
<td>NUMERIC</td>
<td>Length of text columns.</td>
</tr>
<tr>
<td>data_precision</td>
<td>NUMERIC</td>
<td>Precision (number of digits) for NUMBER columns.</td>
</tr>
<tr>
<td>data_scale</td>
<td>NUMERIC</td>
<td>Scale of NUMBER columns.</td>
</tr>
<tr>
<td>nullable</td>
<td>CHARACTER(1)</td>
<td>Whether or not the column is nullable. Possible values are: <strong>Y</strong> - column is nullable; <strong>N</strong> - column does not allow null.</td>
</tr>
<tr>
<td>column_id</td>
<td>NUMERIC</td>
<td>Relative position of the column in the table or view.</td>
</tr>
<tr>
<td>data_default</td>
<td>CHARACTER VARYING</td>
<td>Default value assigned to the column.</td>
</tr>
</tbody>
</table>
### 

The `ALL_TAB_PARTITIONS` view provides information about all of the partitions that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>Owner of the table in which the partition resides.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>composite</td>
<td>TEXT</td>
<td>YES if the table is subpartitioned, NO if the table isn't subpartitioned.</td>
</tr>
<tr>
<td>partition_name</td>
<td>TEXT</td>
<td>Name of the partition.</td>
</tr>
<tr>
<td>subpartition_count</td>
<td>BIGINT</td>
<td>Number of subpartitions in the partition.</td>
</tr>
<tr>
<td>high_value</td>
<td>TEXT</td>
<td>High-partitioning value specified in the <code>CREATE TABLE</code> statement.</td>
</tr>
<tr>
<td>high_value_length</td>
<td>INTEGER</td>
<td>Length of high-partitioning value.</td>
</tr>
<tr>
<td>partition_position</td>
<td>INTEGER</td>
<td>Ordinal position of this partition.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the partition resides. If the tablespace name isn't specified, the default tablespace is <code>PG_DEFAULT</code>.</td>
</tr>
<tr>
<td>pct_free</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>pct_used</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>ini_trans</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>max_trans</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>initial_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>next_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>min_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>max_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>pct_increase</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>freelists</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>freelist_groups</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>logging</td>
<td>CHARACTER VARYING(7)</td>
<td>Included only for compatibility. Always YES.</td>
</tr>
<tr>
<td>compression</td>
<td>CHARACTER VARYING(8)</td>
<td>Included only for compatibility. Always NONE.</td>
</tr>
<tr>
<td>num_rows</td>
<td>NUMERIC</td>
<td>Same as <code>pg_class.reltuples</code>.</td>
</tr>
<tr>
<td>blocks</td>
<td>INTEGER</td>
<td>Same as <code>pg_class.relpages</code>.</td>
</tr>
<tr>
<td>empty_blocks</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>avg_space</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>chain_cnt</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>avg_row_len</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>sample_size</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>last_analyzed</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>buffer_pool</td>
<td>CHARACTER VARYING(7)</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>global_stats</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always YES.</td>
</tr>
<tr>
<td>user_stats</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always NO.</td>
</tr>
<tr>
<td>backing_table</td>
<td>REGCLASS</td>
<td>Name of the partition backing table.</td>
</tr>
</tbody>
</table>

### 

The `ALL_TAB_SUBPARTITIONS` view provides information about all of the subpartitions that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>Owner of the table in which the subpartition resides.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>partition_name</td>
<td>TEXT</td>
<td>Name of the partition.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>subpartition_name</td>
<td>TEXT</td>
<td>Name of the subpartition.</td>
</tr>
<tr>
<td>high_value</td>
<td>TEXT</td>
<td>High-subpartitioning value specified in the CREATE TABLE statement.</td>
</tr>
<tr>
<td>high_value_length</td>
<td>INTEGER</td>
<td>Length of high-partitioning value.</td>
</tr>
<tr>
<td>subpartition_position</td>
<td>INTEGER</td>
<td>Ordinal position of this subpartition.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the subpartition resides. If the tablespace name is not specified, then default tablespace PG_DEFAULT.</td>
</tr>
<tr>
<td>pct_free</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>pct_used</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>init_trans</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>max_trans</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>initial_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>next_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>min_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>max_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>pct_increase</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>freelists</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>freelist_groups</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>logging</td>
<td>CHARACTER VARYING(7)</td>
<td>Included only for compatibility. Always YES.</td>
</tr>
<tr>
<td>compression</td>
<td>CHARACTER VARYING(8)</td>
<td>Included only for compatibility. Always NONE.</td>
</tr>
<tr>
<td>num_rows</td>
<td>NUMERIC</td>
<td>Same as pg_class.reltuples.</td>
</tr>
<tr>
<td>blocks</td>
<td>INTEGER</td>
<td>Same as pg_class.relpages.</td>
</tr>
<tr>
<td>empty_blocks</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>avg_space</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>chain_cnt</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>avg_row_len</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>sample_size</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>last_analyzed</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>buffer_pool</td>
<td>CHARACTER VARYING(7)</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>global_stats</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always YES.</td>
</tr>
<tr>
<td>user_stats</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always NO.</td>
</tr>
<tr>
<td>backing_table</td>
<td>REGCLASS</td>
<td>Name of the subpartition backing table.</td>
</tr>
</tbody>
</table>

**14.3.4.1.24 ALL_TAB_PRIVS**

The ALL_TAB_PRIVS view provides the following types of privileges:

- Object privileges for which a current user is either an object owner, grantor, or grantee.
- Object privileges for which the PUBLIC is the grantee.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantor</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user who granted the privilege.</td>
</tr>
<tr>
<td>grantee</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user with the privilege.</td>
</tr>
<tr>
<td>table_schemaname</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user who owns the object.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the schema in which the object resides.</td>
</tr>
<tr>
<td>tablename</td>
<td>CHARACTER VARYING(128)</td>
<td>Object name.</td>
</tr>
<tr>
<td>privilege</td>
<td>CHARACTER VARYING(40)</td>
<td>Privilege name.</td>
</tr>
<tr>
<td>grantable</td>
<td>CHARACTER VARYING(3)</td>
<td>Indicates whether the privilege was granted with the grant option YES or NO. YES Indicates that the GRANTEE (recipient of the privilege) can in turn grant the privilege to others. The value can be YES if the grantee has administrator privileges.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.25 ALL_TABLES

The `ALL_TABLES` view provides information on all user-defined tables.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the table's owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides. If the tablespace name isn't specified, the default tablespace <code>PG_DEFAULT</code>.</td>
</tr>
<tr>
<td>degree</td>
<td>CHARACTER VARYING(10)</td>
<td>Number of threads per instance to scan a table, or <code>DEFAULT</code>.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING(5)</td>
<td>Whether or not the state of the table is valid. Included only for compatibility. Always set to <code>VALID</code>.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHARACTER(1)</td>
<td>Y if this is a temporary table, N if this isn't a temporary table.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.26 ALL_TRIGGERS

The `ALL_TRIGGERS` view provides information about the triggers on tables that the current user can accessed.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the trigger owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the trigger resides.</td>
</tr>
<tr>
<td>trigger_name</td>
<td>TEXT</td>
<td>Name of the trigger.</td>
</tr>
<tr>
<td>trigger_type</td>
<td>TEXT</td>
<td>The type of the trigger. Possible values are: <code>BEFORE ROW</code>, <code>BEFORE STATEMENT</code>, <code>AFTER ROW</code>, <code>AFTER STATEMENT</code>.</td>
</tr>
<tr>
<td>triggering_event</td>
<td>TEXT</td>
<td>The event that fires the trigger.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>The user name of the owner of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>base_object_type</td>
<td>TEXT</td>
<td>Included only for compatibility. Value is always <code>TABLE</code>.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>referencing_name</td>
<td>TEXT</td>
<td>Included only for compatibility. Value is always <code>REFERENCING NEW AS NEW OLD AS OLD</code>.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).</td>
</tr>
<tr>
<td>description</td>
<td>TEXT</td>
<td>Included only for compatibility.</td>
</tr>
<tr>
<td>trigger_body</td>
<td>TEXT</td>
<td>The body of the trigger.</td>
</tr>
<tr>
<td>action_statement</td>
<td>TEXT</td>
<td>The SQL command that executes when the trigger fires.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.27 ALL_TYPES

The `ALL_TYPES` view provides information about the object types available to the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Owner of the object type.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the type is defined.</td>
</tr>
<tr>
<td>type_name</td>
<td>TEXT</td>
<td>Name of the type.</td>
</tr>
<tr>
<td>type_oid</td>
<td>OID</td>
<td>Object identifier (OID) of the type.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.28 ALL_USERS

The `ALL_USERS` view provides information on all user names.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>TEXT</td>
<td>Name of the user.</td>
</tr>
<tr>
<td>user_id</td>
<td>OID</td>
<td>Numeric user id assigned to the user.</td>
</tr>
<tr>
<td>created</td>
<td>DATE</td>
<td>Timestamp for when an object was created.</td>
</tr>
<tr>
<td>last_ddl_time</td>
<td>DATE</td>
<td>Timestamp for the last modification of an object resulting from a DDL statement, including grants and revokes.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.29 ALL_VIEW_COLUMNS

The `ALL_VIEW_COLUMNS` view provides information on all columns in all user-defined views.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>CHARACTER VARYING</td>
<td>User name of the view’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>column_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>data_type</td>
<td>CHARACTER VARYING</td>
<td>Data type of the column.</td>
</tr>
<tr>
<td>data_length</td>
<td>NUMERIC</td>
<td>Length of text columns.</td>
</tr>
<tr>
<td>data_precision</td>
<td>NUMERIC</td>
<td>Precision (number of digits) for NUMBER columns.</td>
</tr>
<tr>
<td>data_scale</td>
<td>NUMERIC</td>
<td>Scale of NUMBER columns.</td>
</tr>
<tr>
<td>nullable</td>
<td>CHARACTER(1)</td>
<td>Whether or not the column is nullable. Possible values are: Y – column is nullable; N – column does not allow null.</td>
</tr>
<tr>
<td>column_id</td>
<td>NUMERIC</td>
<td>Relative position of the column in the view.</td>
</tr>
<tr>
<td>data_default</td>
<td>CHARACTER VARYING</td>
<td>Default value assigned to the column.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.30 ALL_VIEWS

The `ALL_VIEWS` view provides information about all user-defined views.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the view’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>TEXT</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>text</td>
<td>TEXT</td>
<td>The <code>SELECT</code> statement that defines the view.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.31 DBA_ALL_TABLES

The `DBA_ALL_TABLES` view provides information about all tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the table owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.32 DBA_CONS_COLUMNS

The **DBA_CONS_COLUMNS** view provides information about all columns that are included in constraints that are specified on all tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint's owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column referenced in the constraint.</td>
</tr>
<tr>
<td>position</td>
<td>SMALLINT</td>
<td>Position of the column in the object definition.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>Definition of the constraint.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.33 DBA_CONSTRAINTS

The **DBA_CONSTRAINTS** view provides information about all constraints on tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint's owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint.</td>
</tr>
<tr>
<td>constraint_type</td>
<td>TEXT</td>
<td>The constraint type. Possible values are: C – check constraint; F – foreign key constraint; P – primary key constraint; U – unique key constraint; R – referential integrity constraint; V – constraint on a view; O – with read-only, on a view.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>search_condition</td>
<td>TEXT</td>
<td>Search condition that applies to a check constraint.</td>
</tr>
<tr>
<td>r_owner</td>
<td>TEXT</td>
<td>Owner of a table referenced by a referential constraint.</td>
</tr>
<tr>
<td>r_constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint definition for a referenced table.</td>
</tr>
<tr>
<td>delete_rule</td>
<td>TEXT</td>
<td>The delete rule for a referential constraint. Possible values are: C – cascade; R – restrict; N – no action.</td>
</tr>
<tr>
<td>deferrable</td>
<td>BOOL EAN</td>
<td>Specified if the constraint is deferrable (T or F).</td>
</tr>
<tr>
<td>deferred</td>
<td>BOOL EAN</td>
<td>Specifies if the constraint has been deferred (T or F).</td>
</tr>
<tr>
<td>index_owner</td>
<td>TEXT</td>
<td>User name of the index owner.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>The definition of the constraint.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.34 DBA_COL_PRIVS

The **DBA_COL_PRIVS** view provides a listing of the object privileges granted on columns for all the database users.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
### 14.3.4.1.35 DBA_DB_LINKS

The **DBA_DB_LINKS** view provides information about all database links in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the database link's owner.</td>
</tr>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>Name of the database link.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING</td>
<td>Type of remote server. Value is REDWOOD or EDB.</td>
</tr>
<tr>
<td>username</td>
<td>TEXT</td>
<td>User name of the user logging in.</td>
</tr>
<tr>
<td>host</td>
<td>TEXT</td>
<td>Name or IP address of the remote server.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.36 DBA_DIRECTORIES

The **DBA_DIRECTORIES** view provides information about all directories created with the **CREATE DIRECTORY** command.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>CHARACTER VARYING(30)</td>
<td>User name of the directory's owner.</td>
</tr>
<tr>
<td>directory_name</td>
<td>CHARACTER VARYING(30)</td>
<td>Alias name assigned to the directory.</td>
</tr>
<tr>
<td>directory_path</td>
<td>CHARACTER VARYING(4080)</td>
<td>Path to the directory.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.37 DBA_DEPENDENCIES

The **DBA_DEPENDENCIES** view provides information about the dependencies between all objects in the database except for synonyms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>CHARACTER VARYING(128)</td>
<td>Owner of the dependent object.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the schema in which the dependent object resides.</td>
</tr>
<tr>
<td>name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the dependent object.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING(18)</td>
<td>Type of the dependent object.</td>
</tr>
<tr>
<td>referenced_owner</td>
<td>CHARACTER VARYING(128)</td>
<td>Owner of the referenced object.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.38 DBA_IND_COLUMNS

The DBA_IND_COLUMNS view provides information about all columns included in indexes on all tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>index_owner</td>
<td>TEXT</td>
<td>User name of the index owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the table owner.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table in which the index belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of column or attribute of object column.</td>
</tr>
<tr>
<td>column_position</td>
<td>SMALLINT</td>
<td>Position of the column in the index.</td>
</tr>
<tr>
<td>column_length</td>
<td>SMALLINT</td>
<td>Length of the column in bytes.</td>
</tr>
<tr>
<td>char_length</td>
<td>NUMERIC</td>
<td>Length of the column in characters.</td>
</tr>
<tr>
<td>descend</td>
<td>CHARACTER(1)</td>
<td>Included only for compatibility. Always set to <strong>Y</strong> (descending).</td>
</tr>
</tbody>
</table>

### 14.3.4.1.39 DBA_INDEXES

The DBA_INDEXES view provides information about all indexes in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the index owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the index resides.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>index_type</td>
<td>TEXT</td>
<td>The index type is always <strong>BTREE</strong>. Included only for compatibility.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the owner of the indexed table.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the indexed table.</td>
</tr>
<tr>
<td>table_type</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to <strong>TABLE</strong>.</td>
</tr>
<tr>
<td>uniqueness</td>
<td>TEXT</td>
<td>Indicates if the index is <strong>UNIQUE</strong> or <strong>NONUNIQUE</strong>.</td>
</tr>
<tr>
<td>compression</td>
<td>CHARACTER(1)</td>
<td>Included only for compatibility. Always set to <strong>N</strong> (not compressed).</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>degree</td>
<td>CHARACTER VARYING(10)</td>
<td>Number of threads per instance to scan the index.</td>
</tr>
<tr>
<td>logging</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to <strong>LOGGING</strong>.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Whether or not the state of the object is valid (<strong>VALID</strong> or <strong>INVALID</strong>).</td>
</tr>
<tr>
<td>partitioned</td>
<td>CHARACTER(3)</td>
<td>Indicates that the index is partitioned. Always set to <strong>NO</strong>.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHARACTER(1)</td>
<td>Indicates that an index is on a temporary table. Always set to <strong>N</strong>.</td>
</tr>
<tr>
<td>secondary</td>
<td>CHARACTER(1)</td>
<td>Included only for compatibility. Always set to <strong>N</strong>.</td>
</tr>
<tr>
<td>join_index</td>
<td>CHARACTER(3)</td>
<td>Included only for compatibility. Always set to <strong>NO</strong>.</td>
</tr>
<tr>
<td>dropped</td>
<td>CHARACTER(3)</td>
<td>Included only for compatibility. Always set to <strong>NO</strong>.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.40 DBA_JOBS

The DBA_JOBS view provides information about all jobs in the database.
### 14.3.4.1.41  DBA_OBJECTS

The `DBA_OBJECTS` view provides information about all objects in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the object's owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the object belongs.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>Name of the object.</td>
</tr>
<tr>
<td>object_type</td>
<td>TEXT</td>
<td>Type of the object. Possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.</td>
</tr>
<tr>
<td>created</td>
<td>DATE</td>
<td>Timestamp for when the object was created.</td>
</tr>
<tr>
<td>last_ddl_time</td>
<td>DATE</td>
<td>Timestamp for the last modification of an object resulting from a DDL statement including grants and revokes.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING</td>
<td>Included only for compatibility. Always set to VALID.</td>
</tr>
<tr>
<td>temporary</td>
<td>TEXT</td>
<td>Y if the table is temporary, N if the table is permanent.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.42  DBA_PART_KEY_COLUMNS

The `DBA_PART_KEY_COLUMNS` view provides information about the key columns of the partitioned tables that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Owner of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the table in which the column resides.</td>
</tr>
<tr>
<td>object_type</td>
<td>CHARACTER(S)</td>
<td>Only for compatibility. Always TABLE.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column on which the key is defined.</td>
</tr>
<tr>
<td>column_position</td>
<td>INTEGER</td>
<td>1 for the first column, 2 for the second column, and so on.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.43  DBA_PART_TABLES
The **DBA_PART_TABLES** view provides information about all of the partitioned tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Owner of the partitioned table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>partitioning_type</td>
<td>TEXT</td>
<td>Type used to define table partitions.</td>
</tr>
<tr>
<td>subpartitioning_type</td>
<td>TEXT</td>
<td>Subpartitioning type used to define table subpartitions.</td>
</tr>
<tr>
<td>partition_count</td>
<td>BIGINT</td>
<td>Number of partitions in the table.</td>
</tr>
<tr>
<td>def_subpartition_count</td>
<td>INTEGER</td>
<td>Number of subpartitions in the table.</td>
</tr>
<tr>
<td>partitioning_key_count</td>
<td>INTEGER</td>
<td>Number of partitioning keys specified.</td>
</tr>
<tr>
<td>subpartitioning_key_count</td>
<td>INTEGER</td>
<td>Number of subpartitioning keys specified.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING(8)</td>
<td>Provided only for compatibility. Always VALID.</td>
</tr>
<tr>
<td>def_tablespace_name</td>
<td>CHARACTER VARYING(30)</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_pct_free</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_pct_used</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_init_trans</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_max_trans</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_initial_extent</td>
<td>CHARACTER VARYING(40)</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_next_extent</td>
<td>CHARACTER VARYING(40)</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_min_extents</td>
<td>CHARACTER VARYING(40)</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_max_extents</td>
<td>CHARACTER VARYING(40)</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_pct_increase</td>
<td>CHARACTER VARYING(40)</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_freelists</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_freelist_groups</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>def_logging</td>
<td>CHARACTER VARYING(7)</td>
<td>Provided only for compatibility. Always YES.</td>
</tr>
<tr>
<td>def_compression</td>
<td>CHARACTER VARYING(8)</td>
<td>Provided only for compatibility. Always NONE.</td>
</tr>
<tr>
<td>def_buffer_pool</td>
<td>CHARACTER VARYING(7)</td>
<td>Provided only for compatibility. Always DEFAULT.</td>
</tr>
<tr>
<td>ref_prtn_constraint_name</td>
<td>CHARACTER VARYING(30)</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>interval</td>
<td>CHARACTER VARYING(1000)</td>
<td>Provided only for compatibility. Always NULL.</td>
</tr>
</tbody>
</table>

14.3.4.144 **DBA_POLICIES**

The **DBA_POLICIES** view provides information on all policies in the database. This view is accessible only to superusers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object_owner</td>
<td>TEXT</td>
<td>Name of the owner of the object.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the object resides.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>Name of the object to which the policy applies.</td>
</tr>
<tr>
<td>policy_group</td>
<td>TEXT</td>
<td>Name of the policy group. Included only for compatibility. Always set to an empty string.</td>
</tr>
<tr>
<td>policy_name</td>
<td>TEXT</td>
<td>Name of the policy.</td>
</tr>
<tr>
<td>pf_owner</td>
<td>TEXT</td>
<td>Name of the schema containing the policy function or the schema containing the package that contains the policy function.</td>
</tr>
<tr>
<td>package</td>
<td>TEXT</td>
<td>Name of the package containing the policy function if the function belongs to a package.</td>
</tr>
<tr>
<td>function</td>
<td>TEXT</td>
<td>Name of the policy function.</td>
</tr>
<tr>
<td>sel</td>
<td>TEXT</td>
<td>Whether or not the policy applies to <strong>SELECT</strong> commands. Possible values are <strong>YES</strong> or <strong>NO</strong>.</td>
</tr>
<tr>
<td>ins</td>
<td>TEXT</td>
<td>Whether or not the policy applies to <strong>INSERT</strong> commands. Possible values are <strong>YES</strong> or <strong>NO</strong>.</td>
</tr>
<tr>
<td>upd</td>
<td>TEXT</td>
<td>Whether or not the policy applies to <strong>UPDATE</strong> commands. Possible values are <strong>YES</strong> or <strong>NO</strong>.</td>
</tr>
<tr>
<td>del</td>
<td>TEXT</td>
<td>Whether or not the policy applies to <strong>DELETE</strong> commands. Possible values are <strong>YES</strong> or <strong>NO</strong>.</td>
</tr>
<tr>
<td>idx</td>
<td>TEXT</td>
<td>Whether or not the policy applies to index maintenance. Possible values are <strong>YES</strong> or <strong>NO</strong>.</td>
</tr>
<tr>
<td>chk_option</td>
<td>TEXT</td>
<td>Whether or not the check option is in force for <strong>INSERT</strong> and <strong>UPDATE</strong> commands. Possible values are <strong>YES</strong> or <strong>NO</strong>.</td>
</tr>
<tr>
<td>enable</td>
<td>TEXT</td>
<td>Whether or not the policy is enabled on the object. Possible values are <strong>YES</strong> or <strong>NO</strong>.</td>
</tr>
<tr>
<td>static_policy</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to <strong>NO</strong>.</td>
</tr>
<tr>
<td>policy_type</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to <strong>UNKNOWN</strong>.</td>
</tr>
</tbody>
</table>
14.3.4.1.45 DBA_PRIVS_CAPTURE

The `DBA_PRIVS_CAPTURE` view provides information about all the used privileges. The used privileges are populated by the `DBMS_PRIVILEGE_CAPTURE.GENERATE_RESULT` procedure.

Superuser or the user with `CAPTURE_ADMIN` role can access this view.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>NAME</td>
<td>Name of the capture policy for privilege analysis.</td>
</tr>
<tr>
<td>description</td>
<td>TEXT</td>
<td>Description of the capture policy for privilege analysis.</td>
</tr>
<tr>
<td>type</td>
<td>TEXT</td>
<td>Type of the privilege analysis policy. Possible values are <code>G_DATABASE</code>, <code>G_ROLE</code>, <code>G_CONTEXT</code>, <code>G_ROLE_AND_CONTEXT</code>.</td>
</tr>
<tr>
<td>enabled</td>
<td>BOOLEAN</td>
<td>Whether or not the policy is enabled on the object. Possible values are <code>YES</code> or <code>NO</code>.</td>
</tr>
<tr>
<td>roles</td>
<td>REGROLE[]</td>
<td>Name of the roles whose privileges are to be analyzed, if the <code>type</code> is <code>G_ROLE</code> or <code>G_ROLE_AND_CONTEXT</code>.</td>
</tr>
<tr>
<td>context</td>
<td>TEXT</td>
<td>Condition if the privilege analysis type is <code>G_CONTEXT</code> or <code>G_ROLE_AND_CONTEXT</code>.</td>
</tr>
<tr>
<td>run_name</td>
<td>NAME</td>
<td>Name of the run for which privileges are analyzed.</td>
</tr>
</tbody>
</table>

14.3.4.1.46 DBA_PROFILES

The `DBA_PROFILES` view provides information about existing profiles. The table includes a row for each profile/resource combination.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>profile</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the profile.</td>
</tr>
<tr>
<td>resource_name</td>
<td>CHARACTER VARYING(32)</td>
<td>Name of the resource associated with the profile.</td>
</tr>
<tr>
<td>resource_type</td>
<td>CHARACTER VARYING(8)</td>
<td>Type of resource governed by the profile. <code>PASSWD</code> for all supported resources.</td>
</tr>
<tr>
<td>limit</td>
<td>CHARACTER VARYING(128)</td>
<td>The limit values of the resource.</td>
</tr>
<tr>
<td>common</td>
<td>CHARACTER VARYING(3)</td>
<td>YES for a user-created profile, NO for a system-defined profile.</td>
</tr>
</tbody>
</table>

14.3.4.1.47 DBA_QUEUES

The `DBA_QUEUES` view provides information about any currently defined queues.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the queue owner.</td>
</tr>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the queue.</td>
</tr>
<tr>
<td>queue_table</td>
<td>TEXT</td>
<td>Name of the queue table in which the queue resides.</td>
</tr>
<tr>
<td>qid</td>
<td>OID</td>
<td>System-assigned object ID of the queue.</td>
</tr>
<tr>
<td>queue_type</td>
<td>CHARACTER VARYING</td>
<td>Queue type. Possible values are <code>EXCEPTION_QUEUE</code>, <code>NON_PERSISTENT_QUEUE</code>, or <code>NORMAL_QUEUE</code>.</td>
</tr>
<tr>
<td>max_retries</td>
<td>NUMERIC</td>
<td>Maximum number of dequeue attempts.</td>
</tr>
<tr>
<td>retrydelay</td>
<td>NUMERIC</td>
<td>Maximum time allowed between retries.</td>
</tr>
<tr>
<td>enqueue_enabled</td>
<td>CHARACTER VARYING</td>
<td>YES if the queue allows enqueuing, NO if the queue does not.</td>
</tr>
<tr>
<td>dequeue_enabled</td>
<td>CHARACTER VARYING</td>
<td>YES if the queue allows dequeuing, NO if the queue does not.</td>
</tr>
<tr>
<td>retention</td>
<td>CHARACTER VARYING</td>
<td>Number of seconds that a processed message is retained in the queue.</td>
</tr>
<tr>
<td>user_comment</td>
<td>CHARACTER VARYING</td>
<td>User-specified comment.</td>
</tr>
<tr>
<td>network_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the network on which the queue resides.</td>
</tr>
<tr>
<td>sharded</td>
<td>CHARACTER VARYING</td>
<td>YES if the queue resides on a sharded network, NO if the queue does not.</td>
</tr>
</tbody>
</table>

14.3.4.1.48 DBA_QUEUE_TABLES
The **DBA_QUEUE_TABLES** view provides information about all of the queue tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Role name of the owner of the queue table.</td>
</tr>
<tr>
<td>queue_table</td>
<td>TEXT</td>
<td>User-specified name of the queue table.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING</td>
<td>Type of data stored in the queue table.</td>
</tr>
<tr>
<td>object_type</td>
<td>TEXT</td>
<td>User-defined payload type.</td>
</tr>
<tr>
<td>sort_order</td>
<td>CHARACTER VARYING</td>
<td>Order in which the queue table is sorted.</td>
</tr>
<tr>
<td>recipients</td>
<td>CHARACTER VARYING</td>
<td>Always SINGLE.</td>
</tr>
<tr>
<td>message_grouping</td>
<td>CHARACTER VARYING</td>
<td>Always NONE.</td>
</tr>
<tr>
<td>compatible</td>
<td>CHARACTER VARYING</td>
<td>Release number of the EDB Postgres Advanced Server release with which this queue table is compatible.</td>
</tr>
<tr>
<td>primary_instance</td>
<td>NUMERIC</td>
<td>Always 0.</td>
</tr>
<tr>
<td>secondary_instance</td>
<td>NUMERIC</td>
<td>Always 0.</td>
</tr>
<tr>
<td>owner_instance</td>
<td>NUMERIC</td>
<td>Instance number of the instance that owns the queue table.</td>
</tr>
<tr>
<td>user_comment</td>
<td>CHARACTER VARYING</td>
<td>User comment provided when the table was created.</td>
</tr>
<tr>
<td>secure</td>
<td>CHARACTER VARYING</td>
<td>YES indicates that the queue table is secure. NO indicates that it isn’t.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.49 DBA_ROLE_PRIVS

The **DBA_ROLE_PRIVS** view provides information on all roles that were granted to users. A row is created for each role to which a user was granted.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantee</td>
<td>TEXT</td>
<td>User name to whom the role was granted.</td>
</tr>
<tr>
<td>granted_role</td>
<td>TEXT</td>
<td>Name of the role granted to the grantee.</td>
</tr>
<tr>
<td>admin_option</td>
<td>TEXT</td>
<td>YES if the role was granted with the admin option. NO otherwise.</td>
</tr>
<tr>
<td>default_role</td>
<td>TEXT</td>
<td>YES if the role is enabled when the grantee creates a session.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.50 DBA_ROLES

The **DBA_ROLES** view provides information on all roles with the **NOLOGIN** attribute (groups).

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>role</td>
<td>TEXT</td>
<td>Name of a role having the <strong>NOLOGIN</strong> attribute, that is, a group.</td>
</tr>
<tr>
<td>password_required</td>
<td>TEXT</td>
<td>Included only for compatibility. Always N.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.51 DBA_SEQUENCES

The **DBA_SEQUENCES** view provides information about all user-defined sequences.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence_owner</td>
<td>TEXT</td>
<td>User name of the sequence’s owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the sequence resides.</td>
</tr>
<tr>
<td>sequence_name</td>
<td>TEXT</td>
<td>Name of the sequence.</td>
</tr>
<tr>
<td>min_value</td>
<td>NUMERIC</td>
<td>Lowest value that the server assigns to the sequence.</td>
</tr>
<tr>
<td>max_value</td>
<td>NUMERIC</td>
<td>Highest value that the server assigns to the sequence.</td>
</tr>
<tr>
<td>increment_by</td>
<td>NUMERIC</td>
<td>Value added to the current sequence number to create the next sequent number.</td>
</tr>
<tr>
<td>cycle_flag</td>
<td>CHARACTER VARYING</td>
<td>Specifies whether the sequence wraps when it reaches min_value or max_value.</td>
</tr>
<tr>
<td>order_flag</td>
<td>CHARACTER VARYING</td>
<td>Always returns Y.</td>
</tr>
<tr>
<td>cache_size</td>
<td>NUMERIC</td>
<td>The number of preallocated sequence numbers stored in memory.</td>
</tr>
<tr>
<td>last_number</td>
<td>NUMERIC</td>
<td>The value of the last sequence number saved to disk.</td>
</tr>
</tbody>
</table>
14.3.4.1.52 DBA_SOURCE

The DBA_SOURCE view provides the source code listing of all objects in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the program owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the program belongs.</td>
</tr>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the program.</td>
</tr>
<tr>
<td>type</td>
<td>TEXT</td>
<td>Type of program. Possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.</td>
</tr>
<tr>
<td>line</td>
<td>INTEGER</td>
<td>Source code line number relative to a given program.</td>
</tr>
<tr>
<td>text</td>
<td>TEXT</td>
<td>Line of source code text.</td>
</tr>
</tbody>
</table>

14.3.4.1.53 DBA_SUBPART_KEY_COLUMNS

The DBA_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables that are subpartitioned that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Owner of the table.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the table in which the column resides.</td>
</tr>
<tr>
<td>object_type</td>
<td>CHARACTER(5)</td>
<td>Only for compatibility. Always TABLE.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column on which the key is defined.</td>
</tr>
<tr>
<td>column_position</td>
<td>INTEGER</td>
<td>1 for the first column, 2 for the second column, and so on.</td>
</tr>
</tbody>
</table>

14.3.4.1.54 DBA_SYNONYMS

The DBA_SYNONYM view provides information about all synonyms in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the synonym owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the synonym belongs.</td>
</tr>
<tr>
<td>synonym_name</td>
<td>TEXT</td>
<td>Name of the synonym.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the owner of the table on which the synonym is defined.</td>
</tr>
<tr>
<td>table_schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table on which the synonym is defined.</td>
</tr>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>Name of any associated database link.</td>
</tr>
</tbody>
</table>

14.3.4.1.55 DBA_TAB_COLUMNS

The DBA_TAB_COLUMNS view provides information about all columns in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>CHARACTER VARYING</td>
<td>User name of the owner of the table or view in which the column resides.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the schema in which the table or view resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the table or view in which the column resides.</td>
</tr>
<tr>
<td>column_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>data_type</td>
<td>CHARACTER VARYING</td>
<td>Data type of the column.</td>
</tr>
<tr>
<td>data_length</td>
<td>NUMERIC</td>
<td>Length of text columns.</td>
</tr>
<tr>
<td>data_precision</td>
<td>NUMERIC</td>
<td>Precision (number of digits) for NUMBER columns.</td>
</tr>
<tr>
<td>data_scale</td>
<td>NUMERIC</td>
<td>Scale of NUMBER columns.</td>
</tr>
</tbody>
</table>
nullable | CHARACTER(1) | Whether or not the column is nullable. Possible values are: Y - column is nullable; N - column does not allow null.
column_id | NUMERIC | Relative position of the column in the table or view.
data_default | CHARACTER VARYING | Default value assigned to the column.

### 14.3.4.1.56 DBA_TAB_PARTITIONS

The `DBA_TAB_PARTITIONS` view provides information about all of the partitions that reside in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>Owner of the table in which the partition resides.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>composite</td>
<td>TEXT</td>
<td>YES if the table is subpartitioned, NO if the table isn't subpartitioned.</td>
</tr>
<tr>
<td>partition_name</td>
<td>TEXT</td>
<td>Name of the partition.</td>
</tr>
<tr>
<td>subpartition_count</td>
<td>BIGINT</td>
<td>Number of subpartitions in the partition.</td>
</tr>
<tr>
<td>high_value</td>
<td>TEXT</td>
<td>High-partitioning value specified in the <code>CREATE TABLE</code> statement.</td>
</tr>
<tr>
<td>high_value_length</td>
<td>INTEGER</td>
<td>Length of high-partitioning value.</td>
</tr>
<tr>
<td>partition_position</td>
<td>INTEGER</td>
<td>Ordinal position of this partition.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the partition resides. If the tablespace name isn't specified, the default tablespace is PG_DEFAULT.</td>
</tr>
<tr>
<td>pct_free</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>pct_used</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>ini_trans</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>max_trans</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>initial_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>next_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>min_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>max_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>pct_increase</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>freelists</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>freelists_groups</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>logging</td>
<td>CHARACTER VARYING(7)</td>
<td>Included only for compatibility. Always YES.</td>
</tr>
<tr>
<td>compression</td>
<td>CHARACTER VARYING(8)</td>
<td>Included only for compatibility. Always NONE.</td>
</tr>
<tr>
<td>num_rows</td>
<td>NUMERIC</td>
<td>Same as pg_class.reltuples.</td>
</tr>
<tr>
<td>blocks</td>
<td>INTEGER</td>
<td>Same as pg_class.relpages.</td>
</tr>
<tr>
<td>empty_blocks</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>avg_space</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>chain_cnt</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>avg_row_len</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>sample_size</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>last_analyzed</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>buffer_pool</td>
<td>CHARACTER VARYING(7)</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>global_stats</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always YES.</td>
</tr>
<tr>
<td>user_stats</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always NO.</td>
</tr>
<tr>
<td>backing_table</td>
<td>REGCLASS</td>
<td>Name of the partition backing table.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.57 DBA_TAB_SUBPARTITIONS

The `DBA_TAB_SUBPARTITIONS` view provides information about all of the subpartitions that reside in the database.
### table_owner
- **Type**: TEXT
- **Description**: Owner of the table in which the subpartition resides.

### schema_name
- **Type**: TEXT
- **Description**: Name of the schema in which the table resides.

### table_name
- **Type**: TEXT
- **Description**: Name of the table.

### partition_name
- **Type**: TEXT
- **Description**: Name of the partition.

### subpartition_name
- **Type**: TEXT
- **Description**: Name of the subpartition.

### high_value
- **Type**: TEXT
- **Description**: High-subpartitioning value specified in the CREATE TABLE statement.

### high_value_length
- **Type**: INTEGER
- **Description**: Length of high-partitioning value.

### subpartition_position
- **Type**: INTEGER
- **Description**: Ordinal position of this subpartition.

### tablespace_name
- **Type**: TEXT
- **Description**: Name of the tablespace in which the subpartition resides. If the tablespace name isn't specified, the default tablespace is `PG_DEFAULT`.

### pct_free
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always 0.

### pct_used
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always 0.

### ini_trans
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always 0.

### max_trans
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always 0.

### initial_extent
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### next_extent
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### min_extent
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### max_extent
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### pct_increase
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always 0.

### freelists
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### freelists_groups
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### logging
- **Type**: CHARACTER VARYING(7)
- **Description**: Included only for compatibility. Always YES.

### compression
- **Type**: CHARACTER VARYING(8)
- **Description**: Included only for compatibility. Always NONE.

### num_rows
- **Type**: NUMERIC
- **Description**: Same as `pg_class.reltuples`.

### blocks
- **Type**: INTEGER
- **Description**: Same as `pg_class.relpages`.

### empty_blocks
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### avg_space
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### chain_cnt
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### avg_row_len
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### sample_size
- **Type**: NUMERIC
- **Description**: Included only for compatibility. Always NULL.

### last_analyzed
- **Type**: TIMESTAMP WITHOUT TIME ZONE
- **Description**: Included only for compatibility. Always NULL.

### buffer_pool
- **Type**: CHARACTER VARYING(7)
- **Description**: Included only for compatibility. Always NULL.

### global_stats
- **Type**: CHARACTER VARYING(3)
- **Description**: Included only for compatibility. Always YES.

### user_stats
- **Type**: CHARACTER VARYING(3)
- **Description**: Included only for compatibility. Always NO.

### backing_table
- **Type**: REGCLASS
- **Description**: Name of the subpartition backing table.

### 14.3.4.1.58 DBA_TAB_PRIVS

The `DBA_TAB_PRIVS` view provides a listing of the access privileges granted to database users and to `PUBLIC`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grante e</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user with the privilege.</td>
</tr>
<tr>
<td>owner</td>
<td>CHARACTER VARYING(128)</td>
<td>Object owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the schema in which the object resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Object name.</td>
</tr>
<tr>
<td>grantor</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user who granted the privilege.</td>
</tr>
<tr>
<td>privilege</td>
<td>CHARACTER VARYING(40)</td>
<td>Privilege name.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.59 DBA_TABLES

The `DBA_TABLES` view provides information about all tables in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the table owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides. If the tablespace name isn't specified, the default tablespace is PG_DEFAULT.</td>
</tr>
<tr>
<td>degree</td>
<td>CHARACTER</td>
<td>Number of threads per instance to scan a table, or DEFAULT.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING(5)</td>
<td>Included only for compatibility. Always set to VALID.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHARACTER(1)</td>
<td>Y if the table is temporary. N if the table is permanent.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.60 DBA_TRIGGERS

The `DBA_TRIGGERS` view provides information about all triggers in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the trigger owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the trigger resides.</td>
</tr>
<tr>
<td>trigger_name</td>
<td>TEXT</td>
<td>Name of the trigger.</td>
</tr>
<tr>
<td>trigger_type</td>
<td>TEXT</td>
<td>Type of the trigger. Possible values are: BEFORE ROW, BEFORE STATEMENT, AFTER ROW, AFTER STATEMENT.</td>
</tr>
<tr>
<td>triggering_event</td>
<td>TEXT</td>
<td>Event that fires the trigger.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the owner of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>base_object_type</td>
<td>TEXT</td>
<td>Included only for compatibility. Value is always TABLE.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>referencing_names</td>
<td>TEXT</td>
<td>Included only for compatibility. Value is always REFERENCING NEW AS NEW, OLD AS OLD.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).</td>
</tr>
<tr>
<td>description</td>
<td>TEXT</td>
<td>Included only for compatibility.</td>
</tr>
<tr>
<td>trigger_body</td>
<td>TEXT</td>
<td>The body of the trigger.</td>
</tr>
<tr>
<td>action_statement</td>
<td>TEXT</td>
<td>The SQL command that executes when the trigger fires.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.61 DBA_TYPES

The `DBA_TYPES` view provides information about all object types in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Owner of the object type.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the type is defined.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.62 DBA_USERS

The **DBA_USERS** view provides information about all users of the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>TEXT</td>
<td>User name of the user.</td>
</tr>
<tr>
<td>user_id</td>
<td>OID</td>
<td>ID number of the user.</td>
</tr>
<tr>
<td>password</td>
<td>CHARACTER VARYING(30)</td>
<td>Encrypted password of the user.</td>
</tr>
<tr>
<td>lock_date</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>If the account status is LOCKed, <code>lock_date</code> displays the date and time the account was locked.</td>
</tr>
<tr>
<td>expiry_date</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>The expiration date of the password. Use the <code>edb_get_password_expiry_date(role_id)</code> function to get the current password expiration date.</td>
</tr>
<tr>
<td>default_tablespace</td>
<td>TEXT</td>
<td>Default tablespace associated with the account.</td>
</tr>
<tr>
<td>temporary_tablespace</td>
<td>CHARACTER VARYING(30)</td>
<td>Included only for compatibility. The value is always &quot; (an empty string).</td>
</tr>
<tr>
<td>created</td>
<td>DATE</td>
<td>Timestamp for when the object was created.</td>
</tr>
<tr>
<td>last_ddl_time</td>
<td>DATE</td>
<td>Timestamp for the last modification of an object resulting from a DDL statement including grants and revokes.</td>
</tr>
<tr>
<td>profile</td>
<td>CHARACTER VARYING(30)</td>
<td>The profile associated with the user.</td>
</tr>
<tr>
<td>initial_role_consumer_group</td>
<td>CHARACTER VARYING(30)</td>
<td>Included only for compatibility. The value is always NULL.</td>
</tr>
<tr>
<td>external_name</td>
<td>CHARACTER VARYING(400)</td>
<td>Included only for compatibility. The value is always NULL.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.63 DBA_USED_PRIVS

The **DBA_USED_PRIVS** view provides information about all the used privileges. The used privileges are populated by the `DBMS_PRIVILEGE_CAPTURE.GENERATE_RESULT` procedure.

Supers user or the user with `CAPTURE_ADMIN` role can access this view.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy_name</td>
<td>NAME</td>
<td>Name of the privilege analysis policy.</td>
</tr>
<tr>
<td>run_name</td>
<td>NAME</td>
<td>Name of the run for which privileges are analyzed.</td>
</tr>
<tr>
<td>object_class</td>
<td>TEXT</td>
<td>Name of the database object like table, function, procedure, and so on.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>Name of the object for which used privilege are recorded.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column for the used object privilege.</td>
</tr>
<tr>
<td>application</td>
<td>TEXT</td>
<td>Name of the application through which queries are run. For eg: psql.</td>
</tr>
<tr>
<td>role_name</td>
<td>NAME</td>
<td>Name of the role whose privilege usage is analyzed and recorded.</td>
</tr>
<tr>
<td>privilege_type</td>
<td>TEXT</td>
<td>Name of the used privilege recorded during analysis. Possible values are <code>SELECT</code>, <code>INSERT</code>, <code>UPDATE</code>, <code>DELETE</code>, <code>TRUNCATE</code>, <code>REFERENCES</code>,...</td>
</tr>
</tbody>
</table>
14.3.4.1.64 DBA_UNUSED_PRIVS

The `DBA_UNUSED_PRIVS` view provides information about all the unused privileges. The unused privileges are populated by the `DBMS_PRIVILEGE_CAPTURE.GENERATE_RESULT` procedure.

Superuser or the user with `CAPTURE_ADMIN` role can access this view.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>policy_name</td>
<td>NAME</td>
<td>Name of the privilege analysis policy.</td>
</tr>
<tr>
<td>run_name</td>
<td>NAME</td>
<td>Name of the run for which privileges are analyzed.</td>
</tr>
<tr>
<td>object_class</td>
<td>TEXT</td>
<td>Name of the database object like table, function, procedure, and so on.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>Name of the object for which unused privilege are recorded.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column for the unused object privilege.</td>
</tr>
<tr>
<td>application</td>
<td>TEXT</td>
<td>Name of the application through which queries are run. For eg: psql.</td>
</tr>
<tr>
<td>role_name</td>
<td>NAME</td>
<td>Name of the role whose privilege usage is analyzed and recorded.</td>
</tr>
<tr>
<td>privilege_type</td>
<td>TEXT</td>
<td>Name of the unused privilege recorded during analysis.</td>
</tr>
</tbody>
</table>

14.3.4.1.65 DBA_VIEW_COLUMNS

The `DBA_VIEW_COLUMNS` view provides information on all columns in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>CHARACTER VARYING</td>
<td>User name of the view owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>column_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>data_type</td>
<td>CHARACTER VARYING</td>
<td>Data type of the column.</td>
</tr>
<tr>
<td>data_length</td>
<td>NUMERIC</td>
<td>Length of text columns.</td>
</tr>
<tr>
<td>data_precision</td>
<td>NUMERIC</td>
<td>Precision (number of digits) for NUMBER columns.</td>
</tr>
<tr>
<td>data_scale</td>
<td>NUMERIC</td>
<td>Scale of NUMBER columns.</td>
</tr>
<tr>
<td>nullable</td>
<td>CHARACTER(1)</td>
<td>Whether or not the column is nullable. Possible values are: Y - column is nullable; N - column does not allow null.</td>
</tr>
<tr>
<td>column_id</td>
<td>NUMERIC</td>
<td>Relative position of the column in the view.</td>
</tr>
<tr>
<td>data_default</td>
<td>CHARACTER VARYING</td>
<td>Default value assigned to the column.</td>
</tr>
</tbody>
</table>

14.3.4.1.66 DBA_VIEWS

The `DBA_VIEWS` view provides information about all views in the database.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the view owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>TEXT</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>text</td>
<td>TEXT</td>
<td>Text of the <code>SELECT</code> statement that defines the view.</td>
</tr>
</tbody>
</table>

14.3.4.1.67 USER_ALL_TABLES

The `USER_ALL_TABLES` view provides information about all tables owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
</tbody>
</table>
14.3.4.1.68 USER_CONS_COLUMNS

The **USER_CONS_COLUMNS** view provides information about all columns that are included in constraints in tables that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>User name of the constraint owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column referenced in the constraint.</td>
</tr>
<tr>
<td>position</td>
<td>SMALLINT</td>
<td>Position of the column in the object definition.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>Definition of the constraint.</td>
</tr>
</tbody>
</table>

14.3.4.1.69 USER_CONSTRAINTS

The **USER_CONSTRAINTS** view provides information about all constraints placed on tables that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner</td>
<td>TEXT</td>
<td>Name of the owner of the constraint.</td>
</tr>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the constraint belongs.</td>
</tr>
<tr>
<td>constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint.</td>
</tr>
<tr>
<td>constraint_type</td>
<td>TEXT</td>
<td>The constraint type. Possible values are: C - check constraint; F - foreign key constraint; P - primary key constraint; U - unique key constraint; R - referential integrity constraint; V - constraint on a view; O - with read-only, on a view.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table to which the constraint belongs.</td>
</tr>
<tr>
<td>search_condition</td>
<td>TEXT</td>
<td>Search condition that applies to a check constraint.</td>
</tr>
<tr>
<td>r_owner</td>
<td>TEXT</td>
<td>Owner of a table referenced by a referential constraint.</td>
</tr>
<tr>
<td>r_constraint_name</td>
<td>TEXT</td>
<td>Name of the constraint definition for a referenced table.</td>
</tr>
<tr>
<td>delete_rule</td>
<td>TEXT</td>
<td>The delete rule for a referential constraint. Possible values are: C - cascade; R - restrict; N - no action.</td>
</tr>
<tr>
<td>deferrable</td>
<td>BOOL</td>
<td>Specified if the constraint is deferrable (T or F).</td>
</tr>
<tr>
<td>deferred</td>
<td>BOOL</td>
<td>Specifies if the constraint was deferred (T or F).</td>
</tr>
<tr>
<td>index_owner</td>
<td>TEXT</td>
<td>User name of the index owner.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>constraint_def</td>
<td>TEXT</td>
<td>Definition of the constraint.</td>
</tr>
</tbody>
</table>

14.3.4.1.70 USER_COL_PRIVS

The **USER_COL_PRIVS** view provides a listing of the object privileges granted on a column for which a current user is either an object owner, grantor, or grantee.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
### 14.3.4.1.71 USER_DB_LINKS

The **USER_DB_LINKS** view provides information about all database links that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_link</td>
<td>TEXT</td>
<td>Name of the database link.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING</td>
<td>Type of remote server. Value is either REDWOOD or EDB.</td>
</tr>
<tr>
<td>username</td>
<td>TEXT</td>
<td>User name of the user logging in.</td>
</tr>
<tr>
<td>password</td>
<td>TEXT</td>
<td>Password used to authenticate on the remote server.</td>
</tr>
<tr>
<td>host</td>
<td>TEXT</td>
<td>Name or IP address of the remote server.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.72 USER_DEPENDENCIES

The **USER_DEPENDENCIES** view provides information about dependencies between objects owned by a current user, with the exception of synonyms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the schema in which the dependent object resides.</td>
</tr>
<tr>
<td>name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the dependent object.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING(18)</td>
<td>Type of the dependent object.</td>
</tr>
<tr>
<td>referenced_owner</td>
<td>CHARACTER VARYING(128)</td>
<td>Owner of the referenced object.</td>
</tr>
<tr>
<td>referenced_schema_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the schema in which the referenced object resides.</td>
</tr>
<tr>
<td>referenced_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the referenced object.</td>
</tr>
<tr>
<td>referenced_type</td>
<td>CHARACTER VARYING(18)</td>
<td>Type of the referenced object.</td>
</tr>
<tr>
<td>referenced_link_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>schemaid</td>
<td>NUMERIC</td>
<td>ID of the current schema.</td>
</tr>
<tr>
<td>dependency_type</td>
<td>CHARACTER VARYING(4)</td>
<td>Included only for compatibility. Always set to HARD.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.73 USER_INDEXES

The **USER_INDEXES** view provides information about all indexes on tables that are owned by the current user.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the index belongs.</td>
</tr>
<tr>
<td>index_name</td>
<td>TEXT</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>index_type</td>
<td>TEXT</td>
<td>Included only for compatibility. The index type is always BTREE.</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>User name of the owner of the indexed table.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the indexed table.</td>
</tr>
<tr>
<td>table_type</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to TABLE.</td>
</tr>
<tr>
<td>uniqueness</td>
<td>TEXT</td>
<td>Indicates if the index is UNIQUE or NONUNIQUE.</td>
</tr>
<tr>
<td>compression</td>
<td>CHARACTER(1)</td>
<td>Included only for compatibility. Always set to N (not compressed).</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides if other than the default tablespace.</td>
</tr>
<tr>
<td>degree</td>
<td>CHARACTER VARYING(10)</td>
<td>Number of threads per instance to scan the index.</td>
</tr>
<tr>
<td>logging</td>
<td>TEXT</td>
<td>Included only for compatibility. Always set to LOGGING.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Whether or not the state of the object is valid. (VALID or INVALID).</td>
</tr>
<tr>
<td>partitioned</td>
<td>CHARACTER(3)</td>
<td>Included only for compatibility. Always set to NO.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHARACTER(1)</td>
<td>Included only for compatibility. Always set to N.</td>
</tr>
<tr>
<td>secondary</td>
<td>CHARACTER(1)</td>
<td>Included only for compatibility. Always set to N.</td>
</tr>
<tr>
<td>join_index</td>
<td>CHARACTER(3)</td>
<td>Included only for compatibility. Always set to NO.</td>
</tr>
<tr>
<td>dropped</td>
<td>CHARACTER(3)</td>
<td>Included only for compatibility. Always set to NO.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.74 USER_JOBS

The `USER_JOBS` view provides information about all jobs owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job</td>
<td>INTEGER</td>
<td>Identifier of the job (Job ID).</td>
</tr>
<tr>
<td>log_user</td>
<td>TEXT</td>
<td>Name of the user that submitted the job.</td>
</tr>
<tr>
<td>priv_user</td>
<td>TEXT</td>
<td>Same as log_user. Included only for compatibility.</td>
</tr>
<tr>
<td>schema_user</td>
<td>TEXT</td>
<td>Name of the schema used to parse the job.</td>
</tr>
<tr>
<td>last_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>Last date that this job executed successfully.</td>
</tr>
<tr>
<td>last_sec</td>
<td>TEXT</td>
<td>Same as last_date.</td>
</tr>
<tr>
<td>this_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>Date that the job began executing.</td>
</tr>
<tr>
<td>this_sec</td>
<td>TEXT</td>
<td>Same as this_date.</td>
</tr>
<tr>
<td>next_date</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>Next date that this job will execute.</td>
</tr>
<tr>
<td>next_sec</td>
<td>TEXT</td>
<td>Same as next_date.</td>
</tr>
<tr>
<td>total_time</td>
<td>INTERVAL</td>
<td>Execution time of this job (in seconds).</td>
</tr>
<tr>
<td>broken</td>
<td>TEXT</td>
<td>If Y, no attempt is made to run this job. If N, this job attempts to execute.</td>
</tr>
<tr>
<td>interval</td>
<td>TEXT</td>
<td>Determines how often the job repeats.</td>
</tr>
<tr>
<td>failures</td>
<td>BIGINT</td>
<td>Number of times that the job has failed to complete since it's last successful execution.</td>
</tr>
<tr>
<td>what</td>
<td>TEXT</td>
<td>Job definition (PL/SQL code block) that runs when the job executes.</td>
</tr>
<tr>
<td>nls_env</td>
<td>CHARACTER VARYING(4000)</td>
<td>Always NULL. Provided only for compatibility.</td>
</tr>
<tr>
<td>misc_env</td>
<td>BYTEA</td>
<td>Always NULL. Provided only for compatibility.</td>
</tr>
<tr>
<td>instance</td>
<td>NUMERIC</td>
<td>Always 0. Provided only for compatibility.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.75 USER_OBJECTS

The `USER_OBJECTS` view provides information about all objects that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the object belongs.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>Name of the object.</td>
</tr>
</tbody>
</table>
14.3.4.1.76 USER_PART_TABLES

The **USER_PART_TABLES** view provides information about all of the partitioned tables in the database that are owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>partitioning_type</td>
<td>TEXT</td>
<td>Partitioning type used to define table partitions.</td>
</tr>
<tr>
<td>subpartitioning_type</td>
<td>TEXT</td>
<td>Subpartitioning type used to define table subpartitions.</td>
</tr>
<tr>
<td>partition_count</td>
<td>BIGINT</td>
<td>Number of partitions in the table.</td>
</tr>
<tr>
<td>def_subpartition_count</td>
<td>INTEGER</td>
<td>Number of subpartitions in the table.</td>
</tr>
<tr>
<td>partitioning_key_count</td>
<td>INTEGER</td>
<td>Number of partitioning keys specified.</td>
</tr>
<tr>
<td>subpartitioning_key_count</td>
<td>INTEGER</td>
<td>Number of subpartitioning keys specified.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING(8)</td>
<td>Provided only for compatibility. Always <strong>VALID</strong>.</td>
</tr>
<tr>
<td>def_tablespace_name</td>
<td>CHARACTER VARYING(30)</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_pct_free</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_pct_used</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_ini_trans</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_max_trans</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_initial_extent</td>
<td>CHARACTER VARYING(40)</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_min_extents</td>
<td>CHARACTER VARYING(40)</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_max_extents</td>
<td>CHARACTER VARYING(40)</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_pct_increase</td>
<td>CHARACTER VARYING(40)</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_freelists</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_freelist_groups</td>
<td>NUMERIC</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>def_logging</td>
<td>CHARACTER VARYING(7)</td>
<td>Provided only for compatibility. Always <strong>YES</strong>.</td>
</tr>
<tr>
<td>def_compression</td>
<td>CHARACTER VARYING(8)</td>
<td>Provided only for compatibility. Always <strong>NONE</strong>.</td>
</tr>
<tr>
<td>def_buffer_pool</td>
<td>CHARACTER VARYING(7)</td>
<td>Provided only for compatibility. Always <strong>DEFAULT</strong>.</td>
</tr>
<tr>
<td>ref_ptn_constraint_name</td>
<td>CHARACTER VARYING(30)</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
<tr>
<td>interval</td>
<td>CHARACTER VARYING(100)</td>
<td>Provided only for compatibility. Always <strong>NULL</strong>.</td>
</tr>
</tbody>
</table>

14.3.4.1.77 USER_POLICIES

The **USER_POLICIES** view provides information on policies where the schema containing the object on which the policy applies has the same name as the current session user. This view is accessible only to superusers.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the object resides.</td>
</tr>
<tr>
<td>object_name</td>
<td>TEXT</td>
<td>Name of the object on which the policy applies.</td>
</tr>
<tr>
<td>policy_group</td>
<td>TEXT</td>
<td>Name of the policy group. Included only for compatibility. Always set to an empty string.</td>
</tr>
<tr>
<td>policy_name</td>
<td>TEXT</td>
<td>Name of the policy.</td>
</tr>
<tr>
<td>pf_owner</td>
<td>TEXT</td>
<td>Name of the schema containing the policy function or the schema containing the package that contains the policy function.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.78 USER_QUEUES

The `USER_QUEUES` view provides information about any queue on which the current user has usage privileges.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the queue.</td>
</tr>
<tr>
<td>queue_table</td>
<td>TEXT</td>
<td>Name of the queue table in which the queue resides.</td>
</tr>
<tr>
<td>qid</td>
<td>OID</td>
<td>The system-assigned object ID of the queue.</td>
</tr>
<tr>
<td>queue_type</td>
<td>CHARACTER VARYING</td>
<td>The queue type. Can be <code>EXCEPTION_QUEUE</code>, <code>NON_PERSISTENT_QUEUE</code>, or <code>NORMAL_QUEUE</code>.</td>
</tr>
<tr>
<td>max_retries</td>
<td>NUMERIC</td>
<td>The maximum number of dequeue attempts.</td>
</tr>
<tr>
<td>retrydelay</td>
<td>NUMERIC</td>
<td>The maximum time allowed between retries.</td>
</tr>
<tr>
<td>enqueue_enabled</td>
<td>CHARACTER VARYING</td>
<td>YES if the queue allows enqueuing, NO if the queue does not.</td>
</tr>
<tr>
<td>dequeue_enabled</td>
<td>CHARACTER VARYING</td>
<td>YES if the queue allows dequeuing, NO if the queue does not.</td>
</tr>
<tr>
<td>retention</td>
<td>CHARACTER VARYING</td>
<td>Number of seconds that a processed message is retained in the queue.</td>
</tr>
<tr>
<td>user_comment</td>
<td>CHARACTER VARYING</td>
<td>A user-specified comment.</td>
</tr>
<tr>
<td>network_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the network on which the queue resides.</td>
</tr>
<tr>
<td>sharded</td>
<td>CHARACTER VARYING</td>
<td>YES if the queue resides on a sharded network, NO if the queue does not.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.79 USER_QUEUE_TABLES

The `USER_QUEUE_TABLES` view provides information about all of the queue tables the current user can access.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue_table</td>
<td>TEXT</td>
<td>User-specified name of the queue table.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING</td>
<td>Type of data stored in the queue table.</td>
</tr>
<tr>
<td>object_type</td>
<td>TEXT</td>
<td>User-defined payload type.</td>
</tr>
<tr>
<td>sort_order</td>
<td>CHARACTER VARYING</td>
<td>Order in which the queue table is sorted.</td>
</tr>
<tr>
<td>recipients</td>
<td>CHARACTER VARYING</td>
<td>Always <code>SINGLE</code>.</td>
</tr>
<tr>
<td>message_grouping</td>
<td>CHARACTER VARYING</td>
<td>Always <code>NONE</code>.</td>
</tr>
<tr>
<td>compatible</td>
<td>CHARACTER VARYING</td>
<td>Release number of the EDB Postgres Advanced Server release with which this queue table is compatible.</td>
</tr>
<tr>
<td>primary_instance</td>
<td>NUMERIC</td>
<td>Always 0.</td>
</tr>
<tr>
<td>secondary_instance</td>
<td>NUMERIC</td>
<td>Always 0.</td>
</tr>
<tr>
<td>owner_instance</td>
<td>NUMERIC</td>
<td>Instance number of the instance that owns the queue table.</td>
</tr>
<tr>
<td>user_comment</td>
<td>CHARACTER VARYING</td>
<td>User comment provided when the table was created.</td>
</tr>
<tr>
<td>secure</td>
<td>CHARACTER VARYING</td>
<td><strong>YES</strong> indicates that the queue table is secure, <strong>NO</strong> indicates that it is not.</td>
</tr>
</tbody>
</table>
The **USER_ROLE_PRIVS** view provides information about the privileges that were granted to the current user. A row is created for each role to which a user has been granted.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>TEXT</td>
<td>Name of the user to which the role was granted.</td>
</tr>
<tr>
<td>granted_role</td>
<td>TEXT</td>
<td>Name of the role granted to the grantee.</td>
</tr>
<tr>
<td>admin_option</td>
<td>TEXT</td>
<td>YES if the role was granted with the admin option, NO otherwise.</td>
</tr>
<tr>
<td>default_role</td>
<td>TEXT</td>
<td>YES if the role is enabled when the grantee creates a session.</td>
</tr>
<tr>
<td>os_granted</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always NO.</td>
</tr>
</tbody>
</table>

**14.3.4.1.81 USER_SEQUENCES**

The **USER_SEQUENCES** view provides information about all user-defined sequences that belong to the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the sequence resides.</td>
</tr>
<tr>
<td>sequence_name</td>
<td>TEXT</td>
<td>Name of the sequence.</td>
</tr>
<tr>
<td>min_value</td>
<td>NUMERIC</td>
<td>Lowest value for the server to assign to the sequence.</td>
</tr>
<tr>
<td>max_value</td>
<td>NUMERIC</td>
<td>Highest value for the server to assign to the sequence.</td>
</tr>
<tr>
<td>increment_by</td>
<td>NUMERIC</td>
<td>Value added to the current sequence number to create the next sequent number.</td>
</tr>
<tr>
<td>cycle_flag</td>
<td>CHARACTER VARYING</td>
<td>Specifies whether the sequence wraps when it reaches min_value or max_value.</td>
</tr>
<tr>
<td>order_flag</td>
<td>CHARACTER VARYING</td>
<td>Included only for compatibility. Always Y.</td>
</tr>
<tr>
<td>cache_size</td>
<td>NUMERIC</td>
<td>Number of preallocated sequence numbers in memory.</td>
</tr>
<tr>
<td>last_number</td>
<td>NUMERIC</td>
<td>The value of the last sequence number saved to disk.</td>
</tr>
</tbody>
</table>

**14.3.4.1.82 USER_SOURCE**

The **USER_SOURCE** view provides information about all programs owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the program belongs.</td>
</tr>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the program.</td>
</tr>
<tr>
<td>type</td>
<td>TEXT</td>
<td>Type of program. Possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.</td>
</tr>
<tr>
<td>line</td>
<td>INTEGER</td>
<td>Source code line number relative to a given program.</td>
</tr>
<tr>
<td>text</td>
<td>TEXT</td>
<td>Line of source code text.</td>
</tr>
</tbody>
</table>

**14.3.4.1.83 USER_SUBPART_KEY_COLUMNS**

The **USER_SUBPART_KEY_COLUMNS** view provides information about the key columns of partitioned tables that are subpartitioned that belong to the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>name</td>
<td>TEXT</td>
<td>Name of the table in which the column resides.</td>
</tr>
<tr>
<td>object_type</td>
<td>CHARACTER(5)</td>
<td>For compatibility only. Always TABLE.</td>
</tr>
<tr>
<td>column_name</td>
<td>TEXT</td>
<td>Name of the column on which the key is defined.</td>
</tr>
<tr>
<td>column_position</td>
<td>INTEGER</td>
<td>1 for the first column, 2 for the second column, and so on.</td>
</tr>
</tbody>
</table>

**14.3.4.1.84 USER_SYNONYMS**

The **USER_SYNONYMS** view provides information about all synonyms owned by the current user.
### USER_TAB_COLUMNS

The **USER_TAB_COLUMNS** view displays information about all columns in tables and views owned by the current user.

- **schema_name** (TEXT): Name of the schema in which the table or view resides.
- **table_name** (CHARACTER VARYING): Name of the table or view in which the column resides.
- **column_name** (CHARACTER VARYING): Name of the column.
- **data_type** (CHARACTER VARYING): Data type of the column.
- **data_length** (NUMERIC): Length of text columns.
- **data_precision** (NUMERIC): Precision (number of digits) for NUMBER columns.
- **data_scale** (NUMERIC): Scale of NUMBER columns.
- **nullable** (CHARACTER(1)): Whether or not the column is nullable. Possible values are: **Y** - column is nullable; **N** - column does not allow null.
- **column_id** (NUMERIC): Relative position of the column in the table.
- **data_default** (CHARACTER VARYING): Default value assigned to the column.

### USER_TAB_PARTITIONS

The **USER_TAB_PARTITIONS** view provides information about all of the partitions that are owned by the current user.

- **schema_name** (TEXT): Name of the schema in which the table resides.
- **table_name** (TEXT): Name of the table.
- **composite** (TEXT): YES if the table is subpartitioned, NO if the table isn’t subpartitioned.
- **partition_name** (TEXT): Name of the partition.
- **subpartition_count** (BIGINT): Number of subpartitions in the partition.
- **high_value** (TEXT): High-partitioning value specified in the CREATE TABLE statement.
- **high_value_length** (INTEGER): Length of high-partitioning value.
- **partition_position** (INTEGER): The ordinal position of this partition.
- **tablespace_name** (TEXT): Name of the tablespace in which the partition resides. If the tablespace name isn’t specified, then default tablespace is **PG_DEFAULT**.
- **pct_free** (NUMERIC): Included only for compatibility. Always 0.
- **pct_used** (NUMERIC): Included only for compatibility. Always 0.
- **init_trans** (NUMERIC): Included only for compatibility. Always 0.
- **max_trans** (NUMERIC): Included only for compatibility. Always 0.
- **initial_extent** (NUMERIC): Included only for compatibility. Always NULL.
- **next_extent** (NUMERIC): Included only for compatibility. Always NULL.
- **min_extent** (NUMERIC): Included only for compatibility. Always 0.
- **max_extent** (NUMERIC): Included only for compatibility. Always 0.
- **pct_increase** (NUMERIC): Included only for compatibility. Always 0.
- **freelists** (NUMERIC): Included only for compatibility. Always 0.
- **freelist_groups** (NUMERIC): Included only for compatibility. Always 0.
- **logging** (CHARACTER VARYING(7)): Included only for compatibility. Always YES.
### 14.3.4.1.87 USER_TAB_SUBPARTITIONS

The **USER_TAB_SUBPARTITIONS** view provides information about all of the subpartitions owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>partition_name</td>
<td>TEXT</td>
<td>Name of the partition.</td>
</tr>
<tr>
<td>subpartition_name</td>
<td>TEXT</td>
<td>Name of the subpartition.</td>
</tr>
<tr>
<td>high_value</td>
<td>TEXT</td>
<td>High-subpartitioning value specified in the <code>CREATE TABLE</code> statement.</td>
</tr>
<tr>
<td>high_value_length</td>
<td>INTEGER</td>
<td>Length of high-partitioning value.</td>
</tr>
<tr>
<td>subpartition_position</td>
<td>INTEGER</td>
<td>Ordinal position of this subpartition.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the subpartition resides. If the tablespace name is not specified, then default tablespace is <code>PG_DEFAULT</code>.</td>
</tr>
<tr>
<td>pct_free</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>pct_used</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>init_trans</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>max_trans</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>initial_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>next_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>min_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>max_extent</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>pct_increase</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always 0.</td>
</tr>
<tr>
<td>freelists</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>freelist_groups</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>logging</td>
<td>CHARACTER VARYING(7)</td>
<td>Included only for compatibility. Always YES.</td>
</tr>
<tr>
<td>compression</td>
<td>CHARACTER VARYING(8)</td>
<td>Included only for compatibility. Always NONE.</td>
</tr>
<tr>
<td>num_rows</td>
<td>NUMERIC</td>
<td>Same as <code>pg_class.reltuples</code>.</td>
</tr>
<tr>
<td>blocks</td>
<td>INTEGER</td>
<td>Same as <code>pg_class.relpages</code>.</td>
</tr>
<tr>
<td>empty_blocks</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>avg_space</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>chain_cnt</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>avg_row_len</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>sample_size</td>
<td>NUMERIC</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>last_analyzed</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>buffer_pool</td>
<td>CHARACTER VARYING(7)</td>
<td>Included only for compatibility. Always NULL.</td>
</tr>
<tr>
<td>global_stats</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always YES.</td>
</tr>
</tbody>
</table>
### 14.3.4.1.88 USER_TAB_PRIVS

The `USER_TAB_PRIVS` view provides a listing of the object privileges for which a current user is either an object owner, grantor, or grantee.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grantee</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user with the privilege.</td>
</tr>
<tr>
<td>owner</td>
<td>CHARACTER VARYING(128)</td>
<td>Object owner.</td>
</tr>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the schema in which the object resides.</td>
</tr>
<tr>
<td>table_name</td>
<td>CHARACTER VARYING(128)</td>
<td>Object name.</td>
</tr>
<tr>
<td>grantor</td>
<td>CHARACTER VARYING(128)</td>
<td>Name of the user who granted the privilege.</td>
</tr>
<tr>
<td>privilege</td>
<td>CHARACTER VARYING(48)</td>
<td>Privilege name.</td>
</tr>
<tr>
<td>grantable</td>
<td>CHARACTER VARYING(3)</td>
<td>Indicates whether the privilege was granted with the grant option YES or NO.</td>
</tr>
<tr>
<td>hierarchy</td>
<td>CHARACTER VARYING(3)</td>
<td>The value can be YES or NO. The value can be YES if the privilege is SELECT. Otherwise the value is NO.</td>
</tr>
<tr>
<td>common</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always NO.</td>
</tr>
<tr>
<td>type</td>
<td>CHARACTER VARYING(24)</td>
<td>Type of object.</td>
</tr>
<tr>
<td>inherited</td>
<td>CHARACTER VARYING(3)</td>
<td>Included only for compatibility. Always NO.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.89 USER_TABLES

The `USER_TABLES` view displays information about all tables owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the table belongs.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table.</td>
</tr>
<tr>
<td>tablespace_name</td>
<td>TEXT</td>
<td>Name of the tablespace in which the table resides. If the tablespace name isn't specified, the default tablespace is PG_DEFAULT.</td>
</tr>
<tr>
<td>degree</td>
<td>CHARACTER VARYING(10)</td>
<td>Number of threads per instance to scan a table, or DEFAULT.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING(5)</td>
<td>Included only for compatibility. Always set to VALID.</td>
</tr>
<tr>
<td>temporary</td>
<td>CHARACTER(1)</td>
<td>Y if the table is temporary, N if the table isn't temporary.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.90 USER_TRIGGERS

The `USER_TRIGGERS` view displays information about all triggers on tables owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the trigger resides.</td>
</tr>
<tr>
<td>trigger_name</td>
<td>TEXT</td>
<td>Name of the trigger.</td>
</tr>
<tr>
<td>trigger_type</td>
<td>TEXT</td>
<td>The type of the trigger. Possible values are: BEFORE ROW, BEFORE STATEMENT, AFTER ROW, AFTER STATEMENT.</td>
</tr>
<tr>
<td>triggering_event</td>
<td>TEXT</td>
<td>The event that fires the trigger.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>table_owner</td>
<td>TEXT</td>
<td>The user name of the owner of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>base_object_type</td>
<td>TEXT</td>
<td>Included only for compatibility. Value is always TABLE.</td>
</tr>
<tr>
<td>table_name</td>
<td>TEXT</td>
<td>Name of the table on which the trigger is defined.</td>
</tr>
<tr>
<td>referencing_names</td>
<td>TEXT</td>
<td>Included only for compatibility. Value is always REFERENCING NEW AS NEW OLD AS OLD.</td>
</tr>
<tr>
<td>status</td>
<td>TEXT</td>
<td>Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).</td>
</tr>
<tr>
<td>description</td>
<td>TEXT</td>
<td>Included only for compatibility.</td>
</tr>
<tr>
<td>trigger_body</td>
<td>TEXT</td>
<td>The body of the trigger.</td>
</tr>
<tr>
<td>action_statement</td>
<td>TEXT</td>
<td>The SQL command that executes when the trigger fires.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.91 USER_TYPES

The **USER_TYPES** view provides information about all object types owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>TEXT</td>
<td>Name of the schema in which the type is defined.</td>
</tr>
<tr>
<td>type_name</td>
<td>TEXT</td>
<td>Name of the type.</td>
</tr>
<tr>
<td>type_oid</td>
<td>OID</td>
<td>The object identifier (OID) of the type.</td>
</tr>
<tr>
<td>typecode</td>
<td>TEXT</td>
<td>The typecode of the type. Possible values are: OBJECT, COLLECTION, OTHER.</td>
</tr>
<tr>
<td>attributes</td>
<td>INTEGER</td>
<td>Number of attributes in the type.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.92 USER_USERS

The **USER_USERS** view provides information about the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>TEXT</td>
<td>User name of the user.</td>
</tr>
<tr>
<td>user_id</td>
<td>OID</td>
<td>ID number of the user.</td>
</tr>
<tr>
<td>account_status</td>
<td>CHARACTER VARYING(32)</td>
<td>The current status of the account. Possible values are: OPEN, EXPIRED, EXPIRED(GRACE), EXPIRED &amp; LOCKED, EXPIRED &amp; LOCKED(TIMED), EXPIRED(GRACE) &amp; LOCKED, EXPIRED(GRACE) &amp; LOCKED(TIMED), LOCKED, LOCKED(TIMED). Use the edb_get_role_status(role_id) function to get the current status of the account.</td>
</tr>
<tr>
<td>lock_date</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>If the account status is LOCKED, lock_date displays the date and time the account was locked.</td>
</tr>
<tr>
<td>expiry_date</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>The expiration date of the account.</td>
</tr>
<tr>
<td>default_tablespace</td>
<td>TEXT</td>
<td>The default tablespace associated with the account.</td>
</tr>
<tr>
<td>temporary_tablespace</td>
<td>CHARACTER VARYING(38)</td>
<td>Included only for compatibility. The value is always &quot; (an empty string).</td>
</tr>
<tr>
<td>created</td>
<td>DATE</td>
<td>Timestamp for the creation of an object.</td>
</tr>
<tr>
<td>last_ddl_time</td>
<td>DATE</td>
<td>Timestamp for the last modification of an object resulting from a DDL statement including grants and revokes.</td>
</tr>
<tr>
<td>initial_role_consumption group</td>
<td>CHARACTER VARYING(38)</td>
<td>Included only for compatibility. The value is always NULL.</td>
</tr>
<tr>
<td>external_name</td>
<td>CHARACTER VARYING(48 00)</td>
<td>Included only for compatibility. Always set to NULL.</td>
</tr>
</tbody>
</table>

### 14.3.4.1.93 USER_VIEW_COLUMNS

The **USER_VIEW_COLUMNS** view provides information about all columns in views owned by the current user.
### USER_VIEWS

The `USER_VIEWS` view provides information about all views owned by the current user.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the schema in which the view belongs.</td>
</tr>
<tr>
<td>view_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the view.</td>
</tr>
<tr>
<td>column_name</td>
<td>CHARACTER VARYING</td>
<td>Name of the column.</td>
</tr>
<tr>
<td>data_type</td>
<td>CHARACTER VARYING</td>
<td>Data type of the column.</td>
</tr>
<tr>
<td>data_length</td>
<td>NUMERIC</td>
<td>Length of text columns.</td>
</tr>
<tr>
<td>data_precision</td>
<td>NUMERIC</td>
<td>Precision (number of digits) for NUMBER columns.</td>
</tr>
<tr>
<td>data_scale</td>
<td>NUMERIC</td>
<td>Scale of NUMBER columns.</td>
</tr>
<tr>
<td>nullable</td>
<td>CHARACTER(1)</td>
<td>Whether or not the column is nullable. Possible values are: Y - column is nullable; N - column does not allow null.</td>
</tr>
<tr>
<td>column_id</td>
<td>NUMERIC</td>
<td>Relative position of the column in the view.</td>
</tr>
<tr>
<td>data_default</td>
<td>CHARACTER VARYING</td>
<td>Default value assigned to the column.</td>
</tr>
</tbody>
</table>

### V$VERSION

The `V$VERSION` view provides information about product compatibility.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>banner</td>
<td>TEXT</td>
<td>Displays product compatibility information.</td>
</tr>
</tbody>
</table>

### PRODUCT_COMPONENT_VERSION

The `PRODUCT_COMPONENT_VERSION` view provides version information about product version compatibility.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>CHARACTER VARYING(74)</td>
<td>Name of the product.</td>
</tr>
<tr>
<td>version</td>
<td>CHARACTER VARYING(74)</td>
<td>Version number of the product.</td>
</tr>
<tr>
<td>status</td>
<td>CHARACTER VARYING(74)</td>
<td>Included for compatibility. Always Available.</td>
</tr>
</tbody>
</table>

### System catalog views

System catalog views are present in PostgreSQL. They may have extra information when accessed in EDB Postgres Advanced Server. The `PG_USER` view is an example of a system catalog view that provides additional information specific to EDB Postgres Advanced Server. For the complete list of the System Catalogs see [PostgreSQL System Catalogs](https://www.postgresql.org/docs/14/catalog.html).

### PG_USER

The `PG_USER` view provides information about the database users. It provides information specific to EDB Postgres Advanced Server in addition to the information provided in the [PostgreSQL pg_user](https://www.postgresql.org/docs/14/catalog.html) view.
### 14.3.5 Compatible SQL commands

EDB Postgres Advanced Server supports many SQL commands compatible with Oracle databases. These SQL commands work on both an Oracle database and an EDB Postgres Advanced Server database.

EDB Postgres Advanced Server supports additional commands not described here. These commands might not have an Oracle equivalent, or they might provide the similar or same functionality as an Oracle SQL command but with different syntax.

The SQL commands that follow don't necessarily represent the full syntax, options, and functionality available for each command. In most cases, syntax, options, and functionality that aren't compatible with Oracle databases aren't included in the command description and syntax.

Command functionality that isn't compatible with Oracle databases is noted.

#### 14.3.5.1 ALTER CAPTURE PRIVILEGE POLICY

EDB Postgres Advanced Server includes extra syntax not offered by Oracle with the `ALTER CAPTURE PRIVILEGES POLICY` SQL command.

**Name**

`ALTER CAPTURE PRIVILEGES POLICY` — Allows a superuser or a user with the `CAPTURE_ADMIN` role to modify the attributes of the capture privilege policy.

**Synopsis**

This command is available in five forms.

```
ALTER CAPTURE PRIVILEGES POLICY <name> SET WHERE ( <expression> )
```

```
ALTER CAPTURE PRIVILEGES POLICY <name> DROP WHERE
```

```
ALTER CAPTURE PRIVILEGES POLICY <name> SET ( ROLE | USER ) <role_names>
```

```
ALTER CAPTURE PRIVILEGES POLICY <name> DROP ( ROLE | USER )
```

```
ALTER CAPTURE PRIVILEGES POLICY <name> ( ENABLE | DISABLE ) [ <run_name> ]
```

**Description**

The `ALTER CAPTURE PRIVILEGES POLICY` command allows a database superuser or any user with the system-defined `CAPTURE_ADMIN` privilege to a policy in the database.

Use `DROP CAPTURE PRIVILEGES POLICY` to remove a capture policy.

**Parameters**

- `name`
expression

Boolean expression containing up to 4000 characters.

role_names

The roles whose privileges are to be analyzed.

run_name

Name of the capture run.

Examples

This command alters the attributes of a capture privilege policy:

```sql
## Enable and disable policy on the database
ALTER CAPTURE PRIVILEGES POLICY sampledb_p1 ENABLE sampledb_run1;
ALTER CAPTURE PRIVILEGES POLICY sampledb_p1 DISABLE;
## Enable and disable policy on the role
ALTER CAPTURE PRIVILEGES POLICY sampledb_p2 ENABLE sampledb_role_run1;
ALTER CAPTURE PRIVILEGES POLICY sampledb_p2 DISABLE sampledb_role_run1;
```

See also

CREATE CAPTURE PRIVILEGE POLICY, DROP CAPTURE PRIVILEGE POLICY

14.3.5.2 ALTER DIRECTORY

Name

`ALTER DIRECTORY` – Change the owner of a directory created using the `CREATE DIRECTORY` command.

Synopsis

```sql
ALTER DIRECTORY <name> OWNER TO <rolename>
```

Description

The `ALTER DIRECTORY ...OWNER TO` command changes the owner of a directory. You must have the superuser privilege to execute this command. The new owner of the directory must also have the superuser privilege.

Parameters

name

The name of the directory to alter.
The name of an owner of the directory.

Examples

These examples change ownership. bob and carol are superusers. bob is a current owner of the directory EMPDIR.

```
SELECT * FROM all_directories WHERE directory_name = 'EMPDIR'
order by 1,2,3;
```

```
| owner | directory_name | directory_path |
|-------+----------------+----------------|
| bob   | EMPDIR         | /path          |

(1 row)
```

To change the ownership of directory EMPDIR to carol:

```
ALTER DIRECTORY EMPDIR OWNER TO carol;
```

```
SELECT * FROM all_directories WHERE directory_name = 'EMPDIR'
order by 1,2,3;
```

```
| owner | directory_name | directory_path |
|-------+----------------+----------------|
| carol | EMPDIR         | /path          |

(1 row)
```

See also

CREATE DIRECTORY, DROP DIRECTORY

14.3.5.3 ALTER INDEX

Name

ALTER INDEX — Modify an existing index.

Synopsis

EDB Postgres Advanced Server supports three variations of the ALTER INDEX command compatible with Oracle databases. Use the first variation to rename an index:

```
ALTER INDEX <name> RENAME TO <new_name>
```

Use the second variation of the ALTER INDEX command to rebuild an index:

```
ALTER INDEX <name> REBUILD
```

Use the third variation of the ALTER INDEX command to set the PARALLEL or NOPARALLEL clause:

```
ALTER INDEX <name> { NOPARALLEL | PARALLEL [ <integer> ] }
```

Description
ALTER INDEX changes the definition of an existing index. The RENAME clause changes the name of the index. The REBUILD clause reconstructs an index, replacing the old copy of the index with an updated version based on the index's table.

The REBUILD clause invokes the PostgreSQL REINDEX command. For more information about using the REBUILD clause, see the PostgreSQL core documentation.

The PARALLEL clause sets the degree of parallelism for an index that can be used to parallelize rebuilding an index.

The NOPARALLEL clause resets parallelism to use default values. reloptions shows the parallel_workers parameter as 0.

ALTER INDEX has no effect on stored data.

Parameters

**name**

The name (possibly schema-qualified) of an existing index.

**new_name**

New name for the index.

**PARALLEL**

Include the PARALLEL clause to specify a degree of parallelism. Set the parallel_workers parameter equal to the degree of parallelism for rebuilding an index. If you specify PARALLEL but don't provide a degree of parallelism, the default parallelism is used.

**NOPARALLEL**

Specify NOPARALLEL to reset parallelism to default values.

**integer**

The integer indicates the degree of parallelism, that is, the number of parallel_workers used when rebuilding an index.

Examples

To change the name of an index from name_idx to empname_idx:

```
ALTER INDEX name_idx RENAME TO empname_idx;
```

To rebuild an index named empname_idx:

```
ALTER INDEX empname_idx REBUILD;
```

This example sets the degree of parallelism on an empname_idx index to 7:

```
ALTER INDEX empname_idx PARALLEL 7;
```

See also

CREATE INDEX, DROP INDEX

14.3.5.4 ALTER PROCEDURE

Name

ALTER PROCEDURE — Modify an existing procedure.
Synopsis

```sql
ALTER PROCEDURE <procedure_name> <options> [RESTRICT]
```

Description

Use the ALTER PROCEDURE statement to specify that a procedure is a SECURITY INVOKER or SECURITY DEFINER.

Parameters

- **procedure_name**
  The (possibly schema-qualified) name of a stored procedure.

- **options** can be:
  - `[EXTERNAL] SECURITY DEFINER`
    Specify SECURITY DEFINER to execute the procedure with the privileges of the user that created the procedure. The EXTERNAL keyword is accepted for compatibility but ignored.
  - `[EXTERNAL] SECURITY INVOKER`
    Specify SECURITY INVOKER to execute the procedure with the privileges of the user that's invoking the procedure. The EXTERNAL keyword is accepted for compatibility but ignored.
  The RESTRICT keyword is accepted for compatibility but ignored.

Examples

This command specifies for the `update_balance` to execute with the privileges of the user invoking the procedure:

```sql
ALTER PROCEDURE update_balance SECURITY INVOKER;
```

See also

CREATE PROCEDURE, DROP PROCEDURE

14.3.5.5 ALTER PROFILE

Name

- **ALTER PROFILE** – Alter an existing profile.

Synopsis

```sql
ALTER PROFILE <profile_name> RENAME TO <new_name>;
ALTER PROFILE <profile_name> LIMIT {[<parameter value>] [...];}
```
Use the `ALTER PROFILE` command to modify a user-defined profile. EDB Postgres Advanced Server supports two forms of the command:

- Use `ALTER PROFILE RENAME TO` to change the name of a profile.
- Use `ALTER PROFILE LIMIT` to modify the limits associated with a profile.

Include the `LIMIT` clause and one or more space-delimited `parameter/value` pairs to specify the rules enforced by EDB Postgres Advanced Server. Use `ALTER PROFILE RENAME TO` to change the name of a profile.

### Parameters

- **profile_name**
  - The name of the profile.

- **new_name**
  - The new name of the profile.

- **parameter**
  - The attribute limited by the profile.

- **value**
  - The parameter limit.

EDB Postgres Advanced Server supports these values for each parameter:

- **FAILED_LOGIN_ATTEMPTS** specifies the number of failed login attempts that a user can make before the server locks them out of their account for the length of time specified by `PASSWORD_LOCK_TIME`. Supported values are:
  - An `INTEGER` value greater than 0.
  - `DEFAULT` — The value of `FAILED_LOGIN_ATTEMPTS` specified in the `DEFAULT` profile.
  - `UNLIMITED` — The connecting user can make an unlimited number of failed login attempts.

- **PASSWORD_LOCK_TIME** specifies the length of time that must pass before the server unlocks an account that was locked because of `FAILED_LOGIN_ATTEMPTS`. Supported values are:
  - A `NUMERIC` value of 0 or greater. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
  - `DEFAULT` — The value of `PASSWORD_LOCK_TIME` specified in the `DEFAULT` profile.
  - `UNLIMITED` — The account is locked until manually unlocked by a database superuser.

- **PASSWORD_LIFE_TIME** specifies the number of days to use the current password before the user is prompted to provide a new password. Include the `PASSWORD_GRACE_TIME` clause when using the `PASSWORD_LIFE_TIME` clause to specify the number of days after the password expires before connections by the role are rejected. If you don't specify `PASSWORD_GRACE_TIME`, the password expires on the day specified by the default value of `PASSWORD_GRACE_TIME`. The user can't execute any command until they provide a new password. Supported values are:
  - A `NUMERIC` value of 0 or greater. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
  - `DEFAULT` — The value of `PASSWORD_LIFE_TIME` specified in the `DEFAULT` profile.
  - `UNLIMITED` — The password doesn't have an expiration date.

- **PASSWORD_GRACE_TIME** specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user is allowed to connect but isn't allowed to execute any command until they update their expired password. Supported values are:
  - A `NUMERIC` value of 0 or greater. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
  - `DEFAULT` — The value of `PASSWORD_GRACE_TIME` specified in the `DEFAULT` profile.
  - `UNLIMITED` — The grace period is infinite.

- **PASSWORD_REUSE_TIME** specifies the number of days a user must wait before reusing a password. Use the `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters together. If you specify a finite value for one of these parameters while the other is `UNLIMITED`, old passwords can never be reused. If both parameters are set to `UNLIMITED`, there are no restrictions on password reuse. Supported values are:
  - A `NUMERIC` value of 0 or greater. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify 4 days, 12 hours.
  - `DEFAULT` — The value of `PASSWORD_REUSE_TIME` specified in the `DEFAULT` profile.
  - `UNLIMITED` — The password can be reused without restrictions.

- **PASSWORD_REUSE_MAX** specifies the number of password changes that must occur before a password can be reused. Use the `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters together. If you specify a finite value for one of these parameters while the other is `UNLIMITED`, old passwords can never be reused. If both parameters are set to `UNLIMITED`, there are no restrictions on password reuse. Supported values are:
An INTEGER value of 0 or greater.

- DEFAULT — The value of PASSWORD_REUSE_MAX specified in the DEFAULT profile.
- UNLIMITED — The password can be reused without restrictions.

**PASSWORD_VERIFY_FUNCTION** specifies password complexity. Supported values are:

- The name of a PL/SQL function.
- DEFAULT — The value of PASSWORD_VERIFY_FUNCTION specified in the DEFAULT profile.
- NULL

**PASSWORD_ALLOW_HASHED** specifies whether an encrypted password is allowed. If you specify the value as TRUE, the system allows a user to change the password by specifying a hash-computed encrypted password on the client side. However, if you specify the value as FALSE, then a password must be specified in a plain-text form to validate without error. Supported values are:

- A BOOLEAN value TRUE/ON/YES/1 or FALSE/OFF/NO/0.
- DEFAULT — The value of PASSWORD_ALLOW_HASHED specified in the DEFAULT profile.

**Note**

The **PASSWORD_ALLOW_HASHED** isn't compatible with Oracle.

**Examples**

This example modifies a profile named acctg_profile:

```
ALTER PROFILE acctg_profile
   LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

The acctg_profile counts failed connection attempts when a login role attempts to connect to the server. The profile specifies that if a user doesn't authenticate with the correct password in three attempts, the account is locked for one day.

This example changes the name of acctg_profile to payables_profile:

```
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

**See also**

CREATE PROFILE, DROP PROFILE

### 14.3.5.6 ALTER QUEUE

EDB Postgres Advanced Server includes an extra syntax not offered by Oracle with the ALTER QUEUE SQL command. You can use this syntax with the DBMS_AQADM package.

**Name**

- **ALTER QUEUE** — Allows a superuser or a user with the aq_administrator_role privilege to modify the attributes of a queue.

**Synopsis**

This command is available in four forms.

**Parameters for the first form**

The first form of this command changes the name of a queue:

```
ALTER QUEUE <queue_name> RENAME TO <new_name>
```
The name (optionally schema-qualified) of an existing queue.

**RENAME TO**

To rename the queue, include the `RENAME TO` clause and a new name for the queue.

New name for the queue.

**Parameters for the second form**

The second form of the `ALTER QUEUE` command modifies the attributes of the queue:

```
ALTER QUEUE <queue_name> SET { { <option_name> <option_value> } } [,SET <option_name> ]
```

The name (optionally schema-qualified) of an existing queue.

Include the `SET` clause and `option_name/option_value` pairs to modify the attributes of the queue.

```
option_name option_value
```

The name of an option to associate with the new queue and the corresponding value of the option. If you provide duplicate option names, the server returns an error.

- If `option_name` is `retries`, provide an integer that represents the number of times to attempt a dequeue.
- If `option_name` is `retrydelay`, provide a double-precision value that represents the delay in seconds.
- If `option_name` is `retention`, provide a double-precision value that represents the retention time in seconds.

**Parameters for the third form**

Use the third form of the `ALTER QUEUE` command to enable or disable enqueuing or dequeuing on a queue:

```
ALTER QUEUE <queue_name> ACCESS { START | STOP } [ FOR { enqueue | dequeue } ] [ NOWAIT
```

The name (optionally schema-qualified) of an existing queue.

```
queue_name
```

The `ACCESS` keyword to enable or disable enqueuing or dequeuing on a queue.

**START | STOP**

Use the `START` and `STOP` keywords to indicate the desired state of the queue.

```
FOR enqueue|dequeue
```

Use the `FOR` clause to indicate if you are specifying the state of enqueuing or dequeuing activity on the specified queue.

**NOWAIT**

Include the `NOWAIT` keyword to specify for the server not to wait for the completion of outstanding transactions before changing the state of the queue. You can use the `NOWAIT` keyword only when specifying an `ACCESS` value of `STOP`. The server returns an error if `NOWAIT` is specified with an `ACCESS` value of `START`.

**Parameters for the fourth form**
Use the fourth form to **ADD** or **DROP** callback details for a queue.

```
ALTER QUEUE <queue_name> { ADD | DROP } CALL TO <location_name> [ WITH <callback_option> ]
```

### queue_name

The name (optionally schema-qualified) of an existing queue.

### ADD | DROP

Include the **ADD** or **DROP** keywords to enable add or remove callback details for a queue.

### location_name

Specifies the name of the callback procedure.

### callback_option

Can be **context**. Specify a *RAW* value when including this clause.

### Examples

This example changes the name of a queue from `work_queue_east` to `work_order`:

```
ALTER QUEUE work_queue_east RENAME TO work_order;
```

This example modifies a queue named `work_order`. It sets the number of retries to 100, the delay between retries to 2 seconds, and the length of time that the queue retains dequeued messages to 10 seconds:

```
ALTER QUEUE work_order SET (retries 100, retrydelay 2, retention 10);
```

These commands enable enqueueing and dequeueing in a queue named `work_order`:

```
ALTER QUEUE work_order ACCESS START;
ALTER QUEUE work_order ACCESS START FOR enqueue;
ALTER QUEUE work_order ACCESS START FOR dequeue;
```

These commands disable enqueueing and dequeueing in a queue named `work_order`:

```
ALTER QUEUE work_order ACCESS STOP NOWAIT;
ALTER QUEUE work_order ACCESS STOP FOR enqueue;
ALTER QUEUE work_order ACCESS STOP FOR dequeue;
```

### See also

CREATE QUEUE, DROP QUEUE

### 14.3.5.7 ALTER QUEUE TABLE

EDB Postgres Advanced Server includes extra syntax not offered by Oracle with the **ALTER QUEUE SQL** command. You can use this syntax with the **DBMS_AQADM** package.

### Name

```
ALTER QUEUE TABLE — Modify an existing queue table.
```
Synopsis

Use `ALTER QUEUE TABLE` to change the name of an existing queue table:

```
ALTER QUEUE TABLE <name> RENAME TO <new_name>
```

Description

`ALTER QUEUE TABLE` allows a superuser or a user with the `aq_administrator_role` privilege to change the name of an existing queue table.

Parameters

- `name`
  The name (optionally schema-qualified) of an existing queue table.
- `new_name`
  New name for the queue table.

Examples

This example changes the name of a queue table from `wo_table_east` to `work_order_table`:

```
ALTER QUEUE TABLE wo_queue_east RENAME TO work_order_table;
```

See also

`CREATE QUEUE TABLE`, `DROP QUEUE TABLE`

14.3.5.8  ALTER ROLE... IDENTIFIED BY

Name

`ALTER ROLE` — Change the password associated with a database role.

Synopsis

```
ALTER ROLE <role_name> IDENTIFIED BY <password>
[REPLACE]
<prev_password>
```

Description

A role without the `CREATEROLE` privilege can use this command to change their own password. An unprivileged role must include the `REPLACE` clause and their previous password if `PASSWORD_VERIFY_FUNCTION` isn't `NULL` in their profile. When a non-superuser uses the `REPLACE` clause, the server compares the password provided to the existing password and raises an error if the passwords don't match.

A database superuser can use this command to change the password associated with any role. If a superuser includes the `REPLACE` clause, the clause is ignored. A non-matching value for the previous password doesn't throw an error.
If the role whose password is being changed has the `SUPERUSER` attribute, then a superuser must issue this command. A role with the `CREATEROLE` attribute can use this command to change the password associated with a role that isn't a superuser.

Parameters

- **role_name**: The name of the role whose password to alter.
- **password**: The role's new password.
- **prev_password**: The role's previous password.

Examples

This example changes a role's password:

```sql
ALTER ROLE: Managing database link and DBMS_RLS privileges

EDB Postgres Advanced Server includes extra syntax not offered by Oracle for the `ALTER ROLE` command. This syntax can be useful when assigning privileges related to creating and dropping database links compatible with Oracle databases and fine-grained access control using `DBMS_RLS`.

CREATE DATABASE LINK

A user who holds the `CREATE DATABASE LINK` privilege can create a private database link. The following `ALTER ROLE` command grants privileges to an EDB Postgres Advanced Server role that allow the specified role to create a private database link:

```sql
ALTER ROLE role_name
WITH [CREATEDBLINK | CREATE DATABASE LINK]
```

This command is the functional equivalent of:

```sql
GRANT CREATE DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```sql
ALTER ROLE role_name
WITH [NOCREATEDBLINK | NO CREATE DATABASE LINK]
```

Note

The `CREATEDBLINK` and `NOCREATEDBLINK` keywords are deprecated syntaxes. We recommend using the `CREATE DATABASE LINK` and `NO CREATE DATABASE LINK` syntax options.

CREATE PUBLIC DATABASE LINK

A user who holds the `CREATE PUBLIC DATABASE LINK` privilege can create a public database link. The following `ALTER ROLE` command grants privileges to an EDB Postgres Advanced Server role that allow the specified role to create a public database link:

```sql
ALTER ROLE role_name
```
This command is the functional equivalent of:

```sql
GRANT CREATE PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```sql
ALTER ROLE role_name
  WITH [NOCREATEPUBLICDBLINK | NO CREATE PUBLIC DATABASE LINK]
```

**Note**

The `CREATEPUBLICDBLINK` and `NOCREATEPUBLICDBLINK` keywords are deprecated syntaxes. We recommend using the `CREATE PUBLIC DATABASE LINK` and `NO CREATE PUBLIC DATABASE LINK` syntax options.

---

### Drop Public Database Link

A user who holds the `DROP PUBLIC DATABASE LINK` privilege can drop a public database link. The following `ALTER ROLE` command grants privileges to an EDB Postgres Advanced Server role that allow the specified role to drop a public database link:

```sql
ALTER ROLE role_name
  WITH [DROPUBLICDBLINK | DROP PUBLIC DATABASE LINK]
```

This command is the functional equivalent of:

```sql
GRANT DROP PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```sql
ALTER ROLE role_name
  WITH [NODROPPUBLICDBLINK | NO DROP PUBLIC DATABASE LINK]
```

**Note**

The `DROPPUBLICDBLINK` and `NODROPPUBLICDBLINK` keywords are deprecated syntaxes. We recommend using the `DROP PUBLIC DATABASE LINK` and `NO DROP PUBLIC DATABASE LINK` syntax options.

---

### Exempt Access Policy

A user who holds the `EXEMPT ACCESS POLICY` privilege is exempt from fine-grained access control (DBMS_RLS) policies. A user who holds these privileges can view or modify any row in a table constrained by a DBMS_RLS policy. The following `ALTER ROLE` command grants privileges to an EDB Postgres Advanced Server role that exempt the specified role from any defined DBMS_RLS policies:

```sql
ALTER ROLE role_name
  WITH [POLICYEXEMPT | EXEMPT ACCESS POLICY]
```

This command is the functional equivalent of:

```sql
GRANT EXEMPT ACCESS POLICY TO role_name
```

Use the following command to revoke the privilege:

```sql
ALTER ROLE role_name
  WITH [NOPOLICYEXEMPT | NO EXEMPT ACCESS POLICY]
```

**Note**

The `POLICYEXEMPT` and `NOPOLICYEXEMPT` keywords are deprecated syntaxes. We recommend using the `EXEMPT ACCESS POLICY` and `NO EXEMPT ACCESS POLICY` syntax options.

---

See also
14.3.5.10 ALTER SEQUENCE

Name

ALTER SEQUENCE — Change the definition of a sequence generator.

Synopsis

ALTER SEQUENCE <name> [ INCREMENT BY <increment> ]
[ MINVALUE <minvalue> ] [ MAXVALUE <maxvalue> ]
[ CACHE <cache> | NOCACHE ] [ CYCLE ]

Description

ALTER SEQUENCE changes the parameters of an existing sequence generator. Any parameter not specifically set in the ALTER SEQUENCE command retains its prior setting.

Parameters

name

The name (optionally schema-qualified) of a sequence to alter.

increment

The clause INCREMENT BY increment is optional. A positive value makes an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value is maintained.

minvalue

The optional clause MINVALUE minvalue determines the minimum value a sequence can generate. If not specified, the current minimum value is maintained. You can use the keywords NO MINVALUE to set this behavior back to the defaults of 1 and -2^63-1 for ascending and descending sequences, respectively. However, this term isn’t compatible with Oracle databases.

maxvalue

The optional clause MAXVALUE maxvalue determines the maximum value for the sequence. If not specified, the current maximum value is maintained. You can use the keywords NO MAXVALUE to set this behavior back to the defaults of 2^63-1 and -1 for ascending and descending sequences, respectively. However, this term isn’t compatible with Oracle databases.

cache

The optional clause CACHE cache specifies how many sequence numbers to preallocate and store in memory for faster access. The minimum value is 1. Only one value can be generated at a time, i.e., NOCACHE. If unspecified, the old cache value is maintained.

CYCLE

The CYCLE option allows the sequence to wrap around when the maxvalue or minvalue is reached by an ascending or descending sequence. If the limit is reached, the next number generated is the minvalue or maxvalue. If not specified, the old cycle behavior is maintained. You can use the keywords NO CYCLE to alter the sequence so that it doesn’t recycle. However, this term isn’t compatible with Oracle databases.

Notes

To avoid blocking concurrent transactions that obtain numbers from the same sequence, ALTER SEQUENCE is never rolled back. The changes take effect immediately and aren’t reversible.

ALTER SEQUENCE doesn’t immediately affect NEXTVAL results in backends, other than the current one, that have preallocated (cached) sequence values. They use up all cached values prior to noticing the changed sequence parameters. The current backend is affected immediately.
Examples

Change the increment and cache values of the sequence `serial`:

```
ALTER SEQUENCE serial INCREMENT BY 2 CACHE 5;
```

See also

`CREATE SEQUENCE`, `DROP SEQUENCE`

14.3.11 ALTER SESSION

Name

`ALTER SESSION` – Change a runtime parameter.

Synopsis

```
ALTER SESSION SET <name> = <value>
```

Description

The `ALTER SESSION` command changes runtime configuration parameters. `ALTER SESSION` affects only the value used by the current session. Some of these parameters are provided solely for compatibility with Oracle syntax and have no effect on the runtime behavior of EDB Postgres Advanced Server. Others alter a corresponding EDB Postgres Advanced Server database server runtime configuration parameter.

Parameters

- **name**
  
  Name of a settable runtime parameter.
  
- **value**
  
  New value of parameter.

Configuration parameters

You can modify the following configuration parameters using the `ALTER SESSION` command:

- **NLS_DATE_FORMAT (string)**
  
  Sets the display format for date and time values as well as the rules for interpreting ambiguous date input values. Has the same effect as setting the EDB Postgres Advanced Server `datestyle` runtime configuration parameter.

- **NLS_LANGUAGE (string)**
  
  Sets the message-display language. Has the same effect as setting the EDB Postgres Advanced Server `lc_messages` runtime configuration parameter.

- **NLS_LENGTH_SEMANTICS (string)**
  
  Valid values are `BYTE` and `CHAR`. The default is `BYTE`. This parameter is provided only for syntax compatibility and has no effect in the EDB Postgres Advanced Server.
OPTIMIZER_MODE (string)

Sets the default optimization mode for queries. Valid values are ALL_ROWS, CHOOSE, FIRST_ROWS, FIRST_ROWS_10, FIRST_ROWS_100, and FIRST_ROWS_1000. The default is CHOOSE. This parameter is implemented in EDB Postgres Advanced Server.

QUERY_REWRITE_ENABLED (string)

Valid values are TRUE, FALSE, and FORCE. The default is FALSE. This parameter is provided only for syntax compatibility and has no effect in EDB Postgres Advanced Server.

QUERY_REWRITE_INTEGRITY (string)

Valid values are ENFORCED, TRUSTED, and STALE_TOLERATED. The default is ENFORCED. This parameter is provided only for syntax compatibility and has no effect in EDB Postgres Advanced Server.

Examples

Set the language to U.S. English in UTF-8 encoding. In this example, the value en_US.UTF-8 is in the format for EDB Postgres Advanced Server. This form isn’t compatible with Oracle databases.

```
ALTER SESSION SET NLS_LANGUAGE = 'en_US.UTF-8';
```

Set the date display format:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'dd/mm/yyyy';
```

14.3.5.12 ALTER SYNONYM

Name

ALTER SYNONYM — Change ownership of a synonym object.

Synopsis

```
ALTER SYNONYM <syn_name> OWNER TO <new_owner>
```

Description

The ALTER SYNONYM command changes the owner of the synonym.

Examples

```
-- create user t1 and t2
edb=# create user t1;
CREATE ROLE
edb=# create user t2;
CREATE ROLE

-- grant all the privileges to user t1 and t2 on public schema
edb=# GRANT ALL PRIVILEGES ON SCHEMA public TO GROUP t1;
GRANT
edb=# GRANT ALL PRIVILEGES ON SCHEMA public TO GROUP t2;
GRANT

-- connect to the database as user t1 and create the table
edb=# \c - t1
You are now connected to database "edb" as user "t1".
edb=> create table t(n int);
CREATE TABLE
```
CREATE SYNONYM

-- check the owner of the synonym x is t1
edb=# select * from all_synonyms;

owner | schema_name | objid | synonym_name | table_owner | table_schema_name | table_name | db_link
-------+-------------+-------+--------------+-------------+-------------------+------------+---------
T1    | PUBLIC      | 16390 | X            | T1          | PUBLIC            | T          |         |

-- connect the database as user edb and change the owner of the synonym x to user t2
edb=> \c - edb
You are now connected to database "edb" as user "edb".
edb=# alter synonym x owner to t2;

-- check the owner of the synonym x is changed to t2
edb=# select * from all_synonyms;
owner | schema_name | objid | synonym_name | table_owner | table_schema_name | table_name | db_link
-------+-------------+-------+--------------+-------------+-------------------+------------+---------
T2    | PUBLIC      | 16390 | X            | T1          | PUBLIC            | T          |         |

14.3.5.13 ALTER TABLE

Name

ALTER TABLE -- Change the definition of a table.

Synopsis

ALTER TABLE <name>
    action [, ...
ALTER TABLE <name>
    RENAME COLUMN <column> TO <new_column>
ALTER TABLE <name>
    RENAME TO <new_name>
ALTER TABLE <name>
    { NOPARALLEL | PARALLEL [ <integer> ] }

action is one of:

ADD <column type> [ <column_constraint> [ ... ]
DROP COLUMN <column>
ADD <table_constraint>
DROP CONSTRAINT <constraint_name> [ CASCADE ]

Description

ALTER TABLE changes the definition of an existing table. There are several subforms:

- **ADD column type**
  This form adds a column to the table using the same syntax as CREATE TABLE.

- **DROP COLUMN**
  This form drops a column from a table. Indexes and table constraints involving the column are dropped as well.

- **ADD table_constraint**
  This form adds a constraint to a table. For details, see CREATE TABLE.
This form drops constraints on a table. Currently, constraints on tables don't need unique names, so there might be more than one constraint matching the specified name. All matching constraints are dropped.

**RENAME**

The **RENAME** forms change the name of a table (or an index, sequence, or view) or the name of a column in a table. There is no effect on the stored data.

The **PARALLEL** clause sets the degree of parallelism for a table. The **NOPARALLEL** clause resets the values to their defaults. **reoptions** shows the **parallel_workers** parameter as 0.

A superuser has permission to create a trigger on any user's table, but a user can create a trigger only on the table they own. However, when the ownership of a table is changed, the ownership of the trigger's implicit objects is updated when they're matched with a table owner owning a trigger.

You can use the **ALTER TRIGGER ...ON AUTHORIZATION** command to alter a trigger's implicit object owner. For information, see **ALTER TRIGGER**.

You must own the table to use **ALTER TABLE**.

**Parameters**

**name**

The name (possibly schema-qualified) of an existing table to alter.

**column**

Name of a new or existing column.

**new_column**

New name for an existing column.

**new_name**

New name for the table.

**type**

Data type of the new column.

**table_constraint**

New table constraint for the table.

**constraint_name**

Name of an existing constraint to drop.

**CASCADE**

Automatically drop objects that depend on the dropped constraint.

**PARALLEL**

Specify **PARALLEL** to select a degree of parallelism. You can also specify the degree of parallelism by setting the **parallel_workers** parameter when performing a parallel scan on a table. If you specify **PARALLEL** without including a degree of parallelism, the index uses default parallelism.

**NOPARALLEL**

Specify **NOPARALLEL** to reset parallelism to default values.

**integer**

The **integer** indicates the degree of parallelism, which is the number of **parallel_workers** used in the parallel operation to perform a parallel scan on a table.

**Notes**
When you invoke `ADD COLUMN`, all existing rows in the table are initialized with the column’s default value (null if no `DEFAULT` clause is specified). Adding a column with a non-null default requires rewriting the entire table. This can take a long time for a large table, and it temporarily requires double the disk space. Adding a `CHECK` or `NOT NULL` constraint requires scanning the table to verify that existing rows meet the constraint.

The `DROP COLUMN` form doesn't physically remove the column but makes it invisible to SQL operations. Subsequent insert and update operations in the table store a null value for the column. Thus, dropping a column is quick, but it doesn't immediately reduce the on-disk size of your table since the space occupied by the dropped column isn't reclaimed. The space is reclaimed over time as existing rows are updated.

Changing any part of a system catalog table isn't permitted. Refer to `CREATE TABLE` for a further description of valid parameters.

### Examples

To add a column of type `VARCHAR2` to a table:

```
ALTER TABLE emp ADD address VARCHAR2(30);
```

to drop a column from a table:

```
ALTER TABLE emp DROP COLUMN address;
```

to rename an existing column:

```
ALTER TABLE emp RENAME COLUMN address TO city;
```

to rename an existing table:

```
ALTER TABLE emp RENAME TO employee;
```

to add a check constraint to a table:

```
ALTER TABLE emp ADD CONSTRAINT sal_chk CHECK (sal > 500);
```

to remove a check constraint from a table:

```
ALTER TABLE emp DROP CONSTRAINT sal_chk;
```

to reset the degree of parallelism to 0 on the `emp` table:

```
ALTER TABLE emp NOPARALLEL;
```

This example creates a table named `dept` and then alters the `dept` table to define and enable a unique key on the `dname` column. The constraint `dept_dname_uq` identifies the `dname` column as a unique key. The `USING_INDEX` clause creates an index on a table `dept` with the index statement specified to enable the unique constraint.

```
CREATE TABLE dept
(
    deptno NUMBER(2),
    dname VARCHAR2(14),
    loc VARCHAR2(13)
);

ALTER TABLE dept
   ADD CONSTRAINT dept_dname_uq UNIQUE(dname)
   USING INDEX (CREATE UNIQUE INDEX idx_dept_dname_uq ON dept (dname));
```

This example creates a table named `emp` and then alters the `emp` table to define and enable a primary key on the `ename` column. The `emp_ename_pk` constraint identifies the `ename` column as a primary key of the `emp` table. The `USING_INDEX` clause creates an index on a table `emp` with the index statement specified to enable the primary constraint.

```
CREATE TABLE emp
(
    empno NUMBER(4) NOT NULL,
    ename VARCHAR2(18),
    job VARCHAR2(9),
    sal NUMBER(7,2),
```

```
);
ALTER TABLE emp
  ADD CONSTRAINT emp_ename_pk PRIMARY KEY (ename)
  USING INDEX (CREATE INDEX idx_emp_ename_pk ON emp (ename));

See also
CREATE TABLE, DROP TABLE

### 14.3.5.14 ALTER TRIGGER

#### Name

ALTER TRIGGER — Change the definition of a trigger.

#### Synopsis

EDB Postgres Advanced Server supports three variations of the `ALTER TRIGGER` command. Use the first variation to change the name of a given trigger without changing the trigger definition:

```sql
ALTER TRIGGER <name> ON <table_name> RENAME TO <new_name>
```

Use the second variation of the `ALTER TRIGGER` command if the trigger depends on an extension. If the extension is dropped, the trigger is dropped as well:

```sql
ALTER TRIGGER <name> ON <table_name> DEPENDS ON EXTENSION <extension_name>
```

Use the third variation of the `ALTER TRIGGER` command to change the ownership of a trigger's object:

```sql
ALTER TRIGGER <name> ON <table_name> AUTHORIZATION <rolespec>
```

For information about using non-compatible implementations of the `ALTER TRIGGER` command that are supported by EDB Postgres Advanced Server, see the PostgreSQL core documentation.

#### Description

`ALTER TRIGGER` changes the properties of an existing trigger. You must own the table the trigger acts on to change its properties.

To alter an owner of the trigger's implicit object, you can use the `ALTER TRIGGER ...ON AUTHORIZATION` command. You must have the privilege to execute `ALTER TRIGGER ...ON AUTHORIZATION` command to assign the trigger's implicit object ownership to a user after authorization.

#### Parameters

- **name**

  The name of the trigger to alter.

- **table_name**

  The name of a table on which trigger acts.

- **rolespec**

  Determines an owner of trigger objects.
Examples

This example includes the users bob and carol as superusers. The user bob owns a table emp. The user carol owns a trigger named emp_sal_trig, which is created on table emp:

```
SELECT relname, relowner::regrole FROM pg_class WHERE relname = 'emp';

<table>
<thead>
<tr>
<th>relname</th>
<th>relowner</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp</td>
<td>bob</td>
</tr>
</tbody>
</table>
(1 row)
```

```
SELECT proname, proowner::regrole FROM pg_proc WHERE oid = (SELECT tgfoid FROM pg_trigger WHERE tgname = 'emp_sal_trig') ORDER BY oid;

<table>
<thead>
<tr>
<th>proname</th>
<th>proowner</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp_sal_trig_emp</td>
<td>carol</td>
</tr>
</tbody>
</table>
(1 row)
```

To alter the ownership of table emp from user bob to a new owner edb:

```
ALTER TABLE emp OWNER TO edb;
```

```
ALTER TABLE

SELECT relname, relowner::regrole FROM pg_class WHERE relname = 'emp';

<table>
<thead>
<tr>
<th>relname</th>
<th>relowner</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp</td>
<td>edb</td>
</tr>
</tbody>
</table>
(1 row)
```

The table ownership is changed from the user bob to an owner edb, but the trigger ownership of emp_sal_trig isn't altered and is owned by user carol. Alter the trigger emp_sal_trig on table emp, and grant authorization to an owner edb:

```
ALTER TRIGGER emp_sal_trig ON emp AUTHORIZATION edb;
```

```
ALTER TRIGGER

SELECT proname, proowner::regrole FROM pg_proc WHERE oid = (SELECT tgfoid FROM pg_trigger WHERE tgname = 'emp_sal_trig') ORDER BY oid;

<table>
<thead>
<tr>
<th>proname</th>
<th>proowner</th>
</tr>
</thead>
<tbody>
<tr>
<td>emp_sal_trig_emp</td>
<td>edb</td>
</tr>
</tbody>
</table>
(1 row)
```

The trigger ownership emp_sal_trig on table emp is altered and granted to an owner edb.

See also

CREATE TRIGGER, DROP TRIGGER

### 14.3.5.15 ALTER TABLESPACE

**Name**

ALTER TABLESPACE — Change the definition of a tablespace.

**Synopsis**
ALTER TABLESPACE <name> RENAME TO <newname>

Description

**ALTER TABLESPACE** changes the definition of a tablespace.

Parameters

**name**

The name of an existing tablespace.

**newname**

The new name of the tablespace. The new name can't begin with `pg_`. These names are reserved for system tablespaces.

Examples

Rename tablespace `empspace` to `employee_space`:

```
ALTER TABLESPACE empspace RENAME TO employee_space;
```

See also

DROP TABLESPACE

14.3.5.16 ALTER USER... IDENTIFIED BY

Name

**ALTER USER** — Change a database user account.

Synopsis

```
ALTER USER <role_name> IDENTIFIED BY <password> REPLACE <prev_password>
```

Description

A role without the **CREATEROLE** privilege can use this command to change their own password. An unprivileged role must include the **REPLACE** clause and their previous password if **PASSWORD_VERIFY_FUNCTION** isn't **NULL** in their profile. When a non-superuser uses the **REPLACE** clause, the server compares the password provided to the existing password and raises an error if the passwords don't match.

A database superuser can use this command to change the password associated with any role. If a superuser includes the **REPLACE** clause, the clause is ignored. A non-matching value for the previous password doesn't throw an error.

If the role for which the password is being changed has the **SUPERUSER** attribute, then a superuser must issue this command. A role with the **CREATEROLE** attribute can use this command to change the password associated with a role that isn't a superuser.

Parameters
role_name
The name of the role whose password to alter.

password
The role's new password.

prev_password
The role's previous password.

Examples
Change a user password:

```
ALTER USER john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;
```

See also
CREATE USER, DROP USER

### 14.3.5.17 ALTER USER|ROLE... PROFILE MANAGEMENT CLAUSES

**Name**

```
ALTER USER|ROLE
```

**Synopsis**

```
ALTER USER|ROLE <name> [(WITH] option[...]
```

`option` can be the following compatible clauses:

```
PROFILE <profile_name>
| ACCOUNT
| LOCK|UNLOCK
| PASSWORD EXPIRE [AT '<timestamp>']
```

`option` can be the following non-compatible clauses:

```
| PASSWORD SET AT '<timestamp>'
| LOCK TIME '<timestamp>'
| STORE PRIOR PASSWORD {'<password>' '<timestamp>}' [,...]
```

For information about the administrative clauses of the `ALTER USER` or `ALTER ROLE` command that are supported by EDB Postgres Advanced Server, see the PostgreSQL core documentation.

Only a database superuser can use the `ALTER USER|ROLE` clauses that enforce profile management. The clauses enforce the following behaviors:

- Include the `PROFILE` clause and a `profile_name` to associate a predefined profile with a role or to change the predefined profile associated with a user.

- Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to place the user account in a locked or unlocked state.

- Include the `LOCK TIME '<timestamp>'` clause and a date/time value to lock the role at the specified time and unlock the role at the time indicated by the `PASSWORD_LOCK_TIME` parameter of the profile assigned to this role. If `LOCK TIME` is used with the `ACCOUNT LOCK` clause, only a database superuser can unlock the role with the `ACCOUNT UNLOCK` clause.

- Include the `PASSWORD EXPIRE` clause with the `AT '<timestamp>'` keywords to specify a date/time for the password associated with the role to expire. If you omit the `AT '<timestamp>'` keywords, the password expires immediately.
Include the `PASSWORD SET AT 'timestamp'` keywords to set the password modification date to the time specified.

Include the `STORE PRIOR PASSWORD ['password' 'timestamp'] [, ...]` clause to modify the password history, adding the new password and the time the password was set.

Each login role can have only one profile. To discover the profile that's currently associated with a login role, query the `profile` column of the `DBA_USERS` view.

**Parameters**

**name**

The name of the role to associate with the specified profile.

**password**

The password associated with the role.

**profile_name**

The name of the profile to associate with the role.

**timestamp**

The date and time when the clause is enforced. When specifying a value for `timestamp`, enclose the value in single quotes.

**Notes**

For information about the Postgres-compatible clauses of the `ALTER USER` or `ALTER ROLE` command, see the PostgreSQL core documentation.

**Examples**

This example uses the `ALTER USER... PROFILE` command to associate a profile named `acctg` with a user named `john`:

```
ALTER USER john PROFILE acctg_profile;
```

This example uses the `ALTER ROLE... PROFILE` command to associate a profile named `acctg` with a user named `john`:

```
ALTER ROLE john PROFILE acctg_profile;
```

**See also**

CREATE USER|ROLE... PROFILE MANAGEMENT CLAUSES

---

**14.3.5.18 CALL**

**Name**

**CALL** — Invoke a procedure.

**Synopsis**

```
CALL <procedure_name> '([<argument_list>])'
```
Use the `CALL` statement to invoke a procedure. To use the `CALL` statement, you must have `EXECUTE` privileges on the procedure that the `CALL` statement invokes.

### Parameters

- **procedure_name**
  
  The (optionally schema-qualified) procedure name.

- **argument_list**
  
  Specifies a comma-separated list of arguments required by the procedure. Each member of `argument_list` corresponds to a formal argument expected by the procedure. Each formal argument can be an `IN` parameter, an `OUT` parameter, or an `INOUT` parameter.

**** Note You must specify an `OUT` parameter in the `CALL` statement when calling a package function. The `OUT` parameter acts as an `INOUT` parameter during package overloading.

### Examples

The `CALL` statement can take several forms, depending on the arguments required by the procedure:

```sql
CALL update_balance();
CALL update_balance(1, 2, 3);
```

### 14.3.5.19 COMMENT

#### Name

**COMMENT** — Define or change the comment of an object.

#### Synopsis

```sql
COMMENT ON

{ TABLE <table_name>
 | COLUMN <table_name.column_name>
}

IS '<text>'
```

#### Description

**COMMENT** stores a comment about a database object. To modify a comment, issue a new `COMMENT` command for the same object. Only one comment string is stored for each object. To remove a comment, specify an empty string (two consecutive single quotes with no intervening space) for `text`. Comments are dropped when the object is dropped.

#### Parameters

- **table_name**
  
  The name of the table (optionally schema-qualified) to comment.

- **table_name.column_name**
  
  The name of a column (optionally schema-qualified) in `table_name` to comment.
The new comment.

Notes

There is currently no security mechanism for comments. Any user connected to a database can see all the comments for objects in that database, although only superusers can change comments for objects that they don't own. Don't put security-critical information in a comment.

Examples

Attach a comment to the table `emp`:

```sql
COMMENT ON TABLE emp IS 'Current employee information';
```

Attach a comment to the `empno` column of the `emp` table:

```sql
COMMENT ON COLUMN emp.empno IS 'Employee identification number';
```

Remove these comments:

```sql
COMMENT ON TABLE emp IS ' ';
COMMENT ON COLUMN emp.empno IS ' ';
```

14.3.20 COMMIT

Name

COMMIT — Commit the current transaction.

Synopsis

```sql
COMMIT [ WORK ]
```

Description

COMMIT commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters

WORK

Optional keyword that has no effect.

Notes

Use ROLLBACK to abort a transaction. Issuing COMMIT when not inside a transaction does no harm.
Executing a `COMMIT` in a plpgsql procedure throws an error if there's an Oracle-style SPL procedure on the runtime stack.

Examples

To commit the current transaction and make all changes permanent:

```sql
COMMIT;
```

See also

`ROLLBACK`, `ROLLBACK TO SAVEPOINT`

14.3.5.21  CREATE CAPTURE PRIVILEGES POLICY

EDB Postgres Advanced Server includes extra syntax not offered by Oracle with the `CREATE_CAPTURE_PRIVILEGES POLICY` SQL command.

Name

`CREATE_CAPTURE_PRIVILEGES POLICY` — Create a new capture privilege policy.

Synopsis

Use `CREATE_CAPTURE_PRIVILEGES POLICY` to define a new policy to capture privileges:

```sql
## Create a policy on
database
CREATE_CAPTURE_PRIVILEGES POLICY <name> [ WHERE ( <expression> ) ];
## Create a policy on a
ROLE/USER
CREATE_CAPTURE_PRIVILEGES POLICY <name> ON { ROLE | USER } <role_names> [ WHERE ( <expression> ) ];
```

Description

The `CREATE_CAPTURE_PRIVILEGES POLICY` command allows a database superuser or any user with the system-defined `CAPTURE_ADMIN` privilege to create a policy in the database.

If the name of the policy is schema-qualified, the policy is created in the specified schema. If the `CREATE_CAPTURE_PRIVILEGES POLICY` command doesn't include a schema, the policy is created in the current schema.

Use `DROP_CAPTURE_PRIVILEGES POLICY` to remove a capture policy.

Parameters

`name`

The name optionally schema-qualified of the capture privilege policy to create.

`expression`

Boolean expression containing up to 4000 characters.

`role_names`

Role names containing up to 4000 characters.
The roles whose privileges are to be analyzed.

Examples

This command creates a capture privilege policy named `sampledb_p1` on the database:

```
CREATE CAPTURE PRIVILEGES POLICY sampledb_p1;
```

This command creates a capture privilege policy named `sampledb_p2` on role `sampledb_role1` and `sampledb_user1`:

```
CREATE CAPTURE PRIVILEGES POLICY sampledb_p2 ON ROLE sampledb_role1,sampledb_user1;
```

See also

`ALTER CAPTURE PRIVILEGE POLICY`, `DROP CAPTURE PRIVILEGE POLICY`

14.3.5.22 CREATE DATABASE

Name

`CREATE DATABASE` — Create a database.

Synopsis

```
CREATE DATABASE <name>
```

Description

`CREATE DATABASE` creates a database.

To create a database, you must be a superuser or have the `CREATEDB` privilege. Normally, the creator becomes the owner of the new database. Non-superusers with the `CREATEDB` privilege can create only databases they own.

The new database is a clone of the standard system database `template1`.

Parameters

- `<name>`
  
The name of the database to create.

Notes

You can't execute `CREATE DATABASE` inside a transaction block.

Errors along the line of "could not initialize database directory" are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems.

Examples
To create a database:

```sql
CREATE DATABASE employees;
```

### 14.3.5.23 CREATE PUBLIC DATABASE LINK

#### Name

CREATE \[PUBLIC\] DATABASE LINK — Create a database link.

#### Synopsis

```sql
CREATE \{ PUBLIC \} DATABASE LINK <name>
\{ CONNECT TO \{ CURRENT_USER | \{ \{ <username> \} IDENTIFIED BY '<password>' \} |\{ postgres_fdw '<fdw_connection_string>' \} | \{ oci '<oracle_connection_string>' \} \} \{ USING \{ postgres_fdw '<fdw_connection_string>' | \{ oci '<oracle_connection_string>' \} \} \}"
```

#### Description

**CREATE DATABASE LINK** creates a database link. A database link is an object that allows a reference to a table or view in a remote database in a DELETE, INSERT, SELECT or UPDATE command. Reference a database link by appending `@dblink` to the table or view name referenced in the SQL command, where `dblink` is the name of the database link.

Database links can be public or private. A public database link is one that any user can use. Only the database link's owner can use a private database link. Specify the `PUBLIC` option to create a public database link. Otherwise, a private database link is created.

When you use the `CREATE DATABASE LINK` command, the database link name and the given connection attributes are stored in the EDB Postgres Advanced Server system table named `pg_catalog.edb_dblink`. When using a given database link, the database containing the `edb_dblink` entry defining this database link is called the _local database_. The server and database whose connection attributes are defined in the `edb_dblink` entry is called the _remote database_. You can use `edb_dblink OCI` to access remote Oracle tables and views using any `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement.

You must be connected to the local database when you issue a SQL command containing a reference to a database link. When the SQL command executes, the appropriate authentication and connection is made to the remote database to access the table or view to which the `@dblink` reference is appended.

**Note**

You can't use a database link to access a remote database in a standby database server. Standby database servers are for high availability, load balancing, and replication.

For information about high availability, load balancing, and replication for Postgres database servers, see the _PostgreSQL core documentation_.

**Note**

- For EDB Postgres Advanced Server 15, the **CREATE DATABASE LINK** command is tested against and certified for use with Oracle version 10g Release 2 (10.2), Oracle version 11g Release 2 (11.2), Oracle version 12c Release 1 (12.1), and Oracle version 18c Release 1 (18.2).
- You can set the `edb_dblink_oci.rescans` GUC to `SCROLL` or `SERIALIZABLE` at the server level in the `postgresql.conf` file. You can also set it at the session level using the `SET` command. However, the setting isn't applied to existing `dblink` connections due to `dblink` connection caching.
- When executing `SELECT` on LOB data of more than 4000 characters, we recommend using `edb_dblink_oci.rescans=serializable` to free up the temporary PGA memory and avoid exceeding `PGA_AGGREGATE_LIMIT`.

The `edb_dblink_oci` supports both types of rescans: `SCROLL` and `SERIALIZABLE`. By default it's set to `SERIALIZABLE`. When set to `SERIALIZABLE`, `edb_dblink_oci` uses the `SERIALIZABLE` transaction isolation level on the Oracle side, which corresponds to PostgreSQL's `REPEATABLE READ`. This is necessary because a single PostgreSQL statement can lead to multiple Oracle queries. It thereby uses a serializable isolation level to provide consistent results.

A serialization failure can occur when modifying a table concurrent with long-running DML transactions, for example, `ADD`, `UPDATE`, or `DELETE` statements. If such a failure occurs, the OCI reports `ORA-08177: can't serialize access for this transaction`, and the application must retry the transaction.

A `SCROLL` rescans is quick, but each iteration resets the current row position to 1. A `SERIALIZABLE` rescans has performance benefits over a `SCROLL` rescans.

#### Parameters
PUBLIC

Create a public database link that any user can use. If you omit this parameter, then the database link is private and only the database link’s owner can use it.

name

The name of the database link.

username

The username to use for connecting to the remote database.

CURRENT_USER

Include CURRENT_USER to use the user mapping associated with the role that’s using the link when establishing a connection to the remote server.

password

The password for username.

postgres_fdw

Specifies foreign data wrapper postgres_fdw as the connection to a remote EDB Postgres Advanced Server database. If postgres_fdw isn't installed on the database, use the CREATE EXTENSION command to install postgres_fdw. For more information, see the CREATE EXTENSION command in the PostgreSQL core documentation.

fdw_connection_string

Specifies the connection information for the postgres_fdw foreign data wrapper.

oci

Specifies a connection to a remote Oracle database. This is the default behavior.

oracle_connection_string

Specifies the connection information for an oci connection.

Note

To create a non-public database link, you need the CREATE DATABASE LINK privilege. To create a public database link, you need the CREATE PUBLIC DATABASE LINK privilege.

Setting up an Oracle instant client for OCI database link

To use edb_dblinkoci, you must download and install an Oracle instant client on the host running the EDB Postgres Advanced Server database in which you want to create the database link.

You can download an instant client here.

Oracle instant client for Linux

These instructions apply to Linux hosts running EDB Postgres Advanced Server.

Be sure the libaio library (the Linux-native asynchronous I/O facility) is installed on the Linux host running EDB Postgres Advanced Server.

You can install the libaio library with the following command:

```
yum install libaio
```

If the Oracle instant client that you downloaded doesn't include the file named libcintlsh.so without a version number suffix, create a symbolic link named libcintlsh.so that points to the downloaded version of the library file. To do so, navigate to the instant client directory and execute the following command:

```
ln -s libcintlsh.so.<version> libcintlsh.so
```

Where version is the version number of the libcintlsh.so library. For example:

```
ln -s libcintlsh.so.12.1 libcintlsh.so
```
When executing a SQL command that references a database link to a remote Oracle database, EDB Postgres Advanced Server must know where the Oracle instant client library resides on the EDB Postgres Advanced Server host.

The `LD_LIBRARY_PATH` environment variable must include the path to the Oracle client installation directory containing the `libclntsh.so` file. For example, if the installation directory containing `libclntsh.so` is `/tmp/instantclient`, use:

```
export LD_LIBRARY_PATH=/tmp/instantclient:$LD_LIBRARY_PATH
```

Alternatively, you can set the value of the `oracle_home` configuration parameter in the `postgresql.conf` file. The `oracle_home` configuration parameter is an alternative to the `LD_LIBRARY_PATH` environment variable. For more details on the `oracle_home` configuration parameter, see configuration parameters.

The `ORACLE_HOME` environment variable must be set and include the path to the Oracle home directory. For example,

```
export ORACLE_HOME=/opt/product/version/dbhomeXE
```

Note

You must set either the `LD_LIBRARY_PATH` or the `oracle_home` configuration parameter and the `ORACLE_HOME` environment variable before executing the `pg_ctl` utility to start or restart EDB Postgres Advanced Server.

If you're running the current session as the user account (for example, `enterprisedb`) that directly invokes `pg_ctl` to start or restart EDB Postgres Advanced Server, then set either the `LD_LIBRARY_PATH` environment variable or the `oracle_home` configuration parameter and the `ORACLE_HOME` environment variable before invoking `pg_ctl`.

You can set the `LD_LIBRARY_PATH` and the `ORACLE_HOME` environment variable in the `.bash_profile` file under the home directory of the `enterprisedb` user account. That is, set `LD_LIBRARY_PATH` and `ORACLE_HOME` in the file `~enterprisedb/.bash_profile`. This setting ensures that `LD_LIBRARY_PATH` and `ORACLE_HOME` are set when you log in as `enterprisedb`.

If you're using a Linux service script with the `systemctl` or `service` command to start or restart EDB Postgres Advanced Server, you must set `LD_LIBRARY_PATH` and `ORACLE_HOME` so it's in effect when the script invokes the `pg_ctl` utility.


If the `LD_LIBRARY_PATH` is `/tmp/instantclient`, include the environment variable by specifying:

```
[Service]
Environment=LD_LIBRARY_PATH=/tmp/instantclient:$LD_LIBRARY_PATH
Environment=ORACLE_HOME=/tmp/instantclient
```

Reload systemd:

```
systemctl daemon-reload
```

Restart the EDB Postgres Advanced Server service:

```
systemctl restart edb-as-14
```

The script file that you need to modify to include the `LD_LIBRARY_PATH` setting depends on the EDB Postgres Advanced Server version and the Linux system on which it was installed.

Oracle instant client for Windows

These instructions apply to Windows hosts running EDB Postgres Advanced Server.

When you're executing a SQL command that references a database link to a remote Oracle database, EDB Postgres Advanced Server must know where the Oracle instant client library resides on the EDB Postgres Advanced Server host.

Set the Windows `PATH` system environment variable to include the Oracle client installation directory that contains the `oci.dll` file.

Alternatively, you can set the value of the `oracle_home` configuration parameter in the `postgresql.conf` file. The value specified in the `oracle_home` configuration parameter overrides the Windows `PATH` environment variable.

To set the `oracle_home` configuration parameter in the `postgresql.conf` file, add the following line:

```
oracle_home = 'lib_directory'
```

Substitute the name of the Windows directory that contains `oci.dll` for `lib_directory`. For example:

```
oracle_home = 'C:/tmp/instantclient_10_2'
```
After setting the `PATH` environment variable or the `oracle_home` configuration parameter, you must restart the server for the changes to take effect. Restart the server from the Windows Services console.

**Note**

If `tnsnames.ora` is configured in failover mode, and a client:server failure occurs, the client connection is established with a secondary server (usually a backup server). Later, when the primary server resumes, the client retains its connection to a secondary server until a new session is established. The new client connections is automatically established with the primary server. If the primary and secondary servers are out of sync, then the clients that established a connection to the secondary server and the clients that later connected to the primary server might see a different database view.

**Examples**

Creating an OCI database link

This example uses the `CREATE DATABASE LINK` command to create a database link named `chicago` that connects an instance of EDB Postgres Advanced Server to an Oracle server using an `edb_dblink_oci` connection. The connection information tells EDB Postgres Advanced Server to log in to Oracle as the user `admin` whose password is `mypassword`. Including the `oci` option tells EDB Postgres Advanced Server that this is an `edb_dblink_oci` connection. The connection string `''//127.0.0.1/acctg''` specifies the server address and name of the database.

```
CREATE DATABASE LINK chicago
CONNECT TO admin IDENTIFIED BY 'mypassword'
USING oci
''//127.0.0.1/acctg'';
```

**Note**

You can specify a hostname in the connection string in place of an IP address.

Creating a postgres_fdw database link

This example uses the `CREATE DATABASE LINK` command to create a database link named `bedford`. The database link connects an instance of EDB Postgres Advanced Server to another EDB Postgres Advanced Server instance by way of a `postgres_fdw` foreign data wrapper connection. The connection information tells EDB Postgres Advanced Server to log in as the user `admin` with the password `mypassword`. Including the `postgres_fdw` option tells EDB Postgres Advanced Server that this is a `postgres_fdw` connection. The connection string, `''host=127.0.0.1 port=5444 dbname=marketing''` specifies the server address and name of the database.

```
CREATE DATABASE LINK bedford
CONNECT TO admin IDENTIFIED BY 'mypassword'
USING postgres_fdw
'host=127.0.0.1 port=5444 dbname=marketing';
```

**Note**

You can specify a hostname in the connection string in place of an IP address.

Using a database link

These examples use a database link with EDB Postgres Advanced Server to connect to an Oracle database. The examples assume that a copy of the EDB Postgres Advanced Server sample application's `emp` table was created in an Oracle database. A second EDB Postgres Advanced Server database cluster with the sample application is accepting connections at port `5443`.

Create a public database link, named `oralink`, to an Oracle database named `xe` located at `127.0.0.1` on port `1521`. Connect to the Oracle database with the username `edb` and password `password`.

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password'
USING '///127.0.0.1:1521/xe';
```

Issue a `SELECT` command on the `emp` table in the Oracle database using the database link `oralink`.

```
SELECT * FROM emp@oralink;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7399</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-88 00:00:00</td>
<td>808</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81 00:00:00</td>
<td>1600</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
<td>1250</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
<td>2975</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
<td>1250</td>
<td>1400</td>
<td></td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01-MAY-81 00:00:00</td>
<td>2858</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
<td>2450</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
<td>3800</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Create a private database link, named `fdwlink`, to the EDB Postgres Advanced Server database named `edb` located on host 192.168.2.22 running on port 5444. Connect to the EDB Postgres Advanced Server database with the username `enterprisedb` and password `password`.

```
CREATE DATABASE LINK fdwlink
CONNECT TO enterprisedb
IDENTIFIED BY 'password'
USING postgres_fdw
host=192.168.2.22 port=5444
dbname=edb;
```

Display attributes of database links `oralink` and `fdwlink` from the local `edb_dblink` system table:

```
SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;
```

```
lkname  | lnkuser   | lkconnstr
---------+-----------+----------------------------------------
oralink | edb       | //127.0.0.1:1521/xe
fdwlink | enterprisedb | //192.168.2.22:5444/edb
```

(2 rows)

Perform a join of the `emp` table from the Oracle database with the `dept` table from the EDB Postgres Advanced Server database:

```
SELECT d.deptno, d.dname, e.empno, e.ename, e.job, e.sal, e.comm
FROM emp@oralink e, dept@fdwlink d
WHERE e.deptno = d.deptno
ORDER BY 1, 3;
```

```
deptno | dname    | empno | ename  | job    | sal  | comm
--------+----------+-------+--------+--------+------|------
10      | ACCOUNTING | 7782  | CLARK  | MANAGER | 2450 |
10      | ACCOUNTING | 7839  | KING   | PRESIDENT | 5000 |
10      | ACCOUNTING | 7934  | MILLER | CLERK  | 1380 |
20      | RESEARCH   | 7566  | JONES  | MANAGER | 2975 |
20      | RESEARCH   | 7788  | SCOTT  | ANALYST | 3080 |
20      | RESEARCH   | 7876  | ADAMS  | CLERK  | 1180 |
20      | RESEARCH   | 7902  | FORD   | ANALYST | 3080 |
30      | SALES      | 7698  | BLAKE  | MANAGER | 2850 |
30      | SALES      | 7844  | TURNER | SALESMAN | 1580 |
30      | SALES      | 7900  | JAMES  | CLERK  | 950  |
```

(14 rows)

**Push down for an OCI database link**

When the OCI database link is used to execute SQL statements on a remote Oracle database, sometimes the pushdown of the processing occurs on the foreign server.

Push down refers to the occurrence of processing on the foreign that is, remote server instead of the local client where the SQL statement was issued. Push down can result in performance improvement since the data is processed on the remote server before being returned to the local client.

OCI database link supports push down for:

- WHERE clause
- JOIN operations
- AGGREGATE functions
- ORDER BY clause

For push down to occur, some basic conditions must be met. The tables involved in the join operation must belong to the same foreign server and use the identical connection information to the foreign server. This connection information is the same database link defined with the `CREATE DATABASE LINK` command.

To determine whether to use push down for a SQL statement, display the execution plan by using the `EXPLAIN` command. For information about the `EXPLAIN` command, see the PostgreSQL core documentation.

You can restrict the push downs using the `edb_dbLink_oci_pushdown.config` configuration file. You can define the list of functions and operators in this file that can push down to the remote server. You can easily add or modify the list as per the requirements.
This file lists the objects as aggregates, functions, and operators allowed to push down to the remote server. Put each entry on a single line. Each entry must have two columns:

- Object type that can be ROUTINE (functions, aggregates, and procedures) or OPERATOR.
- The second column is schema-qualified object names with their arguments.

You can format the second column using the following query:

**For ROUTINES:**

```sql
SELECT prnamespace::regnamespace || '.' || oid::regprocedure FROM pg_proc
WHERE prname = '<routine_name>'
```

**For OPERATORS:**

```sql
SELECT opnamespace::regnamespace || '.' || oid::regoperator FROM pg_operator
WHERE oprname = '<operator_name>'
```

Example of `edb_dbLink_oci_pushdown.config` file:

```
ROUTINE pg_catalog.sum(bigint)
ROUTINE pg_catalog.sum(smallint)
ROUTINE pg_catalog.to_number(text)
ROUTINE pg_catalog.to_number(text,text)
OPERATOR pg_catalog.=(integer,integer)
OPERATOR pg_catalog.=(text,text)
OPERATOR pg_catalog.=(smallint,integer)
OPERATOR pg_catalog.=(bigint,integer)
OPERATOR pg_catalog.=(numeric,numeric)
```

To find out whether pushdown is used for a SQL statement, display the execution plan by using the `EXPLAIN` command.

These examples use the following database link:

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password'
USING '//192.168.2.23:1521/xe';
```

This example shows the execution plan of an inner join:

```
EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename
FROM dept@oralink d, emp@oralink e WHERE d.deptno = e.deptno ORDER BY 1,
3;
```

```
QUERY PLAN
--------------------------------------------------------------------------------
Foreign Scan
   Output: d.deptno, d.dname, e.empno, e.ename
   Relations: (_dblink_dept_1 d) INNER JOIN (_dblink_emp_2 e)
   Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM dept r1 INNER
JOIN emp r2 ON ((r1.deptno = r2.deptno)) ORDER BY r1.deptno ASC NULLS LAST, r2.empno ASC NULLS
LAST
(4 rows)
```

The `INNER JOIN` operation occurs under the `Foreign Scan` section. The output of this join is the following:

```
deptno | dname   | empno | ename  
-------+---------+-------+--------
 10    | ACCOUNTING | 7782 | CLARK  
 10    | ACCOUNTING | 7839 | KING   
 10    | ACCOUNTING | 7934 | MILLER 
 20    | RESEARCH  | 7369 | SMITH  
 20    | RESEARCH  | 7566 | JONES  
 20    | RESEARCH  | 7788 | SCOTT  
 20    | RESEARCH  | 7876 | ADAMS  
 20    | RESEARCH  | 7902 | FORD   
 30    | SALES     | 7499 | ALLEN  
 30    | SALES     | 7521 | WARD   
 30    | SALES     | 7654 | MARTIN 
 30    | SALES     | 7698 | BLAKE  
 30    | SALES     | 7844 | TURNER 
 30    | SALES     | 7900 | JAMES  
(14 rows)
```

This example shows the execution plan of an left outer join:

```
EXPLAIN (verbose,costs off) SELECT d.deptno, d.dname, e.empno, e.ename
FROM dept@oralink d, emp@oralink e WHERE d.deptno = e.deptno ORDER BY 1,
3;
```

```
QUERY PLAN
--------------------------------------------------------------------------------
Foreign Scan
   Output: d.deptno, d.dname, e.empno, e.ename
   Relations: (_dblink_dept_1 d) INNER JOIN (_dblink_emp_2 e)
   Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM dept r1 INNER
JOIN emp r2 ON ((r1.deptno = r2.deptno)) ORDER BY r1.deptno ASC NULLS LAST, r2.empno ASC NULLS
LAST
(4 rows)
```

The `INNER JOIN` operation occurs under the `Foreign Scan` section. The output of this join is the following:

```
deptno | dname   | empno | ename  
-------+---------+-------+--------
 10    | ACCOUNTING | 7782 | CLARK  
 10    | ACCOUNTING | 7839 | KING   
 10    | ACCOUNTING | 7934 | MILLER 
 20    | RESEARCH  | 7369 | SMITH  
 20    | RESEARCH  | 7566 | JONES  
 20    | RESEARCH  | 7788 | SCOTT  
 20    | RESEARCH  | 7876 | ADAMS  
 20    | RESEARCH  | 7902 | FORD   
 30    | SALES     | 7499 | ALLEN  
 30    | SALES     | 7521 | WARD   
 30    | SALES     | 7654 | MARTIN 
 30    | SALES     | 7698 | BLAKE  
 30    | SALES     | 7844 | TURNER 
 30    | SALES     | 7900 | JAMES  
(14 rows)
```

This example shows the execution plan of an left outer join:
This example shows a case where the entire processing isn’t pushed down because the \texttt{emp} joined table resides locally instead of on the same foreign server:

```
EXPLAIN (verbose, costs off) SELECT d.deptno, d.dname, e.empno, e.ename FROM dept@oralink d LEFT OUTER JOIN emp@oralink e ON d.deptno = e.deptno ORDER BY 1, 3;
```

**QUERY PLAN**

```
Foreign Scan
  Output: d.deptno, d.dname, e.empno, e.ename
  Relations: (_dblink_dept_1 d) LEFT JOIN (_dblink_emp_2 e)
Remote Query: SELECT r1.deptno, r1.dname, r2.empno, r2.ename FROM dept r1 LEFT JOIN emp r2 ON ((r1.deptno = r2.deptno)) ORDER BY r1.deptno ASC NULLS LAST, r2.empno ASC NULLS LAST
(4 rows)
# The output of this join is the following:

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>empno</th>
<th>ename</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7782</td>
<td>CLARK</td>
</tr>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7839</td>
<td>KING</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7000</td>
<td>SMITH</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7566</td>
<td>JONES</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7088</td>
<td>SCOTT</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7876</td>
<td>ADAMS</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7902</td>
<td>FORD</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7499</td>
<td>ALLEN</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7521</td>
<td>MARD</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7654</td>
<td>MARTIN</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7844</td>
<td>TURNER</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7900</td>
<td>JAMES</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

This example shows the execution plan of ORDER BY clause:

```
EXPLAIN (verbose, costs off) SELECT department_id, COUNT(*), SUM(salary), AVG(salary), MIN(salary), MAX(salary), ROUND(STDDEV(salary)), SUM(salary) * aggpd_pkg.pkgvar AS sum2 FROM employees@oralink WHERE department_id < 50 GROUP BY department_id ORDER BY 1;
```

**QUERY PLAN**

```
Foreign Scan
  Output: department_id, (count(*)), (sum(salary)), (avg(salary)), (min(salary)), (max(salary)), (round(stddev(salary), 0)), ((sum(salary)) * (public.aggd_pkg.pkgvar)::numeric)
  Relations: Aggregate on (_dblink_employees_1 employees)
Remote Query: SELECT department_id, count(*), sum(salary), avg(salary), min(salary), max(salary), round(stddev(salary), 0) FROM employees WHERE (department_id < 50) GROUP BY department_id ORDER BY department_id ASC NULLS LAST
(4 rows)
```

This example shows a case where the entire processing isn’t pushed down because the \texttt{emp} joined table resides locally instead of on the same foreign server:
The output of this join is the same as the previous left-outer-join example.

Creating a foreign table from a database link

Note
This procedure isn’t compatible with Oracle databases.

After you create a database link, you can create a foreign table based on this database link. You can then use the foreign table to access the remote table, referencing it with the foreign table name instead of using the database link syntax. Using the database link requires appending `@dblink` to the table or view name referenced in the SQL command, where `dblink` is the name of the database link.

You can use this technique for either an oci-dblink connection for remote Oracle access or a `postgres_fdw` connection for remote Postgres access.

This example creates a foreign table to access a remote Oracle table.

The following creates a database link named `oralink` for connecting to the Oracle database:

```sql
CREATE DATABASE LINK oralink
CONNECT TO edb IDENTIFIED BY 'password'
USING '//127.0.0.1:1521/xe';
```

The following query shows the database link:

```sql
SELECT lnkname, lnkuser, lnkconnstr
FROM pg_catalog.edb_dblink;
```

```
lnkname | lnkuser | lnkconnstr
---------+---------+---------------------
oralink | edb     | //127.0.0.1:1521/xe
(1 row)
```

When you create the database link, EDB Postgres Advanced Server creates a corresponding foreign server. The following query displays the foreign server:

```sql
SELECT srvname, srvowner, srvfdw, srvtype, srvoptions
FROM pg_foreign_server;
```

```
srvname | srvowner | srvfdw | srvtype | srvoptions
---------+----------+--------+---------+-------------------------------
oralink |       10 |  14005 |         | {connstr=//127.0.0.1:1521/xe} | (1 row)
```

For more information about foreign servers, see the [CREATE SERVER](https://www.postgresql.org/docs/14/sql-createserver.html) command in the [PostgreSQL core documentation](https://www.postgresql.org/)

Create the foreign table:

```sql
CREATE FOREIGN TABLE emp_ora
(
  empno         NUMERIC(4),
  ename         VARCHAR(10),
  job           VARCHAR(9),
  mgr           NUMERIC(4),
  hiredate      TIMESTAMP WITHOUT TIME ZONE,
  sal           NUMERIC(7,2),
  comm          NUMERIC(7,2),
  deptno        NUMERIC(2)
) SERVER oralink
  OPTIONS (table_name 'emp', schema_name 'edb');
```

The output of this join is the same as the previous left-outer-join example.
Note the following in the `CREATE FOREIGN TABLE` command:

- The name specified in the `SERVER` clause at the end of the `CREATE FOREIGN TABLE` command is the name of the foreign server, which is `oralink` in this example. You can see this name in the `srvname` column from the query on `pg_foreign_server`.
- The table name and schema name are specified in the `OPTIONS` clause by the `table` and `schema` options.
- The column names specified in the `CREATE FOREIGN TABLE` command must match the column names in the remote table.
- Generally, `CONSTRAINT` clauses can't be accepted or enforced on the foreign table. They are assumed to have been defined on the remote table.

For more information about the `CREATE FOREIGN TABLE` command, see the PostgreSQL core documentation.

The following is a query on the foreign table:

```sql
SELECT * FROM emp_ora;
```

```plaintext
<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80 00:00:00</td>
<td>800.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81 00:00:00</td>
<td>1600.00</td>
<td>300.00</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>MANDER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
<td>1250.00</td>
<td>500.00</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
<td>2975.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01-MAY-81 00:00:00</td>
<td>2850.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
<td>2450.00</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
<td>3000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08-SEP-81 00:00:00</td>
<td>1500.00</td>
<td>0.00</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7788</td>
<td>23-MAY-87 00:00:00</td>
<td>1100.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
<td>950.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23-JAN-82 00:00:00</td>
<td>1300.00</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
```

(14 rows)

In contrast, the following is a query on the same remote table but using the database link instead of the foreign table:

```sql
SELECT * FROM emp@oralink;
```

```plaintext
<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80 00:00:00</td>
<td>800.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81 00:00:00</td>
<td>1600.00</td>
<td>300.00</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>MANDER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
<td>1250.00</td>
<td>500.00</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
<td>2975.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01-MAY-81 00:00:00</td>
<td>2850.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
<td>2450.00</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
<td>3000.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08-SEP-81 00:00:00</td>
<td>1500.00</td>
<td>0.00</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7788</td>
<td>23-MAY-87 00:00:00</td>
<td>1100.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
<td>950.00</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23-JAN-82 00:00:00</td>
<td>1300.00</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
```

(14 rows)

For backward compatibility, you can still write `USING libpq` instead of `USING postgres_fdw`. However, the `libpq` connector is missing important optimizations that the `postgres_fdw` connector has. Therefore, use the `postgres_fdw` connector when possible. The `libpq` option is deprecated and might be removed in a future EDB Postgres Advanced Server release.

See also

- `DROP DATABASE LINK`

### 14.3.5.24 CREATE DIRECTORY

**Name**
CREATE DIRECTORY — Create an alias for a file system directory path.

Synopsis

```sql
CREATE DIRECTORY <name> AS '<pathname>'
```

Description

The `CREATE DIRECTORY` command creates an alias for a file system directory pathname. You must be a database superuser to use this command.

When you specify the alias as the appropriate parameter to the programs of the `UTL_FILE` package, the operating system files are created in or accessed from the directory corresponding to the alias.

Parameters

```sql
name
```

The directory alias name.

```sql
pathname
```

The fully qualified directory path represented by the alias name. The `CREATE DIRECTORY` command doesn't create the operating system directory. The physical directory must be created independently using operating system commands.

Notes

The operating system user id `enterprisedb` must have the appropriate read and write privileges on the directory if you want to use the `UTL_FILE` package to create or read files using the directory.

The directory alias is stored in the `pg_catalog.edb_dir` system catalog table.

Note

The `edb_dir` table isn't compatible with Oracle databases.

You can also view the directory alias from the Oracle catalog views `SYS.ALL_DIRECTORIES` and `SYS.DBA_DIRECTORIES`. These views are compatible with Oracle databases.

Use the `DROP DIRECTORY` command to delete the directory alias. Deleting a directory alias doesn't affect the corresponding physical file system directory. Delete the file system directory using operating system commands.

In a Linux system, the directory name separator is a forward slash (`/`).

In a Windows system, you can specify the directory name separator as a forward slash (`/`) or two consecutive backslashes (`\`).

Examples

Create an alias named `empdir` for the directory `/tmp/empdir` on Linux:

```sql
CREATE DIRECTORY empdir AS '/tmp/empdir';
```

Create an alias named `empdir` for the directory `C:\TEMP\EMPDIR` on Windows:

```sql
CREATE DIRECTORY empdir AS 'C:/TEMP/EMPDIR';
```

View all of the directory aliases:

```sql
SELECT * FROM pg_catalog.edb_dir;
```
View the directory aliases using a view compatible with Oracle databases:

```
SELECT owner, directory_name, directory_path
FROM SYS.ALL_DIRECTORIES;
```

#### 14.3.5.25 CREATE FUNCTION

**Name**

CREATE FUNCTION — Define a new function.

**Synopsis**

```
CREATE [ OR REPLACE ] FUNCTION <name> [ ( <parameters> ) ]
    [ <declarations> ]
    RETURN <data_type>

    [ IMMUTABLE ]
    [ STABLE ]
    [ VOLATILE ]
    [ DETERMINISTIC ]
    [ NOT ]
    [ LEAKPROOF ]
    [ CALLED ON NULL ]
    [ INPUT ]
    [ RETURNS NULL ON NULL ]
    [ STRICT ]
    [ [ EXTERNAL ] SECURITY ]
    [ INVOKE ]
    [ [ EXTERNAL ] SECURITY ]
    [ DEFINER ]
    [ [ AUTHID ] ]
    [ CURRENT_USER ]
    [ [ AUTHID ] ]
    [ [ PARALLEL [ Unsafe | Restricted | Safe ] ] ]
    [ [ COST ] ]
    [ [ ROWS ] ]
    [ [ SET configuration_parameter ]
        [ TO <value> ]
        [ = <value> ]
        [ FROM CURRENT ]
    ]
    [ ]
    [ [ IS [ AS ] ] ]
```

See also

ALTER DIRECTORY, DROP DIRECTORY
Description

CREATE FUNCTION defines a new function. CREATE OR REPLACE FUNCTION either creates a new function or replaces an existing definition.

If you include a schema name, then the function is created in the specified schema. Otherwise it's created in the current schema. The name of the new function can't match any existing function with the same input argument types in the same schema. However, functions of different input argument types can share a name. This is called overloading.

Note

Overloading functions is an EDB Postgres Advanced Server feature. Overloading stored, standalone functions isn't compatible with Oracle databases.

To update the definition of an existing function, use CREATE OR REPLACE FUNCTION. You can't change the name or argument types of a function this way. (That syntax instead creates a new function.) CREATE OR REPLACE FUNCTION also doesn't let you change the return type of an existing function. To do that, you must drop the function and create it again. Also, you can change the types of any OUT parameters only by dropping the function first.

The user that creates the function becomes the owner of the function.

Parameters

name

The identifier of the function.

parameters

A list of formal parameters.

data_type

The data type of the value returned by the function's RETURN statement.

declarations

Variable, cursor, type, or subprogram declarations. To include subprogram declarations, declare them after all other variable, cursor, and type declarations.

statements

SPL program statements. The BEGIN - END block can contain an EXCEPTION section.

IMMUTABLE

STABLE

VOLATILE

These attributes inform the query optimizer about the behavior of the function. You can specify only one of these attributes.

- Use IMMUTABLE to indicate that the function can't modify the database and must always reach the same result when given the same argument values. This attribute doesn't perform database lookups and uses only information present in its argument list. If you include this clause, you can immediately replace any call of the function with all-constant arguments with the function value.

- Use STABLE to indicate that the function can't modify the database and that, in a single table scan, it consistently returns the same result for the same argument values. However, its result might change across SQL statements. Use this selection for functions that depend on database lookups, parameter variables (such as the current time zone), and so on.

- Use VOLATILE (the default) to indicate that the function value can change even in a single table scan, so no optimizations can be made. Classify any function that has side effects as volatile, even if its result is predictable. This setting prevent calls from being optimized away.

DETERMINISTIC

DETERMINISTIC is a synonym for IMMUTABLE. A DETERMINISTIC function can't modify the database and always reaches the same result when given the same argument values. It doesn't perform database lookups and uses only information directly present in its argument list. If you include this clause, you can immediately replace any call of the function with all-constant arguments with the function value.
A **LEAKPROOF** function has no side effects and reveals no information about the values used to call the function.

**CALLED ON NULL INPUT**

**RETURNS NULL ON NULL INPUT**

**STRICT**

- Use **CALLED ON NULL INPUT** (the default) to call the procedure normally when some of its arguments are **NULL**. Check for **NULL** values, if necessary, and respond appropriately.

- Use **RETURNS NULL ON NULL INPUT** or **STRICT** to indicate that the procedure always returns **NULL** when any of its arguments are **NULL**. If you use these clauses, the procedure doesn't execute when there are **NULL** arguments. Instead, a **NULL** result is assumed.

**EXTERNAL SECURITY DEFINER**

**SECURITY DEFINER** (the default) specifies for the function to execute with the privileges of the user that created it. The optional keyword **EXTERNAL** is allowed for SQL conformance.

**EXTERNAL SECURITY INVOKER**

Use the **SECURITY INVOKER** clause to execute the function with the privileges of the user that calls it. The optional keyword **EXTERNAL** is allowed for SQL conformance.

**AUTHID DEFINER**

**AUTHID CURRENT_USER**

- The **AUTHID DEFINER** clause is a synonym for [EXTERNAL] **SECURITY DEFINER**. If you omit the **AUTHID** clause or specify **AUTHID DEFINER**, the rights of the function owner determine access privileges to database objects.

- The **AUTHID CURRENT_USER** clause is a synonym for [EXTERNAL] **SECURITY INVOKER**. If you specify **AUTHID CURRENT_USER**, the rights of the current user executing the function determine access privileges.

**PARALLEL { UNSAFE | RESTRICTED | SAFE }**

The **PARALLEL** clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query, in contrast to a serial sequential scan.

- When set to **UNSAFE** (the default), the function can't execute in parallel mode. The presence of such a function in a SQL statement forces a serial execution plan.

- When set to **RESTRICTED**, the function can execute in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation isn't chosen for parallelism.

- When set to **SAFE**, the function can execute in parallel mode with no restriction.

**COST** \(\text{execution}_\text{cost}\)

**ROWS** \(\text{result}_\text{rows}\)

**SET configuration_parameter { TO value | = value | FROM CURRENT }**

The **SET** clause causes the specified configuration parameter to be set to the specified value when the function is entered and then restored to its prior value when the function exits. **SET FROM CURRENT** saves the session's current value of the parameter as the value to apply when the function is entered.

If a **SET** clause is attached to a function, then the effects of a **SET LOCAL** command executed inside the function for the same variable are restricted to the function. The configuration parameter's prior value is restored at function exit. An ordinary **SET** command (without **LOCAL**) overrides the **SET** clause, similar to a previous **SET LOCAL** command. The effects of such a command persist after procedure exit unless the current transaction is rolled back.

**PRAGMA AUTONOMOUS_TRANSACTION**

**PRAGMA AUTONOMOUS_TRANSACTION** is the directive that sets the function as an autonomous transaction.

**Note**
The **STRICT**, **LEAKPROOF**, **PARALLEL**, **COST**, **ROWS** and **SET** keywords provide extended functionality for EDB Postgres Advanced Server and aren’t supported by Oracle.

**Notes**

EDB Postgres Advanced Server allows function overloading. That is, you can use the same name for several different functions. However, they must have discrete input **IN**, **IN OUT** ) argument data types.

**Examples**

The function `emp_comp` takes two numbers as input and returns a computed value. The `SELECT` command shows the use of the function.

```sql
CREATE OR REPLACE FUNCTION emp_comp
    (p_sal NUMBER,
     p_comm NUMBER)
    RETURN NUMBER
IS
    BEGIN
        RETURN (p_sal + NVL(p_comm, 0)) * 24;
    END;

SELECT ename "Name", sal "Salary", comm "Commission", emp_comp(sal, comm) "Total Compensation" FROM emp;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Commission</th>
<th>Total Compensation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>800.00</td>
<td></td>
<td>19200.00</td>
</tr>
<tr>
<td>ALLEN</td>
<td>1600.00</td>
<td>300.00</td>
<td>45600.00</td>
</tr>
<tr>
<td>WARD</td>
<td>1250.00</td>
<td>500.00</td>
<td>42000.00</td>
</tr>
<tr>
<td>JONES</td>
<td>2575.00</td>
<td></td>
<td>71400.00</td>
</tr>
<tr>
<td>MARTIN</td>
<td>1250.00</td>
<td>1400.00</td>
<td>63600.00</td>
</tr>
<tr>
<td>BLAKE</td>
<td>2850.00</td>
<td></td>
<td>68400.00</td>
</tr>
<tr>
<td>CLARK</td>
<td>2450.00</td>
<td></td>
<td>58800.00</td>
</tr>
<tr>
<td>SCOTT</td>
<td>3000.00</td>
<td></td>
<td>72000.00</td>
</tr>
<tr>
<td>KING</td>
<td>5000.00</td>
<td></td>
<td>120000.00</td>
</tr>
<tr>
<td>TURNER</td>
<td>1500.00</td>
<td>0.00</td>
<td>36000.00</td>
</tr>
<tr>
<td>ADAMS</td>
<td>1100.00</td>
<td></td>
<td>26400.00</td>
</tr>
<tr>
<td>JAMES</td>
<td>950.00</td>
<td></td>
<td>22800.00</td>
</tr>
<tr>
<td>FORD</td>
<td>3000.00</td>
<td></td>
<td>72000.00</td>
</tr>
<tr>
<td>MILLER</td>
<td>1300.00</td>
<td></td>
<td>31200.00</td>
</tr>
</tbody>
</table>

(14 rows)

The function `sal_range` returns a count of the number of employees whose salary falls in the specified range. The following anonymous block calls the function a number of times using the argument’s default values for the first two calls:

```sql
CREATE OR REPLACE FUNCTION sal_range
    (p_sal_min NUMBER DEFAULT 0,
     p_sal_max NUMBER DEFAULT 10000)
    RETURN INTEGER
IS
    v_count INTEGER;
    BEGIN
        SELECT COUNT(*) INTO v_count FROM emp
        WHERE sal BETWEEN p_sal_min AND p_sal_max;
        RETURN v_count;
    END;

BEGIN
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary: ' || sal_range);
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary of at least $2000.00: ' || sal_range(2000.00));
    DBMS_OUTPUT.PUT_LINE('Number of employees with a salary between
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
This example uses the `AUTHID CURRENT_USER` clause and `STRICT` keyword in a function declaration:

```sql
CREATE OR REPLACE FUNCTION dept_salaries(dept_id int) RETURN NUMBER STRICT
AUTHID CURRENT_USER BEGIN
RETURN QUERY (SELECT sum(salary) FROM emp WHERE deptno = id);
END;
END;
```

Include the `STRICT` keyword to return `NULL` if any input parameter passed is `NULL`. If a `NULL` value is passed, the function doesn't execute.

The `dept_salaries` function executes with the privileges of the role that's calling the function. If the current user doesn't have privileges to perform the `SELECT` statement querying the `emp` table (to display employee salaries), the function reports an error. To use the privileges associated with the role that defined the function, replace the `AUTHID CURRENT_USER` clause with the `AUTHID DEFINER` clause.

Other Pragmas (declared in a package specification)

`PRAGMA RESTRICT_REFERENCES`

EDB Postgres Advanced Server accepts but ignores syntax referencing `PRAGMA RESTRICT_REFERENCES`.

See also

`DROP FUNCTION`

**14.3.5.26 CREATE INDEX**

**Name**

`CREATE INDEX` — Define a new index.

**Synopsis**

```sql
CREATE [ UNIQUE ] INDEX <name> ON <table>
( [ <column> | ( <expression> ) | <constant> ] )
[ TABLESPACE <tablespace> ]
[ ( NOPARALLEL | PARALLEL [ <integer> ] ) ]
```

**Description**

`CREATE INDEX` constructs an index, `name`, on the specified table. Indexes are primarily used to enhance database performance. However, inappropriate use results in slower performance.

The key fields for the index are specified as column names, constants, or as expressions written in parentheses. You can specify multiple fields to create multicolumn indexes.
An index field can be an expression computed from the values of one or more columns of the table row. You can use this feature to get fast access to data based on some transformation of the basic data. For example, an index computed on \texttt{UPPER(col)} allows the clause \texttt{WHERE UPPER(col) = 'JIM'} to use an index.

EDB Postgres Advanced Server provides the B-tree index method. The B-tree index method is an implementation of Lehman-Yao high-concurrency B-trees.

By default, indexes aren't used for \texttt{IS NULL} clauses.

All functions and operators used in an index definition must be immutable. That is, their results must depend only on their arguments and never on any outside influence, such as the contents of another table or the current time. This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression, mark the function immutable when you create it.

If you create an index on a partitioned table, the \texttt{CREATE INDEX} command propagates indexes to the table's subpartitions.

The \texttt{PARALLEL} clause specifies the degree of parallelism used while creating an index. The \texttt{NOPARALLEL} clause resets the parallelism to the default value. \texttt{reloptions} shows the \texttt{parallel_workers} parameter as \texttt{0}.

\textbf{Note}

If you use the \texttt{CREATE INDEX... PARALLEL} command to create an index on a table whose definition included the \texttt{PARALLEL} clause at creation, the server uses the \texttt{PARALLEL} clause provided with the \texttt{CREATE INDEX} command when building a parallel index.

\textbf{Parameters}

\texttt{UNIQUE}

Causes the system to check for duplicate values in the table when the index is created, if data already exists, and each time data is added. Attempts to insert or update data that would result in duplicate entries generates an error.

\texttt{name}

The name of the index to create. You can't create a schema name here. The index is always created in the same schema as its parent table.

\texttt{table}

The name (possibly schema-qualified) of the table to index.

\texttt{column}

The name of a column in the table.

\texttt{expression}

An expression based on one or more columns of the table. Usually, you must write the expression surrounded by parentheses, as shown in the syntax. However, you can omit the parentheses if the expression has the form of a function call.

\texttt{constant}

A constant value that you can use as an index key.

Normally, a row where all indexed columns are \texttt{NULL} isn't included in an index. That means that the optimizer can't use that index for certain queries. To overcome this limitation, you can add a constant to the index, which forces the index to never contain a row where all index columns are \texttt{NULL}.

\texttt{tablespace}

The tablespace in which to create the index. If not specified, \texttt{default_tablespace} is used, or the database's default tablespace if \texttt{default_tablespace} is an empty string.

\texttt{PARALLEL}

Specify \texttt{PARALLEL} to select a degree of parallelism. Set \texttt{parallel_workers} parameter equal to the degree of parallelism to create a parallelized index. Alternatively, if you specify \texttt{PARALLEL} but don't list the degree of parallelism, an index accepts default parallelism.

\texttt{NOPARALLEL}

Specify \texttt{NOPARALLEL} for default execution.

\texttt{integer}

The \texttt{integer} indicates the degree of parallelism, which is a number of \texttt{parallel_workers} used in the parallel operation to perform a parallel scan on an index.
Notes

You can specify up to 32 fields in a multicol mn index.

Examples

To create a B-tree index on the column `ename` in the table `emp`:

```sql
CREATE INDEX name_idx ON emp (ename);
```

To create the same index but have it reside in the `index_tblspc` tablespace:

```sql
CREATE INDEX name_idx ON emp (ename) TABLESPACE index_tblspc;
```

You can include a constant value in the index definition (1) to create an index that never contains a row where all of the indexed columns are `NULL`:

```sql
CREATE INDEX emp_dob_idx ON emp (emp_dob, 1);
```

To create an index on `name_idx` in the table `emp` with degree of parallelism set to 7:

```sql
CREATE UNIQUE INDEX name_idx ON emp (ename) PARALLEL 7;
```

See also

`ALTER INDEX`, `DROP INDEX`

14.3.5.27 CREATE MATERIALIZED VIEW

Name

**CREATE MATERIALIZED VIEW** – Define a new materialized view.

Synopsis

```sql
CREATE MATERIALIZED VIEW <name> 
[(<build_clause>)[(<create_mv_refresh>] AS 
subquery

Where <build_clause> is:

  BUILD [IMMEDIATE | DEFERRED]

Where <create_mv_refresh> is:

  REFRESH [COMPLETE] [ON 
DEMAND]
```

Description

`CREATE MATERIALIZED VIEW` defines a view of a query that isn't updated each time the view is referenced in a query. By default, the view is populated when the view is created. You can include the `BUILD DEFERRED` keywords to delay populating the view.

A materialized view can be schema-qualified. If you specify a schema name when invoking the `CREATE MATERIALIZED VIEW` command, the view is created in the specified schema. The view name must be different from the name of any other view, table, sequence, or index in the same schema.
Parameters

name

The name (optionally schema-qualified) of a view to create.

subquery

A `SELECT` statement that specifies the contents of the view.

build_clause

Include a `build_clause` to specify when to populate the view. Specify `BUILD IMMEDIATE` or `BUILD DEFERRED`:

- `BUILD IMMEDIATE` (the default) populates the view immediately.
- `BUILD DEFERRED` populates the view later, during a `REFRESH` operation.

create_mv_refresh

Include the `create_mv_refresh` clause to specify when to update the contents of a materialized view. The clause contains the `REFRESH` keyword followed by optional `COMPLETE` and `ON DEMAND` keywords, where:

- `COMPLETE` discards the current content and reloads the materialized view by executing the view's defining query when refreshing the materialized view.
- `ON DEMAND` (the default) refreshes the materialized view on demand by calling the `DBMS_MVIEW` package or by calling the Postgres `REFRESH MATERIALIZED VIEW` statement.

Notes

Materialized views are read-only. The server doesn't allow an `INSERT`, `UPDATE`, or `DELETE` on a view.

Permissions of the view owner determine access to tables referenced in the view. The user of a view must have permissions to call all functions the view uses.

For more information about the Postgres `REFRESH MATERIALIZED VIEW` command, see the PostgreSQL core documentation.

Examples

This statement creates a materialized view named `dept_30`:

```sql
CREATE MATERIALIZED VIEW dept_30 BUILD IMMEDIATE AS SELECT * FROM emp WHERE deptno = 30;
```

The view contains information retrieved from the `emp` table about any employee that works in department 30.

14.3.5.28 CREATE PACKAGE

Name

CREATE PACKAGE — Define a new package specification.

Synopsis

```
CREATE [ OR REPLACE ] PACKAGE <name>
[ (name) ]
[ AUTHID { DEFINER | CURRENT_USER } ]
[ IS | AS ]
[ [ <declaration>; ] [, 
  ...] ]
[ ( PROCEDURE
     <proc_name>
```
CREATE PACKAGE defines a new package specification. CREATE OR REPLACE PACKAGE either creates a new package specification or replaces an existing specification.

If you include a schema name, then the package is created in the specified schema. Otherwise it’s created in the current schema. The name of the new package can’t match any existing package in the same schema unless you want to update the definition of an existing package. In that case, use CREATE OR REPLACE PACKAGE.

The user that creates the procedure becomes the owner of the package.

Parameters

name

The name (optionally schema-qualified) of the package to create.

DEFINER | CURRENT_USER

Specifies whether to use the privileges of the package owner (DEFINER) or the privileges of the current user executing a program in the package (CURRENT_USER) to determine whether access is allowed to database objects referenced in the package. DEFINER is the default.

declaration

A public variable, type, cursor, or REF CURSOR declaration.

proc_name

The name of a public procedure.

argname

The name of an argument.

IN | IN OUT | OUT

The argument mode.

argtype

The data types of the program’s arguments.

DEFAULT value

Default value of an input argument.
The name of a public function.

The return data type.

**DETERMINISTIC**

**DETERMINISTIC** is a synonym for **IMMUTABLE**. A **DETERMINISTIC** procedure can’t modify the database and always reaches the same result when given the same argument values. It doesn’t perform database lookups and uses only information directly present in its argument list. If you include this clause, you can immediately replace any call of the procedure with all-constant arguments with the procedure value.

**RNDS | RNPS | TRUST | WNDS | WNPS**

The keywords are accepted for compatibility and ignored.

**Examples**

The package specification `empinfo` contains three public components: a public variable, a public procedure, and a public function:

```sql
CREATE OR REPLACE PACKAGE empinfo IS
    emp_name VARCHAR2(10);
    PROCEDURE get_name (p_empno NUMBER);
    FUNCTION display_counter RETURN INTEGER;
END;
```

**See also**

DROP PACKAGE

**14.3.5.29 CREATE PACKAGE BODY**

**Name**

**CREATE PACKAGE BODY** – Define a new package body.

**Synopsis**

```
CREATE [ OR REPLACE ] PACKAGE BODY <name>
    [ IS | AS ]
    [ declaration; ] [ forward_declaration ] [, ...]
    [ ( PROCEDURE <proc_name>
        [ ( <argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT <value> ] [, ...])
        [ STRICT ]
        [ LEAKPROOF ]
        [ PARALLEL { UNSAFE | RESTRICTED | SAFE } ]
        [ COST <execution_cost> ]
    ]
```
CREATE PACKAGE BODY defines a new package body. CREATE OR REPLACE PACKAGE BODY either creates a new package body or replaces an existing body.

If you include a schema name, then the package body is created in the specified schema. Otherwise it's created in the current schema. The name of the new package body must match an existing package specification in the same schema. The new package body name can't match any existing package body in the same schema unless you want to update the definition of an existing package body. In that case, use CREATE OR REPLACE PACKAGE BODY.

Parameters

name

The name (optionally schema-qualified) of the package body to create.
A private variable, type, cursor, or `REF CURSOR` declaration.

**forward_declaration**

The forward declaration of a procedure or function appears in a package body and is declared in advance of the actual body definition. In a block, you can create multiple subprograms. If they invoke each other, each one requires a forward declaration. You must declare a subprogram before you can invoke it. You can use a forward declaration to declare a subprogram without defining it. The forward declaration and its corresponding definition must reside in the same block.

**proc_name**

The name of a public or private procedure. If `proc_name` exists in the package specification with an identical signature, then it's public. Otherwise, it's private.

**argname**

The name of an argument.

**IN | IN OUT | OUT**

The argument mode.

**argtype**

The data types of the program's arguments.

**DEFAULT value**

Default value of an input argument.

**STRICT**

Use the `STRICT` keyword to specify for the function not to execute if called with a `NULL` argument. Instead the function returns `NULL`.

**LEAKPROOF**

Use the `LEAKPROOF` keyword to specify for the function not to reveal any information about arguments other than through a return value.

**PARALLEL { UNSAFE | RESTRICTED | SAFE }**

The `PARALLEL` clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query, in contrast to a serial sequential scan.

- When set to `UNSAFE` (the default), the procedure or function can't execute in parallel mode. The presence of such a procedure or function forces a serial execution plan.
- When set to `RESTRICTED`, the procedure or function can execute in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that's parallel restricted, that relation isn't chosen for parallelism.
- When set to `SAFE`, the procedure or function can execute in parallel mode with no restriction.

**execution_cost**

`execution_cost` specifies a positive number giving the estimated execution cost for the function, in units of `cpu_operator_cost`. If the function returns a set, this is the cost per returned row. The default is `0.0025`.

**result_rows**

`result_rows` is the estimated number of rows for the query planner to expect the function to return. The default is `1000`.

**SET**

Use the `SET` clause to specify a parameter value for the duration of the function:

- `config_param` specifies the parameter name.
- `value` specifies the parameter value.
- `FROM CURRENT` guarantees that the parameter value is restored when the function ends.

**program_body**

The pragma, declarations, and SPL statements that comprise the body of the function or procedure.
The pragma can be **PRAGMA AUTONOMOUS_TRANSACTION** to set the function or procedure as an autonomous transaction.

The declarations can include variable, type, **REF CURSOR**, or subprogram declarations. If you include subprogram declarations, declare them after all other variable, type, and **REF CURSOR** declarations.

```
func_name
```

The name of a public or private function. If `func_name` exists in the package specification with an identical signature, then it's public. Otherwise it's private.

```
rettype
```

The return data type.

```
DETERMINISTIC
```

Include **DETERMINISTIC** to specify for the function to always return the same result when given the same argument values. A **DETERMINISTIC** function must not modify the database.

---

**Note**

The **DETERMINISTIC** keyword is equivalent to the PostgreSQL **IMMUTABLE** option. If you specify **DETERMINISTIC** for a public function in the package body, you must also specify it for the function declaration in the package specification. For private functions, there's no function declaration in the package specification.

```
statement
```

An SPL program statement. Statements in the package initialization section execute once per session the first time the package is referenced.

---

**Note**

The **STRICT**, **LEAKPROOF**, **PARALLEL**, **COST**, **ROWS** and **SET** keywords provide extended functionality for EDB Postgres Advanced Server and aren't supported by Oracle.

---

### Examples

The following is the package body for the `empinfo` package:

```
CREATE OR REPLACE PACKAGE BODY empinfo IS
  v_counter INTEGER;

  PROCEDURE get_name(
    p_empno NUMBER
  ) IS
    BEGIN
      SELECT ename INTO emp_name FROM emp WHERE empno = p_empno;
      v_counter := v_counter + 1;
      END;

  FUNCTION display_counter
    RETURN INTEGER IS
    BEGIN
      RETURN v_counter;
      END;

  BEGIN
    v_counter := 0;
    DBMS_OUTPUT.PUT_LINE('Initialized counter');
    END;

END;
```

The following two anonymous blocks execute the procedure and function in the `empinfo` package and display the public variable:

```
BEGIN
  empinfo.get_name(7369);
  DBMS_OUTPUT.PUT_LINE('Employee Name : ' || empinfo.emp_name);
  DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);
END;
```

**Initialized counter**

Employee Name : SMITH

Number of queries: 1
This example uses a forward declaration in a package body. The example displays the name and number of employees whose salary falls in the specified range.

```
CREATE OR REPLACE PACKAGE empinfo IS
    FUNCTION emp_comp (p_sal_range INTEGER) RETURN INTEGER;
END empinfo;
CREATE OR REPLACE PACKAGE BODY empinfo IS
    FUNCTION emp_comp (p_sal_range INTEGER) RETURN INTEGER IS
    BEGIN
      dbms_output.put_line ('Employee details
      return sal_range(p_sal_range);
    END;
    FUNCTION sal_range (p_sal_range INTEGER) RETURN INTEGER IS
    emp_cnt INTEGER;
    BEGIN
      select count(*) into emp_cnt from emp where sal <= p_sal_range;
      dbms_output.put_line('Number of employees in the salary range '|| p_sal_range|| ' is :'|| emp_cnt);
      return emp_cnt;
    END;
    PROCEDURE list_emp (p_sal_range INTEGER) IS
    BEGIN
      FOR i IN select ename from emp where sal <= p_sal_range LOOP
        dbms_output.put_line (i);
      END LOOP;
      END;
      END
    empinfo;

SELECT empinfo.emp_comp(1500);
```

This example uses a forward declaration in a package body. The example displays the name and number of employees whose salary falls in the specified range.

Employee details
Number of employees in the salary range 1500 is :7
(SMITH)
(WARD)
(MARTIN)
(TURNER)
(ADAMS)
(JAMES)
(MILLER)
emp_comp
--------
7
(1 row)

See also
CREATE PACKAGE, DROP PACKAGE
# CREATE PROCEDURE

## Name

`CREATE PROCEDURE` – Define a new stored procedure.

## Synopsis

```
CREATE [OR REPLACE] PROCEDURE <name> [ (<parameters>) ]

[ IMUTABLE |
STABLE |
VOLATILE |
DETERMINISTIC |
[ NOT ]
LEAKPROOF |
CALLED ON NULL |
INPUT |
RETURNS NULL ON NULL |
INPUT |
STRICT |
[ EXTERNAL ] SECURITY |
[ EXTERNAL ] SECURITY |
DEFINER |
AUTHID |
DEFINER |
AUTHID |
CURRENT_USER |
PARALLEL [ Unsafe | Restricted | Safe ] |
[ COST |
<execution_cost> |
ROWS |
<result_rows> ] |
SET |
<configuration_parameter> |
[ TO <value> | = <value> | FROM CURRENT ] |
...
[ IS | AS ] |
PRAGMA AUTONOMOUS_TRANSACTION; |
[ <declarations> ] |
BEGIN |
<statements> |
END [ <name> ]
```

## Description

`CREATE PROCEDURE` defines a new stored procedure. `CREATE OR REPLACE PROCEDURE` either creates a new procedure or replaces an existing definition.

If you include a schema name, then the procedure is created in the specified schema. Otherwise it’s created in the current schema. The name of the new procedure can’t match any existing procedure with the same input argument types in the same schema. However, procedures of different input argument types can share a name. This is called overloading.

### Note

Overloading procedures is an EDB Postgres Advanced Server feature. Overloading stored, standalone procedures isn’t compatible with Oracle databases.

To update the definition of an existing procedure, use `CREATE OR REPLACE PROCEDURE`. You can’t change the name or argument types of a procedure this way. That syntax creates a new, distinct procedure. When using `OUT` parameters, you can’t change the types of any `OUT` parameters except by dropping the procedure.
Parameters

name

The identifier of the procedure.

parameters

A list of formal parameters.

declarations

Variable, cursor, type, or subprogram declarations. If you include subprogram declarations, declare them after all other variable, cursor, and type declarations.

statements

SPL program statements. The \texttt{BEGIN} -- \texttt{END} block can contain an \texttt{EXCEPTION} section.

\texttt{IMMUTABLE}

\texttt{STABLE}

\texttt{VOLATILE}

These attributes inform the query optimizer about the behavior of the procedure. You can specify only one of these attributes.

- Use \texttt{IMMUTABLE} to indicate that the procedure can't modify the database and always reaches the same result when given the same argument values. It doesn't perform database lookups and uses only information directly present in its argument list. If you include this clause, you can immediately replace any call of the procedure with all-constant arguments with the procedure value.

- Use \texttt{STABLE} to indicate that the procedure can't modify the database and that, in a single table scan, it consistently returns the same result for the same argument values. However, its result might change across SQL statements. Use this selection for procedures that depend on database lookups, parameter variables (such as the current time zone), and so on.

- Use \texttt{VOLATILE} (the default) to indicate that the procedure value can change even in a single table scan, so no optimizations can be made. You must classify any function that has side effects as volatile, even if its result is predictable, to prevent calls from being optimized away.

\texttt{DETERMINISTIC}

\texttt{DETERMINISTIC} is a synonym for \texttt{IMMUTABLE}. A \texttt{DETERMINISTIC} procedure can't modify the database and always reaches the same result when given the same argument values. It doesn't perform database lookups and uses only information directly present in its argument list. If you include this clause, you can immediately replace any call of the procedure with all-constant arguments with the procedure value.

\texttt{[ NOT ] LEAKPROOF}

A \texttt{LEAKPROOF} procedure has no side effects and reveals no information about the values used to call the procedure.

\texttt{CALLED ON NULL INPUT}

\texttt{RETURNS NULL ON NULL INPUT}

\texttt{STRICT}

- Use \texttt{CALLED ON NULL INPUT} (the default) to call the procedure normally when some of its arguments are \texttt{NULL}. Check for \texttt{NULL} values, if needed, and respond appropriately.

- Use \texttt{RETURNS NULL ON NULL INPUT} or \texttt{STRICT} to indicate that the procedure always returns \texttt{NULL} when any of its arguments are \texttt{NULL}. If you specify these clauses, the procedure doesn't execute when there are \texttt{NULL} arguments. Instead a \texttt{NULL} result is assumed.

\texttt{[ EXTERNAL ] SECURITY DEFINER}

Use \texttt{SECURITY DEFINER} (the default) to specify that the procedure executes with the privileges of the user that created it. The keyword \texttt{EXTERNAL} is allowed for SQL conformance but is optional.

\texttt{[ EXTERNAL ] SECURITY INVOKER}

Use the \texttt{SECURITY INVOKER} clause to execute the procedure with the privileges of the user that calls it. The keyword \texttt{EXTERNAL} is allowed for SQL conformance but is optional.

\texttt{AUTHID DEFINER}

\texttt{AUTHID CURRENT_USER}

- The \texttt{AUTHID DEFINER} clause is a synonym for \texttt{[EXTERNAL] SECURITY DEFINER}. If you omit the \texttt{AUTHID} clause or specify \texttt{AUTHID DEFINER}, the rights of the procedure owner determine access privileges to database objects.
The **AUTHID CURRENT_USER** clause is a synonym for **EXTERNAL SECURITY INVOKER**. If you specify **AUTHID CURRENT_USER**, the rights of the current user executing the procedure determine access privileges.

**PARALLEL { UNSAFE | RESTRICTED | SAFE }**

The **PARALLEL** clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query, in contrast to a serial sequential scan.

- When set to **UNSAFE** (the default), the procedure can't execute in parallel mode. The presence of such a procedure forces a serial execution plan.
- When set to **RESTRICTED**, the procedure can execute in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation isn't chosen for parallelism.
- When set to **SAFE**, the procedure can execute in parallel mode with no restriction.

**COST execution_cost**

**execution_cost** is a positive number giving the estimated execution cost for the procedure, in units of **cpu_operator_cost**. If the procedure returns a set, this is the cost per returned row. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

**ROWS result_rows**

**result_rows** is a positive number giving the estimated number of rows for the planner to expect the procedure to return. This option is allowed only when the procedure is declared to return a set. The default is 1000 rows.

**SET configuration_parameter { TO value | = value | FROM CURRENT }**

The **SET** clause sets the specified configuration parameter to the specified value when the procedure is entered and then restored to its prior value when the procedure exits. **SET FROM CURRENT** saves the session's current value of the parameter as the value to apply when the procedure is entered.

If a **SET** clause is attached to a procedure, then the effects of a **SET LOCAL** command executed inside the procedure for the same variable are restricted to the procedure. The configuration parameter's prior value is restored at procedure exit. An ordinary **SET** command (without **LOCAL**) overrides the **SET** clause, similar to a previous **SET LOCAL** command. The effects of such a command persist after procedure exit, unless the current transaction is rolled back.

**PRAGMA AUTONOMOUS_TRANSACTION**

**PRAGMA AUTONOMOUS_TRANSACTION** is the directive that sets the procedure as an autonomous transaction.

**Note**

- The **STRICT, LEAKPROOF, PARALLEL, COST, ROWS and SET** keywords provide extended functionality for EDB Postgres Advanced Server and aren't supported by Oracle.
- The **IMMUTABLE, STABLE, STRICT, LEAKPROOF, COST, ROWS and PARALLEL { UNSAFE | RESTRICTED | SAFE }** attributes are supported only for EDB SPL procedures.
- By default, stored procedures are created as **SECURITY DEFINERS**. Stored procedures defined in plpgsql are created as **SECURITY INVOKERS**.

**Examples**

This procedure lists the employees in the **emp** table:

```sql
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno NUMBER(4);
    v_ename VARCHAR2(10);
    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
    BEGIN
        OPEN emp_cur;
        DBMS_OUTPUT.PUT_LINE('EMPNO     ENAME');
        LOOP
            FETCH emp_cur INTO v_empno, v_ename;
            EXIT WHEN emp_cur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(v_empno || ' ' || v_ename);
        END LOOP;
        CLOSE emp_cur;
    END;
END;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
This procedure uses IN OUT and OUT arguments to return an employee's number, name, and job based on a search using the given employee number. If that isn't found, then the search uses the given name. An anonymous block calls the procedure.

```
CREATE OR REPLACE PROCEDURE emp_job
(
    p_empno    IN OUT emp.empno%TYPE,
    p_ename    IN OUT emp.ename%TYPE,
    p_job      OUT emp.job%TYPE
)
IS
    v_empno    emp.empno%TYPE;
    v_ename    emp.ename%TYPE;
    v_job      emp.job%TYPE;
BEGIN
    SELECT ename, job INTO v_ename, v_job FROM emp WHERE empno = p_empno;
    p_ename := v_ename;
    p_job := v_job;
    DBMS_OUTPUT.PUT_LINE('Found employee # ' || p_empno);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        SELECT empno, job INTO v_empno, v_job FROM emp
        WHERE ename = p_ename;
        p_empno := v_empno;
        p_job := v_job;
        DBMS_OUTPUT.PUT_LINE('Found employee ' || p_ename);
END;
END;
END;
```

DECLARE
    v_empno    emp.empno%TYPE;
    v_ename    emp.ename%TYPE;
    v_job      emp.job%TYPE;
BEGIN
    v_empno := 0;
    v_ename := 'CLARK';
    emp_job(v_empno, v_ename, v_job);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name : ' || v_ename);
```
This example uses the AUTHID DEFINER and SET clauses in a procedure declaration. The update_salary procedure conveys the privileges of the role that defined the procedure to the role that's calling the procedure while the procedure executes.

```
CREATE OR REPLACE PROCEDURE update_salary(id INT, new_salary NUMBER)
  SET SEARCH_PATH = 'public' SET WORK_MEM = '1MB'
  AUTHID DEFINER IS
  BEGIN
    UPDATE emp SET salary = new_salary WHERE emp_id = id;
  END;
```

Include the SET clause to set the procedure's search path to public and the work memory to 1MB. Other procedures, functions, and objects aren't affected by these settings.

In this example, the AUTHID DEFINER clause temporarily grants privileges to a role that otherwise might not be allowed to execute the statements in the procedure. To use the privileges associated with the role invoking the procedure, replace the AUTHID DEFINER clause with the AUTHID CURRENT_USER clause.

See also

DROP PROCEDURE, ALTER PROCEDURE

14.3.5.31 CREATE PROFILE

Name

```
CREATE PROFILE — Create a profile.
```

Synopsis

```
CREATE PROFILE <profile_name>
  [LIMIT {{<parameter value>}} ...]
```

Description

```
CREATE PROFILE create a profile. Include the LIMIT clause and one or more space-delimited parameter/value pairs to specify the rules enforced by EDB Postgres Advanced Server.
```

EDB Postgres Advanced Server creates a default profile named DEFAULT. When you use the CREATE ROLE command to create a role, the new role is associated with the DEFAULT profile. If you upgrade from a previous version of EDB Postgres Advanced Server to EDB Postgres Advanced Server 10, the upgrade process creates the roles in the upgraded version to the DEFAULT profile.

You must be a superuser to use CREATE PROFILE.

Include the LIMIT clause and one or more space-delimited parameter/value pairs to specify the rules enforced by EDB Postgres Advanced Server.

Parameters

```
profile_name
```

The name of the profile.
The password attribute for the rule to monitor.

The value the parameter must reach before an action is taken by the server.

EDB Postgres Advanced Server supports these values for each parameter:

**FAILED_LOGIN_ATTEMPTS** specifies the number of failed login attempts that a user can make before the server locks them out of their account for the length of time specified by **PASSWORD_LOCK_TIME**. Supported values are:

- An **INTEGER** value greater than 0.
- **DEFAULT** — The value of **FAILED_LOGIN_ATTEMPTS** specified in the **DEFAULT** profile.
- **UNLIMITED** — The connecting user can make an unlimited number of failed login attempts.

**PASSWORD_LOCK_TIME** specifies the length of time that must pass before the server unlocks an account that was locked because of **FAILED_LOGIN_ATTEMPTS**. Supported values are:

- A **NUMERIC** value of 0 or greater. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- **DEFAULT** — The value of **PASSWORD_LOCK_TIME** specified in the **DEFAULT** profile.
- **UNLIMITED** — The account is locked until manually unlocked by a database superuser.

**PASSWORD_LIFE_TIME** specifies the number of days that the current password can be used before the user is prompted to provide a new password. Include the **PASSWORD_GRACE_TIME** clause when using the **PASSWORD_LIFE_TIME** clause to specify the number of days to pass after the password expires before the user is forced to change their password. If you don't specify **PASSWORD_GRACE_TIME**, the password expires on the day specified by the default value of **PASSWORD_LIFE_TIME**, and the user isn't allowed to execute any command until a new password is provided. Supported values are:

- A **NUMERIC** value greater than 0 or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- **DEFAULT** — The value of **PASSWORD_LIFE_TIME** specified in the **DEFAULT** profile.
- **UNLIMITED** — The password doesn't have an expiration date.

**PASSWORD_GRACE_TIME** specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user is allowed to connect but isn't allowed to execute any command until they update their expired password. Supported values are:

- A **NUMERIC** value of 0 or greater. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- **DEFAULT** — The value of **PASSWORD_GRACE_TIME** specified in the **DEFAULT** profile.
- **UNLIMITED** — The grace period is infinite.

**PASSWORD_REUSE_TIME** specifies the number of days a user must wait before reusing a password. Use the **PASSWORD_REUSE_TIME** and **PASSWORD_REUSE_MAX** parameters together. If you specify a finite value for one of these parameters while the other is **UNLIMITED**, old passwords can never be reused. If both parameters are set to **UNLIMITED**, there are no restrictions on password reuse. Supported values are:

- A **NUMERIC** value of 0 or greater. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours.
- **DEFAULT** — The value of **PASSWORD_REUSE_TIME** specified in the **DEFAULT** profile.
- **UNLIMITED** — The password can be reused without restrictions.

**PASSWORD_REUSE_MAX** specifies the number of password changes that must occur before a password can be reused. Use the **PASSWORD_REUSE_TIME** and **PASSWORD_REUSE_MAX** parameters together. If you specify a finite value for one of these parameters while the other is **UNLIMITED**, old passwords can never be reused. If both parameters are set to **UNLIMITED**, there are no restrictions on password reuse. Supported values are:

- An **INTEGER** value greater than 0 or equal to 0.
- **DEFAULT** — The value of **PASSWORD_REUSE_MAX** specified in the **DEFAULT** profile.
- **UNLIMITED** — The password can be reused without restrictions.

**PASSWORD_VERIFY_FUNCTION** specifies password complexity. Supported values are:

- The name of a PL/SQL function.
- **DEFAULT** — The value of **PASSWORD_VERIFY_FUNCTION** specified in the **DEFAULT** profile.
- **NULL**

**PASSWORD_ALLOW_HASHED** specifies whether an encrypted password is allowed for use. If you specify **TRUE**, the system allows a user to change the password by specifying a hash computed using a password client-side. However, if you specify **FALSE**, then a password must be specified in a plain-text form to be validated. Otherwise an error is thrown if a server receives an encrypted password. Supported values are:

- A **BOOLEAN** value **TRUE/ON/YES/1** or **FALSE/OFF/NO/0**.
- **DEFAULT** — The value of **PASSWORD_ALLOW_HASHED** specified in the **DEFAULT** profile.

**Note**

The **PASSWORD_ALLOW_HASHED** isn't compatible with Oracle.
Notes

Use the DROP PROFILE command to remove the profile.

Examples

This command creates a profile named acctg. The profile specifies that if a user hasn't authenticated with the correct password in five attempts, the account is locked for one day:

```
CREATE PROFILE acctg LIMIT
    FAILED_LOGIN_ATTEMPTS 5
    PASSWORD_LOCK_TIME 1;
```

This command creates a profile named sales. The profile specifies that a user must change their password every 90 days:

```
CREATE PROFILE sales LIMIT
    PASSWORD_LIFE_TIME 90
    PASSWORD_GRACE_TIME 3;
```

If the user doesn't change their password before the 90 days specified in the profile passes, they are issued a warning at login. After a grace period of three days, their account isn't allowed to invoke any commands until they change their password.

This command creates a profile named accts. The profile specifies that a user can't reuse a password within 180 days of the last use of the password and must change their password at least five times before reusing it:

```
CREATE PROFILE accts LIMIT
    PASSWORD_REUSE_TIME 180
    PASSWORD_REUSE_MAX 5;
```

This command creates a profile named resources. The profile calls a user-defined function named password_rules that verifies that the password provided meets their standards for complexity:

```
CREATE PROFILE resources LIMIT
    PASSWORD_VERIFY_FUNCTION password_rules;
```

See also

ALTER PROFILE, DROP PROFILE

14.3.5.32   CREATE QUEUE

EDB Postgres Advanced Server includes extra syntax not offered by Oracle with the CREATE QUEUE SQL command. You can use this syntax with DBMS_AQADM.

Name

CREATE QUEUE — Create a queue.

Synopsis

Use CREATE QUEUE to define a new queue:

```
CREATE QUEUE <name> QUEUE TABLE <queue_table_name> [ ( { <option_name option_value> } [, ... ] ) ]
```

Where possible values for option_name and the corresponding option_value are:

- **TYPE** [normal_queue | exception_queue]
- **RETRIES** [INTEGER]
- **RETRYDELAY** [DOUBLE PRECISION]
The **CREATE QUEUE** command allows a database superuser or any user with the system-defined `aq_administrator_role` privilege to create a queue in the current database.

If the name of the queue is schema-qualified, the queue is created in the specified schema. If the **CREATE QUEUE** command doesn't include a schema, the queue is created in the current schema. You can create a queue only in the schema where the queue table resides. The name of the queue must be unique among queues in the same schema.

Use **DROP QUEUE** to remove a queue.

### Parameters

- **name**
  
  The name (optionally schema-qualified) of the queue to create.

- **queue_table_name**
  
  The name of the queue table with which this queue is associated.

- **option_name option_value**
  
  The name of any options to associate with the new queue and the corresponding value for the option. If the call to **CREATE QUEUE** includes duplicate option names, the server returns an error. The following values are accepted:

  - **TYPE** — Specify `normal_queue` to indicate that the queue is a normal queue or `exception_queue` for an exception queue. An exception queue accepts only dequeue operations.
  
  - **RETRIES** — An integer value that specifies the maximum number of attempts to remove a message from a queue.
  
  - **RETRYDELAY** — A double-precision value that specifies the number of seconds after a rollback that the server waits before retrying a message.

  - **RETENTION** — A double-precision value that specifies the number of seconds to save a message in the queue table after dequeueing.

### Examples

This command creates a queue named `work_order` that's associated with a queue table named `work_order_table`:

```sql
CREATE QUEUE work_order QUEUE TABLE work_order_table (RETRIES 5, RETRYDELAY 2);
```

The server allows 5 attempts to remove a message from the queue and enforces a retry delay of 2 seconds between attempts.

### See also

**ALTER QUEUE, DROP QUEUE**

### 14.3.5.33 CREATE QUEUE TABLE

EDB Postgres Advanced Server includes extra syntax not offered by Oracle with the **CREATE QUEUE TABLE** SQL command. You can use this syntax with **DBMS_AQADM**.

#### Name

- **CREATE QUEUE TABLE** — Create a queue table.

#### Synopsis
Use `CREATE QUEUE TABLE` to define a queue table:

```
CREATE QUEUE TABLE <name> OF <type_name> [ ( ( <option_name> <option_value> ) [, ... ] ) ]
```

Possible `option_name` and corresponding `option_value` values are:

- `SORT_LIST` — Specify `option_value` as `priority`, `enq_time`.
- `MULTIPLE_CONSUMERS` — Specify `option_value` as `FALSE`, `TRUE`.
- `MESSAGE_GROUPING` — Specify `option_value` as `NONE`, `TRANSACTIONAL`.
- `STORAGE_CLAUSE` — Specify `option_value` as `TABLESPACE tablespace_name`, `PCTFREE integer`, `PCTUSED integer`, `INITRANS integer`, `MAXTRANS integer`, `STORAGE storage_option`. Values for `storage_option` are one or more of the following: `MINEXTENTS integer`, `MAXEXTENTS integer`, `PCTINCREASE integer`, `INITIAL size_clause`, `NEXT`, `FREELISTS integer`, `OPTIMAL size_clause`, `BUFFER_POOL {KEEP|RECYCLE|DEFAULT}`.

Only the `TABLESPACE` option is enforced. All others are accepted for compatibility and ignored. Use the `TABLESPACE` clause to specify the name of a tablespace in which to create the table.

Description

`CREATE QUEUE TABLE` allows a superuser or a user with the `aq_administrator_role` privilege to create a queue table.

If the call to `CREATE QUEUE TABLE` includes a schema name, the queue table is created in the specified schema. If you don’t provide a schema name, the queue table is created in the current schema.

The name of the queue table must be unique among queue tables in the same schema.

Parameters

- **name**
  The name (optionally schema-qualified) of the new queue table.

- **type_name**
  The name of an existing type that describes the payload of each entry in the queue table. For information about defining a type, see `CREATE TYPE`.

- **option_name option_value**
  The name of any options that to associate with the new queue and the corresponding value for the option. If the call to `CREATE QUEUE` includes duplicate option names, the server returns an error. The following values are accepted:

  - `SORT_LIST` — Use the `SORT_LIST` option to control the dequeuing order of the queue. Specify the names of the columns to use, in ascending order, to sort the queue. The following combinations of `enq_time` and `priority` are possible values:

    - `enq_time`, `priority`
    - `priority`, `enq_time`
    - `priority`
    - `enq_time`

  - `MULTIPLE_CONSUMERS` — A `BOOLEAN` value that indicates if a message can have more than one consumer (`TRUE`) or are limited to one consumer per message (`FALSE`).

  - `MESSAGE_GROUPING` — Specify `none` to dequeue each message individually or `transactional` to add messages to the queue as a result of one transaction and dequeue them as a group.

  - `STORAGE_CLAUSE` — Use `STORAGE_CLAUSE` to specify table attributes. Possible values are `TABLESPACE tablespace_name`, `PCTFREE integer`, `PCTUSED integer`, `INITRANS integer`, `MAXTRANS integer`, `STORAGE storage_option`. Possible values for `storage_option` are:

    - `MINEXTENTS integer`
    - `MAXEXTENTS integer`
    - `PCTINCREASE integer`
    - `INITIAL size_clause`
    - `NEXT`
Examples

You must create a user-defined type before creating a queue table. The type describes the columns and data types in the table. This command creates a type named `work_order`:

```
CREATE TYPE work_order AS (name VARCHAR2, project TEXT, completed BOOLEAN);
```

This command uses the `work_order` type to create a queue table named `work_order_table`:

```
CREATE QUEUE TABLE work_order_table OF work_order (sort_list (enq_time, priority));
```

See also

`ALTER QUEUE TABLE`, `DROP QUEUE TABLE`

14.3.5.34 CREATE ROLE

Name

`CREATE ROLE` — Define a new database role.

Synopsis

```
CREATE ROLE <name> [IDENTIFIED BY <password>] [REPLACE old_password]
```

Description

`CREATE ROLE` adds a role to the EDB Postgres Advanced Server database cluster. A role is an entity that can own database objects and have database privileges. A role can be considered a user, a group, or both depending on how you use it. The new role doesn't have the `LOGIN` attribute, so you can't use it to start a session. (Use the `ALTER ROLE` command to give the role `LOGIN` rights.) You must have `CREATEROLE` privilege or be a database superuser to use `CREATE ROLE`.

If you specify the `IDENTIFIED BY` clause, the `CREATE ROLE` command also creates a schema owned by and with the same name as the newly created role.

Note

The roles are defined at the database cluster level, making them valid in all databases in the cluster.

Parameters

name

The name of the new role.

IDENTIFIED BY password
Sets the role's password. A password is needed only for roles having the LOGIN attribute, but you can still define one for roles without it. If you don’t plan to use password authentication, you can omit this option.

Notes

Use ALTER ROLE to change the attributes of a role. Use DROP ROLE to remove a role. You can modify the attributes specified by CREATE ROLE using ALTER ROLE commands.

Use GRANT and REVOKE to add and remove members of roles that are being used as groups.

The maximum length limit for role name and password is 63 characters.

Examples

Create a role and a schema named admins with a password:

```sql
CREATE ROLE admins IDENTIFIED BY Rt498zb;
```

See also

ALTER ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE

14.3.5.35 CREATE SCHEMA

Name

CREATE SCHEMA — Define a new schema.

Synopsis

```
CREATE SCHEMA AUTHORIZATION <username> <schema_element> [ ... ]
```

Description

This variation of the CREATE SCHEMA command creates a schema owned by <username> and populated with one or more objects. Creating the schema and objects occurs in a single transaction, so either all objects are created or none of them are, including the schema. If you’re using an Oracle database, no new schema is created. <username>, and therefore the schema, must exist before you use the command.

A schema is essentially a namespace. It contains named objects, such as tables and views, whose names might duplicate those of other objects existing in other schemas. You can access named objects either by qualifying their names with the schema name as a prefix or by setting a search path that includes the desired schemas. Unqualified objects are created in the current schema. The current schema is the one at the front of the search path. Use the function CURRENT_SCHEMA to determine the value.

Note

The search path concept and the CURRENT_SCHEMA function aren’t compatible with Oracle databases.

CREATE SCHEMA includes subcommands to create objects in the schema. The subcommands are treated the same as separate commands issued after creating the schema. All the created objects are owned by the specified user.

Parameters
The name of the user to own the new schema. The schema also gets this name. Only superusers can create schemas owned by users other than themselves. In EDB Postgres Advanced Server, the role `username` must already exist, but the schema must not exist. In Oracle, the user and schema must exist.

A SQL statement defining an object to create in the schema. `CREATE TABLE`, `CREATE VIEW`, and `GRANT` are accepted as clauses in `CREATE SCHEMA`. You can create other kinds of objects in separate commands after you create the schema.

### Notes

Non-superusers creating the schema must have the `CREATE` privilege for the current database.

In EDB Postgres Advanced Server, other forms of the `CREATE SCHEMA` command aren’t compatible with Oracle databases.

### Examples

```
CREATE SCHEMA AUTHORIZATION enterprisedb
    CREATE TABLE empjobs (ename VARCHAR2(10), job VARCHAR2(9))
    CREATE VIEW managers AS SELECT ename FROM empjobs WHERE job = 'MANAGER'
    GRANT SELECT ON managers TO PUBLIC;
```

#### 14.3.5.36 CREATE SEQUENCE

### Name

`CREATE SEQUENCE` — Define a new sequence generator.

### Synopsis

```
CREATE SEQUENCE <name> [ INCREMENT BY <increment> ]
    [ ( NOMINVALUE | MINVALUE <minvalue> ) ]
    [ ( NOMAXVALUE | MAXVALUE <maxvalue> ) ]
    [ START WITH <start> ] [ CACHE <cache> | NOCACHE ] [ CYCLE ]
```

### Description

`CREATE SEQUENCE` creates a sequence number generator. This process involves creating and initializing a special single-row table with the specified name. The generator is owned by the user who created it.

If you include a schema name, then the sequence is created in the specified schema. Otherwise it’s created in the current schema. The sequence name differ from the name of any other sequence, table, index, or view in the same schema.

After you create a sequence, use the functions `NEXTVAL` and `CURRVAL` to operate on it. These functions are documented in SQL reference.

### Parameters

`name`
The name (optionally schema-qualified) of the sequence to create.

increment

The optional clause INCREMENT BY increment specifies the value to add to the current sequence value to create a new value. Use a positive value to create ascending sequence and a negative value to create a descending sequence. The default value is 1.

NOMINVALUE | MINVALUE minvalue

The optional clause MINVALUE minvalue determines the minimum value a sequence can generate. The defaults are 1 for ascending and -2^31-1 for descending sequences. You can use the keywords NOMINVALUE to set this behavior to the default.

NOMAXVALUE | MAXVALUE maxvalue

The optional clause MAXVALUE maxvalue determines the maximum value for the sequence. The defaults are 2^31-1 for ascending and -1 for descending sequences. You can use the keywords NOMAXVALUE to set this behavior to the default.

start

The optional clause START WITH start allows the sequence to begin anywhere. The default starting value is minvalue for ascending sequences and maxvalue for descending ones.

cache

The optional clause CACHE cache specifies how many sequence numbers to preallocate and store in memory for faster access. The minimum value is 1, which is also the default. This setting generates only one value at a time, that is, NOCACHE.

CYCLE

The CYCLE option allows the sequence to wrap around when the maxvalue or minvalue is reached by an ascending or descending sequence, respectively. If the limit is reached, the next number generated is minvalue or maxvalue, respectively.

If you omit CYCLE, any calls to NEXTVAL after the sequence reaches its maximum value return an error. You can use the keywords NO CYCLE to use the default behavior. However, this term isn't compatible with Oracle databases.

Notes

Sequences are based on big integer arithmetic, so the range can't exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). Some older platforms might not have compiler support for eight-byte integers. In this case, sequences use regular INTEGER arithmetic (range -2147483648 to +2147483647).

Unexpected results can occur if you use a cache setting greater than 1 for a sequence object to be used concurrently by multiple sessions. Each session allocates and caches successive sequence values during one access to the sequence object and increases the sequence object’s last value accordingly. Then, the next cache-1 that uses NEXTVAL in that session returns the preallocated values without touching the sequence object. So, any numbers allocated but not used in a session are lost when that session ends. “Holes” in the sequence result.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values might be generated out of sequence when all the sessions are considered. For example, with a cache setting of 10, session A might reserve values 1..10 and return NEXTVAL=11. Then session B might reserve values 11..20 and return NEXTVAL=11 before session A generates NEXTVAL=2. Thus, with a cache setting of 1, it’s safe to assume that NEXTVAL values are generated sequentially. With a cache setting greater than 1, assume only that the NEXTVAL values are all distinct, not that they are generated sequentially. Also, the last value reflects the latest value reserved by any session, whether or not it was already returned by NEXTVAL.

Examples

Create an ascending sequence called serial, starting at 101:

CREATE SEQUENCE serial START WITH 101;

Select the next number from this sequence:

SELECT serial.NEXTVAL FROM DUAL;

nextval
---------
101
(1 row)

Create a sequence called supplier_seq with the NOCACHE option:
CREATE SEQUENCE supplier_seq
  MINVALUE 1
  START WITH 1
  INCREMENT BY 1
  NOCACHE;

Select the next number from this sequence:

```
SELECT supplier_seq.NEXTVAL FROM DUAL;
```

nextval
--------
1
(1 row)

See also

ALTER SEQUENCE, DROP SEQUENCE

14.3.5.37 CREATE SYNONYM

Name

CREATE SYNONYM — Define a new synonym.

Synopsis

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM
  [<schema>.]<syn_name>
  FOR <object_schema>.<object_name>[@<dblink_name>];
```

Description

CREATE SYNONYM defines a synonym for certain types of database objects. EDB Postgres Advanced Server supports synonyms for:

- Tables
- Views
- Materialized views
- Sequences
- Stored procedures
- Stored functions
- Types
- Objects that are accessible through a database link
- Other synonyms

Parameters

- **syn_name**
  The name of the synonym. A synonym name must be unique in a schema.

- **schema**
  The name of the schema where the synonym resides. If you don't specify a schema name, the synonym is created in the first existing schema in your search path.

- **object_name**
  The name of the object.
### CREATE TABLE

**Name**

`CREATE TABLE` - Define a new table.

**Synopsis**

```
CREATE [ GLOBAL TEMPORARY | UNLOGGED ] TABLE <table_name>  
  (  
    [ <column_name> <data_type> [ DEFAULT <default_expr> ]  
      [ column_constraint [ ... ] ] [ <table_constraint> ] [,  
      ... ] ]  
  )  
  [ WITH ( ROWIDS = <value> ) ]  
  [ ON COMMIT ( PRESERVE ROWS | DELETE ROWS ) ]  
[ TABLESPACE <tablespace> ]
```
EDB Postgres Advanced Server allows you to create rowids on a foreign table by specifying either the `WITH (ROWIDS)` or `WITH (ROWIDS=true)` option in the `CREATE FOREIGN TABLE` syntax. Specifying the `WITH (ROWIDS);` or `WITH (ROWIDS=true);` option adds a rowid column to a foreign table. For information about `CREATE FOREIGN TABLE`, see the PostgreSQL core documentation.

**Description**

**CREATE TABLE** creates an empty table in the current database. The user who creates the table owns the table.

If you provide a schema name, then the table is created in the specified schema. Otherwise it's created in the current schema. Temporary tables exist in a special schema, so you can't provide a schema name when creating a temporary table. The table name must differ from the name of any table, sequence, index, or view in the same schema.

**CREATE TABLE** also creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables can't have the same name as any existing data type in the same schema.

The **PARALLEL** clause sets the degree of parallelism for a table. If you don't specify the **PARALLEL** clause, the server determines a value based on the relation size.

The **NOPARALLEL** clause resets the parallelism for default execution, and `reloptions` shows the `parallel_workers` parameter as 0.

A table can't have more than 1600 columns. In practice, the effective limit is lower because of tuple-length constraints.

The optional constraint clauses specify constraints or tests that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is a SQL object that helps define the set of valid values in the table in various ways.

You can define table constraints and column constraints. A column constraint is part of a column definition. A table constraint definition isn't tied to a particular column, and it can encompass more than one column. You can also write every column constraint as a table constraint. A column constraint is a notational convenience only if the constraint affects only one column.

**Note**

EDB Postgres Advanced Server allows you to create rowids on a foreign table by specifying either the `WITH (ROWIDS)` or `WITH (ROWIDS=true)` option in the `CREATE FOREIGN TABLE` syntax. Specifying the `WITH (ROWIDS);` or `WITH (ROWIDS=true);` option adds a rowid column to a foreign table. For information about `CREATE FOREIGN TABLE`, see the PostgreSQL core documentation.
Parameters

GLOBAL TEMPORARY

Creates the table as a temporary table. Temporary tables are dropped at the end of a session or, optionally, at the end of the current transaction. (See the ON COMMIT parameter.) Existing permanent tables with the same name aren't visible to the current session while the temporary table exists unless you reference them with schema-qualified names. In addition, temporary tables aren't visible outside the session in which you created them. This aspect of global temporary tables isn't compatible with Oracle databases. Any indexes created on a temporary table are also temporary.

UNLOGGED

Creates the table as an unlogged table. The data written to unlogged tables isn't written to the write-ahead log (WAL), making them faster than an ordinary table. Indexes created on an unlogged table are unlogged. The contents of an unlogged table aren't replicated to a standby server. The unlogged table is not crash-safe. It's truncated after a crash or shutdown.

table_name

The name (optionally schema-qualified) of the table to create.

column_name

The name of a column to create in the new table.

data_type

The data type of the column. This can include array specifiers. For more information on the data types included with EDB Postgres Advanced Server, see SQL reference.

DEFAULT default_expr

The DEFAULT clause assigns a default data value for the column whose column definition it appears in. The value is any variable-free expression. Subqueries and cross references to other columns in the current table aren't allowed. The data type of the default expression must match the data type of the column.

The default expression is used in any insert operation that doesn't specify a value for the column. If you don't specify a default for a column, then the default is null.

CONSTRAINT constraint_name

An optional name for a column or table constraint. If you don't specify one, the system generates a name.

NOT NULL

The column can't contain null values.

NULL

The column can contain null values. This is the default.

This clause is available only for compatibility with nonstandard SQL databases. We don't recommend using it in new applications.

UNIQUE — Column constraint.

UNIQUE (column_name [, ...] ) — Table constraint.

The UNIQUE constraint specifies that a group of one or more distinct columns of a table can contain only unique values. The behavior of the unique table constraint is the same as that for column constraints. However, the unique table constraint can span multiple columns.

For the purpose of a unique constraint, null values aren't considered equal.

Each unique table constraint must name a set of columns that's different from the set of columns named by any other unique or primary key constraint defined for the table. Otherwise it's the same constraint listed twice.

PRIMARY KEY — Column constraint.

PRIMARY KEY ( column_name [, ...] ) — Table constraint.

The primary key constraint specifies that any columns of a table can contain only unique, non-duplicated, non-null values. Technically, PRIMARY KEY is a combination of UNIQUE and NOT NULL. However, identifying a set of columns as primary key also provides metadata about the design of the schema. A primary key implies that other tables might rely on this set of columns as a unique identifier for rows.

You can specify only one primary key for a table, whether as a column constraint or a table constraint.

Use the primary key constraint to name a set of columns that's different from other sets of columns named by any unique constraint defined for the same table.

CHECK (expression)
The **CHECK** clause specifies an expression producing a Boolean result that new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to **TRUE** or "unknown" succeed. If any row of an insert or update operation produces a **FALSE** result, an error exception is raised and the insert or update doesn't alter the database. A check constraint specified as a column constraint must reference only that column's value. An expression appearing in a table constraint can reference multiple columns.

Currently, **CHECK** expressions can't contain subqueries or refer to variables other than columns of the current row.

### REFERENCES

`REFERENCES reftable [ ( refcolumn ] [ ON DELETE action ] -- Column constraint.`

`FOREIGN KEY ( column [, ...] ) REFERENCES reftable [ ( refcolumn [, ...] ) ] [ ON DELETE action ] -- Table constraint.`

These clauses specify a foreign-key constraint, which requires that a group of columns of the new table must contain only values that match values in the referenced columns of some row of the referenced table. If you omit `refcolumn`, the primary key of the `reftable` is used. The referenced columns must be the columns of a unique or primary key constraint in the referenced table.

In addition, when the data in the referenced columns changes, certain actions are performed on the data in this table's columns. The **ON DELETE** clause specifies the action to perform when a referenced row in the referenced table is being deleted. You can't defer referential actions even if the constraint is deferrable. Here are the following possible actions for each clause:

- **CASCADE**
  - Delete any rows referencing the deleted row, or update the value of the referencing column to the new value of the referenced column.

- **SET NULL**
  - Set the referencing columns to `NULL`.

  If the referenced columns change frequently, we recommend adding an index to the foreign key column so that referential actions associated with the foreign key column are performed more efficiently.

- **DEFERRABLE**
- **NOT DEFERRABLE**
  - This parameter controls whether you can defer the constraint. A constraint that isn’t deferrable is checked immediately after every command. You can postpone checking constraints that are deferrable until the end of the transaction using the **SET CONSTRAINTS** command. **NOT DEFERRABLE** is the default. Only foreign key constraints currently accept this clause. All other constraint types aren't deferrable.

- **INITIALLY IMMEDIATE**
- **INITIALLY DEFERRED**
  - If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is **INITIALLY IMMEDIATE**, it's checked after each statement. This is the default. If the constraint is **INITIALLY DEFERRED**, it's checked only at the end of the transaction. You can alter the constraint check time with the **SET CONSTRAINTS** command.

`WITH ( ROWIDS [= value ] )`

The `ROWIDS` option for a table includes `value` equals to **TRUE/ON/1** or **FALSE/OFF/0**. When set to **TRUE/ON/1**, a `ROWID` column is created in the new table. `ROWID` is an auto-incrementing value based on a sequence that starts with `1` and assigned to each row of a table. The default value is always **TRUE**.

By default, a unique index is created on a `ROWID` column. The **ALTER** and **DROP** operations are restricted on a `ROWID` column.

### ON COMMIT

You can control the behavior of temporary tables at the end of a transaction block using **ON COMMIT**. The options are:

- **PRESEVER ROWS**
  - No special action is taken at the ends of transactions. This is the default behavior. This aspect isn't compatible with Oracle databases. The Oracle default is **DELETE ROWS**.

- **DELETE ROWS**
  - All rows in the temporary table are deleted at the end of each transaction block. Essentially, an automatic **TRUNCATE** is done at each commit.

### TABLESPACE

The **TABLESPACE** clause allows selection of the tablespace in which the index associated with a **UNIQUE** or **PRIMARY KEY** constraint is created. If not specified, **default_tablespace** is used, or the database's default tablespace if **default_tablespace** is an empty string.

This clause allows selection of the tablespace in which the index associated with a **UNIQUE** or **PRIMARY KEY** constraint is created. If not specified, **default_tablespace** is used, or the database's default tablespace if **default_tablespace** is an empty string.
If you specify the `create_index_statement` option, the database server creates an index enabling unique or primary key constraints. The columns specified in the constraint and the columns of an index must be the same, but their order of appearance might differ.

**PARALLEL**

Include the `PARALLEL` clause to specify the degree of parallelism for the table. Set the `parallel_workers` parameter equal to the degree of parallelism to perform a parallel scan of a table. Alternatively, if you specify `PARALLEL` but don't include a degree of parallelism, an index uses default parallelism.

**NOPARALLEL**

Specify `NOPARALLEL` for default execution.

**integer**

The `integer` indicates the degree of parallelism, which is a number of `parallel_workers` used in the parallel operation to perform a parallel scan on a table.

**Notes**

EDB Postgres Advanced Server creates an index for each unique constraint and primary key constraint to enforce the uniqueness. Thus, you don't need to create an explicit index for primary key columns. For more information, see `CREATE INDEX`.

**Examples**

Create table `dept` and table `emp`:

```sql
CREATE TABLE dept (
    deptno NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname VARCHAR2(14),
    loc VARCHAR2(13)
);
CREATE TABLE emp (
    empno NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename VARCHAR2(10),
    job VARCHAR2(9),
    mgr NUMBER(4),
    hiredate DATE,
    sal NUMBER(7,2),
    comm NUMBER(7,2),
    deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk
        REFERENCES dept(deptno)
);

Define a unique table constraint for the table `dept`. You can define unique table constraints on one or more columns of the table.

```sql
CREATE TABLE dept (
    deptno NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc VARCHAR2(13)
);

Define a check column constraint:

```sql
CREATE TABLE emp (
    empno NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename VARCHAR2(10),
    job VARCHAR2(9),
    mgr NUMBER(4),
    hiredate DATE,
    sal NUMBER(7,2),
    comm NUMBER(7,2),
    deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk
        REFERENCES dept(deptno)
);```
Define a table constraint:

```sql
CREATE TABLE emp
(
  empno  NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename  VARCHAR2(10),
  job    VARCHAR2(9),
  mgr    NUMBER(4),
  hiredate DATE,
  sal    NUMBER(7,2),
  comm   NUMBER(7,2),
  deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk REFERENCES dept(deptno),
  CONSTRAINT new_emp_ck CHECK (ename IS NOT NULL AND empno > 7000)
);```

Define a check table constraint:

```sql
CREATE TABLE jobhist
(
  empno  NUMBER(4) NOT NULL,
  startdate DATE NOT NULL,
  enddate  DATE,
  job    VARCHAR2(9),
  sal    NUMBER(7,2),
  comm   NUMBER(7,2),
  deptno NUMBER(2),
  chgdsc VARCHAR2(88),
  CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate)
);```

This example assigns a literal constant default value for the column `job` and makes the default value of `hiredate` the date when the row is inserted.

```sql
CREATE TABLE emp
(
  empno  NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename  VARCHAR2(10),
  job    VARCHAR2(9) DEFAULT 'SALESMAN',
  mgr    NUMBER(4),
  hiredate DATE DEFAULT SYSDATE,
  sal    NUMBER(7,2),
  comm   NUMBER(7,2),
  deptno NUMBER(2) CONSTRAINT emp_ref_dept_fk REFERENCES dept(deptno),
);```

Create table `dept` in tablespace `diskvol1`:

```sql
CREATE TABLE dept
(
  deptno  NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname   VARCHAR2(14),
  loc     VARCHAR2(13)
) TABLESPACE diskvol1;
```

This `PARALLEL` example creates a table `sales` and sets a degree of parallelism to 6:

```sql
CREATE TABLE dept
(
  deptno  NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname   VARCHAR2(14),
  loc     VARCHAR2(13)
) TABLESPACE diskvol1;
```
This example creates a table named `sales_order` and sets a degree of parallelism to 0:

```
CREATE TABLE sales_order (deptno number) NOPARALLEL WITH (FILLFACTOR=66);
```

This example creates a table named `dept`. The definition creates a unique key on the `dname` column. The constraint `dept_dname_uq` identifies the `dname` column as a unique key. The preceding statement includes the `USING_INDEX` clause, which explicitly creates an index on a table `dept` with the index statement enabling the unique constraint.

```
CREATE TABLE dept
(
  deptno       NUMBER(2) NOT NULL,
  dname        VARCHAR2(14),
  loc          VARCHAR2(13),
  CONSTRAINT dept_dname_uq UNIQUE(dname)
  USING INDEX (CREATE UNIQUE INDEX idx_dept_dname_uq ON dept(dname))
);
```

This example creates a table named `emp`. The definition creates a primary key on the `ename` column. The `emp_ename_pk` constraint identifies the `ename` column as a primary key of the `emp` table. The preceding statement includes the `USING_INDEX` clause, which explicitly creates an index on a table `emp` with the index statement enabling the primary constraint.

```
CREATE TABLE emp
(
  empno       NUMBER(4) NOT NULL,
  ename       VARCHAR2(18),
  job         VARCHAR2(9),
  sal         NUMBER(7,2),
  deptno      NUMBER(2),
  CONSTRAINT emp_ename_pk PRIMARY KEY (ename)
  USING INDEX (CREATE INDEX idx_emp_ename_pk ON emp (ename))
);
```

See also

ALTER TABLE, DROP TABLE

14.3.5.39 CREATE TABLE AS

Name

`CREATE TABLE AS` - Define a new table from the results of a query.

Synopsis

```
CREATE [ GLOBAL TEMPORARY ] TABLE <table_name> [(<column_name>, ..., )] [ ON COMMIT ( PRESERVE ROWS | DELETE ROWS ) ] [ TABLSPACE tablespace ] AS <query>
```

Description

`CREATE TABLE AS` creates a table and fills it with data computed by a `SELECT` command. The table columns have the names and data types associated with the output columns of the `SELECT`. 
You can override the column names by providing an explicit list of new column names.

CREATE TABLE AS is similar to creating a view. However, it differs because it creates a table and evaluates the query once to fill the new table initially. The new table doesn't track later changes to the source tables of the query. In contrast, a view reevaluates its defining SELECT statement whenever you query it.

Parameters

GLOBAL TEMPORARY

Specify to create the table as a temporary table. For details, see CREATE TABLE.

table_name

The name (optionally schema-qualified) of the table to create.

column_name

The name of a column in the new table. If you don't provide column names, the names are taken from the output column names of the query.

query

A query statement (a SELECT command). For a description of the allowed syntax, see SELECT.

14.3.5.40 CREATE TRIGGER

Name

CREATE TRIGGER — Define a simple trigger.

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER
  <name>
  [ BEFORE | AFTER | INSTEAD OF ]
  [ ( INSERT | UPDATE | DELETE | TRUNCATE )
    [ OR ( INSERT | UPDATE | DELETE | TRUNCATE ) ] [, ...
    ON <table>
    [ REFERENCING ( OLD AS <old> | NEW AS <new> )
      [, ...]
    ]
    [ FOR EACH ROW ]
    [ WHEN <condition> ]
    []
    DECLARE
    [ PRAGMA AUTONOMOUS_TRANSACTION;]
    <declaration>; [, ...]
  ]
BEGIN
  <statement>; [, ...]
  [ EXCEPTION
    [ WHEN <exception> [ OR <exception> ] [, ...
      THEN
      <statement>; [, ...] ] [, ...
    ]
  ]
END
```

Name
CREATE TRIGGER — Define a compound trigger.

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER
  <name>[
    FOR ( INSERT | UPDATE | DELETE | TRUNCATE )
    [ OR ( INSERT | UPDATE | DELETE | TRUNCATE ) ] [,
    ...]
    ON <table>
    ...]
  WHEN <condition>
  ]
  TRIGGER
    [ <private_declaration>; ]
    ...
    [ <procedure_or_function_definition> ]
    ...
    <compound_trigger_definition>
  END
```

Where `private_declaration` is an identifier of a private variable that any procedure or function can access. There can be zero, one, or more private variables. `private_declaration` can be any of the following:

- Variable declaration
- Record declaration
- Collection declaration
- `REF CURSOR` and cursor variable declaration
- `TYPE` definitions for records, collections, and `REF CURSOR` cursors
- Exception
- Object variable declaration

Where `procedure_or_function_definition` :=

```
procedure_definition | function_definition
```

Where `procedure_definition` :=

```
PROCEDURE proc_name[ argument_list ]
  [ options_list ]
  { IS | AS }
  procedure_body
  END [ proc_name ];
```

Where `procedure_body` :=

```
[ declaration; ] [,,
  ...]
BEGIN
  statement;
  [...] EXCEPTION
    { WHEN exception [OR exception] [...] } THEN statement;
  }
  [...] ]
```

Where `function_definition` :=

```
FUNCTION func_name [ argument_list ]
  RETURN rettype [ DETERMINISTIC ]
  [ options_list ]
  { IS | AS }
  function_body
  END [ func_name ];
```
Where function_body :=

[ declaration; ] [, 
...]
BEGIN
statement;
[...]
[ EXCEPTION
  { WHEN exception [ OR exception ] [...] THEN statement; }
[...]
]
]

Where compound_trigger_definition :=

{ compound_trigger_event ) { IS | AS
}
compound_trigger_body
END [ compound_trigger_event ] [ ...
]

Where compound_trigger_event :=

[ BEFORE STATEMENT | BEFORE EACH ROW | AFTER EACH ROW | AFTER STATEMENT | INSTEAD OF EACH ROW
]

Where compound_trigger_body :=

[ declaration; ] [, 
...]
BEGIN
  statement;
[...]
  [ EXCEPTION
  { WHEN exception [OR exception] [...] THEN statement;
  }
[...]
]

Description

CREATE TRIGGER defines a new trigger. CREATE OR REPLACE TRIGGER either creates a trigger or replaces an existing definition.

If you’re using the CREATE TRIGGER keywords to create a new trigger, the name of the new trigger can’t match any existing trigger defined on the same table. Triggers are created in the same schema as the table where the triggering event is defined.

If you’re updating the definition of an existing trigger, use the CREATE OR REPLACE TRIGGER keywords.

When you use syntax that’s compatible with Oracle to create a trigger, the trigger runs as a SECURITY DEFINER function.

Parameters

name

The name of the trigger to create.

BEFORE | AFTER

Determines whether the trigger is fired before or after the triggering event.

INSTEAD OF

Modifies an updatable view. The trigger executes to update the underlying tables appropriately. The INSTEAD OF trigger executes for each row of the view that’s updated or modified.

INSERT | UPDATE | DELETE | TRUNCATE

Defines the triggering event.
**table**

The name of the table or view on which the triggering event occurs.

**condition**

A Boolean expression that determines if the trigger executes. If `condition` evaluates to `TRUE`, the trigger fires.

- If the simple trigger definition includes the `FOR EACH ROW` keywords, the `WHEN` clause can refer to columns of the old or new row values by writing `OLD.column_name` or `NEW.column_name`. INSERT triggers can't refer to `OLD`, and DELETE triggers can't refer to `NEW`.
- If the compound trigger definition includes a statement-level trigger having a `WHEN` clause, then the trigger executes without evaluating the expression in the `WHEN` clause. Similarly, if a compound trigger definition includes a row-level trigger having a `WHEN` clause, then the trigger executes if the expression evaluates to `TRUE`.
- If the trigger includes the `INSTEAD OF` keywords, it can't include a `WHEN` clause. A `WHEN` clause can't contain subqueries.

**REFERENCING { OLD AS old | NEW AS new } ...**

`REFERENCING` clause to reference old rows and new rows. This clause is restricted in that you can replace `old` only with an identifier named `old` or any equivalent that's saved in all lowercase. Examples include `REFERENCING OLD AS old`, `REFERENCING OLD AS OLD`, and `REFERENCING OLD AS "old"`. Also, you can replace `new` only with an identifier named `new` or any equivalent that's saved in all lowercase. Examples include `REFERENCING NEW AS new`, `REFERENCING NEW AS NEW`, and `REFERENCING NEW AS "new"`.

You can specify one or both phrases `OLD AS old` and `NEW AS new` in the `REFERENCING` clause, such as `REFERENCING NEW AS New OLD AS Old`.

This clause isn't compatible with Oracle databases in that you can't use identifiers other than `old` or `new`.

**FOR EACH ROW**

Determines whether to fire the trigger once for every row affected by the triggering event or just once per SQL statement. If you specify this parameter, the trigger is fired once for every affected row (row-level trigger). Otherwise the trigger is a statement-level trigger.

**PRAGMA AUTONOMOUS_TRANSACTION**

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the trigger as an autonomous transaction.

**declaration**

A variable, type, `REF CURSOR`, or subprogram declaration. If you include subprogram declarations, declare them after all other variable, type, and `REF CURSOR` declarations.

**statement**

An SPL program statement. A `DECLARE - BEGIN - END` block is considered an SPL statement. Thus, the trigger body can contain nested blocks.

**exception**

An exception condition name, such as `NO_DATA_FOUND` and `OTHERS`.

**Examples**

This example shows a statement-level trigger that fires after the triggering statement (insert, update, or delete on table `emp`) executes.

```sql
CREATE OR REPLACE TRIGGER user_audit_trig
AFTER INSERT OR UPDATE OR DELETE ON
emp
DECLARE
  v_action VARCHAR2(24);
BEGIN
  IF INSERTING THEN
    v_action := ' added employee(s) on '
 ELSIF UPDATING THEN
    v_action := ' updated employee(s) on '
 ELSIF DELETING THEN
    v_action := ' deleted employee(s) on '
  END IF;
```
CREATE OR REPLACE TRIGGER emp_sal_trig
BEFORE DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
DECLARE
  sal_diff NUMBER;
BEGIN
  IF INSERTING THEN
    DBMS_OUTPUT.PUT_LINE('Inserting employee ' || NEW.empno);
    DBMS_OUTPUT.PUT_LINE('..New salary: ' || NEW.sal);
    END IF;
  IF UPDATING THEN
    sal_diff := NEW.sal - OLD.sal;
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || OLD.sal);
    DBMS_OUTPUT.PUT_LINE('..Raise     : ' || sal_diff);
    END IF;
  IF DELETING THEN
    DBMS_OUTPUT.PUT_LINE('Deleting employee ' || OLD.empno);
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || OLD.sal);
    END IF;
END;

CREATE TABLE emp(EMPNO INT, ENAME TEXT, SAL INT, DEPTNO INT);

CREATE OR REPLACE TRIGGER hr_trigger
FOR INSERT OR UPDATE OR DELETE ON emp
COMPOUND TRIGGER
  -- Global declaration.
  var_sal NUMBER := 10000;

  BEFORE STATEMENT IS
    var_sal := var_sal + 1808;
    DBMS_OUTPUT.PUT_LINE('Before Statement: ' || var_sal);
    END BEFORE STATEMENT;

  BEFORE EACH ROW IS
    var_sal := var_sal + 1808;
    DBMS_OUTPUT.PUT_LINE('Before Each Row: ' || var_sal);
    END BEFORE EACH ROW;

  AFTER EACH ROW IS
    var_sal := var_sal + 1808;
    DBMS_OUTPUT.PUT_LINE('After Each Row: ' || var_sal);
    END AFTER EACH ROW;
The `INSERT` statement produces the following output:

```
INSERT INTO emp (EMPNO, ENAME, SAL, DEPTNO) VALUES(1111, 'SMITH', 10000, 20);
```

The `UPDATE` statement updates the employee salary record, setting the salary to 15000 for a specific employee number:

```
UPDATE emp SET SAL = 15000 where EMPNO = 1111;
```

The `DELETE` statement deletes the employee salary record:

```
DELETE from emp where EMPNO = 1111;
```

The `TRUNCATE` statement removes all the records from the `emp` table:

```
CREATE OR REPLACE TRIGGER hr_trigger
  FOR TRUNCATE ON emp
  COMPOUND TRIGGER
    -- Global declaration.
    var_sal NUMBER := 10000;
    BEFORE STATEMENT IS
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('Before Statement: ' || var_sal);
    END BEFORE STATEMENT;
  AFTER STATEMENT IS
  BEGIN
```

Output:
```
Trigger created.
```

The record is inserted into the `emp` table.

The `INSERT` statement produces the following output:

```
Before Statement: 11000
Before each row: 12000
After each row: 13000
After statement: 14000
```

The `UPDATE` statement produces the following output:

```
sql OUTPUT
Before Statement: 11000
Before each row: 12000
After each row: 13000
After statement: 14000
```

The `DELETE` statement executes the trigger:

```
```

The `DELETE` statement produces the following output:

```
Before Statement: 11000
Before each row: 12000
After each row: 13000
After statement: 14000
```

The `TRUNCATE` statement removes all the records from the `emp` table:

```
CREATE OR REPLACE TRIGGER hr_trigger
  FOR TRUNCATE ON emp
  COMPOUND TRIGGER
    -- Global declaration.
    var_sal NUMBER := 10000;
    BEFORE STATEMENT IS
    var_sal := var_sal + 1000;
    DBMS_OUTPUT.PUT_LINE('Before Statement: ' || var_sal);
    END BEFORE STATEMENT;
  AFTER STATEMENT IS
  BEGIN
```

Output:
```
Trigger created.
```

The `UPDATE` statement updates the employee salary record, setting the salary to 15000 for a specific employee number:
The TRUNCATE statement produces the following output:

```
TRUNCATE emp;
```

Before Statement: 11000
After statement: 12000
TRUNCATE TABLE

**Note**

You can use the TRUNCATE statement only at a BEFORE STATEMENT or AFTER STATEMENT timing point.

This example creates a compound trigger named hr_trigger on the emp table. It is a WHEN condition that checks and prints employee salary whenever an INSERT, UPDATE, or DELETE statement affects the emp table. The database evaluates the WHEN condition for a row-level trigger, and the trigger executes once per row if the WHEN condition evaluates to TRUE. The statement-level trigger executes regardless of the WHEN condition.

```
CREATE OR REPLACE TRIGGER hr_trigger
    FOR INSERT OR UPDATE OR DELETE ON emp
    REFERENCING NEW AS new OLD AS old
    WHEN (old.sal > 5000 OR new.sal < 8000)
    COMPOUND TRIGGER
    BEFORE STATEMENT IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Before Statement');
    END BEFORE STATEMENT;
    BEFORE EACH ROW IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Before Each Row: 
            :NEW.sal');
    END BEFORE EACH ROW;
    AFTER EACH ROW IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('After Each Row: 
            :OLD.sal');
    END AFTER EACH ROW;
    AFTER STATEMENT IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('After Statement');
    END AFTER STATEMENT;
    END
```

Insert the record into table emp:

```
INSERT INTO emp(EMPNO, ENAME, SAL, DEPTNO) VALUES(1111, 'SMITH', 1600, 20);
```

The INSERT statement produces the following output:

```
__OUTPUT__
Before Statement
Before Each Row: 1600
After Each Row: 1600
After Statement
INSERT 0 1
```

The UPDATE statement updates the employee salary record, setting the salary to 7500:

```
var_sal := var_sal + 1000;
DBMS_OUTPUT.PUT_LINE('After Statement: ' || var_sal);
END AFTER STATEMENT;
hr_trigger;
```

Output: Trigger created.

```
TRUNCATE emp;
```

CREATE OR REPLACE TRIGGER hr_trigger
    FOR INSERT OR UPDATE OR DELETE ON emp
    REFERENCING NEW AS new OLD AS old
    WHEN (old.sal > 5000 OR new.sal < 8000)
    COMPOUND TRIGGER
    BEFORE STATEMENT IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Before Statement');
    END BEFORE STATEMENT;
    BEFORE EACH ROW IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Before Each Row: 
            :OLD.sal');
    END BEFORE EACH ROW;
    AFTER EACH ROW IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('After Each Row: 
            :OLD.sal');
    END AFTER EACH ROW;
    AFTER STATEMENT IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('After Statement');
    END AFTER STATEMENT;
    END
```

Insert the record into table emp:

```
INSERT INTO emp(EMPNO, ENAME, SAL, DEPTNO) VALUES(1111, 'SMITH', 1600, 20);
```

The INSERT statement produces the following output:

```
__OUTPUT__
Before Statement
Before Each Row: 1600
After Each Row: 1600
After Statement
INSERT 0 1
```

The UPDATE statement updates the employee salary record, setting the salary to 7500:
The `UPDATE` statement produces the following output:

```
| empno | ename | sal  | deptno |
|--------+-------+------+--------|
| 1111   | SMITH | 7500 | 20     |
```

(1 row)

The `DELETE` statement deletes the employee salary record:

```
DELETE from emp where EMPNO = 1111;
```

The `DELETE` statement produces the following output:

```
| empno | ename | sal  | deptno |
|--------+-------+------+--------|
```

(0 rows)

See also

`ALTER TRIGGER`, `DROP TRIGGER`

### 14.3.5.41 CREATE TYPE

**Name**

`CREATE TYPE` — Define a new user-defined type. The new type can be an object type, a collection type (a nested table type or a varray type), or a composite type.

**Synopsis**

**Object Type**

```
CREATE [ OR REPLACE ] TYPE <name> 
  [ AUTHID ( DEFINER | CURRENT_USER ) ]
  [ IS | AS ] OBJECT
  {{ <attribute> ( <datatype> | <objtype> | <collecttype> )
    [...]
    [ <method_spec> ] [, [...]]
  } [ NOT ] [ FINAL | INSTANTIABLE ]
```

Where `method_spec` is:...
Where _subprogram_spec_ is:

```sql
CREATE TYPE defines a new data type. The types that you can create are an object type, a nested table type, a varray type, or a composite type. Nested table and varray types belong to the category of types known as _collections_. Composite types aren't compatible with Oracle databases. However, SPL programs can access composite types along with other types.

Parameters

name
```
The name (optionally schema-qualified) of the type to create.

```
DEFINER | CURRENT_USER
```

Specifies whether to use the privileges of the object type owner (DEFINER) or of the current user executing a method in the object type (CURRENT_USER) to determine whether access is allowed to database objects referenced in the object type. DEFINER is the default.

```
attribute
```

The name of an attribute in the object type or composite type.

```
datatype
```

The data type that defines an attribute of the object type or composite type, or the elements of the collection type that’s being created.

```
objtype
```

The name of an object type that defines an attribute of the object type or the elements of the collection type that’s being created.

```
collecttype
```

The name of a collection type that defines an attribute of the object type or the elements of the collection type that’s being created.

```
FINAL
NOT FINAL
```

For an object type, specifies whether you can derive a subtype from the object type. FINAL (you can’t derive a subtype from the object type) is the default.

For `method_spec`, specifies whether you can override the method in a subtype. NOT FINAL (you can override the method in a subtype) is the default.

```
INSTANTIABLE
NOT INSTANTIABLE
```

For an object type, specifies whether you can create an object instance of this object type. INSTANTIABLE (you can create an instance of this object type) is the default. If you specify NOT INSTANTIABLE, then you must specify NOT FINAL as well. If `method_spec` for any method in the object type contains the NOT INSTANTIABLE qualifier, then the object type must be defined with NOT INSTANTIABLE and NOT FINAL following the closing parenthesis of the object type specification.

For `method_spec`, specifies whether the object type definition provides an implementation for the method. INSTANTIABLE (the CREATE TYPE BODY command for the object type provides the implementation of the method) is the default. If you specify NOT INSTANTIABLE, then the CREATE TYPE BODY command for the object type can’t contain the implementation of the method.

```
OVERRIDING
```

If you specify OVERRIDING, `method_spec` overrides an identically named method with the same number of identically named method arguments with the same data types, in the same order, and the same return type (if the method is a function) as defined in a supertype.

```
MEMBER
STATIC
```

Specify MEMBER if the subprogram operates on an object instance. Specify STATIC if the subprogram operates independently of any particular object instance.

```
proc_name
```

The name of the procedure to create.

```
SELF [ IN | IN OUT ] name
```

For a member method, there is an implicit, built-in parameter named SELF whose data type is that of the object type being defined. SELF refers to the object instance that’s currently invoking the method. You can explicitly declare SELF as an IN or IN OUT parameter in the parameter list. If explicitly declared, SELF must be the first parameter in the parameter list. If you don’t explicitly declare SELF, its parameter mode defaults to IN OUT for member procedures and IN for member functions.

```
argname
```

The name of an argument. The argument is referenced by this name in the method body.

```
argtype
```

The data types of the method’s arguments. The argument types can be a base data type or a user-defined type such as a nested table or an object type. Don’t specify a length for any base type. For
example, specify VARCHAR2, not VARCHAR2(10).

DEFAULT value

Supplies a default value for an input argument if you don't supply one in the method call. You can't specify DEFAULT for arguments with modes IN OUT or OUT.

func_name

The name of the function to create.

rettype

The return data type, which can be any of the types listed for argtype. Don't specify a length for rettype.

maxsize

The maximum number of elements permitted in the varray.

Examples

Creating an object type

Create object type addr_obj_typ:

```sql
CREATE OR REPLACE TYPE addr_obj_typ AS OBJECT
(
  street VARCHAR2(30),
  city VARCHAR2(28),
  state CHAR(2),
  zip NUMBER(5)
);
```

Create object type emp_obj_typ that includes a member method display_emp:

```sql
CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT
(
  empno NUMBER(4),
  ename VARCHAR2(28),
  addr ADDR_OBJ_TYP,
MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
);
```

Create object type dept_obj_typ that includes a static method get_dname:

```sql
CREATE OR REPLACE TYPE dept_obj_typ AS OBJECT
(
  deptno NUMBER(2),
  STATIC FUNCTION get_dname (p_deptno IN NUMBER) RETURN VARCHAR2,
MEMBER PROCEDURE display_dept
);
```

Creating a collection type

Create a nested table type budget_tbl_typ of data type NUMBER(8,2):

```sql
CREATE OR REPLACE TYPE budget_tbl_typ IS TABLE OF NUMBER(8,2);
```

Creating and using a composite type

This example uses a composite type accessed from an anonymous block.

This example creates the composite type:
CREATE OR REPLACE TYPE emphist_typ AS
(
  empno NUMBER(4),
  ename VARCHAR2(10),
  hiredate DATE,
  job VARCHAR2(9),
  sal NUMBER(7,2)
);

DECLARE
  v_emphist EMPHIST_TYP;
BEGIN
  v_emphist.empno := 9001;
  v_emphist.ename := 'SMITH';
  v_emphist.hiredate := '01-AUG-17';
  v_emphist.job := 'SALESMAN';
  v_emphist.sal := 8000.00;
  DBMS_OUTPUT.PUT_LINE(' EMPNO: ' ||
                        v_emphist.empno);
  DBMS_OUTPUT.PUT_LINE(' ENAME: ' ||
                        v_emphist.ename);
  DBMS_OUTPUT.PUT_LINE(' HIREDATE: ' ||
                        v_emphist.hiredate);
  DBMS_OUTPUT.PUT_LINE(' JOB: ' ||
                        v_emphist.job);
  DBMS_OUTPUT.PUT_LINE(' SAL: ' ||
                        v_emphist.sal);
END;

CREATE OR REPLACE TYPE salhist_typ AS
(
  startdate DATE,
  job VARCHAR2(9),
  sal NUMBER(7,2)
);

CREATE OR REPLACE PACKAGE emp_salhist
IS
  PROCEDURE fetch_emp
  (p_empno IN NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY
emp_salhist
IS
  TYPE emprec_typ IS RECORD
  (empno NUMBER(4),
   ename VARCHAR(10),
   salhist SALHIST_TYP);
  TYPE emp_arr_typ IS TABLE OF emprec_typ INDEX BY
  BINARY_INTEGER;
  emp_arr emp_arr_typ;

This code shows the anonymous block that accesses the composite type:

This example uses a composite type accessed from a user-defined record type declared in a package body.

The following creates the composite type:

This example defines the type declared in the previous example:

The following defines the package specification:

The following defines the package body:
In the declaration of the TYPE emprec_typ IS RECORD data structure in the package body, the salhist field is defined with the SALHIST_TYP composite type created by the CREATE TYPE statement.

The associative array definition TYPE emp_arr_typ IS TABLE OF emprec_typ references the record type data structure emprec_typ that includes the field salhist defined with the SALHIST_TYP composite type.

This example invokes the package procedure that loads the array from a join of the emp and jobhist tables and then displays the array content:

```sql
PROCEDURE fetch_emp ( p_empno IN NUMBER ) IS
  CURSOR emp_cur IS
  SELECT e.empno, e.ename, h.startdate, h.job, h.sal
  FROM emp e, jobhist h
  WHERE e.empno = p_empno
  AND e.empno = h.empno;
  i INTEGER := 0;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    STARTDATE  JOB         SAL ');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   --------- ');
    FOR r_emp IN emp_cur LOOP
      i := i + 1;
      emp_arr(i) := (r_emp.empno, r_emp.ename, (r_emp.startdate, r_emp.job, r_emp.sal));
    END LOOP;
    FOR i IN 1..emp_arr.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE(emp_arr(i).empno || ' ' || RPAD(emp_arr(i).ename,8) || ' ' || TO_CHAR(emp_arr(i).salhist.startdate,'DD-MON-YY') || ' ' || RPAD(emp_arr(i).salhist.job,10) || ' ' || TO_CHAR(emp_arr(i).salhist.sal,'99,999.99'));
    END LOOP;
  END;
END;
END;
```

In the declaration of the TYPE emprec_typ IS RECORD data structure in the package body, the salhist field is defined with the SALHIST_TYP composite type created by the CREATE TYPE salhist_typ statement.

The associative array definition TYPE emp_arr_typ IS TABLE OF emprec_typ references the record type data structure emprec_typ that includes the field salhist defined with the SALHIST_TYP composite type.

This example invokes the package procedure that loads the array from a join of the emp and jobhist tables and then displays the array content:

```sql
EXEC emp_salhist.fetch_emp(7788);
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>STARTDATE</th>
<th>JOB</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>19-APR-87</td>
<td>CLERK</td>
<td>1,000.00</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>13-APR-88</td>
<td>CLERK</td>
<td>1,040.00</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>05-MAY-90</td>
<td>ANALYST</td>
<td>3,000.00</td>
</tr>
</tbody>
</table>

EDB-SPL Procedure successfully completed

Notes

For packages only, you can include a composite type in a user-defined record type declared with the TYPE IS RECORD statement in the package specification or package body. This nested structure isn't permitted in other SPL programs such as functions, procedures, and triggers.

In the CREATE TYPE command, if you include a schema name, then the type is created in the specified schema. Otherwise it's created in the current schema. The name of the new type can't match any existing type in the same schema unless you want to update the definition of an existing type. In that case, use CREATE OR REPLACE TYPE.

You can't use the OR REPLACE option to add, delete, or modify the attributes of an existing object type. Use the DROP TYPE command to first delete the existing object type. You can use the OR REPLACE option to add, delete, or modify the methods in an existing object type.

You can use the PostgreSQL form of the ALTER TYPE ALTER ATTRIBUTE command to change the data type of an attribute in an existing object type. However, the ALTER TYPE command can't add or delete attributes in the object type.
The user that creates the type becomes the owner of the type.

See also

CREATE TYPE BODY, DROP TYPE

14.3.5.42 CREATE TYPE BODY

Name

CREATE TYPE BODY — Define a new object type body.

Synopsis

```
CREATE [ OR REPLACE ] TYPE BODY
  <name>
  ( [ IS | AS ]
    <method_spec>
  )
END
```

Where `method_spec` is:

```
subprogram_spec
```

Where `subprogram_spec` is:

```
{ MEMBER | STATIC }
{ PROCEDURE <proc_name>
  [ [ SELF [ IN | IN OUT ] <name> ]
    [ , <argname> [ IN | IN OUT | OUT ] ]
  <argtype>
    [ DEFAULT <value> ]
  ]
  ...)
  [ IS | AS ]
  <program_body>
END;
| FUNCTION <func_name>
[ [ SELF [ IN | IN OUT ] <name> ]
  [ , <argname> [ IN | IN OUT | OUT ] ]
  <argtype>
    [ DEFAULT <value> ]
  ]
  ...)
  RETURN <rettype>
{ IS | AS ]
  <program_body>
END;
}
```

Description

CREATE TYPE BODY defines a new object type body. CREATE OR REPLACE TYPE BODY either creates a new object type body or replaces an existing body.
If you include a schema name, then the object type body is created in the specified schema. Otherwise it's created in the current schema. The name of the new object type body must match an existing object type specification in the same schema. The new object type body name can't match any existing object type body in the same schema unless you want to update the definition of an existing object type body. In that case, use \texttt{CREATE OR REPLACE TYPE BODY}.

### Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{name}</td>
<td>The name (optionally schema-qualified) of the object type for which to create a body.</td>
</tr>
<tr>
<td>\textit{MEMBER}</td>
<td>Specify \texttt{MEMBER} if the subprogram operates on an object instance. Specify \texttt{STATIC} if the subprogram operates independently of any particular object instance.</td>
</tr>
<tr>
<td>\textit{proc_name}</td>
<td>The name of the procedure to create.</td>
</tr>
<tr>
<td>\textit{SELF [ IN</td>
<td>IN OUT ] name}</td>
</tr>
<tr>
<td>\textit{arg_name}</td>
<td>The name of an argument. The argument is referenced by this name in the method body.</td>
</tr>
<tr>
<td>\textit{arg_type}</td>
<td>The data types of the method's arguments. The argument types can be a base data type or a user-defined type such as a nested table or an object type. Don't specify a length for any base type. For example, specify \texttt{VARCHAR2}, not \texttt{VARCHAR2(10)}.</td>
</tr>
<tr>
<td>\textit{DEFAULT value}</td>
<td>Supplies a default value for an input argument if one isn't supplied in the method call. You can't specify \texttt{DEFAULT} for arguments with modes \texttt{IN OUT} or \texttt{OUT}.</td>
</tr>
<tr>
<td>\textit{program_body}</td>
<td>The pragma, declarations, and SPL statements that make up the body of the function or procedure. Use the pragma \texttt{PRAGMA AUTONOMOUS_TRANSACTION} to set the function or procedure as an autonomous transaction.</td>
</tr>
<tr>
<td>\textit{func_name}</td>
<td>The name of the function to create.</td>
</tr>
<tr>
<td>\textit{ret_type}</td>
<td>The return data type, which can be any of the types listed for \textit{arg_type}. Don't specify a length for \textit{ret_type}.</td>
</tr>
</tbody>
</table>

### Examples

Create the object type body for object type \texttt{emp\_obj\_typ}:

```sql
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
  MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
  IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE('Employee No   : ' || empno);
      DBMS_OUTPUT.PUT_LINE('Name          : ' || ename);
      DBMS_OUTPUT.PUT_LINE('Street        : ' || addr.street);
    END;
END;
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
See also

CREATE TYPE, DROP TYPE

14.3.5.43 CREATE USER

Name

CREATE USER — Define a new database user account.

Synopsis

CREATE USER <name> [IDENTIFIED BY <password>]

Description

CREATE USER adds a user to an EDB Postgres Advanced Server database cluster. You must be a database superuser to use this command.

When you invoke the CREATE USER command, a schema is created with the same name as the new user. The new schema is owned by the new user. Objects with unqualified names that this user creates are created in this schema.

Parameters

name
The name of the user.

password

The user's password. You can change the password later using ALTE USER.

Notes

The maximum length allowed for the user name and password is 63 characters.

Examples

Create a user named john:

```sql
CREATE USER john IDENTIFIED BY abc;
```

See also

DROP USER

14.3.5.44 CREATE USER|ROLE... PROFILE MANAGEMENT CLAUSES

Name

CREATE USER|ROLE — Create a user or role.

Synopsis

```
CREATE USER|ROLE <name> [[WITH] option [...]]
```

Where option is any of the following compatible clauses:

- `PROFILE <profile_name>`
- `ACCOUNT [LOCK |UNLOCK]`
- `PASSWORD EXPIRE [AT '<timestamp>']`

Alternatively, option can be any of the following non-compatible clauses:

- `LOCK TIME '<timestamp>'`

For information about the administrative clauses of the CREATE USER or CREATE ROLE commands that are supported by EDB Postgres Advanced Server, see the PostgreSQL core documentation.

Description

CREATE ROLE|USER_PROFILE adds a role with an associated profile to an EDB Postgres Advanced Server database cluster.

By default, roles created with the CREATE USER command are login roles and roles created with the CREATE ROLE command aren't login roles. To create a login account with the CREATE ROLE command, you must include the LOGIN keyword.

Only a database superuser can use the CREATE USER|ROLE clauses that enforce profile management. These clauses enforce the following behaviors:

- Include the PROFILE clause and a profile_name to associate a predefined profile with a role or to change the predefined profile associated with a user.
Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to place the user account in a locked or unlocked state.

Include the `LOCK TIME 'timestamp'` clause and a date and time value to lock the role at the specified time. Unlock the role at the time indicated by the `ACCOUNT UNLOCK` clause.

Include the `PASSWORD_LOCK_TIME` parameter of the profile assigned to this role. If you use `LOCK TIME` with the `ACCOUNT LOCK` clause, only a database superuser can unlock the role by using the `ACCOUNT UNLOCK` clause.

Include the `PASSWORD_EXPIRE` clause with the optional `AT 'timestamp'` keywords to specify a date and time when the password associated with the role expires. If you omit the `AT 'timestamp'` keywords, the password expires immediately.

Each login role can have only one profile. To find the profile that's currently associated with a login role, query the `profile` column of the `DBA_USERS` view.

**Parameters**

- **name**
  
  The name of the role.

- **profile_name**
  
  The name of the profile associated with the role.

- **timestamp**
  
  The date and time when the clause is enforced. When specifying a value for `timestamp`, enclose the value in single quotes.

**Examples**

This example uses `CREATE USER` to create a login role named `john` that is associated with the `acctg_profile` profile:

```sql
CREATE USER john PROFILE acctg_profile IDENTIFIED BY "1safepwd";
```

`john` can log in to the server, using the password `1safepwd`.

This example uses `CREATE ROLE` to create a login role named `john` that is associated with the `acctg_profile` profile:

```sql
CREATE ROLE john PROFILE acctg_profile LOGIN PASSWORD "1safepwd";
```

`john` can log in to the server, using the password `1safepwd`.

**See also**

`ALTER USER|ROLE… PROFILE MANAGEMENT CLAUSES`

**14.3.5.45 CREATE VIEW**

**Name**

`CREATE VIEW` — Define a new view.

**Synopsis**

```sql
CREATE [ OR REPLACE ] VIEW <name> [ ( <column_name> [ , ... ] ) ]
AS <query>
```
Description

CREATE VIEW defines a view of a query. The view isn't physically materialized. Instead, the query runs every time a query references the view.

CREATE OR REPLACE VIEW is similar. However, if a view of the same name exists, the command replaces it.

If you provide a schema name, then the view is created in the specified schema. Otherwise it's created in the current schema. The view name must differ from the name of any other view, table, sequence, or index in the same schema.

Parameters

name

The name (optionally schema-qualified) of a view to create.

column_name

An optional list of names to use for columns of the view. If not given, the column names are deduced from the query.

query

A query (that is, a SELECT statement) that provides the columns and rows of the view. See SELECT for information about valid queries.

Notes

Currently, views are read-only. The system doesn't allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts and the other commands on the view into appropriate actions on other tables.

Permissions of the view owner determine access to tables referenced in the view. However, functions called in the view are treated the same as if they were called directly from the query using the view. Therefore, the user of a view must have permissions to call all functions the view uses.

Examples

Create a view consisting of all employees in department 30:

```
CREATE VIEW dept_30 AS SELECT * FROM emp WHERE deptno = 30;
```

See also

DROP VIEW

14.3.5.46   DELETE

Name

DELETE = Delete rows of a table.

Synopsis

```
DELETE [ <optimizer_hint> ] [ FROM ] <table>@<dblink> 
[ WHERE <condition> ]
[ RETURNING <return_expression> [, ...]
  [ INTO { <record> | <variable> [, ...] }]
```
Description

DELETE deletes rows that satisfy the WHERE clause from the specified table. Omitting the WHERE clause deletes all rows in the table, leaving an empty table. You need the DELETE privilege on the table to delete from it. You also need the SELECT privilege for any table whose values are read in the condition.

The FROM keyword is optional if EDB Postgres Advanced Server is installed in Oracle-compatible mode. It's required if EDB Postgres Advanced Server is installed in PostgreSQL mode.

Note

The TRUNCATE command is a faster way to remove all rows from a table.

You can specify the RETURNING INTO (record | variable [, ...]) clause only if you use the DELETE command in an SPL program. The result set of the DELETE command can't include more than one row. If it does, an error occurs. If the result set is empty, then the contents of the target record or variables are set to null.

You can specify the RETURNING BULK COLLECT INTO collection [, ...] clause only if you use the DELETE command in an SPL program. If you specify more than one collection as the target of the BULK COLLECT INTO clause, then each collection must consist of a single, scalar field. That is, collection can't be a record.

The result set of the DELETE command can contain zero, one, or more rows. The return_expression evaluated for each row of the result set becomes an element in collection, starting with the first element. Any existing rows in collection are deleted. If the result set is empty, then collection is empty.

Parameters

**optimizer_hint**
Comment-embedded hints to the optimizer for selecting an execution plan.

**table**
The name (optionally schema-qualified) of an existing table.

**dblink**
Database link name identifying a remote database. For more information, see CREATE DATABASE LINK.

**condition**
A value expression that returns a value of type BOOLEAN that determines the rows to delete.

**return_expression**
An expression that can include one or more columns from table. If you specify a column name from table in return_expression, the value substituted for the column when return_expression is evaluated is the value from the deleted row.

**record**
A record whose field to assign the evaluated return_expression. The first return_expression is assigned to the first field in record, the second return_expression is assigned to the second field in record, and so on. The number of fields in record must exactly match the number of expressions, and the fields must be type-compatible with their assigned expressions.

**variable**
A variable to which to assign the evaluated return_expression. If you specify more than one return_expression and variable, the first return_expression is assigned to the first variable, the second return_expression is assigned to the second variable, and so on. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword, and the variables must be type-compatible with their assigned expressions.

**collection**
A collection in which an element is created from the evaluated return_expression. You can have a single collection, which can be a collection of a single field or a collection of a record type. Alternatively, you can have more than one collection. In that case, each collection must consist of a single field. The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding return_expression and collection field must be type-compatible.

Examples
Delete all rows for employee 7900 from the jobhist table:

```
DELETE FROM jobhist WHERE empno = 7900;
```

Clear the table jobhist:

```
DELETE FROM jobhist;
```

See also

TRUNCATE

### 14.3.5.47 DROP CAPTURE PRIVILEGES POLICY

EDB Postgres Advanced Server includes syntax not offered by Oracle with the DROP CAPTURE PRIVILEGES POLICY SQL command.

#### Name

**DROP CAPTURE PRIVILEGES POLICY** — Drop an existing capture privilege policy.

#### Synopsis

Use **DROP CAPTURE PRIVILEGES POLICY** to drop an existing policy to capture privileges:

```
DROP CAPTURE PRIVILEGES POLICY [ IF EXISTS ] <name> [ CASCADE | RESTRICT ]
```

#### Description

**DROP CAPTURE PRIVILEGES POLICY** allows a superuser or a user with the CREATE_CAPTURE privilege to drop an existing queue.

#### Parameters

**name**

The name optionally schema-qualified of the capture privilege policy that's being dropped.

#### Examples

This example drops a capture privilege policy named `sampledb_p2`:

```
DROP CAPTURE PRIVILEGES POLICY
sampledb_p2;
```

See also

CREATE CAPTURE PRIVILEGE POLICY, ALTER CAPTURE PRIVILEGE POLICY
14.3.5.48  DROP DATABASE LINK

Name

DROP DATABASE LINK  — Remove a database link.

Synopsis

DROP [ PUBLIC ] DATABASE LINK

<name>

Description

DROP DATABASE LINK  drops existing database links. To execute this command, you must be a superuser or the owner of the database link.

Parameters

name

The name of a database link to remove.

PUBLIC

Indicates that name is a public database link.

Examples

Remove the public database link named oralink:

DROP PUBLIC DATABASE LINK
oralink;

Remove the private database link named edblink:

DROP DATABASE LINK
edblink;

See also

CREATE PUBLIC DATABASE LINK

14.3.5.49  DROP DIRECTORY

Name

DROP DIRECTORY  — Remove a directory alias for a file system directory path.

Synopsis
DROP DIRECTORIES <name>

Description

`DROP DIRECTORY` drops an existing alias for a file system directory path that was created with the `CREATE DIRECTORY` command. To execute this command, you must be a superuser.

When you delete a directory alias, the corresponding physical file system directory isn't affected. To delete the file system directory, use operating system commands.

Parameters

`name`

The name of a directory alias to remove.

Examples

Remove the directory alias `empdir`:

```
DROP DIRECTORY empdir;
```

See also

`CREATE DIRECTORY`, `ALTER DIRECTORY`

14.3.5.50 DROP FUNCTION

Name

`DROP FUNCTION` — Remove a function.

Synopsis

```
DROP FUNCTION [ IF EXISTS ]

<name>

[ ([ [ <argmode> ] [ <argname> <argtype> ] [, ...]])

] [ CASCADE | RESTRICT
```

Description

`DROP FUNCTION` removes the definition of an existing function. To execute this command, you must be a superuser or the owner of the function. If this is an overloaded function, you must specify all input (IN, IN OUT) argument data types to the function.

Note

This requirement isn't compatible with Oracle databases. In Oracle, you specify only the function name. EDB Postgres Advanced Server allows you to overload function names. Therefore, the function signature given by the input argument data types is required in the EDB Postgres Advanced Server `DROP FUNCTION` command of an overloaded function.

The `IF EXISTS`, `CASCADE`, and `RESTRICT` parameters aren't compatible with Oracle databases. Only EDB Postgres Advanced Server uses them.
Parameters

**IF EXISTS**

Specifies not to throw an error if the function doesn’t exist. A notice is issued instead.

**name**

The name (optionally schema-qualified) of an existing function.

**argmode**

The mode of an argument: **IN**, **IN OUT**, or **OUT**. If omitted, the default is **IN**. **DROP FUNCTION** ignores **OUT** arguments, since only the input arguments are needed to determine the function’s identity. You need to list only the **IN** and **IN OUT** arguments.

!!! Note Specifying `argmode` isn’t compatible with Oracle databases. It applies only to EDB Postgres Advanced Server.

**argname**

The name of an argument. **DROP FUNCTION** ignores argument names, since only the argument data types are needed to determine the function’s identity.

!!! Note Specifying `argname` isn’t compatible with Oracle databases. It applies only to EDB Postgres Advanced Server.

**argtype**

The data type of an argument of the function.

!!! Note Specifying `argtype` isn’t compatible with Oracle databases. It applies only to EDB Postgres Advanced Server.

**CASCADE**

Drop objects that depend on the function, such as operators or triggers, and objects that depend on those objects.

**RESTRICT**

Prevent dropping the function if any objects depend on it. This is the default.

Examples

This example removes the `emp_comp` function:

```sql
DROP FUNCTION emp_comp(NUMBER, NUMBER);
```

See also

CREATE FUNCTION

14.3.5.51 DROP INDEX

Name

**DROP INDEX** — Remove an index.

Synopsis

```sql
DROP INDEX <name>
```
Description

DROP INDEX drops an existing index from the database system. To execute this command, you must be a superuser or the owner of the index. If any objects depend on the index, an error occurs, and the index isn't dropped.

Parameters

name

The name (optionally schema-qualified) of an index to remove.

Examples

This example removes the index name_idx:

```
DROP INDEX name_idx;
```

See also

CREATE INDEX, ALTER INDEX

14.3.5.52   DROP PACKAGE

Name

DROP PACKAGE — Remove a package.

Synopsis

```
DROP PACKAGE [ BODY ]

<name>
```

Description

DROP PACKAGE drops an existing package. To execute this command, you must be a superuser or the owner of the package. Specify BODY to remove only the package body without dropping the package specification. Omit BODY to remove both the package specification and body.

Parameters

name

The name (optionally schema-qualified) of a package to remove.

Examples

This example removes the emp_admin package:

```
DROP PACKAGE emp_admin;
```
14.3.5.53 DROP PROCEDURE

Name

DROP PROCEDURE — Remove a procedure.

Synopsis

DROP PROCEDURE [ IF EXISTS ]
<name>
[ ( [ [ argmode ] ] [ argname ] argtype ] [, ...] ) ]
[ CASCADE | RESTRICT ]

Description

DROP PROCEDURE removes the definition of an existing procedure. To execute this command, you must be a superuser or the owner of the procedure. For an overloaded procedure, you must specify all input (IN, IN OUT) argument data types to the procedure.

Note

This requirement isn't compatible with Oracle databases. In Oracle, specify only the procedure name. EDB Postgres Advanced Server allows overloading of procedure names. Therefore the procedure signature given by the input argument data types is required in the EDB Postgres Advanced Server DROP PROCEDURE command for an overloaded procedure.

The IF EXISTS, CASCADE, and RESTRICT parameters aren't compatible with Oracle databases. Only EDB Postgres Advanced Server uses them.

Parameters

IF EXISTS

Specifies not to throw an error if the procedure doesn't exist. A notice is issued instead.

name

The name (optionally schema-qualified) of an existing procedure.

argmode

The mode of an argument: IN, IN OUT, or OUT. The default is IN. DROP PROCEDURE ignores OUT arguments, since only the input arguments are needed to determine the procedure's identity. List only the IN and IN OUT arguments.

!!! Note Specifying argmode isn't compatible with Oracle databases. It applies only to EDB Postgres Advanced Server.

argname

The name of an argument. DROP PROCEDURE ignores argument names, since only the argument data types are needed to determine the procedure's identity.

!!! Note Specifying argname isn't compatible with Oracle databases. It applies only to EDB Postgres Advanced Server.

argtype

The data type of an argument of the procedure.

!!! Note Specifying argtype isn't compatible with Oracle databases. It applies only to EDB Postgres Advanced Server.
CASCADE

Drop objects that depend on the procedure and all objects that depend on those objects.

RESTRICT

Prevent dropping the procedure if any objects depend on it. This is the default.

Examples

This example removes the `select_emp` procedure:

```
DROP PROCEDURE select_emp;
```

See also

CREATE PROCEDURE, ALTER PROCEDURE

14.3.5.54 DROP PROFILE

Name

DROP PROFILE — Drop a user-defined profile.

Synopsis

```
DROP PROFILE [IF EXISTS] profile_name [CASCADE | RESTRICT];
```

Description

Include the `IF EXISTS` clause to prevent an error if the specified profile doesn't exist. Instead, issue a notice.

Include the optional `CASCADE` clause to reassign any users that are currently associated with the profile to the `default` profile and then drop the profile. Include the optional `RESTRICT` clause to prevent dropping any profile that's associated with a role. This is the default behavior.

Parameters

`profile_name`

The name of the profile to drop.

Examples

This example drops a profile named `acctg_profile`. The command first associates any roles associated with the `acctg_profile` profile with the `default` profile and then drops the `acctg_profile` profile.

```
DROP PROFILE acctg_profile CASCADE;
```

This example drops a profile named `acctg_profile`. The `RESTRICT` clause prevents dropping `acctg_profile` if any roles are associated with the profile.
DBMS_AQADM.

Description

DROP QUEUE allows a superuser or a user with the aq_administrator_role privilege to drop an existing queue.

Parameters

name

The name (possibly schema-qualified) of the queue that's being dropped.

IF EXISTS

Include the IF EXISTS clause if you don't want to return an error if the queue doesn't exist. Instead, issue a notice.

Examples

This example drops a queue named work_order:

```sql
DROP QUEUE work_order;
```

See also

CREATE QUEUE, ALTER QUEUE

14.3.5.56 DROP QUEUE TABLE
EDB Postgres Advanced Server includes syntax not offered by Oracle with the `DROP QUEUE TABLE SQL` command. You can use this syntax with `DBMS_AQADM`.

### Name

**DROP QUEUE TABLE** — Drop a queue table.

### Synopsis

Use `DROP QUEUE TABLE` to delete a queue table:

```
DROP QUEUE TABLE [ IF EXISTS ] <name> [, ...
[CASCADE | RESTRICT]
```

### Description

`DROP QUEUE TABLE` allows a superuser or a user with the `aq_administrator_role` privilege to delete a queue table.

### Parameters

**name**

The name (possibly schema-qualified) of the queue table to delete.

**IF EXISTS**

Include the `IF EXISTS` clause if you don’t want to return an error if the queue table doesn’t exist. Instead, issue a notice.

**CASCADE**

Include the `CASCADE` keyword to delete objects that depend on the queue table.

**RESTRICT**

Include the `RESTRICT` keyword to prevent deleting the queue table if any objects depend on it. This is the default.

### Examples

This example deletes a queue table named `work_order_table` and any objects that depend on it:

```
DROP QUEUE TABLE work_order_table CASCADE;
```

### See also

`CREATE QUEUE TABLE, ALTER QUEUE TABLE`

### 14.3.5.57 DROP SYNONYM

Name
**DROP SYNONYM** — Remove a synonym.

**Synopsis**

```sql
DROP [PUBLIC] SYNONYM
[<schema>.]<syn_name>
```

**Description**

**DROP SYNONYM** deletes existing synonyms. To execute this command, you must be a superuser or the owner of the synonym and have **USAGE** privileges on the schema in which the synonym resides.

**Parameters**

- **syn_name**
  The name of the synonym. A synonym name must be unique in a schema.

- **schema**
  The name of the schema where the synonym resides.

Like any other object that can be schema qualified, you can have two synonyms with the same name in your search path. To specify the name of the synonym that you're dropping, include a schema name. Unless a synonym is schema qualified in the **DROP SYNONYM** command, EDB Postgres Advanced Server deletes the first instance of the synonym it finds in your search path.

You can optionally include the **PUBLIC** clause to drop a synonym that resides in the **public** schema. The **DROP PUBLIC SYNONYM** command, compatible with Oracle databases, drops a synonym that resides in the **public** schema:

```sql
DROP PUBLIC SYNONYM syn_name;
```

**Examples**

This example drops the synonym **personnel**:

```sql
DROP SYNONYM personnel;
```

**See also**

**CREATE SYNONYM**

**14.3.5.58 DROP ROLE**

**Name**

**DROP ROLE** — Remove a database role.

**Synopsis**

```sql
DROP ROLE <name> [ CASCADE ]
```
Description

DROP ROLE removes the specified role. To drop a superuser role, you must be a superuser. To drop non-superuser roles, you must have the CREATEROLE privilege.

Before dropping the role, you must drop all the objects it owns or reassign their ownership and revoke any privileges the role was granted. You can't remove a role if any database of the cluster references it.

You don't need to remove role memberships involving the role. DROP ROLE revokes any memberships of the target role in other roles and of other roles in the target role. The other roles aren't dropped or otherwise affected.

Alternatively, if the only objects owned by the role belong in a schema that's owned by the role and has the same name as the role, you can specify the CASCADE option. In this case, the issuer of the DROP ROLE name CASCADE command must be a superuser. The named role, the schema, and all objects in the schema are deleted.

Parameters

name

The name of the role to remove.

CASCADE

Drops the schema owned by and with the same name as the role as long as there are no other dependencies on the role or the schema. All objects owned by the role belonging to the schema are also dropped.

Examples

This example drops a role:

```
DROP ROLE admins;
```

See also

CREATE ROLE, SET ROLE, GRANT, REVOKE

14.3.5.59 DROP SEQUENCE

Name

DROP SEQUENCE — Remove a sequence.

Synopsis

```
DROP SEQUENCE <name> [, ...]
```

Description

DROP SEQUENCE removes sequence number generators. To execute this command, you must be a superuser or the owner of the sequence.

Parameters

name
The name (optionally schema-qualified) of a sequence.

Examples

Remove the sequence `serial`:

```
DROP SEQUENCE serial;
```

See also

`ALTER SEQUENCE`, `CREATE SEQUENCE`

14.3.60 DROP TABLE

Name

`DROP TABLE` — Remove a table.

Synopsis

```
DROP TABLE <name> [CASCADE | RESTRICT | CASCADE CONSTRAINTS]
```

Description

`DROP TABLE` removes tables from the database. Only the owner can remove a table. To empty a table of rows without removing the table, use `DELETE`. `DROP TABLE` removes any of the target table’s indexes, rules, triggers, and constraints.

Parameters

`name`

The name (optionally schema-qualified) of the table to drop.

`RESTRICT`

Include the `RESTRICT` keyword to prevent dropping the table if any objects depend on it. This is the default behavior. The `DROP TABLE` command reports an error if any objects depend on the table.

`CASCADE`

Include the `CASCADE` clause to drop any objects that depend on the table.

`CASCADE CONSTRAINTS`

Include the `CASCADE CONSTRAINTS` clause to drop any dependent constraints on the specified table, excluding other object types.

Examples

This example drops a table named `emp` that has no dependencies:

```
DROP TABLE emp;
```
The outcome of a `DROP TABLE` command varies depending on whether the table has any dependencies. You can control the outcome by specifying a drop behavior. For example, suppose you create two tables, `orders` and `items`, where the `items` table depends on the `orders` table:

```
CREATE TABLE orders
  (order_id int PRIMARY KEY, order_date date, ...);
CREATE TABLE items
  (order_id REFERENCES orders, quantity int, ...);
```

EDB Postgres Advanced Server performs one of the following actions when dropping the `orders` table, depending on the drop behavior that you specify:

- If you specify `DROP TABLE orders RESTRICT`, EDB Postgres Advanced Server reports an error.
- If you specify `DROP TABLE orders CASCADE`, EDB Postgres Advanced Server drops the `orders` table and the `items` table.
- If you specify `DROP TABLE orders CASCADE CONSTRAINTS`, EDB Postgres Advanced Server drops the `orders` table and removes the foreign key specification from the `items` table. It doesn't drop the `items` table.

See also

`CREATE TABLE`, `ALTER TABLE`

### 14.3.5.61 DROP TABLESPACE

#### Name

```
DROP TABLESPACE
```

#### Synopsis

```
DROP TABLESPACE <tablespacename>
```

#### Description

`DROP TABLESPACE` removes a tablespace from the system.

Only a superuser or the tablespace owner can drop a tablespace. The tablespace must be empty of all database objects before you drop it. Objects in other databases might still reside in the tablespace even if no objects in the current database are using the tablespace.

#### Parameters

- `tablespacename`

  The name of a tablespace.

#### Examples

This example removes the tablespace `employee_space` from the system:

```
DROP TABLESPACE employee_space;
```

See also

`CREATE TABLESPACE`, `ALTER TABLESPACE`
14.3.5.62 DROP TRIGGER

Name

DROP TRIGGER — Remove a trigger.

Synopsis

DROP TRIGGER <name>

Description

DROP TRIGGER removes a trigger from its associated table. A superuser or the owner of the associated table can run the command.

Parameters

name

The name of a trigger to remove.

Examples

Remove a trigger named emp_salary_trig:

DROP TRIGGER emp_salary_trig;

See also

CREATE TRIGGER, ALTER TRIGGER

14.3.5.63 DROP TYPE

Name

DROP TYPE — Remove a type definition.

Synopsis

DROP TYPE [ BODY ]
<name>

Description
DROP TYPE removes the type definition. To execute this command, you must be a superuser or the owner of the type.

The optional BODY qualifier applies only to object type definitions, not to collection types or to composite types. If you specify BODY, only the object type body is removed and not the object type specification. If you omit BODY, both the object type specification and body are removed.

The type isn’t deleted if other database objects depend on the named type.

Parameters

name

The name of a type definition to remove.

Examples

Drop the object type named addr_obj_typ:

```
DROP TYPE addr_obj_typ;
```

Drop the nested table type named budget_tbl_typ:

```
DROP TYPE budget_tbl_typ;
```

See also

CREATE TYPE, CREATE TYPE BODY

14.3.5.64 DROP USER

Name

DROP USER — Remove a database user account.

Synopsis

```
DROP USER <name> [ CASCADE ]
```

Description

DROP USER removes the specified user. You must be a superuser to drop a superuser. To drop non-superusers, you need the CREATEROLE privilege.

You can’t remove a user if any database of the cluster references it. Before dropping the user, you must drop all the objects it owns or reassign their ownership and revoke any privileges the user was granted.

However, you don’t need to remove role memberships involving the user. DROP USER revokes any memberships of the target user in other roles and of other roles in the target user. The other roles aren’t dropped or otherwise affected.

Alternatively, if the only objects owned by the user belong in a schema that’s owned by the user and has the same name as the user, you can specify the CASCADE option. In this case, the issuer of the DROP USER name CASCADE command must be a superuser. The named user, the schema, and all objects in the schema are deleted.

Parameters
The name of the user to remove.

**CASCADE**

If specified, drops the schema owned by and with the same name as the user as long as there are no other dependencies on the user or the schema. It also drops all objects owned by the user belonging to the schema.

### Examples

Drop a user account named `john` that owns no objects and doesn't have privileges on any other objects:

```sql
DROP USER john;
```

Drop the user account `john` that doesn't have privileges on any objects. The account also doesn't own any objects outside of a schema named `john` that's owned by the user `john`:

```sql
DROP USER john CASCADE;
```

### See also

*CREATE USER, ALTER USER*

**14.3.65 DROP VIEW**

### Name

**DROP VIEW** — Remove a view.

### Synopsis

```
DROP VIEW <name>
```

### Description

**DROP VIEW** drops an existing view. To execute this command, you must be a database superuser or the owner of the view. The named view isn't deleted if other objects depend on this view, such as a view of a view.

The form of the **DROP VIEW** command compatible with Oracle doesn't support a **CASCADE** clause. To drop a view and its dependencies, use the PostgreSQL-compatible form of the **DROP VIEW** command. For more information, see the PostgreSQL core documentation.

### Parameters

**name**

The name (optionally schema-qualified) of the view to remove.

### Examples

This example removes the view named `dept_30`:

```sql
DROP VIEW dept_30;
```
14.3.5.66  EXEC

Name

EXEC — Execute a function.

Synopsis

EXEC function_name
['([argument_list])]'

Description

EXECUTE

Parameters

procedure_name

The (optionally schema-qualified) function name.

argument_list

A comma-separated list of arguments required by the function. Each member of argument_list corresponds to a formal argument expected by the function. Each formal argument can be an IN parameter, an OUT parameter, or an INOUT parameter.

Examples

The EXEC statement can take one of several forms, depending on the arguments required by the function:

EXEC update_balance;
EXEC update_balance();
EXEC update_balance(1, 2, 3);

14.3.5.67  GRANT

Name

GRANT — Define access privileges.

Synopsis
GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES }
} [
 [...]
 ,
 ALL [ PRIVILEGES ]
}
ON tablename
TO { username | groupname | PUBLIC } [, 
...
] [ WITH GRANT OPTION ]

GRANT { { INSERT | UPDATE | REFERENCES } (column [, ...])
} [
 [...]
 ON tablename
TO { username | groupname | PUBLIC } [, 
...
] [ WITH GRANT OPTION ]

GRANT { SELECT | ALL [ PRIVILEGES ]
}
ON sequencename
TO { username | groupname | PUBLIC } [, 
...
] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ]
}
ON FUNCTION
programe
([ [ [ argmode ] [ argname ] argtype ] [, ...] ])
TO { username | groupname | PUBLIC } [, 
...
] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ]
}
ON PROCEDURE
programe
([ [ [ argmode ] [ argname ] argtype ] [, ...] ])
TO { username | groupname | PUBLIC } [, 
...
] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ]
}
ON PACKAGE
packagename
TO { username | groupname | PUBLIC } [, 
...
] [ WITH GRANT OPTION ]

GRANT role [, 
...
] 
TO { username | groupname | PUBLIC } [, 
...
] [ WITH ADMIN OPTION ]

GRANT { CONNECT | RESOURCE | DBA } [, 
...
] 
TO { username | groupname } [, 
...
] [ WITH ADMIN OPTION ]

GRANT CREATE [ PUBLIC ] DATABASE LINK
TO { username | groupname }

GRANT DROP PUBLIC DATABASE LINK
TO { username | groupname }

GRANT EXEMPT ACCESS
POLICY
TO { username | groupname }
The GRANT command has three basic variants:

- One that grants privileges on a database object (table, view, sequence, or program)
- One that grants membership in a role
- One that grants system privileges

In EDB Postgres Advanced Server, the concept of users and groups was unified into a single type of entity called a role. In this context, a user is a role that has the LOGIN attribute. The role can be used to create a session and connect to an application. A group is a role that doesn't have the LOGIN attribute. You can't use the role to create a session or connect to an application.

A role can be a member of one or more other roles, so the traditional concept of users belonging to groups is still valid. However, with the generalization of users and groups, users can "belong" to users, groups can "belong" to groups, and groups can "belong" to users, forming a general multi-level hierarchy of roles. User names and group names share the same namespace. Therefore, you don't need to specify whether a grantee is a user or a group in the GRANT command.

GRANT on database objects

This variant of the GRANT command gives specific privileges on a database object to a role. These privileges are added to those already granted, if any.

The keyword PUBLIC grants the privileges to all roles, including those that you create later. PUBLIC is an implicitly defined group that always includes all roles. Any role has the sum of privileges granted directly to it; privileges granted to any role it is a member of, and privileges granted to PUBLIC.

If you specify the WITH GRANT OPTION, the recipient of the privilege can grant it to others. Grant options aren't granted to PUBLIC.

You don't need to grant privileges to the owner of an object (usually the user that created it), as the owner has all privileges by default. The owner can, however, revoke some of their own privileges for safety. The right to drop an object or to alter its definition isn't described by a grantable privilege. It's inherent in the owner and can't be granted or revoked. The owner implicitly has all grant options for the object as well.

Depending on the type of object, the initial default privileges can include granting some privileges to PUBLIC. The default is no public access for tables and EXECUTE privilege for functions, procedures, and packages. The object owner can revoke these privileges.

Note

For maximum security, issue the REVOKE in the same transaction that creates the object. This approach prevents a window from occurring in which another user can use the object.

The possible privileges are:

SELECT

Allows SELECT from any column of the specified table, view, or sequence. For sequences, this privilege also allows the use of the currval function.

INSERT

Allows you to insert a new row into the specified table.

UPDATE

Allows UPDATE of a column of the specified table. SELECT ... FOR UPDATE also requires this privilege in addition to the SELECT privilege.

DELETE

Allows you to delete a row from the specified table.

REFERENCES

To create a foreign key constraint, you need this privilege on both the referencing and referenced tables.

EXECUTE

Allows the use of the specified package, procedure, or function. When applied to a package, allows the use of all of the package’s public procedures, public functions, public variables, records, cursors, and other public objects and object types. This is the only type of privilege that applies to functions, procedures, and packages.

The EDB Postgres Advanced Server syntax for granting the EXECUTE privilege isn't fully compatible with Oracle databases. EDB Postgres Advanced Server requires that you qualify the program name with one of the keywords FUNCTION, PROCEDURE, or PACKAGE. You must omit these keywords in Oracle.

For functions, EDB Postgres Advanced Server requires all input (IN, IN OUT) argument data types after the function name, including an empty parenthesis if there are no function arguments. For procedures, you must specify all input argument data types if the procedure has any input arguments. In Oracle, omit function and procedure signatures. All programs share the same namespace in Oracle, whereas functions, procedures, and packages have their own namespaces in EDB Postgres Advanced Server to allow program name overloading to a certain extent.
ALL PRIVILEGES

Grants all of the available privileges at once.

GRANT on roles

This variant of the GRANT command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If you specify the WITH ADMIN OPTION, the member can grant membership in the role to others and revoke membership in the role.

Database superusers can grant or revoke membership in any role to anyone. Roles having the CREATEROLE privilege can grant or revoke membership in any role that isn’t a superuser.

There are three predefined roles.

CONNECT

Granting the CONNECT role is equivalent to giving the grantee the LOGIN privilege. The grantor must have the CREATEROLE privilege.

RESOURCE

Granting the RESOURCE role is equivalent to granting the CREATE and USAGE privileges on a schema that has the same name as the grantee. This schema must exist before you give the grant. The grantor must have the privilege to grant CREATE or USAGE privileges on this schema to the grantee.

DBA

Granting the DBA role is equivalent to making the grantee a superuser. The grantor must be a superuser.

Notes

Use the REVOKE command to revoke access privileges.

When a non-owner of an object attempts to grant privileges on the object, the command fails if the user has no privileges on the object. As long as a privilege is available, the command proceeds, but it grants only those privileges for which the user has grant options. The GRANT ALL PRIVILEGES form issues a warning if no grant options are held, while the other forms issue a warning if grant options for any of the privileges named in the command aren’t held. In principle, these statements apply to the object owner as well. However, since the owner is always treated as holding all grant options, the cases can never occur.

Database superusers can access all objects regardless of object privilege settings. This is comparable to the rights of root in a Unix system. As with root, only operate as a superuser when you have to.

If a superuser issues a GRANT or REVOKE command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted by such a command appear as if granted by the object owner. (For role membership, the membership appears as if granted by the containing role.)

GRANT and REVOKE can also be done by:

- A role that isn’t the owner of the affected object but is a member of the role that owns the object.
- A role that is a member of a role that holds privileges WITH GRANT OPTION on the object. In this case, the privileges are recorded as having been granted by the role that owns the object or holds the privileges WITH GRANT OPTION.

For example, if table t1 is owned by role g1, of which role u1 is a member, then u1 can grant privileges on t1 to u2. However, those privileges appear as if granted directly by g1. Any other member of role g1 can revoke them later.

If the role executing GRANT holds the required privileges indirectly by way of more than one role membership path, the containing role recorded as having done the grant is unspecified. In such cases, best practice is to use SET ROLE to become the specific role you want to do the GRANT as.

Currently, EDB Postgres Advanced Server doesn’t support granting or revoking privileges for individual columns of a table. One workaround is to create a view having just the desired columns and then grant privileges to that view.

Examples

Grant insert privilege to all users on table emp:

```sql
GRANT INSERT ON emp TO PUBLIC;
```
Grant all available privileges to user **mary** on view **salesemp**:

```
GRANT ALL PRIVILEGES ON salesemp TO mary;
```

While this example does grant all privileges if executed by a superuser or the owner of **emp**, when executed by someone else it grants only those permissions for which that user has grant options.

Grant membership in role **admins** to user **joe**:

```
GRANT admins TO joe;
```

Grant **CONNECT** privilege to user **joe**:

```
GRANT CONNECT TO joe;
```

**See also**

`REVOKE`, `SET ROLE`

**GRANT on system privileges**

This variant of the `GRANT` command gives a role the ability to perform certain system operations in a database. System privileges relate to the ability to create or delete certain database objects that aren’t necessarily in the confines of one schema. Only database superusers can grant system privileges.

**CREATE [PUBLIC] DATABASE LINK**

The `CREATE [PUBLIC] DATABASE LINK` privilege allows the specified role to create a database link. Include the **PUBLIC** keyword to allow the role to create public database links. Omit the **PUBLIC** keyword to allow the specified role to create private database links.

**DROP PUBLIC DATABASE LINK**

The `DROP PUBLIC DATABASE LINK` privilege allows a role to drop a public database link. You don’t need system privileges to drop a private database link. The link owner or a database superuser can drop a private database link.

**EXEMPT ACCESS POLICY**

The `EXEMPT ACCESS POLICY` privilege allows a role to execute a SQL command without invoking any policy function that’s associated with the target database object. The role is exempt from all security policies in the database.

The `EXEMPT ACCESS POLICY` privilege can’t be inherited by membership to a role that has the `EXEMPT ACCESS POLICY` privilege. For example, the following sequence of `GRANT` commands doesn’t result in user **joe** obtaining the `EXEMPT ACCESS POLICY` privilege. This is true even though **joe** is granted membership to the `enterprisedb` role, which was granted the `EXEMPT ACCESS POLICY` privilege:

```
GRANT EXEMPT ACCESS POLICY TO enterprisedb;
GRANT enterprisedb TO joe;
```

The `rolpolicyexempt` column of the system catalog table `pg_authid` is set to **true** if a role has the `EXEMPT ACCESS POLICY` privilege.

**Examples**

Grant `CREATE PUBLIC DATABASE LINK` privilege to user **joe**:

```
GRANT CREATE PUBLIC DATABASE LINK TO joe;
```

Grant `DROP PUBLIC DATABASE LINK` privilege to user **joe**:

```
GRANT DROP PUBLIC DATABASE LINK TO joe;
```

Grant the `EXEMPT ACCESS POLICY` privilege to user **joe**:

```
GRANT EXEMPT ACCESS POLICY TO joe;
```
Using the ALTER ROLE command to assign system privileges

The EDB Postgres Advanced Server ALTER ROLE command also supports syntax that you can use to assign:

- The privilege required to create a public or private database link.
- The privilege required to drop a public database link.
- The EXEMPT ACCESS POLICY privilege.

The ALTER ROLE syntax is equivalent to the respective commands compatible with Oracle databases.

See also

REVOKE, ALTER ROLE

14.3.5.68 INSERT

Name

INSERT — Create rows in a table.

Synopsis

```
INSERT INTO <table>[@<dblink> | [ AS ] <alias>] [ ( <column> [, ...] ) ]
VALUES ( ( <expression> | DEFAULT ) [, ...] )
[ RETURNING <return_expression> [, ...
    ( INTO [ <record> | <variable> [, ...] ]
    ] BULK COLLECT INTO <collection> [, ... ]
    ] ]
[ <query> ]
```

Description

**INSERT** allows you to insert new rows into a table. You can insert a single row or several rows as a result of a query.

You can list the columns in the target list in any order. Each column not present in the target list is inserted using a default value, either its declared default value or null.

If the expression for each column doesn't have the correct data type, type conversion is attempted.

You can specify the **RETURNING INTO [ record | variable [, ... ] ]** clause only when the **INSERT** command is used in an SPL program and only when the **VALUES** clause is used.

You can specify the **RETURNING BULK COLLECT INTO collection [, ...]** clause only if the **INSERT** command is used in an SPL program. If you specify more than one **collection** as the target of the **BULK COLLECT INTO** clause, then each **collection** must consist of a single, scalar field. That is, **collection** can't be a record. The **return_expression** evaluated for each inserted row becomes an element in **collection**, starting with the first element. Any existing rows in **collection** are deleted. If the result set is empty, then **collection** is empty.

You need **INSERT** privilege to a table to insert into it. If you use the **query** clause to insert rows from a query, you also need **SELECT** privilege on any table used in the query.

Parameters

- **table**
  
  The name (optionally schema-qualified) of an existing table.

- **alias**
A substitute name, which is an alias for the table, view, materialized view, or a subquery to reference in a statement during query execution.

**dblink**

Database link name identifying a remote database. See [CREATE DATABASE LINK](#) for information on database links.

**column**

The name of a column in `table`.

**expression**

An expression or value to assign to `column`.

**DEFAULT**

This column is filled with its default value.

**query**

A query (SELECT statement) that supplies the rows to insert. See [SELECT](#) for a description of the syntax.

**return_expression**

An expression that can include one or more columns from `table`. If you specify a column name from `table` in `return_expression`, the value substituted for the column when `return_expression` is evaluated is determined as follows:

- If the column specified in `return_expression` is assigned a value in the INSERT command, then the assigned value is used in the evaluation of `return_expression`.

- If the column specified in `return_expression` isn't assigned a value in the INSERT command, and there's no default value for the column in the table's column definition, then null is used in the evaluation of `return_expression`.

- If the column specified in `return_expression` isn't assigned a value in the INSERT command, and there's a default value for the column in the table's column definition, then the default value is used in the evaluation of `return_expression`.

**record**

A record whose field to assign the evaluated `return_expression`. The first `return_expression` is assigned to the first field in `record`, the second `return_expression` is assigned to the second field in `record`, and so on. The number of fields in `record` must exactly match the number of expressions, and the fields must be type-compatible with their assigned expressions.

**variable**

A variable to which to assign the evaluated `return_expression`. If you specify more than one `return_expression` and `variable`, the first `return_expression` is assigned to the first `variable`, the second `return_expression` is assigned to the second `variable`, and so on. The number of variables specified following the INTO keyword must exactly match the number of expressions following the RETURNING keyword. The variables must be type-compatible with their assigned expressions.

**collection**

A collection in which an element is created from the evaluated `return_expression`. There can be either:

- A single collection, which can be a collection of a single field or a collection of a record type.

- More than one collection, in which case each collection must consist of a single field.

The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding `return_expression` and `collection` field must be type-compatible.

**Examples**

Insert a single row into table `emp`:

```
INSERT INTO emp VALUES (8021,'JOHN','SALESMAN',7698,'22-FEB-07',1250,500,30);
```

This example omits the column `comm`, which means it uses the default value of null:

```
INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, deptno)
```
This example uses the DEFAULT clause for the hiredate and comm columns rather than specifying a value:

```sql
VALUES (8022, 'PETERS', 'CLERK', 7598, '03-DEC-86', 950, 30);
```

This example creates a table for the department names and then inserts into the table by selecting from the dname column of the dept table:

```sql
CREATE TABLE deptnames
(
    deptname VARCHAR2 (14)
);
INSERT INTO deptnames SELECT dname FROM dept;
```

This example creates an alias enm for the table emp and inserts rows into a table:

```sql
INSERT INTO emp AS enm (enm.empno, enm.ename, enm.job, enm.mgr, enm.hiredate, enm.sal, enm.deptno)
VALUES (7499, 'SMITH', 'ANALYST', 7902, '03-DEC-06', 2500, 20);
```

14.3.5.69 MERGE

**Name**

MERGE — conditionally insert, update, or delete rows of a table.

**Synopsis**

```sql
MERGE INTO target_table_name [ target_alias ]
USING data_source [ source_alias ] ON ( join_condition )
merge_update_clause
merge_insert_clause
-- where merge_update_clause
IS
WHEN MATCHED THEN
    UPDATE SET column = value, ... [ WHERE where_condition ]
[ DELETE WHERE where_condition ]
-- and merge_insert_clause
IS
WHEN NOT MATCHED THEN
    INSERT [(col_list)] VALUES (val_list) [ WHERE where_condition ]
```

**Description**

MERGE allows you to select rows from one or more sources for update or insertion into a table. You specify the join condition to determine whether to update or insert into the target table. You specify conditional UPDATE and INSERT statements using the WHERE clause in the MERGE statement.

MERGE provides a single SQL statement that can conditionally INSERT, UPDATE, or DELETE rows, a task which would otherwise require multiple procedural language statements.

For more information, see MERGE statement.

**Parameters**

- `target_table_name`
The name (optionally schema-qualified) of the target table to merge into.

```plaintext
target_alias
```

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table.

```plaintext
data_source
```

The name (optionally schema-qualified) of the source table or a source query. The source table is the source table name, view name, or transition table name. The source query (SELECT or VALUES statement) supplies the rows to be merged into the target_table_name.

```plaintext
source_alias
```

A substitute name for the data source. When an alias is provided, it completely hides the actual name of the table.

```plaintext
join_condition
```

An expression resulting in a value of type boolean similar to a WHERE clause that specifies which rows in the data_source match rows in the target_table_name.

```plaintext
merge_update_clause
```

Specifies the new column values of the target table. It performs this update if the join_condition of the ON clause is true.

```plaintext
merge_insert_clause
```

The merge_insert_clause specifies values to insert into the column of the target table if the condition of the ON clause is false. If you omit the column list after the INSERT keyword, then the number of columns in the target table must match the number of values in the VALUES clause.

To insert all of the source rows into the table, you can use a constant filter predicate in the ON clause condition. An example of a constant filter predicate is `ON (0=1)`. It recognizes such a predicate and makes an unconditional insert of all source rows into the table. This approach is different from omitting the merge_update_clause. In that case, the database still must perform a join. With constant filter predicate, no join is performed.

Specify the where_clause if you want the database to execute the insert operation only if the specified condition is true. The condition can refer only to the data source table. Database skips the insert operation for all rows for which the condition is not true.

You can specify this clause by itself or with the merge_insert_clause. If you specify both, then they can be in either order.

```plaintext
Examples
```

Create tables `target` and `source`:

```sql
CREATE TABLE target (tid integer, balance integer);
CREATE TABLE source (sid integer, delta integer);
```

Add rows to both the tables:

```sql
# Insert rows into target table
INSERT INTO target VALUES (1, 0);
INSERT INTO target VALUES (2, 20);
INSERT INTO target VALUES (3, 0);

# Insert rows into source table
INSERT INTO source VALUES (1, 100);
INSERT INTO source VALUES (2, 200);
INSERT INTO source VALUES (3, 300);
INSERT INTO source VALUES (4, 100);
```
This example shows how to UPDATE and INSERT rows on the target table using the MERGE statement:

```
MERGE INTO target t
USING source s
ON (t.tid = s.sid)
WHEN MATCHED THEN
  UPDATE SET balance = s.delta
WHEN NOT MATCHED THEN
  INSERT VALUES (s.sid, s.delta);
```

This example shows how to UPDATE, INSERT, and conditionally DELETE rows on the target table using MERGE statement:

```
MERGE INTO target t
USING source s
ON (t.tid = s.sid)
WHEN MATCHED THEN
  UPDATE SET balance = t.balance + 100
  DELETE WHERE sid = 1
WHEN NOT MATCHED THEN
  INSERT VALUES (s.sid, s.delta);
```

This example shows how to UPDATE, INSERT, and DELETE rows on the target table conditionally using the MERGE statement:

```
MERGE INTO target t
USING source s
ON (t.tid = s.sid)
WHEN MATCHED THEN
  UPDATE SET balance = s.delta WHERE balance = 0
  DELETE WHERE sid = 1
WHEN NOT MATCHED THEN
  INSERT VALUES (s.sid, s.delta) WHERE s.sid >= 5;
```

14.3.5.70  

**LOCK**

**Name**

LOCK — Lock a table.

**Synopsis**

```
LOCK TABLE <name> [, ...] IN <lockmode> MODE [ NOWAIT ]
```

Where **lockmode** is one of:

- ROW SHARE
- ROW EXCLUSIVE
- SHARE
- SHARE ROW EXCLUSIVE
- EXCLUSIVE

**Description**

The **LOCK TABLE** command obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If you specify **NOWAIT**, **LOCK TABLE** does not wait to acquire the desired lock. If the lock can't be acquired immediately, the command is aborted and an error occurs. Once obtained, the lock is held for the remainder of the current transaction. There's no **UNLOCK TABLE** command; locks are always released at transaction end.
When acquiring locks for commands that reference tables, EDB Postgres Advanced Server always uses the least restrictive lock mode possible. `LOCK TABLE` provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the isolation level read committed and needs to ensure that data in a table remains stable for the rest of the transaction. To achieve this, you can obtain `SHARE` lock mode over the table before querying. This approach prevents concurrent data changes and ensures subsequent reads of the table see a stable view of committed data. That's because `SHARE` lock mode conflicts with the `ROW EXCLUSIVE` lock acquired by writers, and your `LOCK TABLE` name in `SHARE` mode waits until any concurrent holders of `ROW EXCLUSIVE` mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding, and none can begin until you release the lock.

To achieve a similar effect when running a transaction at the isolation level serializable, you have to execute the `LOCK TABLE` statement before executing any data modification statement. A serializable transaction’s view of data is frozen when its first data modification statement begins. A later `LOCK TABLE` still prevents concurrent writes, but it won’t ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort changes the data in the table, then use `SHARE ROW EXCLUSIVE` lock mode instead of `SHARE` mode.

This approach ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire `SHARE` mode and then be unable to also acquire `ROW EXCLUSIVE` mode to perform their updates. A transaction’s own locks never conflict, so a transaction can acquire `ROW EXCLUSIVE` mode when it holds `SHARE` mode but not if anyone else holds `SHARE` mode.

To avoid deadlocks, make sure all transactions acquire locks on the same objects in the same order. If multiple lock modes are involved for a single object, then always have transactions acquire the most restrictive mode first.

### Parameters

- **name**
  
  The name (optionally schema-qualified) of an existing table to lock.

  The command `LOCK TABLE a, b;` is equivalent to `LOCK TABLE a; LOCK TABLE b`. The tables are locked one by one in the order specified in the `LOCK TABLE` command.

- **lockmode**
  
  Specifies the locks this lock conflicts with.

  If you don't specify a lock mode, then the server uses the most restrictive mode: `ACCESS EXCLUSIVE`. `ACCESS EXCLUSIVE` isn't compatible with Oracle databases. In EDB Postgres Advanced Server, this configuration mode ensures that no other transaction can access the locked table.

- **NOWAIT**
  
  Specifies for `LOCK TABLE` not to wait for any conflicting locks to be released. If you can't immediately acquire the specified lock without waiting, the transaction is aborted.

### Notes

All forms of `LOCK` require `UPDATE` or `DELETE` privileges.

- `LOCK TABLE` is useful only inside a transaction block since the lock is dropped as soon as the transaction ends. A `LOCK TABLE` command appearing outside any transaction block forms a self-contained transaction, so the lock is dropped as soon as you obtain it.

- `LOCK TABLE` deals only with table-level locks, and so the mode names involving `ROW` are all misnomers. These mode names are generally read as indicating the intention of the user to acquire row-level locks in the locked table. Also, `ROW EXCLUSIVE` mode is a sharable table lock. Keep in mind that all the lock modes have identical semantics as far as `LOCK TABLE` is concerned, differing only in the rules about the modes that conflict with each other.

### 14.3.5.71 REVOKE

#### Name

`REVOKE` – Remove access privileges.

#### Synopsis

```
REVOKE { { SELECT | INSERT | UPDATE | DELETE | REFERENCES }
   [ , . . . ] | ALL [ PRIVILEGES ] }
```
The `REVOKE` command revokes privileges that were granted to one or more roles. The key word `PUBLIC` refers to the implicitly defined group of all roles.

See `GRANT` for the meaning of the privilege types.

A role has the sum of:

- Privileges granted directly to it
- Privileges granted to any role it is presently a member of
- Privileges granted to `PUBLIC`
Thus, for example, revoking the `SELECT` privilege from `PUBLIC` doesn't necessarily mean that all roles lose `SELECT` privilege on the object. Roles that were granted the privilege directly or from another role still have it.

If the privilege was granted with the grant option, the grant option for the privilege is revoked along with the privilege.

If a user holds a privilege with grant option and granted it to other users, then the privileges held by those other users are called dependent privileges. If the privilege or the grant option held by the first user is revoked, any dependent privileges are also revoked if `CASCADE` is specified. Without the `CASCADE` option, the revoke action fails. This recursive revocation affects only privileges that were granted through a chain of users that's traceable to the subject of this `REVOKE` command. The affected users can keep the privilege if it was also granted through other users.

**Note**

The `CASCADE` option isn't compatible with Oracle databases. By default, Oracle always cascades dependent privileges. EDB Postgres Advanced Server requires the `CASCADE` keyword for the `REVOKE` command to succeed.

When revoking membership in a role, `GRANT OPTION` is called `ADMIN OPTION`. The behavior is similar.

**Notes**

A user can revoke only the privileges that were granted directly by that user. If, for example, user A granted a privilege with grant option to user B, and user B granted it to user C, then user A can't revoke the privilege directly from C. Instead, user A can revoke the grant option from user B and use the `CASCADE` option to revoke the privilege from user C. For another example, if both A and B granted the same privilege to C, A can revoke their own grant but not B's grant. C still has the privilege.

When a non-owner of an object attempts to revoke privileges on the object, the command fails if the user doesn't have privileges on the object. As long as some privilege is available, the command proceeds, but it revokes only those privileges for which the user has grant options. The `REVOKE ALL PRIVILEGES` forms issue a warning message if no grant options are held. The other forms issue a warning if grant options for any of the privileges named in the command aren't held. In principle, these statements apply to the object owner as well. However, since the owner is always treated as holding all grant options, the cases can never occur.

If a superuser issues a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. Since all privileges ultimately come from the object owner (possibly indirectly by way of chains of grant options), a superuser can revoke all privileges. This might require use of `CASCADE`.

A role that is not the owner of the affected object can also use `REVOKE`. That role must be a member of the role that owns the object or a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case, the command is performed as if issued by the containing role that owns the object or holds the privileges `WITH GRANT OPTION`. For example, if table t1 is owned by role g1, of which role u2 is a member, then u1 can revoke privileges on t1 that are recorded as being granted by g1. This includes grants made by u2 as well as by other members of role g1.

If the role executing `REVOKE` holds privileges indirectly by more than one role membership path, the containing role that performs the command is unspecified. In such cases, best practice is to use `SET ROLE` to become the specific role you want to do the `REVOKE` as. Otherwise, you might revoke privileges other than the ones you intended or not revoke anything at all.

**Note**

The EDB Postgres Advanced Server `ALTER ROLE` command also supports syntax that revokes the system privileges required to create a public or private database link or exemptions from fine-grained access control policies (DBMS_RLS). The `ALTER ROLE` syntax is functionally equivalent to the respective `REVOKE` command, compatible with Oracle databases.

**Examples**

Revoke insert privilege for the public on table `emp`:

```
REVOKE INSERT ON emp FROM PUBLIC;
```

Revoke all privileges from user `mary` on view `salesemp`. This actually means "revoke all privileges that I granted."

```
REVOKE ALL PRIVILEGES ON salesemp FROM mary;
```

Revoke membership in role `admins` from user `joe`:

```
REVOKE admins FROM joe;
```

Revoke `CONNECT` privilege from user `joe`:

```
REVOKE CONNECT FROM joe;
```

Revoke `CREATE DATABASE LINK` privilege from user `joe`:

```
REVOKE CREATE DATABASE LINK FROM joe;
```

Revoke the `EXEMPT ACCESS POLICY` privilege from user `joe`:
### ROLLBACK

#### Name

```
ROLLBACK  [ WORK
```

#### Description

ROLLBACK rolls back the current transaction and discards all the updates made by the transaction.

#### Parameters

**WORK**

Optional keyword that has no effect.

#### Notes

- Use `COMMIT` to successfully terminate a transaction.
- Issuing `ROLLBACK` when not inside a transaction does no harm.

**Note**

Executing a `ROLLBACK` in a plpgsql procedure throws an error if there's an Oracle-style SPL procedure on the runtime stack.

#### Examples

Abort all changes:

```
ROLLBACK;
```

#### See also

- `COMMIT`
- `ROLLBACK TO SAVEPOINT`
- `SAVEPOINT`
14.3.5.73 ROLLBACK TO SAVEPOINT

Name

ROLLBACK TO SAVEPOINT — Roll back to a savepoint.

Synopsis

ROLLBACK [ WORK ] TO [ SAVEPOINT ]
<savepoint_name>

Description

Roll back all commands that were executed after the savepoint was set. The savepoint remains valid and you can roll back to it again if you need to.

ROLLBACK TO SAVEPOINT destroys all savepoints that were established after the named savepoint.

Parameters

savepoint_name

The savepoint to which to roll back.

Notes

Specifying a savepoint name that wasn’t established is an error.

ROLLBACK TO SAVEPOINT isn’t supported in SPL programs.

Examples

Undo the effects of the commands executed from the point when savepoint depts was established:

\set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
ROLLBACK TO SAVEPOINT depts;

See also

COMMIT, ROLLBACK, SAVEPOINT

14.3.5.74 SAVEPOINT

Name
**SAVEPOINT** — Define a new savepoint in the current transaction.

**Synopsis**

**SAVEPOINT** <savepoint_name>

**Description**

**SAVEPOINT** establishes a new savepoint in the current transaction.

A savepoint is a mark inside a transaction that allows all commands that are executed after it to be rolled back. This restores the transaction state to what it was at the savepoint.

**Parameters**

savepoint_name

The name of the savepoint.

**Notes**

Use **ROLLBACK TO SAVEPOINT** to roll back to a savepoint.

You can establish savepoints only when inside a transaction block. You can define multiple savepoints in a transaction.

When another savepoint is established with the same name as a previous savepoint, the old savepoint is kept. However, only the more recent one is used when rolling back.

**SAVEPOINT** isn't supported in SPL programs.

**Examples**

Establish a savepoint and then undo the effects of all commands executed after it:

```
\set AUTOCOMMIT off
INSERT INTO dept VALUES (50, 'HR', 'NEW YORK');
SAVEPOINT depts;
INSERT INTO emp (empno, ename, deptno) VALUES (9001, 'JONES', 50);
INSERT INTO emp (empno, ename, deptno) VALUES (9002, 'ALICE', 50);
SAVEPOINT emps;
INSERT INTO jobhist VALUES (9001,'17-SEP-07',NULL,'CLERK',808,NULL,50,'New Hire');
INSERT INTO jobhist VALUES (9002,'20-SEP-07',NULL,'CLERK',708,NULL,50,'New Hire');
ROLLBACK TO depts;
COMMIT;
```

This transaction commits a row into the dept table. The inserts into the emp and jobhist tables are rolled back.

**See also**

**COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT**

14.3.5.75 **SELECT**
Name

**SELECT** – Retrieve rows from a table or view.

Synopsis

```sql
SELECT [ optimizer_hint ] [ ALL | DISTINCT | UNIQUE ]
   [ * | expression [ AS output_name ] [, ...]
   FROM from_item [, ... ]
   [ WHERE condition ]
   [ [ START WITH start_expression ]
      CONNECT BY ( PRIOR parent_expr = child_expr )
      child_expr = PRIOR parent_expr
   ]
   [ ORDER SIBLINGS BY expression [ ASC | DESC ] [, ... ] ]
   [ GROUP BY ( expression | ROLLUP ( expr_list ) ]
      CUBE ( expr_list ) | GROUPING SETS ( expr_list ) ] [, ... ]
   [ LEVEL ]
   [ HAVING condition [, ... ]
   ]
   [ ( UNION [ ALL ] | INTERSECT | MINUS ) select ]
   [ ORDER BY expression [ ASC | DESC ] [, ... ] ]
   [ FOR UPDATE [ WAIT n | NOWAIT | SKIP LOCKED ] ]
```

Where `from_item` can be one of:

- `table_name[[dblink ] [ alias]
  ( select )
  alias
  from_item [ NATURAL ] join_type
  from_item
  [ ON join_condition | USING ( join_column [, ... ] ) ]`

Description

**SELECT** retrieves rows from one or more tables. The general processing of **SELECT** is as follows:

1. All elements in the **FROM** list are computed. (Each element in the **FROM** list is a real or virtual table.) If you specify more than one element in the **FROM** list, they are cross joined. See **FROM** clause.
2. If you specify the **WHERE** clause, all rows that don’t satisfy the condition are eliminated from the output. See **WHERE** clause.
3. If you specify the **GROUP BY** clause, the output is divided into groups of rows that match one or more values. The **HAVING** clause eliminates groups that don’t satisfy the given condition. See **GROUP BY** clause and **HAVING** clause.
4. Using the operators **UNION**, **INTERSECT**, and **MINUS**, you can combine the output of more than one **SELECT** statement to form a single result set. The **UNION** operator returns all rows that are in one or both of the result sets. The **INTERSECT** operator returns all rows that are strictly in both result sets. The **MINUS** operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated. In the case of the **UNION** operator, if you specify **ALL**, then duplicates aren’t eliminated. See **UNION** clause, **INTERSECT** clause, and **MINUS** clause.
5. The actual output rows are computed using the **SELECT** output expressions for each selected row. See **SELECT** list.
6. The **CONNECT BY** clause is used to select data that has a hierarchical relationship. Such data has a parent-child relationship between rows. See **CONNECT BY** clause.
7. If you specify the **ORDER BY** clause, the returned rows are sorted in the specified order. Otherwise, the rows are returned in the order the system finds fastest to produce. See **ORDER BY** clause.
8. **DISTINCT | UNIQUE** eliminates duplicate rows from the result. **ALL** (the default) returns all candidate rows, including duplicates. See **DISTINCT | UNIQUE** clause.
9. The **FOR UPDATE** clause causes the **SELECT** statement to lock the selected rows against concurrent updates. See **FOR UPDATE** clause.

You must have **SELECT** privilege on a table to read its values. The use of **FOR UPDATE** requires **UPDATE** privilege as well.

Parameters

`optimizer_hint`
FROM clause

The FROM clause specifies one or more source tables for a SELECT statement. The syntax is:

```
FROM source [, ...]
```

Where `source` can be one of following elements:

- `table_name[dblink ]`
  The name (optionally schema-qualified) of an existing table or view. `dblink` is a database link name identifying a remote database. See the CREATE DATABASE LINK command for information on database links.

- `alias`
  A substitute name for the FROM item containing the alias. Use an alias for brevity or to eliminate ambiguity for self-joins where the same table is scanned multiple times. Providing an alias completely hides the name of the table or function. For example, given `FROM foo AS f`, the remainder of the SELECT must refer to this FROM item as `f`, not `foo`.

- `select`
  A sub-SELECT can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. Surround the sub-SELECT with parentheses, and provide an alias for it.

- `join_type`
  One of the following:

  - INNER JOIN
  - LEFT [ OUTER ] JOIN
  - RIGHT [ OUTER ] JOIN
  - FULL [ OUTER ] JOIN
  - CROSS JOIN

  For the INNER and OUTER join types, you must specify a join condition, namely one of NATURAL, ON `join_condition`, or USING (join_column [, ... ]). For CROSS JOIN, you can't use any of these clauses.

A JOIN clause combines two FROM items. Use parentheses to determine the order of nesting. Without parentheses, joins nest left-to-right. In any case JOINs bind more tightly than the commas separating FROM items.

```
CROSS JOIN and INNER JOIN produce a simple Cartesian product, the same result that you get from listing the two items at the top level of FROM but restricted by any join condition. CROSS JOIN is equivalent to INNER JOIN ON (TRUE), that is, no rows are removed by qualification. These join types are a notational convenience, since you can accomplish the same thing using FROM and WHERE.
```

LEFT OUTER JOIN returns all rows in the qualified Cartesian product, that is, all combined rows that pass its join condition. It also returns one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Only the JOIN clause's own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.

Conversely, RIGHT OUTER JOIN returns all the joined rows plus one row for each unmatched right-hand row extended with nulls on the left. This is a notational convenience, since you can convert it to a LEFT OUTER JOIN by switching the left and right inputs.

```
FULL OUTER JOIN returns all the joined rows, one row for each unmatched left-hand row extended with nulls on the right, and one row for each unmatched right-hand row extended with nulls on the left.
```

ON `join_condition`

A clause of the form USING (a, b, ... ) is shorthand for ON left_table.a = right_table.a AND left_table.b = right_table.b .... Also, USING implies that only one of each pair of equivalent columns is included in the join output, not both.
**NATURAL** is shorthand for a **USING** list that mentions all columns in the two tables that have the same names.

If you specify multiple sources, the result is the Cartesian product (cross join) of all the sources. Usually qualification conditions are added to restrict the returned rows to a small subset of the Cartesian product.

**Example**

This example selects all of the entries from the **dept** table:

```
SELECT * FROM dept;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>

**WHERE clause**

The optional **WHERE** clause has the form:

```
WHERE condition
```

where **condition** is any expression that evaluates to a result of type **BOOLEAN**. Any row that doesn't satisfy this condition is eliminated from the output. A row satisfies the condition if it returns **TRUE** when the actual row values are substituted for any variable references.

**Example**

This example joins the contents of the **emp** and **dept** tables. The value of the **deptno** column in the **emp** table is equal to the value of the **deptno** column in the **dept** table.

```
SELECT d.deptno, d.dname, e.empno, e.ename, e.mgr, e.hiredate
FROM emp e, dept d
WHERE d.deptno = e.deptno;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>empno</th>
<th>ename</th>
<th>mgr</th>
<th>hiredate</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7934</td>
<td>MILLER</td>
<td>7782</td>
<td>23-JAN-82 00:00:00</td>
</tr>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7782</td>
<td>CLARK</td>
<td>7839</td>
<td>09-JUN-81 00:00:00</td>
</tr>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>7839</td>
<td>KING</td>
<td></td>
<td>17-NOV-81 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7788</td>
<td>SCOTT</td>
<td>7566</td>
<td>19-APR-87 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7566</td>
<td>JONES</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7369</td>
<td>SMITH</td>
<td>7982</td>
<td>17-DEC-80 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7876</td>
<td>ADAMS</td>
<td>7788</td>
<td>23-MAY-87 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7902</td>
<td>FORD</td>
<td>7566</td>
<td>03-DEC-81 00:00:00</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>7902</td>
<td>JAMES</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7521</td>
<td>WARD</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7698</td>
<td>TURNER</td>
<td>7698</td>
<td>30-SEP-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7499</td>
<td>ALLEN</td>
<td>7698</td>
<td>01-MAY-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7698</td>
<td>BLAKE</td>
<td>7839</td>
<td>01-MAY-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7654</td>
<td>MARTIN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>7900</td>
<td>JAMES</td>
<td>7698</td>
<td>03-DEC-81 00:00:00</td>
</tr>
</tbody>
</table>

**GROUP BY clause**

The optional **GROUP BY** clause has the form:

```
GROUP BY { expression | ROLLUP ( expr_list )
| CUBE ( expr_list ) | GROUPING SETS ( expr_list ) } [,...
```

**GROUP BY** condenses all selected rows that share the same values for the grouped expressions into a single row. **expression** can be an input column name or the name or ordinal number of an
output column (SELECT list item). Or it can be an arbitrary expression formed from input-column values. In case of ambiguity, a GROUP BY name is interpreted as an input-column name rather than an output column name.

ROLLUP, CUBE, and GROUPING SETS are extensions to the GROUP BY clause for supporting multidimensional analysis.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group. Without GROUP BY, an aggregate produces a single value computed across all the selected rows. When GROUP BY is present, it isn't valid for the SELECT list expressions to refer to ungrouped columns except in aggregate functions, since there is more than one possible value to return for an ungrouped column.

Example

This example computes the sum of the sal column in the emp table, grouping the results by department number:

```sql
SELECT deptno, SUM(sal) AS total
FROM emp
GROUP BY deptno;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8750.00</td>
</tr>
<tr>
<td>20</td>
<td>10875.00</td>
</tr>
<tr>
<td>30</td>
<td>9400.00</td>
</tr>
</tbody>
</table>

(H3 rows)

HAVING clause

The optional HAVING clause has the form:

```
HAVING condition
```

Where condition is the same as specified for the WHERE clause.

HAVING eliminates group rows that don't satisfy the specified condition. HAVING is different from WHERE. WHERE filters individual rows before applying GROUP BY, while HAVING filters group rows created by GROUP BY. Each column referenced in condition must unambiguously reference a grouping column, unless the reference appears in an aggregate function.

Example

Sum the column sal of all employees, group the results by department number, and show those group totals that are less than 10000:

```sql
SELECT deptno, SUM(sal) AS total
FROM emp
GROUP BY deptno
HAVING SUM(sal) < 10000;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8750.00</td>
</tr>
<tr>
<td>30</td>
<td>9400.00</td>
</tr>
</tbody>
</table>

(H2 rows)

SELECT List

The SELECT list (between the keywords SELECT and FROM) specifies expressions that form the output rows of the SELECT statement. The expressions can refer to columns computed in the FROM clause, and they usually do. Using the clause AS output_name, you can specify another name for an output column. This name is primarily used to label the column for display. You can also use it to refer to the column’s value in ORDER BY and GROUP BY clauses but not in the WHERE or HAVING clauses. In those clauses, you must write out the expression.

Instead of an expression, you can write * in the output list as a shorthand for all the columns of the selected rows.

Example

The SELECT list in this example specifies for the result set to include the empno column, the ename column, the mgr column, and the hiredate column:
The `UNION` clause

The `UNION` clause has the form:

```
select_statement UNION [ ALL ] select_statement
```

Where `select_statement` is any `SELECT` statement without an `ORDER BY` or `FOR UPDATE` clause. You can attach `ORDER BY` to a sub-expression if you enclose it in parentheses. Without parentheses, these clauses apply to the result of the `UNION`, not to its right-hand input expression.

The `UNION` operator computes the set union of the rows returned by the involved `SELECT` statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two `SELECT` statements that represent the direct operands of the `UNION` must produce the same number of columns, and corresponding columns must have compatible data types.

The result of `UNION` doesn't contain any duplicate rows unless you specify the `ALL` option. `ALL` prevents eliminating duplicates.

Without parentheses, multiple `UNION` operators in the same `SELECT` statement are evaluated left to right.

Currently, you can't specify `FOR UPDATE` either for a `UNION` result or for any input of a `UNION`.

The `INTERSECT` clause

The `INTERSECT` clause has the form:

```
select_statement INTERSECT select_statement
```

Where `select_statement` is any `SELECT` statement without an `ORDER BY` or `FOR UPDATE` clause.

The `INTERSECT` operator computes the set intersection of the rows returned by the involved `SELECT` statements. A row is in the intersection of two result sets if it appears in both result sets.

The result of `INTERSECT` doesn't contain any duplicate rows.

Without parentheses, multiple `INTERSECT` operators in the same `SELECT` statement are evaluated left to right. `INTERSECT` binds more tightly than `UNION`. That is, `A UNION B INTERSECT C` is read as `A UNION (B INTERSECT C)`.

The `MINUS` clause

The `MINUS` clause has this general form:

```
select_statement MINUS select_statement
```

Where `select_statement` is any `SELECT` statement without an `ORDER BY` or `FOR UPDATE` clause.

The `MINUS` operator computes the set of rows that are in the result of the left `SELECT` statement but not in the result of the right one.
The result of `MINUS` doesn't contain any duplicate rows.

Without parentheses, multiple `MINUS` operators in the same `SELECT` statement are evaluated left to right. `MINUS` binds at the same level as `UNION`.

**CONNECT BY clause**

The `CONNECT BY` clause determines the parent-child relationship of rows when performing a hierarchical query. It has the general form:

```
CONNECT BY ( PRIOR parent_expr = child_expr
   | child_expr = PRIOR parent_expr )
```

Where `parent_expr` is evaluated on a candidate parent row. If `parent_expr = child_expr` results in `TRUE` for a row returned by the `FROM` clause, then this row is considered a child of the parent.

You can specify the following optional clauses with the `CONNECT BY` clause.

**START WITH start_expression**

The rows returned by the `FROM` clause on which `start_expression` evaluates to `TRUE` become the root nodes of the hierarchy.

**ORDER SIBLINGS BY expression [ ASC | DESC ] [, ...]**

Sibling rows of the hierarchy are ordered by `expression` in the result set.

**Note**

EDB Postgres Advanced Server doesn't support the use of `AND` or other operators in the `CONNECT BY` clause.

**ORDER BY clause**

The optional `ORDER BY` clause has the form:

```
ORDER BY expression [ ASC | DESC ] [, ...]
```

Where `expression` can be the name or ordinal number of an output column (`SELECT` list item), or it can be an arbitrary expression formed from input-column values.

The `ORDER BY` clause sorts the result rows according to the specified expressions. If two rows are equal according to the left-most expression, they are compared according to the next expression, and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

The ordinal number refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that doesn't have a unique name. This is never required because you can always assign a name to a result column using the `AS` clause.

You also can use arbitrary expressions in the `ORDER BY` clause, including columns that don't appear in the `SELECT` result list. Thus the following statement is valid:

```
SELECT ename FROM emp ORDER BY enmpno;
```

A limitation of this feature is that an `ORDER BY` clause applying to the result of a `UNION`, `INTERSECT`, or `MINUS` clause can specify only an output column name or number, not an expression.

If an `ORDER BY` expression is a simple name that matches both a result column name and an input column name, `ORDER BY` interprets it as the result column name. This behavior is the opposite of the choice that `GROUP BY` makes in the same situation. This inconsistency is compatible with the SQL standard.

Optionally, you can add the key word `ASC` (ascending) or `DESC` (descending) after any expression in the `ORDER BY` clause. `ASC` is the default.

The null value sorts higher than any other value. In other words, with ascending sort order, null values sort at the end. With descending sort order, null values sort at the beginning.

Character-string data is sorted according to the locale-specific collation order that was established when the database cluster was initialized.

**Note**

If you specify `SELECT DISTINCT` or if a `SELECT` statement includes the `SELECT DISTINCT ...ORDER BY` clause, then all the expressions in `ORDER BY` must be present in the select list of the `SELECT DISTINCT` query.
Examples

These examples sort the individual results according to the contents of the second column (`dname`):

```
SELECT * FROM dept ORDER BY
dname;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>

(4 rows)

```
SELECT * FROM dept ORDER BY 2;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>dname</th>
<th>loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>

(4 rows)

This example uses the `SELECT DISTINCT ...ORDER BY` clause to fetch the `job` and `deptno` from table `emp`:

```
CREATE TABLE emp(EMPNO NUMBER(4) NOT NULL,
    ENAME VARCHAR2(10),
    JOB VARCHAR2(9),
    DEPTNO NUMBER(2));

INSERT INTO emp VALUES (7369, 'SMITH', 'CLERK', 20);
INSERT 0 1
INSERT INTO emp VALUES (7499, 'ALLEN', 'SALESMAN', 30);
INSERT 0 1
INSERT INTO emp VALUES (7521, 'WARD', 'SALESMAN', 30);
INSERT 0 1
INSERT INTO emp VALUES (7566, 'JONES', 'MANAGER', 20);
INSERT 0 1

SELECT DISTINCT e.job, e.deptno FROM emp e ORDER BY e.job,
e.deptno;
```

<table>
<thead>
<tr>
<th>job</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLERK</td>
<td>20</td>
</tr>
<tr>
<td>MANAGER</td>
<td>20</td>
</tr>
<tr>
<td>SALESMAN</td>
<td>30</td>
</tr>
</tbody>
</table>

(3 rows)

**DISTINCT | UNIQUE clause**

If a `SELECT` statement specifies `DISTINCT` or `UNIQUE`, all duplicate rows are removed from the result set. One row is kept from each group of duplicates. The `DISTINCT` or `UNIQUE` clauses are synonymous when used with a `SELECT` statement. The `ALL` keyword specifies the opposite, which is that all rows are kept (the default).

Error messages resulting from the improper use of a `SELECT` statement that includes the `DISTINCT` or `UNIQUE` keywords include both the `DISTINCT | UNIQUE` keywords:

```
psql: ERROR:  FOR UPDATE is not allowed with DISTINCT/UNIQUE clause
```

**FOR UPDATE clause**

The `FOR UPDATE` clause takes the form:
FOR UPDATE causes the rows retrieved by the SELECT statement to be locked as though for update. This prevents a row from being modified or deleted by other transactions until the current transaction ends. Any transaction that attempts to UPDATE, DELETE, or SELECT FOR UPDATE a selected row is blocked until the current transaction ends. If an UPDATE, DELETE, or SELECT FOR UPDATE from another transaction has already locked a selected row or rows, SELECT FOR UPDATE waits for the first transaction to complete. It then locks and returns the updated row, or it returns no row if the row was deleted.

You can’t use FOR UPDATE in contexts where you can't clearly identify returned rows with individual table rows, for example, with aggregation.

Use FOR UPDATE options to specify locking preferences:

- Include the WAIT n keywords to specify the number of seconds or fractional seconds for the SELECT statement to wait for a row locked by another session. Use a decimal form to specify fractional seconds. For example, WAIT 1.5 waits one and a half seconds. Specify up to four digits to the right of the decimal.
- Include the NOWAIT keyword to report an error immediately if the current session can't lock a row.
- Include SKIP LOCKED to lock rows, if possible, and skip rows that are already locked by another session.

### 14.3.5.76 SET CONSTRAINTS

**Name**

SET CONSTRAINTS — Set constraint-checking modes for the current transaction.

**Synopsis**

```
SET CONSTRAINTS { ALL | name [, ..] } { DEFERRED | IMMEDIATE }
```

**Description**

SET CONSTRAINTS sets the behavior of constraint checking in the current transaction. IMMEDIATE constraints are checked at the end of each statement. DEFERRED constraints are checked when the transaction commits. Each constraint has its own IMMEDIATE or DEFERRED mode.

When you create a constraint, you give it one of three characteristics: DEFERRABLE INITIALLY DEFERRED, DEFERRABLE INITIALLY IMMEDIATE, or NOT DEFERRABLE. The third class is always IMMEDIATE and isn’t affected by the SET CONSTRAINTS command. The first two classes start every transaction in the indicated mode, but you can change their behavior in a transaction by using SET CONSTRAINTS.

SET CONSTRAINTS with a list of constraint names changes the mode of just those constraints. Those constraints must all be deferrable. If multiple constraints match any given name, all are affected. SET CONSTRAINTS ALL changes the mode of all deferrable constraints.

When SET CONSTRAINTS changes the mode of a constraint from DEFERRED to IMMEDIATE, the new mode takes effect retroactively. Any outstanding data modifications that normally are checked at the end of the transaction are instead checked while SET CONSTRAINTS executes. If any such constraint is violated, the SET CONSTRAINTS fails and doesn’t change the constraint mode. Thus, you can use SET CONSTRAINTS to force constraint checking to occur at a specific point in a transaction.

Currently, only foreign key constraints are affected by this setting. Check and unique constraints are never deferrable.

**Notes**

This command alters only the behavior of constraints in the current transaction. If you execute this command outside of a transaction block, it doesn’t have any effect.

### 14.3.5.77 SET ROLE

**Name**

SET ROLE — Set the current user identifier of the current session.
**Synopsis**

```sql
SET ROLE { rolename | NONE }
```

**Description**

This command sets the current user identifier of the current SQL session context to `<rolename>`. After `SET ROLE`, permissions checking for SQL commands is carried out as though the named role is the one that logged in originally.

The specified role name must be a role that the current session user is a member of. A superuser can select any role.

`NONE` resets the current user identifier to the current session user identifier. Any user can execute these forms.

**Notes**

You can use this command to add or restrict privileges. If the session user role has the `INHERITS` attribute, then it automatically has all the privileges of every role that it can set the role to. In this case, `SET ROLE` drops all the privileges assigned directly to the session user and to the other roles it is a member of, which leaves only the privileges available to the named role. If the session user role has the `NOINHERITS` attribute, `SET ROLE` drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role. When a superuser chooses to set a role to a non-superuser role, they lose their superuser privileges.

**Examples**

User `mary` takes on the identity of role `admins`:

```
SET ROLE admins;
```

User `mary` reverts back to their own identity:

```
SET ROLE NONE;
```

14.3.5.78 **SET TRANSACTION**

**Name**

`SET TRANSACTION` — Set the characteristics of the current transaction.

**Synopsis**

```sql
SET TRANSACTION transaction_mode
```

Where `transaction_mode` is one of:

```sql
ISOLATION LEVEL { SERIALIZABLE | READ COMMITTED }
READ WRITE | READ ONLY
```

**Description**

The `SET TRANSACTION` command sets the characteristics of the current transaction. It has no effect on any later transactions. The available transaction characteristics are the transaction isolation level and the transaction access mode (read/write or read-only). The isolation level of a transaction determines the data the transaction can see when other transactions are running concurrently.
A statement can see only rows committed before it began. This is the default.

**SERIALIZABLE**

All statements of the current transaction can see only rows committed before the first query or data-modification statement was executed in this transaction.

You can't change the transaction isolation level after the first query or data-modification statement (SELECT, INSERT, DELETE, UPDATE, or FETCH) of a transaction is executed. The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default.

When a transaction is read-only, the following SQL commands aren't allowed:

- INSERT, UPDATE, and DELETE if the table they write to isn't a temporary table
- All CREATE, ALTER, and DROP commands
- COMMENT, GRANT, REVOKE, TRUNCATE
- EXECUTE if the command it executes is among those listed.

This is a high-level notion of read-only that doesn't prevent all writes to disk.

### 14.3.5.79 TRUNCATE

**Name**

TRUNCATE — Empty a table.

**Synopsis**

```
TRUNCATE TABLE <name> [DROP STORAGE]
```

**Description**

TRUNCATE removes all rows from a table. It has the same effect as an unqualified DELETE. However, since it doesn't scan the table, it's faster. TRUNCATE is most useful on large tables.

The DROP STORAGE clause is accepted for compatibility but is ignored.

**Parameters**

- *name*

  The name (optionally schema-qualified) of the table to truncate.

**Notes**

You can't use TRUNCATE if there are foreign-key references to the table from other tables. Checking validity in such cases requires table scans, which TRUNCATE aims to avoid.

TRUNCATE doesn't run any user-defined ON DELETE triggers for the table.

**Examples**

This command truncates a table named accounts:

```
TRUNCATE TABLE accounts;
```
See also

DROP VIEW, DELETE

14.3.5.80 UPDATE

Name

`UPDATE` — Update rows of a table.

Synopsis

```sql
edn# \h UPDATE
Command:  UPDATE
Description: update rows of a table
Syntax:
  [ WITH [ RECURSIVE ] with_query [, ...]
  UPDATE [ <optimizer_hint> ] [ ONLY ] <table_name>[@<dblink> ] [ * ] [ [ AS ] alias
  ]
  SET { { <column_name> = ( <expression> | DEFAULT }
  |          ( <column_name> [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] )
  |          ( <column_name> [, ...] ) = ( sub-SELECT
  ) [, ...]
  )
  ROW = <row_or_record_var>
  )
  [ FROM <from_item> [, ...]
  ]
  [ WHERE <condition> | WHERE CURRENT OF <cursor_name>
  ]
  [ RETURNING <return_expression> [, ...]
  ]
  [ INTO { <record> | <variable> [, ...]
  } | BULK COLLECT INTO <collection> [, ...] ]
```

URL:
https://www.postgresql.org/docs/15/sql-update.html
https://www.enterprisedb.com/docs

Description

`UPDATE` changes the values of the specified columns in all rows that satisfy the condition. You need to mention only the columns you want to modify in the `SET` clause. Columns not explicitly modified retain their values.

`SET ROW` enables us to update a target record using a record type variable or row type objects. The record or row used, must have compatible data types with the table's columns in order.

You can specify the `RETURNING INTO { record | variable [, ...] }` clause only in an SPL program. In addition, the result set of the `UPDATE` command must not return more than one row. Otherwise an exception is thrown. If the result set is empty, then the contents of the target record or variables are set to null.

In an SPL program, you can specify the `RETURNING BULK COLLECT INTO collection [, ...]` clause only if you use the `UPDATE` command. If you specify more than one collection as the target of the `BULK COLLECT INTO` clause, then each collection must consist of a single scalar field. That is, `collection` can't be a record. The result set of the `UPDATE` command can contain zero or more rows. `return_expression` evaluated for each row of the result set becomes an element in `collection` starting with the first element. Any existing rows in `collection` are deleted. If the result set is empty, then `collection` is empty.

You need the `UPDATE` privilege on the table to update it. You also need the `SELECT` privilege to any table whose values are read in `expression` or `condition`.

Parameters
Comment-embedded hints to the optimizer for selecting an execution plan.

The name (optionally schema-qualified) of the table to update.

Database link name identifying a remote database. See the `CREATE DATABASE LINK` command for information on database links.

The name of a column in the table.

An expression to assign to the column. The expression can use the old values of this and other columns in the table.

Set the column to its default value, which is null if you don’t assign a default expression to it.

An expression that returns a value of type `BOOLEAN`. Update only rows for which this expression returns true.

An expression that can include one or more columns from the table. If a column name from the table is specified in `return_expression`, the value substituted for the column when `return_expression` is evaluated is determined as follows:

- If the column specified in `return_expression` is assigned a value in the `UPDATE` command, then the assigned value is used when evaluating `return_expression`
- If the column specified in `return_expression` isn’t assigned a value in the `UPDATE` command, then the column’s current value in the affected row is used when evaluating `return_expression`

A record whose field to assign the evaluated `return_expression`. The first `return_expression` is assigned to the first field in `record`, the second `return_expression` is assigned to the second field in `record`, and so on. The number of fields in `record` must exactly match the number of expressions, and the fields must be type-compatible with their assigned expressions.

A variable to which to assign the evaluated `return_expression`. If more than one `return_expression` and `variable` are specified, the first `return_expression` is assigned to the first `variable`, the second `return_expression` is assigned to the second `variable`, and so on. The number of variables specified following the `INTO` keyword must exactly match the number of expressions following the `RETURNING` keyword, and the variables must be type-compatible with their assigned expressions.

A collection in which an element is created from the evaluated `return_expression`. There can be either:

- A single collection, which can be a collection of a single field or a collection of a record type
- More than one collection, in which case each collection must consist of a single field

The number of return expressions must match in number and order the number of fields in all specified collections. Each corresponding `return_expression` and `collection` field must be type-compatible.

Examples

Change the location to `AUSTIN` for department 20 in the `dept` table:

```
UPDATE dept
SET loc = 'AUSTIN'
WHERE deptno = 20;
```

For all employees with `job = SALESMAN` in the `emp` table, update the salary by 10%, and increase the commission by 500:

```
UPDATE emp
SET salary = salary * 1.1,
    commission = commission + 500;
```
### 14.3.6 List of Oracle compatible configuration parameters

EDB Postgres Advanced Server supports developing and running applications compatible with PostgreSQL and Oracle. You can alter some system behaviors to act in a more PostgreSQL- or in a more Oracle-compliant manner. You control these behaviors by using configuration parameters.

- **edb_redwood_date** — Controls whether or not a time component is stored in `DATE` columns. For behavior compatible with Oracle databases, set `edb_redwood_date` to `TRUE`.  
- **edb_redwood_raw_names** — Controls whether database object names appear in uppercase or lowercase letters when viewed from Oracle system catalogs. For behavior compatible with Oracle databases, `edb_redwood_raw_names` is set to its default value of `FALSE`. To view database object names as they are actually stored in the PostgreSQL system catalogs, set `edb_redwood_raw_names` to `TRUE`.  
- **edb_redwood_strings** — Equates `NULL` to an empty string for purposes of string concatenation operations. For behavior compatible with Oracle databases, set `edb_redwood_strings` to `TRUE`.  
- **edb_stmt_level_tx** — Isolates automatic rollback of an aborted SQL command to statement level rollback only – the entire, current transaction is not automatically rolled back as is the case for default PostgreSQL behavior. For behavior compatible with Oracle databases, set `edb_stmt_level_tx` to `TRUE`; however, use only when absolutely necessary.  
- **oracle_home** — Point EDB Postgres Advanced Server to the correct Oracle installation directory.

See Configuration parameters compatible with Oracle databases for access to more detailed information.

### 14.4 SQL reference

This SQL reference information applies to organizations migrating their Oracle applications to use EDB Postgres Advanced Server.

A subset of the EDB Postgres Advanced Server SQL language is compatible with Oracle databases. These SQL syntaxes, data types, and functions work in both EDB Postgres Advanced Server and Oracle.

**Note**

The EDB Postgres Advanced Server also includes syntax and commands for extended functionality (functionality that doesn't provide database compatibility for Oracle or support Oracle-styled applications) that aren't included here.

#### 14.4.1 SQL syntax

Understanding the general syntax of SQL forms the foundation for understanding how to apply the SQL commands to define and modify data.

##### 14.4.1.1 Lexical structure

The lexical structure of SQL has several aspects:

- **SQL input** consists of a sequence of commands.  
- A command is composed of a sequence of tokens, terminated by a semicolon `;`. The end of the input stream also terminates a command.  
- The valid tokens depend on the syntax of the command.  
- A token can be a key word, an identifier, a quoted identifier, a literal or constant, or a special character symbol. Tokens are normally separated by whitespace (space, tab, new line) but don't need to be if there's no ambiguity. This is generally the case only if a special character is adjacent to some other token type.  
- Comments can occur in SQL input. They aren't tokens. They are equivalent to whitespace.
For example, the following is syntactically valid SQL input:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there!');
```

This is a sequence of three commands, one per line, although that format isn’t required. You can enter more than one command on a line, and commands can usually split across lines.

The SQL syntax isn’t very consistent regarding the tokens that identify commands and the ones that are operands or parameters. The first few tokens are generally the command name, so the example contains a SELECT, an UPDATE, and an INSERT command. But, for instance, the UPDATE command always requires a SET token to appear in a certain position, and this variation of INSERT also requires a VALUES token to be complete. The precise syntax rules for each command are described in SQL reference.

### 14.4.1.2 Identifiers and key words

Tokens such as SELECT, UPDATE, or VALUES are examples of key words, that is, words that have a fixed meaning in the SQL language. The tokens MY_TABLE and A are examples of identifiers. They identify names of tables, columns, or other database objects, depending on the command you use them in. Therefore, they’re sometimes called names. Key words and identifiers have the same lexical structure, meaning that you can’t know whether a token is an identifier or a key word without knowing the language.

SQL identifiers and key words must begin with a letter (a-z or A-Z). Subsequent characters in an identifier or key word can be letters, underscores, digits (0-9), dollar signs ($), or the number sign (#). Identifier and key word names aren’t case sensitive. Therefore these two commands are equivalent:

```
UPDATE MY_TABLE SET A = 5;
```

The equivalent command is:

```
UPDATE my_TABLE SET a = 5;
```

A convention often used is to write key words in upper case and names in lower case, for example:

```
UPDATE my_table SET a = 5;
```

A second kind of identifier is the delimited identifier or quoted identifier. It’s formed by enclosing an arbitrary sequence of characters in double quotes ("). A delimited identifier is always an identifier, never a key word. So you can use "select" to refer to a column or table named select. An unquoted select is taken as a key word and therefore provokes a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

```
UPDATE "my_table" SET "a" = 5;
```

Quoted identifiers can contain any character except the character with the numeric code zero.

To include a double-quote, use two double quotes. This allows you to construct table or column names that are otherwise not possible (such as ones containing spaces or ampersands). The length limitation still applies.

Quoting an identifier also makes it case sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers FOO, foo, and "Foo" are considered the same by EDB Postgres Advanced Server, but "Foo" and "FOO" are different from these three and each other. The folding of unquoted names to lower case isn’t compatible with Oracle databases. In Oracle syntax, unquoted names are folded to upper case. For example, foo is equivalent to "FOO" and not "foo". If you want to write portable applications, either always quote a particular name or never quote it.

### 14.4.1.3 Constants

The kinds of implicitly typed constants in EDB Postgres Advanced Server are strings and numbers. You can also specify constants with explicit types, which can enable more accurate representation and more efficient handling by the system.

#### String constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes (’), for example ‘This is a string’. To include a single-quote character in a string constant, include two adjacent single quotes, for example, ‘Dianne’s horse’. (The two single quotes differ from a double-quote character (“).)

#### Numeric constants
Numeric constants are accepted in these general forms:

\[ \text{digits} \quad \text{[digits][e[+-]digits]} \quad \text{digitse[+-]digits} \]

Where \( \text{digits} \) is one or more decimal digits (0 through 9). You must enter at least one digit before or after the decimal point, if one is used. At least one digit must follow the exponent marker \( (e) \), if one is present. Don't embed any spaces or other characters in the constant. Leading plus or minus signs aren't considered part of the constant. They are operators applied to the constant.

These are some examples of valid numeric constants:

- 42
- 3.5
- 4
- .001
- 5e2
- 1.925e-3

A numeric constant that doesn't contain a decimal point or an exponent is initially presumed to be type INTEGER if its value fits in type INTEGER (32 bits). Otherwise it's presumed to be type BIGINT if its value fits in type BIGINT (64 bits). If its value then doesn't fit in type BIGINT, it's taken to be type NUMBER. Constants that contain decimal points or exponents are always initially presumed to be type NUMBER.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant is automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it as described in Constants of other types.

Constants of other types

CAST

You can enter a constant of an arbitrary type using the following notation:

\[ \text{CAST}('\text{string}' \text{ AS } \text{type}) \]

The string constant's text is passed to the input conversion routine for the type called \( \text{type} \). The result is a constant of the indicated type. You can omit the explicit type cast if there's no ambiguity as to the type of constant (for example, when it's assigned directly to a table column), in which case it's automatically coerced.

You can also use \text{CAST} to specify runtime type conversions of arbitrary expressions.

CAST (MULTISET)

\text{CAST (MULTISET)} is an extension to \text{CAST} that converts subquery results into a nested table type. The synopsis is:

\[ \text{CAST} \left( \text{MULTISET} \left( \langle \text{subquery} \rangle \right) \text{ AS } \langle \text{datatype} \rangle \right) \]

Where \( \text{subquery} \) is a query returning one or more rows, and \( \text{datatype} \) is a nested table type.

\text{CAST (MULTISET)} is used to store a collection of data in a table.

Example

This example uses \text{MULTISET}:

```sql
EDB Postgres Advanced Server (EPAS) Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved. 1200
```
14.4.1 Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, for example:

```
-- This is a standard SQL comment
```

Alternatively, you can use C-style block comments:

```
/* multiline comment
   block */
```

The comment begins with /* and extends to the matching occurrence of */.

14.4.2 Data types

14.4.2.1 Numeric types

Numeric types consist of four-byte integers, four-byte and eight-byte floating-point numbers, and fixed-precision decimals. The following table lists the available types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage size</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY_INTEGER</td>
<td>4 bytes</td>
<td>Signed integer, alias for INTEGER</td>
<td>-2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>8 bytes</td>
<td>Variable-precision, inexact</td>
<td>15 decimal digits precision</td>
</tr>
<tr>
<td>INTEGER</td>
<td>4 bytes</td>
<td>Usual choice for integer</td>
<td>-2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Variable</td>
<td>User-specified precision, exact</td>
<td>Up to 1000 digits of precision</td>
</tr>
<tr>
<td>NUMBER(p [, s ] )</td>
<td>Variable</td>
<td>Exact numeric of maximum precision, p, and optional scale, s</td>
<td>Up to 1000 digits of precision</td>
</tr>
<tr>
<td>PLS_INTEGER</td>
<td>4 bytes</td>
<td>Signed integer, alias for INTEGER</td>
<td>-2,147,483,648 to +2,147,483,647</td>
</tr>
<tr>
<td>REAL</td>
<td>4 bytes</td>
<td>Variable-precision, inexact</td>
<td>6 decimal digits precision</td>
</tr>
<tr>
<td>ROWID</td>
<td>8 bytes</td>
<td>Signed 8 bit integer.</td>
<td>-9223372036854775808 to 9223372036854775807</td>
</tr>
</tbody>
</table>

Integer types
The `BINARY_INTEGER`, `INTEGER`, `PLS_INTEGER`, and `ROWID` types store whole numbers (without fractional components) as specified in the numeric types table. Attempts to store values outside of the allowed range result in an error.

**Arbitrary precision numbers**

The type `NUMBER` can store an almost unlimited number of digits of precision and perform calculations exactly. We especially recommend it for storing monetary amounts and other quantities where exactness is required. However, the `NUMBER` type is very slow compared to the floating-point types described in **Floating point types**.

The scale of a `NUMBER` is the count of decimal digits in the fractional part, to the right of the decimal point. The precision of a `NUMBER` is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers have a scale of zero.

You can configure both the precision and the scale of the `NUMBER` type. To declare a column of type `NUMBER`, use the syntax:

```
NUMBER(precision, scale)
```

The precision must be positive, and the scale must be zero or positive. Alternatively, this syntax selects a scale of 0:

```
NUMBER(precision)
```

Specifying `NUMBER` without any precision or scale creates a column in which you can store numeric values of any precision and scale, up to the implementation limit on precision. A column of this kind doesn't coerce input values to any particular scale, whereas `NUMBER` columns with a declared scale coerce input values to that scale. (The SQL standard requires a default scale of 0, that is, coercion to integer precision. For maximum portability, it's best to specify the precision and scale explicitly.)

If the precision or scale of a value is greater than the declared precision or scale of a column, the system attempts to round the value. If the value can't be rounded to satisfy the declared limits, an error occurs.

**Floating-point types**

The data types `REAL` and `DOUBLE PRECISION` are inexact, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values can't be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and isn't discussed further here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the `NUMBER` type instead.
- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), evaluate the implementation carefully.
- Comparing two floating-point values for equality might not work as expected.

On most platforms, the `REAL` type has a range of at least $1 \times 10^{-37}$ to $1 \times 10^{+37}$ with a precision of at least six decimal digits. The `DOUBLE PRECISION` type typically has a range of around $1 \times 10^{-387}$ to $1 \times 10^{+308}$ with a precision of at least 15 digits. Values that are too large or too small cause an error. Rounding might occur if the precision of an input number is too high. Numbers too close to zero that you can't represent as distinct from zero cause an underflow error.

EDB Postgres Advanced Server also supports the SQL standard notations `FLOAT` and `FLOAT(p)` for specifying inexact numeric types. Here, `p` specifies the minimum acceptable precision in binary digits. EDB Postgres Advanced Server accepts `FLOAT(1)` to `FLOAT(24)` as selecting the `REAL` type and `FLOAT(25)` to `FLOAT(53)` as selecting `DOUBLE PRECISION`. Values of `p` outside the allowed range draw an error. `FLOAT` with no precision specified is taken to mean `DOUBLE PRECISION`.

### 14.4.2.2 Character types

The following table lists the general-purpose character types available in EDB Postgres Advanced Server:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR[(n)]</td>
<td>Fixed-length character string, blank-padded to the size specified by <code>n</code></td>
</tr>
<tr>
<td>CLOB</td>
<td>Large variable-length up to 1 GB</td>
</tr>
<tr>
<td>LONG</td>
<td>Variable unlimited length.</td>
</tr>
<tr>
<td>NVARCHAR(n)</td>
<td>Variable-length national character string, with limit</td>
</tr>
<tr>
<td>NVARCHAR2(n)</td>
<td>Variable-length national character string, with limit</td>
</tr>
<tr>
<td>STRING</td>
<td>Alias for <code>VARCHAR2</code></td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>Variable-length character string, with limit (considered deprecated, but supported for compatibility)</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>VARCHAR2(n)</td>
<td>Variable-length character string, with limit</td>
</tr>
</tbody>
</table>

Where $n$ is a positive integer, these types can store strings up to $n$ characters in length. An attempt to assign a value that exceeds the length of $n$ results in an error, unless the excess characters are all spaces. In this case, the string is truncated to the maximum length.

The storage requirement for data of these types is the actual string plus 1 byte if the string is less than 127 bytes or 4 bytes if the string is 127 bytes or greater. In the case of CHAR, the padding also requires storage. Long strings are compressed by the system automatically, so the physical requirement on disk might be less. Long values are stored in background tables so they don't interfere with rapid access to the shorter column values.

The database character set determines the character set used to store textual values.

### CHAR

If you don't specify a value for $n$, $n$ defaults to 1. If the string to assign is shorter than $n$, values of type CHAR are space-padded to the specified width ($n$) and are stored and displayed that way.

Padding spaces are treated as semantically insignificant. That is, trailing spaces are disregarded when comparing two values of type CHAR, and they are removed when converting a CHAR value to one of the other string types.

If you explicitly cast an over-length value to a CHAR($n$) type, the value is truncated to $n$ characters without raising an error (as specified by the SQL standard).

<table>
<thead>
<tr>
<th>VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the string to assign is shorter than $n$, values of type VARCHAR, VARCHAR2, NVARCHAR, and NVARCHAR2 store the shorter string without padding.</td>
</tr>
</tbody>
</table>

**Note**

The trailing spaces are semantically significant in VARCHAR values.

If you explicitly cast a value to a VARCHAR type, an over-length value is truncated to $n$ characters without raising an error (as specified by the SQL standard).

### CLOB

You can store a large character string in a CLOB type. CLOB is semantically equivalent to VARCHAR2 except no length limit is specified. Generally, use a CLOB type if you don't know the maximum string length.

The longest possible character string that you can store in a CLOB type is about 1 GB.

**Note** The CLOB data type is actually a DOMAIN based on the PostgreSQL TEXT data type. For information on a DOMAIN, see the PostgreSQL core documentation.

Thus, use of the CLOB type is limited by what can be done for TEXT, such as a maximum size of approximately 1 GB.

For larger amounts of data, instead of using the CLOB data type, use the PostgreSQL large objects feature that relies on the pg_largeobject system catalog. For information on large objects, see the PostgreSQL core documentation.

#### 14.4.2.3 Binary data

The following table shows data types that allow the storage of binary strings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>The length of the binary string</td>
<td>Fixed-length binary string, with a length between 1 and 8300</td>
</tr>
<tr>
<td>BLOB</td>
<td>The actual binary string plus 1 byte if the binary string is less than 127 bytes, or 4 bytes if the binary string is 127 bytes or greater</td>
<td>Variable-length binary string, with a maximum size of 1 GB</td>
</tr>
<tr>
<td>VARCHARY</td>
<td>The length of the binary string</td>
<td>Variable-length binary string, with a length between 1 and 8300</td>
</tr>
</tbody>
</table>

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from characters strings by two characteristics:

- Binary strings specifically allow storing octets of value zero and other "non-printable" octets (defined as octets outside the range 32 to 126).
- Operations on binary strings process the actual bytes, whereas the encoding and processing of character strings depends on locale settings.

#### 14.4.2.4 Date/time types
The following discussion of the date/time types assumes that the configuration parameter `edb_redwood_date` is set to `TRUE` whenever a table is created or altered.

EDB Postgres Advanced Server supports the date/time types shown in the table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage size</th>
<th>Description</th>
<th>Low value</th>
<th>High value</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>8 bytes</td>
<td>Date and time</td>
<td>4713 BC</td>
<td>5874897 AD</td>
<td>1 second</td>
</tr>
<tr>
<td>INTERVAL DAY TO SECOND [(p)]</td>
<td>12 bytes</td>
<td>Period of time</td>
<td>-1780000000 years</td>
<td>1780000000 years</td>
<td>1 microsecond / 14 digits</td>
</tr>
<tr>
<td>INTERVAL YEAR TO MONTH</td>
<td>12 bytes</td>
<td>Period of time</td>
<td>-1780000000 years</td>
<td>1780000000 years</td>
<td>1 microsecond / 14 digits</td>
</tr>
<tr>
<td>TIMESTAMP [(p)]</td>
<td>8 bytes</td>
<td>Date and time with time zone</td>
<td>4713 BC</td>
<td>5874897 AD</td>
<td>1 microsecond</td>
</tr>
<tr>
<td>TIMESTAMP [(p)] WITH TIME ZONE</td>
<td>8 bytes</td>
<td>Date and time with time zone</td>
<td>4713 BC</td>
<td>5874897 AD</td>
<td>1 microsecond</td>
</tr>
</tbody>
</table>

When `DATE` appears as the data type of a column in the data definition language (DDL) commands `CREATE TABLE` or `ALTER TABLE`, it's translated to `TIMESTAMP` at the time the table definition is stored in the database. Thus, a time component is also stored in the column along with the date.

- `DATE` can appear as a data type of:
  - A variable in an SPL declaration section
  - The data type of a formal parameter in an SPL procedure or an SPL function
  - The return type of an SPL function

In these cases, it's always translated to `TIMESTAMP` and thus can handle a time component if present.

`TIMESTAMP` accepts an optional precision value `p` that specifies the number of fractional digits retained in the seconds field. The allowed range of `p` is from 0 to 6. The default is 6.

When `TIMESTAMP` values are stored as double-precision floating-point numbers (the default), the effective limit of precision might be less than 6. `TIMESTAMP` values are stored as seconds before or after midnight 2000-01-01. Microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. When `TIMESTAMP` values are stored as 8-byte integers (a compile-time option), microsecond precision is available over the full range of values. However, 8-byte integer timestamps have a more limited range of dates than shown in the table: from 4713 BC up to 294276 AD.

`TIMESTAMP (p) WITH TIME ZONE` is similar to `TIMESTAMP (p)` but includes the time zone as well.

**INTERVAL types**

`INTERVAL` values specify a period of time. Values of `INTERVAL` type are composed of fields that describe the value of the data. The following table lists the fields allowed in an `INTERVAL` type.

<table>
<thead>
<tr>
<th>Field name</th>
<th>INTERVAL values allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>Integer value (positive or negative)</td>
</tr>
<tr>
<td>MONTH</td>
<td>0 through 11</td>
</tr>
<tr>
<td>DAY</td>
<td>Integer value (positive or negative)</td>
</tr>
<tr>
<td>HOUR</td>
<td>0 through 23</td>
</tr>
<tr>
<td>MINUTE</td>
<td>0 through 59</td>
</tr>
<tr>
<td>SECOND</td>
<td>0 through 59.9(p) where p is the precision of fractional seconds</td>
</tr>
</tbody>
</table>

The fields must be presented in descending order, from `YEARS` to `MONTHS` and from `DAYS` to `HOURS`, `MINUTES`, and then `SECONDS`.

EDB Postgres Advanced Server supports two `INTERVAL` types compatible with Oracle databases.

The first variation supported by EDB Postgres Advanced Server is `INTERVAL DAY TO SECOND [(p)]`. `INTERVAL DAY TO SECOND [(p)]` stores a time interval in days, hours, minutes, and seconds.

`p` specifies the precision of the `SECOND` field.

EDB Postgres Advanced Server interprets this value as as 1 day, 2 hours, 3 minutes, 5 seconds and 678 thousandths of a second:

`INTERVAL '1 2:34:5.678' DAY TO SECOND(3)`

EDB Postgres Advanced Server interprets this value as 1 day and 23 hours:

`INTERVAL '1 23' DAY TO HOUR`

EDB Postgres Advanced Server interprets this value as 2 hours and 34 minutes:
INTERVAL '2:34' HOUR TO MINUTE

EDB Postgres Advanced Server interprets this value as 2 hours, 34 minutes, 56 seconds and 13 thousandths of a second. The fractional second is rounded up to 13 because of the specified precision.

INTERVAL '2:34:56.129' HOUR TO SECOND(2)

The second variation supported by EDB Postgres Advanced Server that's compatible with Oracle databases is INTERVAL YEAR TO MONTH. This variation stores a time interval in years and months.

EDB Postgres Advanced Server interprets this value as 12 years and 3 months:

INTERVAL '12-3' YEAR TO MONTH

EDB Postgres Advanced Server interprets this value as 12 years and 3 months:

INTERVAL '456' YEAR(2)

EDB Postgres Advanced Server interprets this value as 25 years:

INTERVAL '300' MONTH

Date/time input

Date and time input is accepted in ISO 8601 SQL-compatible format, the Oracle default `dd-MON-yy` format, as well as a number of other formats provided that there's no ambiguity as to which component is the year, month, and day. However, we strongly recommend using the `TO_DATE` function to avoid ambiguities.

Enclose any date or time literal input in single quotes, like text strings. The following SQL standard syntax is also accepted:

```
type 'value'
```

*`type`* is either `DATE` or `TIMESTAMP`.

*`value`* is a date/time text string.

Dates

The following block shows some possible input formats for dates, all of which equate to January 8, 1999:

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>January 8, 1999</td>
<td></td>
</tr>
<tr>
<td>1999-01-08</td>
<td></td>
</tr>
<tr>
<td>1999-Jan-08</td>
<td></td>
</tr>
<tr>
<td>Jan-08-1999</td>
<td></td>
</tr>
<tr>
<td>08-Jan-09</td>
<td></td>
</tr>
<tr>
<td>08-Jan-99</td>
<td></td>
</tr>
<tr>
<td>Jan-08-99</td>
<td></td>
</tr>
<tr>
<td>19990108</td>
<td></td>
</tr>
<tr>
<td>990108</td>
<td></td>
</tr>
</tbody>
</table>

You can assign the date values to a `DATE` or `TIMESTAMP` column or variable. The hour, minute, and seconds fields is set to zero if you don't append the date value with a time value.

Times

Some examples of the time component of a date or time stamp are shown in the table.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>04:05:06.789</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05:06</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>048506</td>
<td>ISO 8601</td>
</tr>
<tr>
<td>04:05 AM</td>
<td>Same as 04:05; AM does not affect value</td>
</tr>
<tr>
<td>04:05 PM</td>
<td>Same as 16:05; input hour must be &lt;= 12</td>
</tr>
</tbody>
</table>
Time stamps

Valid input for time stamps consists of a concatenation of a date and a time. You can format the date portion of the time according to any of the examples shown in Dates. The time portion of the time stamp can be formatted according to any of examples shown in the table in Times.

This example shows a time stamp that follows the Oracle default format:

08-JAN-99 04:05:06

This example shows a time stamp that follows the ISO 8601 standard:

1999-01-08 04:05:06

Date/time output

The default output format of the date/time types is either:

- `(dd-MON-yy)`, referred to as the Redwood date style, compatible with Oracle databases
- `(yyyy-mm-dd)` referred to as the ISO 8601 format

The format you use depends on the application interface to the database. Applications that use JDBC, such as SQL Interactive, always present the date in ISO 8601 form. Other applications such as PSQL present the date in Redwood form.

The following table shows examples of the output formats for the two styles: Redwood and ISO 8601:

<table>
<thead>
<tr>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redwood style</td>
<td>31-DEC-05 07:37:16</td>
</tr>
<tr>
<td>ISO 8601/SQL standard</td>
<td>1997-12-17 07:37:16</td>
</tr>
</tbody>
</table>

Internals

EDB Postgres Advanced Server uses Julian dates for all date/time calculations. Julian dates correctly predict or calculate any date after 4713 BC based on the assumption that the length of the year is 365.2425 days.

14.4.2.5 Boolean types

EDB Postgres Advanced Server provides the standard SQL type `BOOLEAN`. `BOOLEAN` can have one of only two states: `TRUE` or `FALSE`. A third state, `UNKNOWN`, is represented by the SQL `NULL` value.

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>1 byte</td>
<td>Logical Boolean (true/false)</td>
</tr>
</tbody>
</table>

The valid literal value for representing the true state is `TRUE`. The valid literal for representing the false state is `FALSE`.

14.4.2.6 XML type

The `XMLTYPE` data type is used to store XML data. Its advantage over storing XML data in a character field is that it checks the input values for how well they're formed. Also, support functions perform type-safe operations on it.

The XML type can store well-formed "documents," as defined by the XML standard, as well as "content" fragments, which are defined by the production `XMLDecl? content` in the XML standard. Roughly, this means that content fragments can have more than one top-level element or character node.

Note

Oracle doesn't support storing content fragments in `XMLTYPE` columns.
This example shows creating and inserting a row into a table with an XMLTYPE column:

```sql
CREATE TABLE books
(
    content XMLTYPE
);
```

```sql
INSERT INTO books VALUES (XMLPARSE ('<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>'));
```

```sql
SELECT * FROM books;
```

<table>
<thead>
<tr>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;book&gt;&lt;title&gt;Manual&lt;/title&gt;&lt;chapter&gt;...&lt;/chapter&gt;&lt;/book&gt;</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

14.4.3 Functions and operators

EDB Postgres Advanced Server provides functions and operators for the built-in data types.

14.4.3.1 Logical operators

The usual logical operators are available: `AND`, `OR`, `NOT`.

SQL uses a three-valued Boolean logic where the null value represents "unknown". Observe the following truth tables.

**AND/OR Truth Table**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a AND b</th>
<th>a OR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>Null</td>
<td>Null</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>Null</td>
<td>False</td>
<td>Null</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

**NOT Truth Table**

<table>
<thead>
<tr>
<th>a</th>
<th>NOT a</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>Null</td>
<td>Null</td>
</tr>
</tbody>
</table>

The operators `AND` and `OR` are commutative. You can switch the left and right operand without affecting the result.

14.4.3.2 Comparison operators

The usual comparison operators are shown in the table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>
Comparison operators are available for all data types where they make sense. All comparison operators are binary operators that return values of type \( \text{BOOLEAN} \). Expressions like \( 1 < 2 < 3 \) aren't valid because there is no \( < \) operator to compare a Boolean value with \( 3 \).

In addition to the comparison operators, the special \( \text{BETWEEN} \) construct is available.

\[
a \text{ BETWEEN } x \text{ AND } y
\]

is equivalent to:

\[
a \geq x \text{ AND } a \leq y
\]

Similarly:

\[
a \text{ NOT BETWEEN } x \text{ AND } y
\]

is equivalent to:

\[
a < x \text{ OR } a > y
\]

There's no difference between the two forms apart from the CPU cycles required to rewrite the first one into the second one internally.

To check whether a value is null, use the constructs:

\[
\text{expression IS NULL}
\]

\[
\text{expression IS NOT NULL}
\]

Don't write \( \text{expression = NULL} \) because \( \text{NULL} \) isn't "equal to" \( \text{NULL} \). The null value represents an unknown value, and it isn't known whether two unknown values are equal. This behavior conforms to the SQL standard.

Some applications might expect that \( \text{expression = NULL} \) returns true if \( \text{expression} \) evaluates to the null value. We highly recommend that you modify these applications to comply with the SQL standard.

### 14.4.3.3 Mathematical functions and operators

#### Overview of mathematical operators

Mathematical operators are provided for many EDB Postgres Advanced Server types. The following table shows the available mathematical operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example 1</th>
<th>Example 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>2 + 3</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>2 - 3</td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>2 * 3</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>Division (See the following note)</td>
<td>4 / 2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation Operator</td>
<td>2 ** 3</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

**Note**

If the \( \text{db_dialect} \) configuration parameter in the \( \text{postgresql.conf} \) file is set to \( \text{redwood} \), then division of a pair of \( \text{INTEGER} \) data types doesn't result in a truncated value. Any fractional result is retained as shown by the following example:

```
edb=# SET db_dialect TO redwood;
edb=# SET;
edb=# SHOW db_dialect;
```
This behavior is compatible with Oracle databases where there is no native `INTEGER` data type. Any `INTEGER` data type specification is internally converted to `NUMBER(38)`, which results in retaining any fractional result.

If the `db_dialect` configuration parameter is set to `postgres`, then division of a pair of `INTEGER` data types results in a truncated value:

```sql
edb=# SET db_dialect TO postgres;
SET db_dialect TO postgres;

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS INTEGER) FROM dual;
?column?
---------
3
(1 row)
```

This behavior is compatible with PostgreSQL databases where division involving any pair of `INTEGER`, `SMALLINT`, or `BIGINT` data types results in truncating the result. The same truncated result is returned by EDB Postgres Advanced Server when `db_dialect` is set to `postgres`.

Even when `db_dialect` is set to `redwood`, only division with a pair of `INTEGER` data types results in no truncation of the result. Division that includes only `SMALLINT` or `BIGINT` data types, with or without an `INTEGER` data type, results in truncation in the PostgreSQL fashion without retaining the fractional portion. In this example, `INTEGER` and `SMALLINT` are involved in the division:

```sql
edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS SMALLINT) FROM dual;
?column?
---------
3
(1 row)
```

### Available mathematical functions

The following table shows the available mathematical functions. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with `DOUBLE PRECISION` data are mostly implemented on top of the host system’s C library. Accuracy and behavior in boundary cases might therefore vary depending on the host system.

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ABS(x)</code></td>
<td>Same as <code>x</code></td>
<td>Absolute value</td>
<td><code>ABS(-17.4)</code></td>
<td>17.4</td>
</tr>
<tr>
<td><code>CEIL(DOUBLE PRECISION or NUMBER)</code></td>
<td>Same as input</td>
<td>Smallest integer not less than argument</td>
<td><code>CEIL(-42.8)</code></td>
<td>-42</td>
</tr>
<tr>
<td><code>EXP(DOUBLE PRECISION or NUMBER)</code></td>
<td>Same as input</td>
<td>Exponential</td>
<td><code>EXP(1.0)</code></td>
<td>2.7182818</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>284598452</td>
</tr>
</tbody>
</table>
### Available trigonometric functions

The following table shows the available trigonometric functions. All trigonometric functions take arguments and return values of type \texttt{DOUBLE PRECISION}.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACOS(x)</td>
<td>Inverse cosine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASIN(x)</td>
<td>Inverse sine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATAN(x)</td>
<td>Inverse tangent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATAN2(x, y)</td>
<td>Inverse tangent of ( x/y )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COS(x)</td>
<td>Cosine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIN(x)</td>
<td>Sine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAN(x)</td>
<td>Tangent</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 14.4.3.4 String functions and operators

These functions and operators are for examining and manipulating string values. Strings in this context include values of the types \texttt{CHAR}, \texttt{VARCHAR2}, and \texttt{CLOB}. Unless otherwise noted, all of these functions work on all of these types, but be aware of potential effects of automatic padding when using the \texttt{CHAR} type. Generally, the functions described here also work on data of nonstring types by converting that data to a string representation first.

#### Overview of string functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>WIDTH_BUCKET(5.35, 0.024, 10.06, 5)</td>
<td>3</td>
</tr>
</tbody>
</table>
More string manipulation functions are available and are listed in the following table. Some of them are used internally to implement the SQL-standard string functions listed in the above table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Return type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td></td>
<td>string</td>
<td>CLOB</td>
<td>String concatenation.</td>
</tr>
<tr>
<td>CONCAT(string, string)</td>
<td>CLOB</td>
<td>String concatenation.</td>
<td>'a'</td>
<td></td>
</tr>
<tr>
<td>HEXTORAM(varchar2)</td>
<td>RAW</td>
<td>Converts a VARCHAR2 value to a RAW value.</td>
<td>HEXTORAM('3B3132')</td>
<td>3B3132</td>
</tr>
<tr>
<td>RAWTOHEX(raw)</td>
<td>VARCHAR2</td>
<td>Converts a RAW value to a HEXADECIMAL value.</td>
<td>RAWTOHEX('012')</td>
<td>3012</td>
</tr>
<tr>
<td>INSTR(string, set, [ start [, occurrence ] ])</td>
<td>INTEGER</td>
<td>Finds the location of a set of characters in a string, starting at position start in the string,</td>
<td>INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PI',1,3)</td>
<td>38</td>
</tr>
<tr>
<td>INSTRB(string, set)</td>
<td>INTEGER</td>
<td>Returns the position of the set within the string. Returns 0 if set is not found.</td>
<td>INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PI')</td>
<td>13</td>
</tr>
<tr>
<td>INSTRB(string, set, start)</td>
<td>INTEGER</td>
<td>Returns the position of the set within the string, beginning at start. Returns 0 if set is not found.</td>
<td>INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PI', 1, 2)</td>
<td>38</td>
</tr>
<tr>
<td>INSTRB(string, set, start, occurrence)</td>
<td>INTEGER</td>
<td>Returns the position of the specified occurrence of set within the string, beginning at start. Returns 0 if set is not found.</td>
<td>INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PI', 14)</td>
<td>30</td>
</tr>
<tr>
<td>LOWER(string)</td>
<td>CLOB</td>
<td>Converts string to lower case.</td>
<td>LOWER('TOM')</td>
<td>TOM</td>
</tr>
<tr>
<td>NLS_CHARSET_ID(string)</td>
<td>NUMBER</td>
<td>Returns the character set ID number corresponding to the character set name string. The string argument is a runtime VARCHAR2 value.</td>
<td>NLS_CHARSET_ID('utf8')</td>
<td>UTF8</td>
</tr>
<tr>
<td>NLS_CHARSET_NAME(number)</td>
<td>VARCHAR2</td>
<td>Returns the name of the character set corresponding to the ID number. The character set name is returned as a VARCHAR2 value in the database character set.</td>
<td>NLS_CHARSET_NAME(6)</td>
<td>UTF8</td>
</tr>
<tr>
<td>NLS_CHARSETDECL_LEN(byte_count, char_set_id)</td>
<td>NUMBER</td>
<td>Returns the declaration length in number of characters of an NCHAR column. The byte_count argument is the width of the column. The char_set_id argument is the character set ID of the column.</td>
<td>NLS_CHARSETDECL_LEN(100,nls_charset_id('utf8'))</td>
<td>UTF8</td>
</tr>
<tr>
<td>SUBSTR(string, start [, count])</td>
<td>CLOB</td>
<td>Extracts substring starting from start and going for count characters. If count isn’t specified, the string is clipped from the start till the end.</td>
<td>SUBSTR('This is a test',6,2)</td>
<td>15</td>
</tr>
<tr>
<td>SUBSTR(string, start [, count])</td>
<td>CLOB</td>
<td>Same as SUBSTR except start and count are in number of bytes.</td>
<td>SUBSTR('abc',3), assuming a double-byte character set.</td>
<td>c</td>
</tr>
<tr>
<td>SUBSTR2(string, start [, count])</td>
<td>CLOB</td>
<td>Alias for SUBSTR.</td>
<td>SUBSTR2('This is a test',6,2)</td>
<td>15</td>
</tr>
<tr>
<td>SUBSTR2(string, start [, count])</td>
<td>CLOB</td>
<td>Alias for SUBSTR.</td>
<td>SUBSTR2('abc',3), (assuming a double-byte character set)</td>
<td>c</td>
</tr>
<tr>
<td>SUBSTR4(string, start [, count])</td>
<td>CLOB</td>
<td>Alias for SUBSTR.</td>
<td>SUBSTR4('This is a test',6,2)</td>
<td>15</td>
</tr>
<tr>
<td>SUBSTR4(string, start [, count])</td>
<td>CLOB</td>
<td>Alias for SUBSTR.</td>
<td>SUBSTR4('abc',3), (assuming a double-byte character set)</td>
<td>c</td>
</tr>
<tr>
<td>TRIM([ LEADING</td>
<td>TRAILING ] [ BOTH ] \ [ characters ] FROM string)</td>
<td>CLOB</td>
<td>Removes the longest string containing only the characters (a space by default) from the start, end, or both ends of the string.</td>
<td>TRIM(both 'x' FROM 'xTomxx')</td>
</tr>
<tr>
<td>LTRIM(string, [ set])</td>
<td>CLOB</td>
<td>Removes all the characters specified in set from the left of a given string, if set isn’t specified, a blank space is used by default.</td>
<td>LTRIM('abcdefghi', 'abc')</td>
<td>defghi</td>
</tr>
<tr>
<td>RTRIM(string, [ set])</td>
<td>CLOB</td>
<td>Removes all the characters specified in set from the right of a given string, if set isn’t specified, a blank space is used by default.</td>
<td>RTRIM('abcdefghi', 'ghi')</td>
<td>abcdef</td>
</tr>
<tr>
<td>UPPER(string)</td>
<td>CLOB</td>
<td>Converts string to upper case.</td>
<td>UPPER('tom')</td>
<td>TOM</td>
</tr>
</tbody>
</table>

ASCII(string) INTEGER ASCII code of the first byte of the argument. | ASCII('x') | 120 |
<table>
<thead>
<tr>
<th>Function</th>
<th>Return type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR(INTEGER)</td>
<td>CLOB</td>
<td>Character with the given ASCII code.</td>
<td>CHR(65)</td>
<td>A</td>
</tr>
<tr>
<td>DECODE(expr1, expr2a, expr2b, [.., default])</td>
<td></td>
<td>Same as argument types of expr1. Finds first match of expr1 with expr2a, expr2b, etc. When match found, returns corresponding parameter pair, expr1b, expr2b, etc. If no match found, returns default. If no match found and default not specified, returns null.</td>
<td>DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')</td>
<td>Th re e</td>
</tr>
<tr>
<td>INITCAP(string)</td>
<td>CLOB</td>
<td>Converts the first letter of each word to upper case and the rest to lower case. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.</td>
<td>INITCAP('hi thom as')</td>
<td>Th om as</td>
</tr>
<tr>
<td>LENGTH</td>
<td>INTEGER</td>
<td>Returns the number of characters in a string value.</td>
<td>LENGTH('Côte d''Azur')</td>
<td>1 1</td>
</tr>
<tr>
<td>LENGTHC</td>
<td>INTEGER</td>
<td>This function is identical in functionality to LENGTH; the function name is supported for compatibility.</td>
<td>LENGTHC('Côte d''Azur')</td>
<td>1 1</td>
</tr>
<tr>
<td>LENGTH2</td>
<td>INTEGER</td>
<td>This function is identical in functionality to LENGTH; the function name is supported for compatibility.</td>
<td>LENGTH2('Côte d''Azur')</td>
<td>1 1</td>
</tr>
<tr>
<td>LENGTH4</td>
<td>INTEGER</td>
<td>This function is identical in functionality to LENGTH; the function name is supported for compatibility.</td>
<td>LENGTH4('Côte d''Azur')</td>
<td>1 1</td>
</tr>
<tr>
<td>LENGTHB</td>
<td>INTEGER</td>
<td>Returns the number of bytes required to hold the given value.</td>
<td>LENGTHB('Côte d''Azur')</td>
<td>1 2</td>
</tr>
<tr>
<td>RPAD(string, length INTEGER [ , fill ]);</td>
<td>CLOB</td>
<td>Fills up string to size length by prepending the characters fill (a space by default). If string is already longer than length, then it is truncated (on the right).</td>
<td>RPAD('hi', 5, 'xy')</td>
<td>xy xh 1</td>
</tr>
<tr>
<td>REPLACE(string, search_string [ , replace_string ]);</td>
<td>CLOB</td>
<td>Replaces one value in a string with another. If you don’t specify a value for replace_string, the search_string value, when found, is removed.</td>
<td>REPLACE('GEORGE', 'GE', 'EG')</td>
<td>EG OR EG</td>
</tr>
<tr>
<td>TRANSATE(string, from, to)</td>
<td>CLOB</td>
<td>Any character in string that matches a character in the from set is replaced by the corresponding character in the to set.</td>
<td>TRANSLATE('12345', '14', 'ax')</td>
<td>a2 3x 5</td>
</tr>
</tbody>
</table>

Truncation of string text resulting from concatenation with NULL

**Note**

This functionality isn’t compatible with Oracle databases, which can lead to some inconsistency when converting data from Oracle to EDB Postgres Advanced Server.

For EDB Postgres Advanced Server, when a column value is NULL, the concatenation of the column with a text string can result in either of the following:

- Return of the text string
- Disappearance of the text string (that is, a null result)

The result depends on the data type of the NULL column and the way in which the concatenation is done.

If you use the string concatenation operator `||`, then the types that have implicit coercion to text, as listed in the table Data Types with Implicit Coercion to Text, don’t truncate the string if one of the input parameters is NULL. For other types, it truncates the string unless the explicit type cast is used (that is, `::text`). Also, to see the consistent behavior in the presence of nulls, you can use the CONCAT function.

The following query lists the data types that have implicit coercion to text:

```sql
SELECT castsource::regtype, casttarget::regtype, castfunc::regproc,
       CASE castcontext
           WHEN 'e' THEN 'explicit'
           WHEN 'a' THEN 'implicit in assignment'
           WHEN 'i' THEN 'implicit in expressions'
       END as castcontext,
       CASE castmethod
           WHEN 'f' THEN 'function'
           WHEN 'i' THEN 'input/output function'
           WHEN 'b' THEN 'binary-coercible'
       END as castmethod
FROM pg_cast
WHERE casttarget::regtype::text = 'text'
```
The result of the query is listed in the following table.

<table>
<thead>
<tr>
<th>castsource</th>
<th>casttarget</th>
<th>castfunc</th>
<th>castcontext</th>
<th>castmethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>text</td>
<td>pg_catalog.text</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>character varying</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>binary-coercible</td>
</tr>
<tr>
<td>&quot;char&quot;</td>
<td>text</td>
<td>pg_catalog.text</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>name</td>
<td>text</td>
<td>pg_catalog.text</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>pg_node_tree</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>binary-coercible</td>
</tr>
<tr>
<td>pg_ndistinct</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>pg_dependencies</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>integer</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>smallint</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>oid</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>date</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>double precision</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>real</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>time with time zone</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>time without time zone</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>timestamp with time zone</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>interval</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>bigint</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>numeric</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>timestamp without time zone</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>record</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>boolean</td>
<td>text</td>
<td>pg_catalog.text</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
<tr>
<td>bytea</td>
<td>text</td>
<td>-</td>
<td>implicit in expressions</td>
<td>function</td>
</tr>
</tbody>
</table>

For information on the column output, see the `pg_cast` system catalog in the PostgreSQL core documentation.

So for example, data type `UUID` isn’t in this list and therefore doesn’t have the implicit coercion to text. As a result, certain concatenation attempts with a `NULL` `UUID` column results in a truncated text result.

The following table is created for this example with a single row with all `NULL` column values:

```sql
CREATE TABLE null_concat_types (
    boolean_type BOOLEAN,
    uuid_type       UUID,
    char_type       CHARACTER
);
```
Columns boolean_type and char_type have the implicit coercion to text while column uuid_type doesn't. Thus, string concatenation with the concatenation operator ' || ' against columns boolean_type or char_type results in the following:

```
SELECT 'x=' || boolean_type || 'y' FROM null_concat_types;

?column?
-------
x=y
(1 row)
```

```
SELECT 'x=' || char_type || 'y' FROM null_concat_types;

?column?
-------
x=y
(1 row)
```

But concatenation with column uuid_type results in the loss of the x= string:

```
SELECT 'x=' || uuid_type || 'y' FROM null_concat_types;

?column?
-------
y
(1 row)
```

However, using explicit casting with ::text prevents the loss of the x= string:

```
SELECT 'x=' || uuid_type::text || 'y' FROM null_concat_types;

?column?
-------
x=y
(1 row)
```

Using the CONCAT function also preserves the x= string:

```
SELECT CONCAT('x=',uuid_type) || 'y' FROM null_concat_types;

?column?
-------
x=y
(1 row)
```

Thus, depending on the data type of a NULL column, use explicit casting or the CONCAT function to avoid losing some text strings.

### 14.4.3.5 Pattern matching string functions

EDB Postgres Advanced Server offers support for the REGEXP_COUNT, REGEXP_INSTR, and REGEXP_SUBSTR functions. These functions search a string for a pattern specified by a regular expression and return information about occurrences of the pattern in the string. Use a POSIX-style regular expression. For more information about forming a POSIX-style regular expression, see the PostgreSQL core documentation.

**REGEXP_COUNT**

REGEXP_COUNT searches a string for a regular expression and returns a count of the times that the regular expression occurs. The signature is:

```
INTEGER REGEXP_COUNT
 ( srcstr
  TEXT,
  pattern TEXT,
  position DEFAULT
)
```
**Parameters**

**srcstr**

- **srcstr** specifies the string to search.

**pattern**

- **pattern** specifies the regular expression **REGEXP_COUNT** searches for.

**position**

- **position** is an integer value that indicates the position in the source string where **REGEXP_COUNT** begins searching. The default value is 1.

**modifier**

- **modifier** specifies values that control the pattern-matching behavior. The default value is NULL. For a complete list of the modifiers supported by EDB Postgres Advanced Server, see the PostgreSQL core documentation.

**Example**

In this example, **REGEXP_COUNT** returns a count of the number of times the letter 'i' is used in the character string 'reinitializing':

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 1) FROM DUAL;

  regexp_count
  --------------
          5
   (1 row)
```

The command instructs **REGEXP_COUNT** to begin counting in the first position. If you modify the command to start the count on the sixth position, **REGEXP_COUNT** returns 3:

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 6) FROM DUAL;

  regexp_count
  --------------
          3
   (1 row)
```

The count now excludes any occurrences of the letter 'i' that occur before the sixth position.

**REGEXP_INSTR**

**REGEXP_INSTR** searches a string for a POSIX-style regular expression. This function returns the position in the string where the match was located. The signature is:

```
INTEGER REGEXP_INSTR
   ( srcstr TEXT, pattern TEXT, position INT DEFAULT 1, occurrence INT DEFAULT 1, returnparam INT DEFAULT 0, modifier TEXT DEFAULT NULL, subexpression INT DEFAULT 0,
   )
```

Parameters


**srcstr**

specifies the string to search.

**pattern**

specifies the regular expression that `REGEXP_INSTR` searches for.

**position**

specifies an integer value that indicates the start position in a source string. The default value is 1.

**occurrence**

specifies the match to return if more than one occurrence of the pattern occurs in the string to search. The default value is 1.

**returnparam**

is an integer value that specifies the location in the string for `REGEXP_INSTR` to return. The default value is 0. Specify:

- 0 to return the location in the string of the first character that matches `pattern`.
- A value greater than 0 to return the position of the first character following the end of the `pattern`.

**modifier**

specifies values that control the pattern-matching behavior. The default value is NULL. For a complete list of the modifiers supported by EDB Postgres Advanced Server, see the PostgreSQL core documentation.

**subexpression**

is an integer value that identifies the portion of the `pattern` that's returned by `REGEXP_INSTR`. The default value of `subexpression` is 0.

If you specify a value for `subexpression`, you must include one or more sets of parentheses in the `pattern` that isolate a portion of the value you are searching for. The value specified by `subexpression` indicates the set of parentheses to return. For example, if `subexpression` is 2, `REGEXP_INSTR` returns the position of the second set of parentheses.

**Example**

In this example, `REGEXP_INSTR` searches a string that contains a phone number for the first occurrence of a pattern that contains three consecutive digits:

```sql
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;

regexp_instr
-----------
 1
(1 row)
```

The command instructs `REGEXP_INSTR` to return the position of the first occurrence. If we modify the command to return the start of the second occurrence of three consecutive digits, `REGEXP_INSTR` returns 5. The second occurrence of three consecutive digits begins in the fifth position.

```sql
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;

regexp_instr
-----------
 5
(1 row)
```

**REGEXP_SUBSTR**

The `REGEXP_SUBSTR` function searches a string for a pattern specified by a POSIX-compliant regular expression. `REGEXP_SUBSTR` returns the string that matches the pattern specified in the call to the function. The signature of the function is:

```sql
TEXT REGEXP_SUBSTR
( srcstr TEXT, pattern TEXT,....
```
Parameters

**srcstr**

*srcstr* specifies the string to search.

**pattern**

*pattern* specifies the regular expression *REGEXP_SUBSTR* searches for.

**position**

*position* specifies an integer value that indicates the start position in a source string. The default value is 1.

**occurrence**

*occurrence* specifies the match returned if more than one occurrence of the search pattern occurs. The default value is 1.

**modifier**

*modifier* specifies values that control the pattern-matching behavior. The default value is *NULL*. For a complete list of the modifiers supported by EDB Postgres Advanced Server, see the PostgreSQL core documentation.

**subexpression**

*subexpression* is an integer value that identifies the portion of the *pattern* that's returned by *REGEXP_SUBSTR*. The default value of *subexpression* is 0.

If you specify a value for *subexpression*, you must include one or more sets of parentheses in the *pattern* that isolate a portion of the value being searched for. The value specified by *subexpression* indicates the set of parentheses to return. For example, if *subexpression* is 2, *REGEXP_SUBSTR* returns the value contained in the second set of parentheses.

Example

In this example, *REGEXP_SUBSTR* searches a string that contains a phone number for the first set of three consecutive digits:

```
EDB> SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;

regexp_substr
-------------
800
(1 row)
```

It locates the first occurrence of three digits and returns the string **800**. If we modify the command to check for the second occurrence of three consecutive digits, *REGEXP_SUBSTR* returns **555**, the contents of the second substring.

```
EDB> SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;

regexp_substr
-------------
555
(1 row)
```

14.4.3.6 Pattern matching using the LIKE operator

EDB Postgres Advanced Server provides pattern matching using the traditional SQL *LIKE* operator. The syntax for the *LIKE* operator is:

```
string LIKE pattern [ ESCAPE escape-character ]
```
string NOT LIKE pattern [ ESCAPE escape-character ]

Every pattern defines a set of strings. The LIKE expression returns TRUE if string is contained in the set of strings represented by pattern. As expected, the NOT LIKE expression returns FALSE if LIKE returns TRUE and vice versa. An equivalent expression is NOT (string LIKE pattern).

If pattern doesn’t contain percent signs or underscore, then the pattern represents only the string. In that case, LIKE acts like the equals operator. An underscore (_;) in pattern matches any single character. A percent sign (%) matches any string of zero or more characters.

Some examples:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'abc' LIKE 'abc'</td>
<td>true</td>
</tr>
<tr>
<td>'abc' LIKE 'ab'</td>
<td>true</td>
</tr>
<tr>
<td>'abc' LIKE '_<em>b</em>'</td>
<td>true</td>
</tr>
<tr>
<td>'abc' LIKE 'c'</td>
<td>false</td>
</tr>
</tbody>
</table>

LIKE pattern matches always cover the entire string. To match a pattern anywhere in a string, the pattern must start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, place an escape character before the character in pattern. The default escape character is the backslash, but you can select a different one by using the ESCAPE clause. To match the escape character, enter two escape characters.

The backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in an SQL statement. Thus, writing a pattern that matches a literal backslash means writing four backslashes in the statement. You can avoid this by selecting a different escape character with ESCAPE. Then a backslash isn’t special to LIKE anymore. However, it’s still special to the string literal parser, so you still need two of them.

It’s also possible to select no escape character by writing ESCAPE ‘’. This disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

### 14.4.3.7 Data type formatting functions

The EDB Postgres Advanced Server formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. These all functions follow a common calling convention: the first argument is the value to format and the second argument is a string template with percent signs (%) and underscore (_). The template is used to format the output value.

#### Overview of data type formatting functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_BLOB( raw )</td>
<td>BLOB</td>
<td>Convert a RAW value to BLOB value.</td>
<td>TO_BLOB('abc')</td>
<td>\x616263</td>
</tr>
<tr>
<td>TO_CLOB( string )</td>
<td>CLOB</td>
<td>Convert a CHAR, VARCHAR, VARCHAR2, NCHAR, NVARCHAR2, or CLOB values to CLOB values.</td>
<td>TO_CLOB('aaaa')</td>
<td>aaaa</td>
</tr>
<tr>
<td>TO_CHAR( DATE [ , format ] )</td>
<td>VARCHAR2</td>
<td>Convert a date/time to a string with output, format. The default format is DD-MON-YY.</td>
<td>TO_CHAR(SYSDATE, 'MM/DD/YYYY HH12:MI:SS AM')</td>
<td>07/25/2007 09:43:02 AM</td>
</tr>
<tr>
<td>TO_CHAR( TIMESTAMP [ , format ] )</td>
<td>VARCHAR2</td>
<td>Convert a timestamp to a string with output, format. The default format is DD-MON-YY.</td>
<td>TO_CHAR (CURRENT_TIMESTAMP, 'MM/DD/YYYY HH12:MI:SS AM')</td>
<td>08/30/2015 08:55:22 PM</td>
</tr>
<tr>
<td>TO_CHAR( INTEGER [ , format ] )</td>
<td>VARCHAR2</td>
<td>Convert an integer to a string with output, format.</td>
<td>TO_CHAR(2412, '999,999S')</td>
<td>2,412</td>
</tr>
<tr>
<td>TO_CHAR( NUMBER [ , format ] )</td>
<td>VARCHAR2</td>
<td>Convert a decimal number to a string with output, format.</td>
<td>TO_CHAR(18123.05, '999,999.99')</td>
<td>18,123.05</td>
</tr>
<tr>
<td>TO_CHAR( DOUBLE PRECISION, format )</td>
<td>VARCHAR2</td>
<td>Convert a floating-point number to a string with output, format.</td>
<td>TO_CHAR(CAST(123.5282 AS REAL), '999.99S')</td>
<td>123.53</td>
</tr>
<tr>
<td>TO_DATE( string [ , format ] )</td>
<td>TIMESTAMP</td>
<td>Convert a date or timestamp formatted string to a TIMESTAMP data type.</td>
<td>TO_DATE('2007-07-04 13:39:10', 'YYYY-MM-DD HH24:MI:SS')</td>
<td>04-JUL-07 13:39:10</td>
</tr>
<tr>
<td>TO_DSINTERVAL( string )</td>
<td>INTERVAL DAY TO SECOND</td>
<td>Convert a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 datatype to an INTERVAL DAY TO SECOND type.</td>
<td>TO_DSINTERVAL('180 13:30:00')</td>
<td>80 days 13:30:00</td>
</tr>
<tr>
<td>TO_NCHAR( string )</td>
<td>NCHAR2</td>
<td>Convert a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 value to a national character set.</td>
<td>TO_NCHAR('test')</td>
<td>test</td>
</tr>
<tr>
<td>TO_NCHAR( NUMBER [ , format ] )</td>
<td>NVARCHAR2</td>
<td>Convert a number formatted string to a national character data type.</td>
<td>TO_NCHAR(7654321, 'C9G999G999D99')</td>
<td>7,654,321.08</td>
</tr>
<tr>
<td>TO_NCHAR( DATE [ , format ] )</td>
<td>NVARCHAR2</td>
<td>Convert a date/time to a formatted string of national character data type.</td>
<td>TO_NCHAR( TIMESTAMP '2022-04-20 17:31:12.66', 'Day: MONTH DD, YYYY')</td>
<td>Wednesday, April 28, 2022</td>
</tr>
</tbody>
</table>
### Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Return type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TO_NUMBER(string [, format])</td>
<td>NUMBER</td>
<td>Convert a number formatted string to a NUMBER data type.</td>
<td>TO_NUMBER('2,412-', '999,999S')</td>
<td>-2412</td>
</tr>
<tr>
<td>TO_TIMESTAMP(string, format)</td>
<td>TIMESTAMP</td>
<td>Convert a timestamp formatted string to a TIMESTAMP WITH TIME ZONE data type.</td>
<td>TO_TIMESTAMP('05 Dec 2000 08:30:25 pm', 'DD Mon YYYY hh12:mi:ss pm')</td>
<td>05-DEC-00 20:30:25 +05:30</td>
</tr>
<tr>
<td>TO_TIMESTAMP_TZ(string[, format])</td>
<td>TIMESTAMP</td>
<td>Convert a timestamp formatted string to a TIMESTAMP WITH TIME ZONE data type.</td>
<td>TO_TIMESTAMP_TZ ('2003/12/13 10:13:18 -8:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM')</td>
<td>13-DEC-03 23:43:18 +05:30</td>
</tr>
<tr>
<td>FROM_TZ(timestamp_value, timezone_value)</td>
<td>TIMESTAMP</td>
<td>Convert a timestamp value and a timezone to a TIMESTAMP WITH TIME ZONE value.</td>
<td>FROM_TZ(TIMESTAMP '2017-08-08 08:09:10', 'Asia/Kolkata')</td>
<td>08-AUG-17 08:09:10 +05:30</td>
</tr>
</tbody>
</table>

### Available template patterns for date values

This table shows the template patterns available for formatting date values using the TO_CHAR, TO_DATE, TO_TIMESTAMP, and TO_TIMESTAMP_TZ functions.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HH</td>
<td>Hour of day (01-12)</td>
</tr>
<tr>
<td>HH12</td>
<td>Hour of day (01-12)</td>
</tr>
<tr>
<td>HH24</td>
<td>Hour of day (00-23)</td>
</tr>
<tr>
<td>MI</td>
<td>Minute (00-59)</td>
</tr>
<tr>
<td>SS</td>
<td>Second (00-59)</td>
</tr>
<tr>
<td>SSSSS</td>
<td>Seconds past midnight (0-86399)</td>
</tr>
<tr>
<td>FFnn</td>
<td>Fractional seconds where n is an optional integer from 1 to 9 for the number of digits to return. The default is 6.</td>
</tr>
<tr>
<td>AM or A.M. or PM or P.M.</td>
<td>Meridian indicator (uppercase)</td>
</tr>
<tr>
<td>am or a.m. or pm or p.m.</td>
<td>Meridian indicator (lowercase)</td>
</tr>
<tr>
<td>Y, YYYY</td>
<td>Year (4 and more digits) with comma</td>
</tr>
<tr>
<td>YEAR</td>
<td>Year (spelled out)</td>
</tr>
<tr>
<td>SYEAR</td>
<td>Year (spelled out) (BC dates prefixed by a minus sign)</td>
</tr>
<tr>
<td>YYYY</td>
<td>Year (4 and more digits)</td>
</tr>
<tr>
<td>SYYYYY</td>
<td>Year (4 and more digits) (BC dates prefixed by a minus sign)</td>
</tr>
<tr>
<td>YYY</td>
<td>Last 3 digits of year</td>
</tr>
<tr>
<td>YY</td>
<td>Last 2 digits of year</td>
</tr>
<tr>
<td>Y</td>
<td>Last digit of year</td>
</tr>
<tr>
<td>IYYYY</td>
<td>ISO year (4 and more digits)</td>
</tr>
<tr>
<td>IYYYY</td>
<td>Last 3 digits of ISO year</td>
</tr>
<tr>
<td>IY</td>
<td>Last 2 digits of ISO year</td>
</tr>
<tr>
<td>I</td>
<td>Last 1 digit of ISO year</td>
</tr>
<tr>
<td>BC or B.C. or AD or A.D.</td>
<td>Era indicator (uppercase)</td>
</tr>
<tr>
<td>bc or b.c. or ad or a.d.</td>
<td>Era indicator (lowercase)</td>
</tr>
<tr>
<td>MONTH</td>
<td>Full uppercase month name</td>
</tr>
<tr>
<td>Month</td>
<td>Full mixed-case month name</td>
</tr>
<tr>
<td>month</td>
<td>Full lowercase month name</td>
</tr>
<tr>
<td>NON</td>
<td>Abbreviated uppercase month name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>Pattern</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>Mon</td>
<td>Abbreviated mixed-case month name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>non</td>
<td>Abbreviated lowercase month name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>MM</td>
<td>Month number (01-12)</td>
</tr>
<tr>
<td>Day</td>
<td>Full uppercase day name</td>
</tr>
<tr>
<td>Day</td>
<td>Full mixed-case day name</td>
</tr>
<tr>
<td>day</td>
<td>Full lowercase day name</td>
</tr>
<tr>
<td>DY</td>
<td>Abbreviated uppercase day name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>Dy</td>
<td>Abbreviated mixed-case day name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>dy</td>
<td>Abbreviated lowercase day name (3 chars in English, localized lengths vary)</td>
</tr>
<tr>
<td>DDD</td>
<td>Day of year (001-366)</td>
</tr>
<tr>
<td>DD</td>
<td>Day of month (01-31)</td>
</tr>
<tr>
<td>D</td>
<td>Day of week (1-7; Sunday is 1)</td>
</tr>
<tr>
<td>W</td>
<td>Week of month (1-5); the first week starts on the first day of the month</td>
</tr>
<tr>
<td>WM</td>
<td>Week number of year (1-53); the first week starts on the first day of the year</td>
</tr>
<tr>
<td>IM</td>
<td>ISO week number of year; the first Thursday of the new year is in week 1</td>
</tr>
<tr>
<td>CC</td>
<td>Century (2 digits); the 21st century starts on 2001-01-01</td>
</tr>
<tr>
<td>SCC</td>
<td>Same as CC except BC dates are prefixed by a minus sign</td>
</tr>
<tr>
<td>J</td>
<td>Julian Day (days since January 1, 4712 BC)</td>
</tr>
<tr>
<td>Q</td>
<td>Quarter</td>
</tr>
<tr>
<td>RM</td>
<td>Month in Roman numerals (I-XII); I=January (uppercase)</td>
</tr>
<tr>
<td>rm</td>
<td>Month in Roman numerals (i-xii; i=January) (lowercase)</td>
</tr>
<tr>
<td>RR</td>
<td>First 2 digits of the year when given only the last 2 digits of the year. Result is based upon an algorithm using the current year and the given 2-digit year. The first 2 digits of the given 2-digit year are the same as the first 2 digits of the current year with the following exceptions: if the given 2-digit year is &lt;50 and the last 2 digits of the current year is &gt;= 50, then the first 2 digits for the given year is 1 greater than the first 2 digits of the current year. If the given 2-digit year is &gt;= 50 and the last 2 digits of the current year is &lt;50, then the first 2 digits for the given year is 1 less than the first 2 digits of the current year.</td>
</tr>
<tr>
<td>RRRR</td>
<td>Affects only <code>TO_DATE</code> function. Allows specification of 2-digit or 4-digit year. If 2-digit year given, then returns first 2 digits of year like RR format. If 4-digit year given, returns the given 4-digit year.</td>
</tr>
<tr>
<td>TZH</td>
<td>Time-zone hours</td>
</tr>
<tr>
<td>TZM</td>
<td>Time-zone minutes</td>
</tr>
</tbody>
</table>

**Date and time modifiers**

You can apply certain modifiers to any template pattern to alter its behavior. For example, `FMMonth` is the `Month` pattern with the `FM` modifier. This table shows the modifier patterns for date/time formatting.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>FM</td>
<td>Fill mode (suppress padding blanks and zeros)</td>
<td><code>FMMonth</code></td>
</tr>
<tr>
<td>TH</td>
<td>Uppercase ordinal number suffix</td>
<td><code>DDTH</code></td>
</tr>
<tr>
<td>th</td>
<td>Lowercase ordinal number suffix</td>
<td><code>DDth</code></td>
</tr>
<tr>
<td>FX</td>
<td>Fixed format global option (see usage notes)</td>
<td><code>FX Month DD Day</code></td>
</tr>
<tr>
<td>SP</td>
<td>Spell mode</td>
<td><code>DDSP</code></td>
</tr>
</tbody>
</table>

**Usage notes for date/time formatting:**

- **FM** suppresses leading zeroes and trailing blanks that are otherwise added to make the output of a pattern fixed width.

- **`TO_TIMESTAMP`, `TO_TIMESTAMP_TZ`, and `TO_DATE`** skip multiple blank spaces in the input string if you don’t use the **FX** option. You must use **FX** as the first item in the template. For example:

  ```sql
  TO_TIMESTAMP('2060 - JUN', 'YYYY-MON') is correct, but
  TO_TIMESTAMP('2060  JUN', 'FXYYYY MON') and
  TO_TIMESTAMP_TZ('2080  JUN', 'FXYYYY MON') returns an error
  because TO_TIMESTAMP and TO_TIMESTAMP_TZ expects one space
  only.
  ```

- **Ordinary text is allowed in `TO_CHAR` templates and is output literally.**

- **In conversions from string to `timestamp`, `timestamp tz`, or `date`, the CC field is ignored if there is a `Y`, `YYY`, or `Y,YYY` field. If CC is used with `Y` or `Y`, then the year is
This table shows some examples of the use of the \texttt{TO\_CHAR} and \texttt{TO\_DATE} functions:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS')}</td>
<td>'Tuesday, 06 05:39:18'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(CURRENT_TIMESTAMP, 'FM-Day, FMDD HH12:MI:SS')}</td>
<td>'Tuesday, 6 05:39:18'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(-0.1, '99.99')}</td>
<td>'-.10'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(0.1, '.9')}</td>
<td>'0.1'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(12, '9990999.9')}</td>
<td>'012.0'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(12, 'FM9990999.9')}</td>
<td>'0012.'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(485, '999')}</td>
<td>'485'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(485, '999MI')}</td>
<td>'485'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(485, 'FM999MI')}</td>
<td>'485'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(485, '999PR')}</td>
<td>'&lt;485&gt;'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(485, 'L999')}</td>
<td>'$485$'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(485, 'RN')}</td>
<td>'CDLXXXV'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(485, 'FMNN')}</td>
<td>'CDLXXXV'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(5.2, 'FMNN')}</td>
<td>'V'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(12, '99999')}</td>
<td>'12800'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(12.4, '99999')}</td>
<td>'12400'</td>
</tr>
<tr>
<td>\texttt{TO_CHAR(12.45, '9999')}</td>
<td>'125'</td>
</tr>
</tbody>
</table>

This table shows some examples of the use of the \texttt{TO\_TIMESTAMP\_TZ} function:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{TO_TIMESTAMP_TZ('12-JAN-2010', 'DD-MONTH-YYYY')}</td>
<td>'12-JAN-10 00:00:00 +05:30'</td>
</tr>
<tr>
<td>\texttt{TO_TIMESTAMP_TZ('03-APR-07 09:12:21 P.M', 'DD-MON-YY HH12:MI:SS A.M')}</td>
<td>'03-APR-07 09:12:21 +05:30'</td>
</tr>
<tr>
<td>\texttt{TO_TIMESTAMP_TZ('2003/12/13 10:13:18 -8:00', 'YYYY/MM/DD HH:MI:SS TZH:TZM')}</td>
<td>'13-DEC-03 23:43:18 +05:30'</td>
</tr>
<tr>
<td>\texttt{TO_TIMESTAMP_TZ('20-MAR-20 04:30:00 +08:00', 'DD-MON-YY HH:MI:SS TZH:TZM')}</td>
<td>'20-MAR-20 02:00:00 +05:30'</td>
</tr>
<tr>
<td>\texttt{TO_TIMESTAMP_TZ('10-Sep-02 14:10:10.123000', 'DD-MON-RR HH24:MI:SS FF')}</td>
<td>'10-SEP-02 14:10:12.123 +05:30'</td>
</tr>
<tr>
<td>\texttt{TO_TIMESTAMP_TZ('20-MAR-20 04:38:00.123456 PM +83:00')}</td>
<td>'20-MAR-20 09:38:00.123456 +04:08'</td>
</tr>
</tbody>
</table>

Available template patterns for numeric values

This table lists the template patterns available for formatting numeric values:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Value with the specified number of digits</td>
</tr>
<tr>
<td>0</td>
<td>Value with leading zeroes</td>
</tr>
<tr>
<td>(period)</td>
<td>Decimal point</td>
</tr>
</tbody>
</table>
The pattern \( \_ \) results in a value with the same number of digits as there are \( \_ \)s. If a digit isn't available, the server ignores the corresponding \( \_ \)s. The $ pattern doesn't support +, and the $ pattern doesn't support decimal points in the expression.

This table shows some examples of the use of the `TO_NUMBER` function:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TO_NUMBER('-65', 'S99')</code></td>
<td>'-65'</td>
</tr>
<tr>
<td><code>TO_NUMBER('$65', 'L99')</code></td>
<td>'65'</td>
</tr>
<tr>
<td><code>TO_NUMBER('9678584', 'S99999999999')</code></td>
<td>'9678584'</td>
</tr>
<tr>
<td><code>TO_NUMBER('123,456,789', 'S99999999999')</code></td>
<td>'123456789'</td>
</tr>
<tr>
<td><code>TO_NUMBER('121.73', 'S9999999')</code></td>
<td>'121.73'</td>
</tr>
<tr>
<td><code>TO_NUMBER('1210.73')</code></td>
<td>'1210.73'</td>
</tr>
<tr>
<td><code>TO_NUMBER('0101.010','FM99999999.99999')</code></td>
<td>'101.010'</td>
</tr>
</tbody>
</table>

**Numeric modifiers**

This table shows the modifier pattern for numeric formatting:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>FM</td>
<td>Fill mode (suppress trailing zeroes and padding blanks)</td>
<td>FM99.99</td>
</tr>
</tbody>
</table>

**Output formats for string and date functions**

In an output template string (for `TO_CHAR`), certain patterns are recognized and replaced with appropriately formatted data from the value to format. Any text that isn't a template pattern is copied verbatim. Similarly, in an input template string (for anything but `TO_CHAR`), template patterns identify the parts of the input data string to look at and the values to find there.

If you don't specify a date, month, or year when calling `TO_TIMESTAMP`, `TO_TIMESTAMP_TZ`, or `TO_DATE`, then by default the output format considers the first date of a current month or current year. In the following example, date, month, and year isn't specified in the input string. `TO_TIMESTAMP`, `TO_TIMESTAMP_TZ`, and `TO_DATE` returns a default value of the first date of a current month and current year.

```sql
edb=# select to_timestamp('12', 'HH');

 to_timestamp
-------------
01-MAY-20 12:00:00 +05:30
(1 row)

edb=# select to_timestamp_tz('12', 'HH');

 to_timestamp_tz
------------------
01-MAY-20 12:00:00 +05:30
(1 row)

edb=# select to_date('12', 'HH');

 to_date
---------
01-MAY-20 12:00:00
(1 row)
```

**Altering the output format**
Alter the output format of \texttt{TO_DSINTERVAL(string)} using the \texttt{intervalstyle} GUC setting. For example:

```sql
edb=# SET intervalstyle = 'sql_standard';
SET

edb=# select to_dsinterval('80 13:30:00') from dual;
to_dsinterval
-----------------------
80 13:30:00

edb=# SET intervalstyle = 'postgres_verbose';
SET
edb=# select to_dsinterval('80 13:30:00') from dual;
to_dsinterval
------------------------
@ 80 days 13 hours 30 mins
```

Creating an index with the \texttt{IMMUTABLE} form of the \texttt{TO_CHAR} function

Certain cases of the \texttt{TO_CHAR} function results in usage of an \texttt{IMMUTABLE} form of the function. A function is \texttt{IMMUTABLE} if the function doesn't modify the database, and the function returns the same, consistent value dependent upon only its input parameters. That is, the settings of configuration parameters, the locale, the content of the database, and so on don't affect the results returned by the function.

For more information about function volatility categories \texttt{VOLATILE}, \texttt{STABLE}, and \texttt{IMMUTABLE}, see the function volatility doc.

A particular advantage of an \texttt{IMMUTABLE} function is that you can use it in the \texttt{CREATE INDEX} command to create an index based on that function.

For the \texttt{TO_CHAR} function to use the \texttt{IMMUTABLE} form, you must satisfy these conditions:

- The first parameter of the \texttt{TO_CHAR} function must be of data type \texttt{TIMESTAMP}.
- The format specified in the second parameter of the \texttt{TO_CHAR} function must not affect the return value of the function based on factors such as language and locale. For example, you can use a format of \texttt{'YYYY-MM-DD HH24:MI:SS'} for an \texttt{IMMUTABLE} form of the function since the result of the function is the date and time expressed solely in numeric form, regardless of locale settings. However, you can't use a format of \texttt{'DD-MON-YYYY'} for an \texttt{IMMUTABLE} form of the function because the three-character abbreviation of the month can return different results depending upon the locale setting.

Format patterns that result in a non-immutable function include any variations of spelled out or abbreviated months (\texttt{MONTH}, \texttt{MON}), days (\texttt{DAY}, \texttt{DY}), median indicators (\texttt{AM}, \texttt{PM}), or era indicators (\texttt{BC}, \texttt{AD}).

For this example, a table with a \texttt{TIMESTAMP} column is created:

```sql
CREATE TABLE ts_tbl (ts_col TIMESTAMP);
```

This example shows the successful creation of an index with the \texttt{IMMUTABLE} form of the \texttt{TO_CHAR} function.

```sql
edb=# CREATE INDEX ts_idx ON ts_tbl (TO_CHAR(ts_col, 'YYYY-MM-DD HH24:MI:SS'));
CREATE INDEX
edb=# \\dS ts_idx

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
</table>
| to_char| character varying | to_char(ts_col, 'YYYY-MM-DD HH24:MI:SS::character varying)'btree, for table "public.ts_tbl"
```

This results in an error because the format specified in the \texttt{TO_CHAR} function prevents the use of the \texttt{IMMUTABLE} form since the three-character month abbreviation, \texttt{MON}, can result in different return values based on the locale setting.

```sql
edb=# CREATE INDEX ts_idx_2 ON ts_tbl (TO_CHAR(ts_col, 'DD-MON-YYYY'));
ERROR: functions in index expression must be marked IMMUTABLE
```

Difference \texttt{TO_CHAR}, \texttt{TO_NUMBER}, and \texttt{SUBSTR} functions

Initialize the database using the \texttt{INITDBOPTS} variable to create clusters in a mode compatible with Oracle databases. The clusters created in PostgreSQL mode don't include compatibility features. To
create a new cluster in PostgreSQL mode, use the "--no-redwood-compat" option, which offers a similar behavior to PostgreSQL.

A `SELECT` statement that contains one of the functions shown in the following table returns the indicated result in EDB Postgres Advanced Server and PostgreSQL. Listed are the following differences for `TO_CHAR`, `TO_NUMBER`, and `SUBSTR` functions.

**TO_CHAR:** Converts the timestamp to string according to a given format.

**Note**
- The 'syear', 'syyyy', 'year', 'rrrr', 'sp', 'spth', 'scc', 'rr', 'ff', 'ff7', 'ff8', 'ff9', and lowercase 'of', 'tzh', and 'tzm' patterns aren't supported in PostgreSQL.
- The uppercase 'TH' modifier adds fewer padding zeros in EDB Postgres Advanced Server.
- The upper case 'TH' pattern, which displays the suffix in uppercase, isn't supported in EDB Postgres Advanced Server. If you specify the 'TH/th' pattern, the suffix is always displayed in lower case.
- A modifier can appear in an 'FM' pattern, and each occurrence toggles the modifier's effect. To toggle the effect of the 'FM' pattern, specify it an even number of times to turn off the effect and specify an odd number of times to turn on the effect of the 'FM' pattern in PostgreSQL. However, in EDB Postgres Advanced Server, specifying the 'FM' pattern always turns on the effect of the 'FM' pattern.

This table shows the examples for `TO_CHAR(timestamp, text)` function:

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
<th>EDB Postgres Advanced Server</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'of');</code></td>
<td></td>
<td>+05:30</td>
<td>of</td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'OF');</code></td>
<td></td>
<td>+05:30</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'syear');</code></td>
<td>twenty twenty one</td>
<td>s1ear</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'syyyy');</code></td>
<td>2021</td>
<td>s2021</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'year');</code></td>
<td>twenty twenty one</td>
<td>1ear</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'rrrr');</code></td>
<td>2021</td>
<td>rrrr</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'hsp');</code></td>
<td>five</td>
<td>05sp</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'hsph');</code></td>
<td>fifth</td>
<td>05spth</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'scc');</code></td>
<td>21</td>
<td>s21</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'rr');</code></td>
<td>21</td>
<td>rr</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'ff');</code></td>
<td>923282</td>
<td>ff</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'ff7');</code></td>
<td>4192318</td>
<td>ff7</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'ff8');</code></td>
<td>36181700</td>
<td>ff8</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'ff9');</code></td>
<td>640485000</td>
<td>ff9</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'tzh');</code></td>
<td>+05</td>
<td>1sth</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'TZH');</code></td>
<td></td>
<td>+05</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'TZZM');</code></td>
<td>30</td>
<td>1stm</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'tzn');</code></td>
<td>1134</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'idd');</code></td>
<td>16</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'id');</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'ssss');</code></td>
<td>5555</td>
<td>64490</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'thth');</code></td>
<td>Invalid format</td>
<td>0th</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'nth');</code></td>
<td>invalid format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'FMMF');</code></td>
<td>+5:30</td>
<td>FM+05:30</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(current_timestamp, 'FMMF');</code></td>
<td>+5:30</td>
<td>FM+05:30</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(timestamptz'5-MAY-21 17:37:14 +05:30', 'DDTH');</code></td>
<td>05th</td>
<td>05TH</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(timestamptz'5-JAN-0001 17:37:14 +05:30', 'Y,YYYTH');</code></td>
<td>1</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(timestamptz'5-JAN-0001 17:37:14 +05:30', 'Y, YYYYTH');</code></td>
<td>01st</td>
<td>001ST</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(timestamptz'5-JAN-0001 17:37:14 +05:30', 'YYYYTH');</code></td>
<td>01st</td>
<td>001ST</td>
<td></td>
</tr>
<tr>
<td>SELECT <code>TO_CHAR(timestamptz'5-JAN-0001 17:37:14 +05:30', 'YYYY');</code></td>
<td>01st</td>
<td>001ST</td>
<td></td>
</tr>
</tbody>
</table>
**TO_NUMBER**: Converts the string to numeric according to a given format.

**Note**
- PostgreSQL doesn't strictly check the format options. In contrast, EDB Postgres Advanced Server checks the format except in a few cases when the input string has '-' or ',' at the beginning. You don't have to provide the pattern for these characters in the corresponding location in the format string.
- In PostgreSQL, you don't have to match all the digits of the input string with the patterns. It outputs the number of digits from the input string for which the valid patterns are present. EDB Postgres Advanced Server, however, outputs an error if you don't provide patterns for all the digits present in the input string.
- PostgreSQL ignores any invalid characters present in the input or the format string. EDB Postgres Advanced Server throws an error. This kind of behavior can lead to many differences with different types of patterns.
- The patterns 'PL', 'PR', and 'SG' aren't supported in EDB Postgres Advanced Server.

This table shows the examples for the `TO_NUMBER(text, text)` function:

<table>
<thead>
<tr>
<th>Examples</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT TO_NUMBER('$1', '00');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('$$1', '$$0');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('1$', '0$');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('$1$', '$0$');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('1$$1', '0$L0');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('1,11', '0,0');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('1-1', '0S0');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('1-1', '000');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('1.11', '000');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('1..1', '0.0');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('$11-', '$00S');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('11', '');</td>
<td>Returns blank</td>
</tr>
<tr>
<td>SELECT TO_NUMBER('+11', '99S');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('11', 'PL99');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('11-', '99PR');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('1+1-1', '9SG9SG9');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('01,000', '99G999');</td>
<td></td>
</tr>
<tr>
<td>SELECT TO_NUMBER('01,000', 'FM99G999');</td>
<td></td>
</tr>
</tbody>
</table>

**SUBSTR**: Returns a substring of a character value.

**Note**
- The index of the string starts from 1. In PostgreSQL, if zero is the start parameter to fetch only one character, the result returns blank. If you try to fetch two characters, the result shows one character since zero is specified in the start position. However, in EDB Postgres Advanced Server, if zero is the start parameter, internally it converts to 1 and starts the searching.
- Negative values at the start parameter aren't allowed in PostgreSQL. In EDB Postgres Advanced Server, negative values cause a reverse search to occur.
- PostgreSQL throws an error if a numeric value is passed for start and length parameter. EDB Postgres Advanced Server converts it to an integer.

This table shows the examples for `SUBSTR(string, from, count)`, `SUBSTR(string, from)`, `SUBSTR(string, from, count)::bytea`, and `SUBSTR(string, from)::bytea` function:

<table>
<thead>
<tr>
<th>Examples</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT (SUBSTR('ABCDEFGH', 0, 1));</td>
<td>A</td>
</tr>
</tbody>
</table>
### 14.4.3.8 Date/time functions and operators

The date/time functions table shows the available functions for date/time value processing.

#### Overview of basic arithmetic operators

The following table shows the behaviors of the basic arithmetic operators (+, -). For formatting functions, refer to `IMMUTABLE TO_CHAR(TIMESTAMP, format)`. For more information on date/time data types, see [Date/time types](#).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus (+)</td>
<td>DATE '2001-09-28' + 7</td>
<td>05-OCT-01 00:00:00</td>
</tr>
<tr>
<td>plus (+)</td>
<td>TIMESTAMP '2001-09-28 13:30:00' + 3</td>
<td>01-OCT-01 13:30:00</td>
</tr>
<tr>
<td>minus (-)</td>
<td>DATE '2001-10-01' - 7</td>
<td>24-SEP-01 00:00:00</td>
</tr>
<tr>
<td>minus (-)</td>
<td>TIMESTAMP '2001-09-28 13:30:00' - 3</td>
<td>25-SEP-01 13:30:00</td>
</tr>
<tr>
<td>minus (-)</td>
<td>TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00'</td>
<td>@ 1 day 15 hours</td>
</tr>
</tbody>
</table>

#### Overview of date and time functions

In the date/time functions of the following table the use of the **DATE** and **TIMESTAMP** data types are interchangeable.

<table>
<thead>
<tr>
<th>Function</th>
<th>Return type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_MONTHS (DATE, NUMBER)</td>
<td>DATE</td>
<td>Adds months to a date.</td>
<td>ADD_MONTHS ('28-FEB-97', 3..8)</td>
<td>31-MAY-97 08:00:00</td>
</tr>
<tr>
<td>CURRENT_DATE</td>
<td>DATE</td>
<td>Current date.</td>
<td>CURRENT_DATE</td>
<td>04-JUL-07</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>Returns the current date and time.</td>
<td>CURRENT_TIMESTAMP</td>
<td>04-JUL-07 15:33:28.484</td>
</tr>
</tbody>
</table>

---

Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT (SUBSTR('ABCD', -1, 1));</td>
<td>D</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', -4, 1));</td>
<td>A</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', 1, -1));</td>
<td>Returns blank</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', 1.1, 1));</td>
<td>A</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', -1));</td>
<td>D</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', 1.1));</td>
<td>ABCD</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCDEFGH', 0, 1))::bytea;</td>
<td>\x41</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', 5,1))::bytea;</td>
<td>Returns blank</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', -5, 1))::bytea;</td>
<td>Returns blank</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', 1, -1))::bytea;</td>
<td>\x41</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', -1));</td>
<td>D</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', 1.1));</td>
<td>ABCD</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', 1, 1.1));</td>
<td>\x41</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCDEFGH', -1, 1))::bytea;</td>
<td>\x48</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', 1, -1));</td>
<td>\x41</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', -1));</td>
<td>D</td>
</tr>
<tr>
<td>SELECT (SUBSTR('ABCD', 1.1));</td>
<td>ABCD</td>
</tr>
</tbody>
</table>

---

**Examples**

14.4.3.8 Date/time functions and operators

The date/time functions table shows the available functions for date/time value processing.

Overview of basic arithmetic operators

The following table shows the behaviors of the basic arithmetic operators (+, -). For formatting functions, refer to `IMMUTABLE TO_CHAR(TIMESTAMP, format)`. For more information on date/time data types, see [Date/time types](#).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus (+)</td>
<td>DATE '2001-09-28' + 7</td>
<td>05-OCT-01 00:00:00</td>
</tr>
<tr>
<td>plus (+)</td>
<td>TIMESTAMP '2001-09-28 13:30:00' + 3</td>
<td>01-OCT-01 13:30:00</td>
</tr>
<tr>
<td>minus (-)</td>
<td>DATE '2001-10-01' - 7</td>
<td>24-SEP-01 00:00:00</td>
</tr>
<tr>
<td>minus (-)</td>
<td>TIMESTAMP '2001-09-28 13:30:00' - 3</td>
<td>25-SEP-01 13:30:00</td>
</tr>
<tr>
<td>minus (-)</td>
<td>TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00'</td>
<td>@ 1 day 15 hours</td>
</tr>
</tbody>
</table>

Overview of date and time functions

In the date/time functions of the following table the use of the **DATE** and **TIMESTAMP** data types are interchangeable.

<table>
<thead>
<tr>
<th>Function</th>
<th>Return type</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_MONTHS (DATE, NUMBER)</td>
<td>DATE</td>
<td>Adds months to a date.</td>
<td>ADD_MONTHS ('28-FEB-97', 3..8)</td>
<td>31-MAY-97 08:00:00</td>
</tr>
<tr>
<td>CURRENT_DATE</td>
<td>DATE</td>
<td>Current date.</td>
<td>CURRENT_DATE</td>
<td>04-JUL-07</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>Returns the current date and time.</td>
<td>CURRENT_TIMESTAMP</td>
<td>04-JUL-07 15:33:28.484</td>
</tr>
<tr>
<td>Function</td>
<td>Return type</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>EXTRACT(field FROM TIMESTAMP)</td>
<td>DOUBLE PRECISION</td>
<td>Gets subfield.</td>
<td>EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40')</td>
<td>20</td>
</tr>
<tr>
<td>LAST_DAY(DATE)</td>
<td>DATE</td>
<td>Returns the last day of the month represented by the given date. If the given date contains a time portion, it's carried forward to the result unchanged.</td>
<td>LAST_DAY('14-APR-98')</td>
<td>30-APR-98 08:00:00</td>
</tr>
<tr>
<td>LOCALTIMESTAMP</td>
<td>TIMESTAMP</td>
<td>Current date and time (start of current transaction).</td>
<td>LOCALTIMESTAMP</td>
<td>04-JUL-07 15:33:23.484</td>
</tr>
<tr>
<td>MONTHS_BETWEEN (DATE, DATE)</td>
<td>NUMBER</td>
<td>Number of months between two dates.</td>
<td>MONTHS_BETWEEN ('28-FEB-97', '30-NOV-96')</td>
<td>3</td>
</tr>
<tr>
<td>NEXT_DAY(DATE, dayofweek)</td>
<td>DATE</td>
<td>Date falling on dayofweek following specified date.</td>
<td>NEXT_DAY('16-APR-07', 'FRI')</td>
<td>20-APR-07 08:00:00</td>
</tr>
<tr>
<td>NEW_TIME(DATE, VARCHAR, VARCHAR)</td>
<td></td>
<td>Converts a date and time to an alternate time zone.</td>
<td>NEW_TIME (TO_DATE '2005/05/29 01:45', 'AST', 'PST')</td>
<td>04:00:00</td>
</tr>
<tr>
<td>ROUND(DATE [, format ])</td>
<td>DATE</td>
<td>Date rounded according to format.</td>
<td>ROUND(TO_DATE '29-MAY-05', 'MON')</td>
<td>01-JUN-05 01:00:00</td>
</tr>
<tr>
<td>SYS_EXTRACT_UTC(TIMESTAMP WITH TIME ZONE)</td>
<td>TIMESTAMP</td>
<td>Returns Coordinated Universal Time.</td>
<td>SYS_EXTRACT_UTC(CAST '24-MAR-11 12:30:00PM -04:00' AS TIMESTAMP WITH TIME ZONE))</td>
<td>24-MAR-11 16:30:00</td>
</tr>
<tr>
<td>SYSDATE</td>
<td>DATE</td>
<td>Returns current date and time.</td>
<td>SYSDATE</td>
<td>01-AUG-12 11:12:34</td>
</tr>
<tr>
<td>SYSTIMESTAMP()</td>
<td>TIMESTAMP</td>
<td>Returns current date and time.</td>
<td>SYSTIMESTAMP</td>
<td>01-AUG-12 11:11:23.665229 -07:00</td>
</tr>
<tr>
<td>TRUNC(DATE [ format ])</td>
<td>DATE</td>
<td>Truncates according to format.</td>
<td>TRUNC(TO_DATE '29-MAY-05', 'MON')</td>
<td>01-MAY-05 08:00:00</td>
</tr>
<tr>
<td>DBTIMEZONE [+</td>
<td>-]TZH:TZM</td>
<td>+</td>
<td>-]TZH:TZM or time zone region name</td>
<td>Returns time zone offset in the format [+</td>
</tr>
<tr>
<td>SESSIONTIMEZONE [+</td>
<td>-]TZH:TZM</td>
<td>+</td>
<td>-]TZH:TZM or time zone region name</td>
<td>Returns time zone offset in the format [+</td>
</tr>
</tbody>
</table>

**ADD_MONTHS**

The **ADD_MONTHS** functions adds or subtracts the specified number of months to or from the given date. Use a negative value to subtract. The resulting day of the month is the same as the day of the month of the given date except when the day is the last day of the month. In that case, the resulting date always falls on the last day of the month.

Any fractional portion of the number of months parameter is truncated before performing the calculation.

If the given date contains a time portion, it's carried forward to the result unchanged.

These examples show the **ADD_MONTHS** function:

```sql
SELECT ADD_MONTHS('13-JUN-07', 4) FROM DUAL;
add_months
-----------
13-OCT-07 00:00:00
(1 row)

SELECT ADD_MONTHS('31-DEC-06', 2) FROM DUAL;
add_months
-----------
28-FEB-07 00:00:00
(1 row)

SELECT ADD_MONTHS('31-MAY-04', -3) FROM DUAL;
add_months
-----------
29-FEB-04 00:00:00
(1 row)
```
CURRENT_DATE/TIME

EDB Postgres Advanced Server provides functions that return values related to the current date and time. These functions return values based on the start time of the current transaction:

- `CURRENT_DATE`
- `CURRENT_TIMESTAMP`
- `LOCALTIMESTAMP`
- `LOCALTIMESTAMP(precision)`

`CURRENT_DATE` returns the current date and time based on the start time of the current transaction. The value of `CURRENT_DATE` doesn't change if called multiple times in a transaction.

```
SELECT CURRENT_DATE FROM DUAL;
```

```
date
-----
06-AUG-07
```

`CURRENT_TIMESTAMP` returns the current date and time. When called from a single SQL statement, it returns the same value for each occurrence in the statement. If called from multiple statements in a transaction, it might return different values for each occurrence. If called from a function, it might return a value different from the one returned by `current_timestamp` in the caller.

```
SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP FROM DUAL;
```

```
current_timestamp | current_timestamp
-----------------|------------------
02-SEP-13 17:52:29.261473 +05:00 | 02-SEP-13 17:52:29.261474 +05:00
```

You can optionally give `LOCALTIMESTAMP` a precision parameter. The parameter causes the result to be rounded to the specified number of fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

```
SELECT LOCALTIMESTAMP FROM DUAL;
```

```
timestamp
----------
06-AUG-07 16:11:35.973
```

```
SELECT LOCALTIMESTAMP(2) FROM DUAL;
```

```
timestamp
----------
06-AUG-07 16:11:44.58
```

Since these functions return the start time of the current transaction, their values don’t change during the transaction. The intent is to allow a single transaction to have a consistent notion of the “current” time, so that multiple modifications in the same transaction have the same timestamp. Other database systems might advance these values more frequently.

TIMEZONE functions

EDB Postgres Advanced Server provides functions that return timezone set values.

`DBTIMEZONE`

`DBTIMEZONE` returns the value of the database time zone. The return type is a time zone offset in the format [+|-]TZH:TZW or a time zone region name. The return type depends on the database time zone value specified in the recent `CREATE DATABASE` or `ALTER DATABASE` statement.

```
ALTER DATABASE edb SET TIMEZONE='--
89:00';
SELECT dbtimezone FROM DUAL;
```

```
dbtimezone
----------
--89:00
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
**DB Time Zone**

The **dbtimezone** function retrieves the time zone of the current session. It returns a time zone offset in the format `[-+][H:H][M:M]` or a time zone region name. The return type depends on the session time zone value specified in the recent ALTER SESSION or SET statement.

```sql
ALTER DATABASE edb SET
TIMEZONE='Europe/London';
SELECT dbtimezone FROM DUAL;
```

<table>
<thead>
<tr>
<th>dbtimezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Europe/London</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

**SESSIONTIMEZONE**

The **SESSIONTIMEZONE** function returns the time zone of the current session. The return type is a time zone offset in the format `[-+][H:H][M:M]` or a time zone region name. The return type depends on the session time zone value specified in the recent ALTER SESSION or SET statement.

```sql
ALTER SESSION SET timezone to
'Amercia/New_York';
SELECT sessiontimezone() FROM dual;
```

<table>
<thead>
<tr>
<th>sessiontimezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>America/New_York</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

```sql
SET timezone TO
'(+5:30)';
SELECT sessiontimezone() FROM dual;
```

<table>
<thead>
<tr>
<th>sessiontimezone</th>
</tr>
</thead>
<tbody>
<tr>
<td>+5:30</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

**EXTRACT**

The **EXTRACT** function retrieves subfields such as year or hour from date/time values. The **EXTRACT** function returns values of type DOUBLE PRECISION. The following are valid field names:

- **YEAR**
  - The year field.

```sql
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

<table>
<thead>
<tr>
<th>date_part</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

- **MONTH**
  - The number of the month in the year (1 - 12).

```sql
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

<table>
<thead>
<tr>
<th>date_part</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

- **DAY**
  - The day of the month field (1 - 31).

```sql
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

<table>
<thead>
<tr>
<th>date_part</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
HOUR

The hour field (0 - 23).

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

date_part
-----------
20
(1 row)

MINUTE

The minutes field (0 - 59).

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

date_part
-----------
38
(1 row)

SECOND

The seconds field, including fractional parts (0 - 59).

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;
```

date_part
-----------
40
(1 row)

MONTHS_BETWEEN

The MONTHS_BETWEEN function returns the number of months between two dates. The result is a numeric value that's positive if the first date is greater than the second date. It's negative if the first date is less than the second date.

The result is always a whole number of months if the day of the month of both date parameters is the same or both date parameters fall on the last day of their respective months.

These examples show the MONTHS_BETWEEN function:

```
SELECT MONTHS_BETWEEN('15-DEC-06','15-OCT-06') FROM DUAL;
```

months_between
----------------
2
(1 row)

```
SELECT MONTHS_BETWEEN('15-OCT-06','15-DEC-06') FROM DUAL;
```

months_between
----------------
-2
(1 row)

```
SELECT MONTHS_BETWEEN('31-JUL-00','01-JUL-00') FROM DUAL;
```

months_between
----------------
0.967741935
(1 row)

```
SELECT MONTHS_BETWEEN('01-JAN-07','01-JAN-06') FROM DUAL;
```

months_between
----------------
12
(1 row)
**NEXT_DAY**

The `NEXT_DAY` function returns the first occurrence of the given weekday strictly greater than the given date. You must specify at least the first three letters of the weekday, for example, `SAT`. If the given date contains a time portion, it's carried forward to the result unchanged.

These examples show the `NEXT_DAY` function.

```sql
SELECT NEXT_DAY(TO_DATE('13-AUG-07', 'DD-MON-YY'), 'SUNDAY') FROM DUAL;
next_day
---------------------
19-AUG-07 00:00:00
(1 row)
```

```sql
SELECT NEXT_DAY(TO_DATE('13-AUG-07', 'DD-MON-YY'), 'MON') FROM DUAL;
next_day
---------------------
20-AUG-07 00:00:00
(1 row)
```

**NEW_TIME**

The `NEW_TIME` function converts a date and time from one time zone to another. `NEW_TIME` returns a value of type `DATE`. The syntax is:

```
NEW_TIME(DATE, time_zone1, time_zone2)
```

`time_zone1` and `time_zone2` must be string values from the `Time zone` column of the following table.

<table>
<thead>
<tr>
<th>Time zone</th>
<th>Offset from UTC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>UTC+4</td>
<td>Atlantic Standard Time</td>
</tr>
<tr>
<td>ADT</td>
<td>UTC+3</td>
<td>Atlantic Daylight Time</td>
</tr>
<tr>
<td>BST</td>
<td>UTC+11</td>
<td>Bering Standard Time</td>
</tr>
<tr>
<td>BDT</td>
<td>UTC+10</td>
<td>Bering Daylight Time</td>
</tr>
<tr>
<td>CST</td>
<td>UTC+6</td>
<td>Central Standard Time</td>
</tr>
<tr>
<td>CDT</td>
<td>UTC+5</td>
<td>Central Daylight Time</td>
</tr>
<tr>
<td>EST</td>
<td>UTC+5</td>
<td>Eastern Standard Time</td>
</tr>
<tr>
<td>EDT</td>
<td>UTC+4</td>
<td>Eastern Daylight Time</td>
</tr>
<tr>
<td>GMT</td>
<td>UTC</td>
<td>Greenwich Mean Time</td>
</tr>
<tr>
<td>HST</td>
<td>UTC+10</td>
<td>Alaska-Hawaii Standard Time</td>
</tr>
<tr>
<td>HDT</td>
<td>UTC+9</td>
<td>Alaska-Hawaii Daylight Time</td>
</tr>
<tr>
<td>MST</td>
<td>UTC+7</td>
<td>Mountain Standard Time</td>
</tr>
<tr>
<td>MDT</td>
<td>UTC+6</td>
<td>Mountain Daylight Time</td>
</tr>
<tr>
<td>NST</td>
<td>UTC+3:30</td>
<td>Newfoundland Standard Time</td>
</tr>
<tr>
<td>PST</td>
<td>UTC+8</td>
<td>Pacific Standard Time</td>
</tr>
<tr>
<td>PDT</td>
<td>UTC+7</td>
<td>Pacific Daylight Time</td>
</tr>
<tr>
<td>YST</td>
<td>UTC+9</td>
<td>Yukon Standard Time</td>
</tr>
<tr>
<td>YDT</td>
<td>UTC+8</td>
<td>Yukon Daylight Time</td>
</tr>
</tbody>
</table>

This example shows the `NEW_TIME` function:

```sql
Pacific Standard Time
---------------------
13-AUG-07 00:35:15
(1 row)
```
NUMTODSINTERVAL

The `NUMTODSINTERVAL` function converts a numeric value to a time interval that includes day-through-second interval units. When calling the function, specify the smallest fractional interval type to include in the result set. The valid interval types are `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

This example converts a numeric value to a time interval that includes days and hours:

```sql
SELECT numtodsinterval(100, 'hour');
```

```
numtodsinterval
---------------
4 days 04:00:00
(1 row)
```

This example converts a numeric value to a time interval that includes minutes and seconds:

```sql
SELECT numtodsinterval(100, 'second');
```

```
numtodsinterval
---------------
1 min 49 secs
(1 row)
```

NUMTOYMINTERVAL

The `NUMTOYMINTERVAL` function converts a numeric value to a time interval that includes year-through-month interval units. When calling the function, specify the smallest fractional interval type to include in the result set. The valid interval types are `YEAR` and `MONTH`.

This example converts a numeric value to a time interval that includes years and months:

```sql
SELECT numtoyminterval(100, 'month');
```

```
numtoyminterval
---------------
8 years 4 mons
(1 row)
```

This example converts a numeric value to a time interval that includes years only:

```sql
SELECT numtoyminterval(100, 'year');
```

```
numtoyminterval
---------------
108 years
(1 row)
```

ROUND

The `ROUND` function returns a date rounded according to a specified template pattern. If you omit the template pattern, the date is rounded to the nearest day. The following table shows the template patterns for the `ROUND` function.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CC</code>, <code>SCC</code></td>
<td>Returns January 1, <code>cc 01</code> where <code>cc</code> is first 2 digits of the given year if last 2 digits &lt;= 50, or 1 greater than the first 2 digits of the given year if last 2 digits &gt; 50; (for AD years)</td>
</tr>
<tr>
<td><code>SYYY</code>, <code>YYY</code>, <code>YEAR</code>, <code>YYEAR</code>, <code>YY</code>, <code>Y</code></td>
<td>Returns January 1, <code>yyyy</code> where <code>yyyy</code> is rounded to the nearest year; rounds down on June 30, rounds up on July 1</td>
</tr>
<tr>
<td><code>IYYY</code>, <code>IYY</code>, <code>IY</code>, <code>I</code></td>
<td>Rounds to the beginning of the ISO year, which is determined by rounding down if the month and day is on or before June 30th or by rounding up if the month and day is July 1st or later</td>
</tr>
<tr>
<td><code>Q</code></td>
<td>Returns the first day of the quarter determined by rounding down if the month and day is on or before the 15th of the second month of the quarter or by rounding up if the month and day is on the 16th of the second month or later of the quarter</td>
</tr>
<tr>
<td><code>MONTH</code>, <code>MON</code>, <code>MM</code>, <code>RM</code></td>
<td>Returns the first day of the specified month if the day of the month is on or prior to the 15th; returns the first day of the following month if the day of the month is on the 16th or later</td>
</tr>
<tr>
<td><code>Wd</code></td>
<td>Rounds to the nearest date that corresponds to the same day of the week as the first day of the year</td>
</tr>
<tr>
<td>Pattern</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>IM</td>
<td>Rounds to the nearest date that corresponds to the same day of the week as the first day of the ISO year</td>
</tr>
<tr>
<td>W</td>
<td>Rounds to the nearest date that corresponds to the same day of the week as the first day of the month</td>
</tr>
<tr>
<td>DDD, DD, J</td>
<td>Rounds to the start of the nearest day; 11:59:59 AM or earlier rounds to the start of the same day; 12:00:00 PM or later rounds to the start of the next day</td>
</tr>
<tr>
<td>DAY, DY, D</td>
<td>Rounds to the nearest Sunday</td>
</tr>
<tr>
<td>HH, HH12, HH24</td>
<td>Round to the nearest hour</td>
</tr>
<tr>
<td>MI</td>
<td>Rounds to the nearest minute</td>
</tr>
</tbody>
</table>

These examples show the use of the `ROUND` function.

These examples round to the nearest hundred years:

```sql
SELECT TO_CHAR(ROUND(TO_DATE('1950', 'YYYY'), 'CC'), 'DD-MON-YYYY') "Century" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-1901 (1 row)</td>
</tr>
</tbody>
</table>

```sql
SELECT TO_CHAR(ROUND(TO_DATE('1951', 'YYYY'), 'CC'), 'DD-MON-YYYY') "Century" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Century</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-2001 (1 row)</td>
</tr>
</tbody>
</table>

These examples round to the nearest year:

```sql
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-1999', 'DD-MON-YYYY'), 'Y'), 'DD-MON-YYYY') "Year" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-1999 (1 row)</td>
</tr>
</tbody>
</table>

```sql
SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-1999', 'DD-MON-YYYY'), 'Y'), 'DD-MON-YYYY') "Year" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-2000 (1 row)</td>
</tr>
</tbody>
</table>


(An ISO year begins on the first Monday from which a seven-day span, Monday through Sunday, contains at least four days of the new year. It's possible for the beginning of an ISO year to start in December of the prior year.)

```sql
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-2004', 'DD-MON-YYYY'), 'IYYYY'), 'DD-MON-YYYY') "ISO Year" FROM DUAL;
```

<table>
<thead>
<tr>
<th>ISO Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>29-DEC-2003 (1 row)</td>
</tr>
</tbody>
</table>

```sql
SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-2004', 'DD-MON-YYYY'), 'IYYYY'), 'DD-MON-YYYY') "ISO Year" FROM DUAL;
```

<table>
<thead>
<tr>
<th>ISO Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>03-JAN-2005 (1 row)</td>
</tr>
</tbody>
</table>

These examples round to the nearest quarter:

```sql
SELECT ROUND(TO_DATE('15-FEB-07', 'DD-MON-YY'), 'Q') "Quarter" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-07 00:00:00 (1 row)</td>
</tr>
</tbody>
</table>
These examples round to the nearest month:

```sql
SELECT ROUND(TO_DATE('15-DEC-07', 'DD-MON-YY'), 'MONTH') "Month" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-DEC-07</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

These examples round to the nearest week. The first day of 2007 lands on a Monday. In the first example, January 18 is closest to the Monday that lands on January 15. In the second example, January 19 is closer to the Monday that falls on January 22.

```sql
SELECT ROUND(TO_DATE('18-JAN-07', 'DD-MON-YY'), 'WW') "Week" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-JAN-07</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

```sql
SELECT ROUND(TO_DATE('19-JAN-07', 'DD-MON-YY'), 'WW') "Week" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>22-JAN-07</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

These examples round to the nearest ISO week. An ISO week begins on a Monday. In the first example, January 1, 2004 is closest to the Monday that lands on December 29, 2003. In the second example, January 2, 2004 is closer to the Monday that lands on January 5, 2004.

```sql
SELECT ROUND(TO_DATE('01-JAN-04', 'DD-MON-YY'), 'IW') "ISO Week" FROM DUAL;
```

<table>
<thead>
<tr>
<th>ISO Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>29-DEC-03</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

```sql
SELECT ROUND(TO_DATE('02-JAN-04', 'DD-MON-YY'), 'IW') "ISO Week" FROM DUAL;
```

<table>
<thead>
<tr>
<th>ISO Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>05-Jan-04</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

These examples round to the nearest week, where a week is considered to start on the same day as the first day of the month:

```sql
SELECT ROUND(TO_DATE('05-MAR-07', 'DD-MON-YY'), 'W') "Week" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>08-MAR-07</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

```sql
SELECT ROUND(TO_DATE('04-MAR-07', 'DD-MON-YY'), 'W') "Week" FROM DUAL;
```

<table>
<thead>
<tr>
<th>Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-MAR-07</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
These examples round to the nearest day:

<table>
<thead>
<tr>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>04-AUG-07</td>
</tr>
<tr>
<td>05-AUG-07</td>
</tr>
</tbody>
</table>

These examples round to the start of the nearest day of the week (Sunday):

<table>
<thead>
<tr>
<th>Day of Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>05-AUG-07</td>
</tr>
<tr>
<td>12-AUG-07</td>
</tr>
</tbody>
</table>

These examples round to the nearest hour:

<table>
<thead>
<tr>
<th>Hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>09-AUG-07 08:00</td>
</tr>
<tr>
<td>09-AUG-07 09:00</td>
</tr>
</tbody>
</table>

These examples round to the nearest minute:

<table>
<thead>
<tr>
<th>Minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>09-AUG-07 08:30</td>
</tr>
<tr>
<td>09-AUG-07 09:30</td>
</tr>
</tbody>
</table>

/* Day */ FROM DUAL;

SELECT ROUND( TO_DATE('04-AUG-07 11:59:59 AM', 'DD-MON-YY HH:MI:SS AM'), 'J' )
"Day" FROM DUAL;

Day
-----------------------------------
04-AUG-07 00:00:00
(1 row)

SELECT ROUND( TO_DATE('04-AUG-07 12:00:00 PM', 'DD-MON-YY HH:MI:SS AM'), 'J' )
"Day" FROM DUAL;

Day
-----------------------------------
05-AUG-07 00:00:00
(1 row)

SELECT ROUND( TO_DATE('08-AUG-07', 'DD-MON-YY'), 'DAY' )
"Day of Week" FROM DUAL;

Day of Week
-----------------
05-AUG-07 00:00:00
(1 row)

SELECT ROUND( TO_DATE('09-AUG-07', 'DD-MON-YY'), 'DAY' )
"Day of Week" FROM DUAL;

Day of Week
-----------------
12-AUG-07 00:00:00
(1 row)

SELECT TO_CHAR(ROUND( TO_DATE('09-AUG-07 08:29', 'DD-MON-YY HH:MI'), 'HH'), 'DD-MON-YY HH24:MI:SS' )
"Hour" FROM DUAL;

Hour
------------------
09-AUG-07 08:29
(1 row)

SELECT TO_CHAR(ROUND( TO_DATE('09-AUG-07 08:30', 'DD-MON-YY HH:MI'), 'HH'), 'DD-MON-YY HH24:MI:SS' )
"Hour" FROM DUAL;

Hour
------------------
09-AUG-07 09:00
(1 row)

SELECT TO_CHAR(ROUND( TO_DATE('09-AUG-07 08:30:29', 'DD-MON-YY HH:MI:SS'), 'MI' ), 'DD-MON-YY HH24:MI:SS' )
"Minute" FROM DUAL;

Minute
------------------
09-AUG-07 08:30
(1 row)

SELECT TO_CHAR(ROUND( TO_DATE('09-AUG-07 08:30:30', 'DD-MON-YY HH:MI:SS'), 'MI' ), 'DD-MON-YY HH24:MI:SS' )
"Minute" FROM DUAL;

Minute
------------------
09-AUG-07 08:31
(1 row)
SYSDATE

The `SYSDATE` function returns the current date and time (timestamp without timezone) of the operating system on which the database server resides. The function is `STABLE` and requires no arguments.

When called from a single SQL statement, it returns the same value for each occurrence in the statement. If called from multiple statements in a transaction, it might return different values for each occurrence. If called from a function, it might return a value different from the one returned by `SYSDATE` in the caller.

This example shows a call to `SYSDATE`:

```
SELECT SYSDATE, SYSDATE FROM DUAL;
```

<table>
<thead>
<tr>
<th>SYSDATE</th>
<th>SYSDATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>28-APR-20 16:45:28</td>
<td>28-APR-20 16:45:28</td>
</tr>
<tr>
<td>(1 row)</td>
<td></td>
</tr>
</tbody>
</table>

TRUNC

The `TRUNC` function returns a date truncated according to a specified template pattern. If you omit the template pattern, the date is truncated to the nearest day. The following table shows the template patterns for the `TRUNC` function.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC, SCC</td>
<td>Returns January 1, cc 01 where cc is first 2 digits of the given year</td>
</tr>
<tr>
<td>YYYY, YYYY, YEAR, SYEAR, YY, Y</td>
<td>Returns January 1, yyyy where yyyy is the given year</td>
</tr>
<tr>
<td>IYYY, IYY, IY, I</td>
<td>Returns the start date of the ISO year containing the given date</td>
</tr>
<tr>
<td>Q</td>
<td>Returns the first day of the quarter containing the given date</td>
</tr>
<tr>
<td>MONTH, MON, MM, RM</td>
<td>Returns the first day of the specified month</td>
</tr>
<tr>
<td>WM</td>
<td>Returns the start of the ISO week containing the given date</td>
</tr>
<tr>
<td>W</td>
<td>Returns the largest date just prior to or the same as the given date that corresponds to the same day of the week as the first day of the year</td>
</tr>
<tr>
<td>DDD, DD, D</td>
<td>Returns the start of the day for the given date</td>
</tr>
<tr>
<td>DAY, DY, D</td>
<td>Returns the start of the week (Sunday) containing the given date</td>
</tr>
<tr>
<td>HH, HH12, HH24</td>
<td>Returns the start of the hour</td>
</tr>
<tr>
<td>MI</td>
<td>Returns the start of the minute</td>
</tr>
</tbody>
</table>

Following are examples that use the `TRUNC` function.

This example truncates down to the hundred-years unit:

```
SELECT TO_CHAR(TRUNC(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;
```

```
Century
-----------
01-JAN-1901
(1 row)
```

This example truncates down to the year:

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM DUAL;
```

```
Year
-----------
01-JAN-1999
(1 row)
```

This example truncates down to the beginning of the ISO year:

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year" FROM DUAL;
```

```
ISO Year
-----------
```

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
This example truncates down to the start date of the quarter:

```
SELECT TRUNC(TO_DATE('16-FEB-07', 'DD-MON-YY'), 'Q') "Quarter" FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>Quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-07 00:00:00</td>
</tr>
</tbody>
</table>
```

This example truncates to the start of the month:

```
SELECT TRUNC(TO_DATE('16-DEC-07', 'DD-MON-YY'), 'MONTH') "Month" FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-DEC-07 00:00:00</td>
</tr>
</tbody>
</table>
```

This example truncates down to the start of the week determined by the first day of the year. The first day of 2007 lands on a Monday, so the Monday just prior to January 19 is January 15.

```
SELECT TRUNC(TO_DATE('19-JAN-07', 'DD-MON-YY'), 'W') "Week" FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-JAN-07 00:00:00</td>
</tr>
</tbody>
</table>
```


```
SELECT TRUNC(TO_DATE('29-DEC-03', 'DD-MON-YY'), 'IW') "ISO Week" FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>ISO Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>29-DEC-03 00:00:00</td>
</tr>
</tbody>
</table>
```

This example truncates to the start of the week, where a week is considered to start on the same day as the first day of the month:

```
SELECT TRUNC(TO_DATE('21-MAR-07', 'DD-MON-YY'), 'W') "Week" FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-MAR-07 00:00:00</td>
</tr>
</tbody>
</table>
```

This example truncates to the start of the day:

```
SELECT TRUNC(TO_DATE('04-AUG-07 12:00:00 PM', 'DD-MON-YY HH:MI:SS AM'), 'J') "Day" FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>04-AUG-07 00:00:00</td>
</tr>
</tbody>
</table>
```

This example truncates to the start of the week (Sunday):

```
SELECT TRUNC(TO_DATE('09-AUG-07', 'DD-MON-YY'), 'DAY') "Day of Week" FROM DUAL;
```

```
<table>
<thead>
<tr>
<th>Day of Week</th>
</tr>
</thead>
<tbody>
<tr>
<td>05-AUG-07 00:00:00</td>
</tr>
</tbody>
</table>
```

This example truncates to the start of the hour:

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30', 'DD-MON-YY HH:MI'), 'HH'), 'HH24:MI:SS') "Hour" FROM DUAL;
```
14.4.3.9 Sequence manipulation functions

Sequence objects, which are also called sequence generators or sequences, are special single-row tables created with the `CREATE SEQUENCE` command. You usually use a sequence to generate unique identifiers for rows of a table. The sequence functions provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

`sequence` is the identifier assigned to the sequence in the `CREATE SEQUENCE` command.

**sequence.NEXTVAL**

Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute `NEXTVAL` concurrently, each safely receives a distinct sequence value.

**sequence.CURRVAL**

Return the value most recently obtained by `NEXTVAL` for this sequence in the current session. An error is reported if `NEXTVAL` was never called for this sequence in this session. Because this is returning a session-local value, it gives a predictable answer whether or not other sessions executed `NEXTVAL` since the current session did.

If a sequence object was created with default parameters, `NEXTVAL` calls on it return successive values beginning with 1. Use special parameters in the `CREATE SEQUENCE` command for other behaviors.

**Important**

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a `NEXTVAL` operation is never rolled back. After a value is fetched it is considered used, even if the transaction that did the `NEXTVAL` later aborts. This means that aborted transactions can leave unused "holes" in the sequence of assigned values.

14.4.3.10 Conditional expressions

SQL-compliant conditional expressions are available in EDB Postgres Advanced Server.

**CASE**

The SQL `CASE` expression is a generic conditional expression, similar to if/else statements in other languages:

```
CASE WHEN condition THEN result
  [ WHEN ...
  ]
  [ ELSE result
END
```

You can use `CASE` clauses wherever an expression is valid. `condition` is an expression that returns a `BOOLEAN` result. If the result is `TRUE`, then the value of the `CASE` expression is the `result` that follows the condition. If the result is `FALSE`, any subsequent `WHEN` clauses are searched in the same manner. If no `WHEN` condition is `TRUE`, then the value of the `CASE` expression is the `result` in the `ELSE` clause. If the `ELSE` clause is omitted and no condition matches, the result is `NULL`.

For example:
SELECT * FROM test;

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

SELECT a,
CASE WHEN a=1 THEN 'one'
     WHEN a=2 THEN 'two'
     ELSE 'other'
END
FROM test;

<table>
<thead>
<tr>
<th>a</th>
<th>case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
</tr>
<tr>
<td>2</td>
<td>two</td>
</tr>
<tr>
<td>3</td>
<td>other</td>
</tr>
</tbody>
</table>

The data types of all the result expressions must be convertible to a single output type.

The following simple CASE expression is a specialized variant of the general form:

```sql
CASE expression
  WHEN value THEN result
  [ WHEN ... ]
  [ ELSE result ]
END
```

The expression is computed and compared to all the value specifications in the WHEN clauses until one is found that is equal. If no match is found, the result in the ELSE clause (or a null value) is returned.

This same example can be written using the simple CASE syntax:

```sql
SELECT a,
CASE a WHEN 1 THEN 'one'
     WHEN 2 THEN 'two'
     ELSE 'other'
END
FROM test;
```

A CASE expression doesn’t evaluate any subexpressions that aren’t needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

```sql
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

COALESCE

The COALESCE function returns the first of its arguments that isn’t null. Null is returned only if all arguments are null.

```sql
COALESCE(value [, value2 ] ...)
```

It’s often used to substitute a default value for null values when data is retrieved for display or further computation. For example:

```sql
SELECT COALESCE(description, short_description, '(none)') ...
```
Like a `CASE` expression, `COALESCE` doesn't evaluate arguments that aren't needed to determine the result. Arguments to the right of the first non-null argument aren't evaluated. This SQL-standard function provides capabilities similar to `NVL` and `IFNULL`, which are used in some other database systems.

### NULLIF

The `NULLIF` function returns a null value if `value1` and `value2` are equal. Otherwise it returns `value1`.

```sql
NULLIF(value1, value2)
```

You can use this to perform the inverse operation of the `COALESCE` example:

```sql
SELECT NULLIF(value1, '(none)') ...
```

If `value1` is `(none)`, return a null. Otherwise return `value1`.

### NVL

The `NVL` function returns the first of its arguments that isn't null. `NVL` evaluates the first expression. If that expression evaluates to `NULL`, `NVL` returns the second expression.

```sql
NVL(expr1, expr2)
```

The return type is the same as the argument types. All arguments must have the same data type or be coercible to a common type. `NVL` returns `NULL` if all arguments are `NULL`. `'"` is considered as unknown, and if the arguments data type aren't coercible to the common data type, then `NVL` throws an error.

### Examples

This example computes a bonus for noncommissioned employees. If an employee is a commissioned employee, this expression returns the employee's commission. If the employee isn't a commissioned employee, that is, their commission is `NULL`, this expression returns a bonus that's 10% of their salary.

```sql
bonus = NVL(emp.commission, emp.salary * .10)
```

In this example, the type of `1` is numeric and the type of `'"` is considered as unknown. Therefore PostgreSQL decides that the common type is numeric. It tries to interpret the empty string as a numeric value, which produces the indicated error:

```sql
edb=# select nvl('','1');
ERROR:  invalid input syntax for type numeric:
  ""
```

In this example, if `33` is type casted to double precision, it converts to double precision and returns the value as double precision. If `33` is type casted to numeric, it converts to numeric and returns the value as numeric.

```sql
edb=# select NVL(33::double precision,0), pg_typeof(NVL(33::double precision,0)), pg_typeof(NVL(33::numeric,0));
```

<table>
<thead>
<tr>
<th>nvl</th>
<th>pg_typeof</th>
<th>pg_typeof</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>double precision</td>
<td>numeric</td>
</tr>
<tr>
<td>(1 row)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### NVL2

`NVL2` evaluates an expression and returns either the second or third expression, depending on the value of the first expression. If the first expression isn't `NULL`, `NVL2` returns the value in `expr2`. If the first expression is `NULL`, `NVL2` returns the value in `expr3`.

```sql
NVL2(expr1, expr2, expr3)
```

The return type is the same as the argument types. All arguments must have the same data type or be coercible to a common type.

This example computes a bonus for commissioned employees. If a given employee is a commissioned employee, this expression returns an amount equal to 110% of their commission. If the employee isn't a commissioned employee, that is, their commission is `NULL`, this expression returns `0`.

```sql
bonus = NVL2(emp.commission, emp.commission * 1.1, 0)
```
The **NANVL** function returns the first of its arguments that is not-a-number (NaN) value. **NANVL** evaluates the first expression. If that expression evaluates to a NaN value, **NANVL** returns the second expression. If the first expression evaluates to a number value, **NANVL** returns the first expression. The **NANVL** function is useful only for floating-point numbers of type **BINARY_FLOAT** or **BINARY_DOUBLE**.

**NANVL(expr1,expr2)**

This function takes any numeric data type or any non-numeric data type that can be implicitly converted to a numeric data type as arguments. EDB Postgres Advanced Server determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

### Examples

```
SELECT nanvl(124346, 1), nanvl('NaN', 2), nanvl(124346::int, 3), nanvl(124346::int8, 4);
```

<table>
<thead>
<tr>
<th>nanvl</th>
<th>nanvl</th>
<th>nanvl</th>
<th>nanvl</th>
</tr>
</thead>
<tbody>
<tr>
<td>124346</td>
<td>2</td>
<td>124346</td>
<td>124346</td>
</tr>
</tbody>
</table>

(1 row)

```
SELECT nanvl('NaN', 1::int2), nanvl(124346::int, 2::int2), nanvl(124346::int8, 3::int2);
```

<table>
<thead>
<tr>
<th>nanvl</th>
<th>nanvl</th>
<th>nanvl</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>124346</td>
<td>124346</td>
</tr>
</tbody>
</table>

(1 row)

```
SELECT nanvl('NaN',2::int2);
```

<table>
<thead>
<tr>
<th>nanvl</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

(1 row)

The **LNNVL** function provides a concise way to evaluate a condition when one or both operands of the condition might be null.

**LNNVL(condition)**

The function returns **TRUE** if the condition is **FALSE** or **UNKNOWN** and **FALSE** if the condition is **TRUE**.

The condition can evaluate any scalar values but can't be a compound condition containing **AND**, **OR**, or **BETWEEN**.

Use the **LNNVL** function:

- In the **WHERE** clause of a query.
- In the **WHEN** condition in a searched **CASE** expression.
- Anywhere a scalar expression appears, even in contexts where the **IS [NOT] NULL**, **AND**, or **OR** conditions aren't valid but otherwise are required to account for potential nulls.

### Examples

Create a table and insert data:

```
CREATE TABLE t1 (id INT, col1 INT);
INSERT INTO t1 VALUES (10, NULL), (1,1),(2,2), (3,3);
```

Use the **LNNVL** function in a **SELECT** query:

```
SELECT * FROM t1 WHERE lnnvl(col1 > 2) ORDER by id;
```

<table>
<thead>
<tr>
<th>id</th>
<th>col1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
### 14.4.3.11 Aggregate functions

Aggregate functions compute a single result value from a set of input values.

#### Built-in aggregate functions

The built-in aggregate functions are listed in the following tables.

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument type</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG((\text{expression}))</td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>NUMBER for any integer type, DOUBLE PRECISION for a floating-point argument, otherwise the same as the argument data type</td>
<td>The average (arithmetic mean) of all input values</td>
</tr>
<tr>
<td>COUNT((\star))</td>
<td>Any</td>
<td>BIGINT</td>
<td>Number of input rows</td>
</tr>
<tr>
<td>COUNT((\text{expression}))</td>
<td>Any numeric, string, date/time, or bytea type</td>
<td>BIGINT</td>
<td>Number of input rows for which the value of expression is not null</td>
</tr>
<tr>
<td>MAX((\text{expression}))</td>
<td>Any numeric, string, date/time, or bytea type</td>
<td>Same as argument type</td>
<td>Maximum value of expression across all input values</td>
</tr>
<tr>
<td>MIN((\text{expression}))</td>
<td>Any numeric, string, date/time, or bytea type</td>
<td>Same as argument type</td>
<td>Minimum value of expression across all input values</td>
</tr>
<tr>
<td>SUM((\text{expression}))</td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>BIGINT for SMALLINT or INTEGER arguments, NUMBER for BIGINT arguments, DOUBLE PRECISION for floating-point arguments, otherwise the same as the argument data type</td>
<td>Sum of expression across all input values</td>
</tr>
</tbody>
</table>
Except for `COUNT`, these functions return a null value when no rows are selected. In particular, `SUM` of no rows returns null, not zero. You can use the `COALESCE` function to substitute zero for null when necessary.

### Aggregate functions for statistical analysis

The following table shows the aggregate functions typically used in statistical analysis. Where the description mentions `N`, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when `N` is zero.

<table>
<thead>
<tr>
<th>Function</th>
<th>Argument type</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CORR(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Correlation coefficient</td>
</tr>
<tr>
<td><code>COVAR_POP(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Population covariance</td>
</tr>
<tr>
<td><code>COVAR_SAMP(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Sample covariance</td>
</tr>
<tr>
<td><code>REGR_AVGX(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Average of the independent variable (sum (X) / N)</td>
</tr>
<tr>
<td><code>REGR_AVGY(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Average of the dependent variable (sum (Y) / N)</td>
</tr>
<tr>
<td><code>REGR_COUNT(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Number of input rows in which both expressions are nonnull</td>
</tr>
<tr>
<td><code>REGR_INTERCEPT(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs</td>
</tr>
<tr>
<td><code>REGR_R2(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Square of the correlation coefficient</td>
</tr>
<tr>
<td><code>REGR_SLOPE(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Slope of the least-squares-fit linear equation determined by the (X, Y) pairs</td>
</tr>
<tr>
<td><code>REGR_SXX(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Sum (X^2) – sum (X)^2 / N (“sum of squares” of the independent variable)</td>
</tr>
<tr>
<td><code>REGR_SXY(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Sum (X*Y) – sum (X) * sum (Y) / N (“sum of products” of independent times dependent variable)</td>
</tr>
<tr>
<td><code>REGR_SYY(Y, X)</code></td>
<td>DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
<td>Sum (Y^2) – sum (Y)^2 / N (“sum of squares” of the dependent variable)</td>
</tr>
<tr>
<td><code>STDDEV(expression)</code></td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Historic alias for <code>STDDEV_SAMP</code></td>
</tr>
<tr>
<td><code>STDDEV_POP(expression)</code></td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Population standard deviation of the input values</td>
</tr>
<tr>
<td><code>STDDEV_SAMP(expression)</code></td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Sample standard deviation of the input values</td>
</tr>
<tr>
<td><code>VARIANCE(expression)</code></td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Historical alias for <code>VAR_SAMP</code></td>
</tr>
<tr>
<td><code>VAR_POP(expression)</code></td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Population variance of the input values (square of the population standard deviation)</td>
</tr>
<tr>
<td><code>VAR_SAMP(expression)</code></td>
<td>INTEGER, REAL, DOUBLE PRECISION, NUMBER</td>
<td>DOUBLE PRECISION for floating-point arguments, otherwise NUMBER</td>
<td>Sample variance of the input values (square of the sample standard deviation)</td>
</tr>
</tbody>
</table>

### LISTAGG

`LISTAGG` is an aggregate function that concatenates data from multiple rows into a single row in an ordered manner. You can optionally include a custom delimiter for your data.

The `LISTAGG` function mandates the use of an `ORDER BY` clause under a `WITHIN GROUP` clause to concatenate values of the measure column and then generate the ordered aggregated data.

#### Objective

- You can use `LISTAGG` without any grouping. In this case, the `LISTAGG` function operates on all rows in a table and returns a single row.
- You can use `LISTAGG` with the `GROUP BY` clause. In this case, the `LISTAGG` function operates on each group and returns an aggregated output for each group.
- You can use `LISTAGG` with the `OVER` clause. In this case, the `LISTAGG` function partitions a query result set into groups based on the expression in the `query_partition_by_clause` and then aggregates data in each group.

#### Synopsis

```
LISTAGG( <measure_expr> [ , <delimiter> ] ) WITHIN GROUP( <order_by_clause> ) 
[ OVER <query_partition_by_clause> ]
```
Parameters

measure_expr

*measure_expr* (mandatory) specifies the column or expression that assigns a value to aggregate. **NULL** values are ignored.

delimiter

delimiter (optional) specifies a string that separates the concatenated values in the result row. The delimiter can be a **NULL** value, string, character literal, column name, or constant expression. If ignored, the **LISTAGG** function uses a **NULL** value by default.

order_by_clause

order_by_clause (mandatory) determines the sort order in which the concatenated values are returned.

query_partition_by_clause

query_partition_by_clause (optional) allows the **LISTAGG** function to be used as an analytic function and sets the range of records for each group in the **OVER** clause.

Return type

The **LISTAGG** function returns a string value.

Examples

This example concatenates the values in the **emp** table and lists all the employees separated by a *delimiter* comma. First, create a table named **emp**. Then insert records into the **emp** table.

```sql
CREATE TABLE emp (EMPNO NUMBER(4) NOT NULL,
ENAME VARCHAR2(18),
JOB VARCHAR2(9),
MGR NUMBER(4),
HIREDATE DATE,
SAL NUMBER(7, 2),
COMM NUMBER(7, 2),
DEPTNO NUMBER(2));

INSERT INTO emp VALUES (7499, 'ALLEN', 'SALESMAN',
TO_DATE('20-FEB-1981', 'DD-MON-YYYY'), 1600, 300, 30);
INSERT INTO emp VALUES (7521, 'WARD', 'SALESMAN',
TO_DATE('22-FEB-1981', 'DD-MON-YYYY'), 1250, 500, 30);
INSERT INTO emp VALUES (7566, 'JONES', 'MANAGER',
TO_DATE('2-APR-1981', 'DD-MON-YYYY'), 2975, NULL, 20);
INSERT INTO emp VALUES (7654, 'MARTIN', 'SALESMAN',
TO_DATE('28-SEP-1981', 'DD-MON-YYYY'), 1250, 1480, 30);
INSERT INTO emp VALUES (7698, 'BLAKE', 'MANAGER',
TO_DATE('1-MAY-1981', 'DD-MON-YYYY'), 2850, NULL, 30);
```
This example uses aPARTITION BY clause with LISTAGG in theemp table. It generates output based on a partition bydeptno that applies to each partition and not on the entire table.

This example includes theGROUP BY clause.

MEDIAN

TheMEDIANfunction calculates the middle value of an expression from a given range of values. NULL values are ignored. TheMEDIANfunction returns an error if a query doesn’t reference the user-defined table.

Objective

- You can useMEDIANwithout any grouping. In this case, theMEDIANfunction operates on all rows in a table and returns a single row.
- You can useMEDIANwith theOVERclause. In this case, theMEDIANfunction partitions a query result set into groups based on theexpressionspecified in thePARTITION BYclause. It then aggregates data in each group.

Synopsis

```sql
MEDIAN( <median_expression> ) [ OVER ( [ PARTITION BY... ] ) ]
```

Parameters

- **median_expression**: (mandatory) is a target column or expression that theMEDIANfunction operates on and returns a median value. It can be a numeric, datetime, or interval data type.
- **PARTITION BY**: (optional) allows you to useMEDIANas an analytic function and sets the range of records for each group in theOVERclause.

Return types

The return type is determined by the input data type ofexpression. The following table shows the return type for each input type.

<table>
<thead>
<tr>
<th>Input type</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>NUMERIC</td>
</tr>
</tbody>
</table>
### Examples

In this example, a query returns the median salary for each department in the `emp` table:

```sql
eddb=# SELECT * FROM emp;

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>mgr</th>
<th>hiredate</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17-DEC-80 00:00:00</td>
<td>808.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20-FEB-81 00:00:00</td>
<td>1600.00</td>
<td>300.0</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22-FEB-81 00:00:00</td>
<td>1250.00</td>
<td>500.0</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02-APR-81 00:00:00</td>
<td>2975.00</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28-SEP-81 00:00:00</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
</tr>
</tbody>
</table>
(5 rows)
```

```sql
eddb=# SELECT MEDIAN (SAL) FROM emp;

______
median
--------
1250
(1 row)
```

This example uses the `PARTITION BY` clause with `MEDIAN` in the `emp` table and returns the median salary based on a partition by `deptno`:

```sql
eddb=# SELECT EMPNO, ENAME, DEPTNO, MEDIAN (SAL) OVER (PARTITION BY DEPTNO) FROM emp;

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>deptno</th>
<th>median</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>20</td>
<td>1887.5</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>20</td>
<td>1887.5</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>30</td>
<td>1250</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>30</td>
<td>1250</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>30</td>
<td>1250</td>
</tr>
</tbody>
</table>
(5 rows)
```

You can compare the `MEDIAN` function with `PERCENTILE_CONT`. In this example, `MEDIAN` generates the same result as `PERCENTILE_CONT`:

```sql
eddb=# SELECT MEDIAN (SAL), PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY SAL) FROM emp;

<table>
<thead>
<tr>
<th>median</th>
<th>percentile_cont</th>
</tr>
</thead>
<tbody>
<tr>
<td>1250</td>
<td>1250</td>
</tr>
</tbody>
</table>
(1 row)
```

### STATS_MODE

The `STATS_MODE` function takes a set of values as an argument and returns the value that occurs with the highest frequency. If multiple values appear with the same frequency, the `STATS_MODE` function arbitrarily chooses the first value and returns only that one value.
You can use \texttt{STATS\_MODE} without any grouping. In this case, the \texttt{STATS\_MODE} function operates on all the rows in a table and returns a single value.

You can use \texttt{STATS\_MODE} as an ordered-set aggregate function using the \texttt{WITHIN GROUP} clause. In this case, the \texttt{STATS\_MODE} function operates on the ordered data set.

You can use \texttt{STATS\_MODE} with the \texttt{GROUP BY} clause. In this case, the \texttt{STATS\_MODE} function operates on each group and returns the most frequent and aggregated output for each group.

**Synopsis**

\begin{verbatim}
STATS\_MODE( <expr>
}
\end{verbatim}

Or

\begin{verbatim}
STATS\_MODE() WITHIN GROUP ( ORDER BY sort_expression
}
\end{verbatim}

**Parameters**

\begin{verbatim}
expr
\end{verbatim}

An expression or value to assign to the column.

**Return type**

The \texttt{STATS\_MODE} function returns a value that appears frequently. However, if all the values of a column are \texttt{NULL}, \texttt{STATS\_MODE} returns \texttt{NULL}.

**Examples**

This example returns the mode of salary in the \texttt{emp} table:

\begin{verbatim}
edb=# SELECT * FROM emp;

empno | ename  | job    | mgr | hiredate      | sal   | comm  | deptno
------|--------|--------|-----|---------------|-------|-------|-------
7369  | SMITH  | CLERK  | 792 | 17-DEC-80 00:00:00 | 800.00 |       |   20
7499  | ALLEN  | SALESMAN| 7698| 20-FEB-81 00:00:00 | 1600.00| 300.0 |   30
7521  | WARD   | SALESMAN| 7698| 22-FEB-81 00:00:00 | 1250.00| 500.0 |   30
7566  | JONES  | MANAGER | 7839| 02-APR-81 00:00:00 | 2975.00|       |   20
7654  | MARTIN | SALESMAN| 7698| 28-SEP-81 00:00:00 | 1250.00| 1400.0|   30
(5 rows)

edb=# SELECT STATS\_MODE(SAL) FROM emp;

stats_mode
--------
1250.00
(1 row)
\end{verbatim}

This example uses \texttt{GROUP BY} and \texttt{ORDER BY} clauses with \texttt{STATS\_MODE} in the \texttt{emp} table and returns the salary based on a partition by \texttt{deptno}:

\begin{verbatim}
edb=# SELECT STATS\_MODE(SAL) FROM emp GROUP BY DEPTNO ORDER BY DEPTNO;

stats_mode
--------
800.00
1250.00
(2 rows)
\end{verbatim}

This example uses the \texttt{WITHIN GROUP} clause with \texttt{STATS\_MODE} to perform aggregation on the ordered data set:

\begin{verbatim}
SELECT STATS\_MODE() WITHIN GROUP(ORDER BY SAL) FROM emp;

stats_mode
--------
1250.00
(1 row)
\end{verbatim}
**BIT_AND_AGG**

*BIT_AND_AGG* is a bitwise aggregation function that performs a bitwise AND operation and returns a value based on the data type of input argument.

**Objective**

- You can use the *BIT_AND_AGG* function with the GROUP BY clause. In this case, the *BIT_AND_AGG* function operates on a group and returns the result of a bitwise AND operation.
- You can use the DISTINCT or UNIQUE keywords with the *BIT_AND_AGG* function to ensure that unique values in the *expr* are used for computation.

**Synopsis**

```
BIT_AND_AGG ( [DISTINCT | ALL | UNIQUE] <expr>
)
```

**Parameters**

*expr*

An expression or value to assign to the column.

**Return Type**

The *BIT_AND_AGG* function returns the same value as the data type of the input argument. However, if all the values of a column are NULL, the *BIT_AND_AGG* returns NULL.

**Examples**

This example applies the *BIT_AND_AGG* function to the *sal* column in an *emp* table and groups the result by *deptno*:

```
edb=# SELECT deptno,BIT_AND_AGG(DISTINCT sal) FROM emp GROUP BY deptno;
```

<table>
<thead>
<tr>
<th>deptno</th>
<th>bit_and_agg</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>2975.00</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>

(2 rows)

**BIT_OR_AGG**

*BIT_OR_AGG* is a bitwise aggregation function that performs a bitwise OR operation and returns a value based on the data type of input argument.

**Objective**

- You can use the *BIT_OR_AGG* function with the GROUP BY clause. In this case, the *BIT_OR_AGG* function operates on a group and returns the result of a bitwise OR operation.
- You can use the DISTINCT or UNIQUE keywords with the *BIT_OR_AGG* function to ensure that unique values in the *expr* are used for computation.

**Synopsis**

```
BIT_OR_AGG ( [DISTINCT | ALL | UNIQUE] <expr>
)
```

**Parameters**

*expr*

An expression or value to assign to the column.
The `BIT_OR_AGG` function returns the same value as the data type of the input argument. However, if all the values of a column are `NULL`, `BIT_OR_AGG` returns `NULL`.

### Examples

This example applies `BIT_OR_AGG` to the `sal` column in an `emp` table and groups the result by `deptno`:

```sql
edb=# SELECT deptno,BIT_OR_AGG(DISTINCT sal) FROM emp GROUP BY deptno;

| deptno | bit_or_agg |
|--------+-----------|
| 20     | 2975.00   |
| 30     | 4866      |
(2 rows)
```

### 14.4.3.12 Subquery expressions

SQL-compliant subquery expressions are available in EDB Postgres Advanced Server. All of these expression forms return Boolean (TRUE/FALSE) results.

**EXISTS**

The argument of `EXISTS` is an arbitrary `SELECT` statement or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of `EXISTS` is `TRUE`. If the subquery returns no rows, the result of `EXISTS` is `FALSE`.

```sql
EXISTS(subquery)
```

The subquery can refer to variables from the surrounding query that act as constants during any one evaluation of the subquery.

The subquery is generally executed only far enough to determine whether at least one row is returned, not all the way to completion. We recommend that you don't write a subquery that has any side effects (such as calling sequence functions). Whether the side effects occur might be difficult to predict.

Since the result depends only on whether any rows are returned and not on the contents of those rows, the output list of the subquery is normally uninteresting. A common coding convention is to write all `EXISTS` tests in the form `EXISTS(SELECT 1 WHERE ...)`. There are exceptions to this rule, however, such as subqueries that use `INTERSECT`.

This simple example is like an inner join on `deptno`, but it produces at most one output row for each `dept` row, even though there are multiple matching `emp` rows:

```sql
SELECT dname FROM dept WHERE EXISTS (SELECT 1 FROM emp WHERE emp.deptno = dept.deptno);

dname
--------
ACCOUNTING  
RESEARCH    
SALES       
(3 rows)
```

**IN**

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of `IN` is `TRUE` if any equal subquery row is found. The result is `FALSE` if no equal row is found (including the special case where the subquery returns no rows).

```sql
expression IN (subquery)
```

If the left-hand expression yields `NULL`, or if there are no equal right-hand values and at least one right-hand row yields `NULL`, the result of the `IN` construct is `NULL`, not `FALSE`. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, you can't assume that the subquery will evaluate completely.

**NOT IN**

Copyright © 2009 - 2023 EnterpriseDB Corporation. All rights reserved.
The right-hand side is a parenthesized subquery that must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of `NOT IN` is **TRUE** if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is **FALSE** if any equal row is found.

<table>
<thead>
<tr>
<th>expression</th>
<th><code>NOT IN</code> (subquery)</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td><em>operator</em> <code>ANY</code> (subquery)</td>
</tr>
<tr>
<td>expression</td>
<td><em>operator</em> <code>SOME</code> (subquery)</td>
</tr>
<tr>
<td>SOME</td>
<td>is a synonym for <code>ANY</code>. <code>IN</code> is equivalent to <code>&lt;&gt; ANY</code>.</td>
</tr>
</tbody>
</table>

If the left-hand expression yields `NULL`, or if there are no equal right-hand values and at least one right-hand row yields `NULL`, the result of the `NOT IN` construct is `NULL`, not `TRUE`. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, you can't assume that the subquery will evaluate completely.

**ANY/SOME**

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of `ANY` is **TRUE** if any true result is obtained. The result is **FALSE** if no true result is found (including the special case where the subquery returns no rows).

<table>
<thead>
<tr>
<th>expression</th>
<th><code>operator</code> <code>ANY</code> (subquery)</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td><code>operator</code> <code>SOME</code> (subquery)</td>
</tr>
</tbody>
</table>

If there are no successes and at least one right-hand-row yields `NULL` for the operator’s result, the result of the `ANY` construct is `NULL`, not `FALSE`. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, you can't assume that the subquery will evaluate completely.

**ALL**

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of `ALL` is **TRUE** if all rows yield true (including the special case where the subquery returns no rows). The result is **FALSE** if any false result is found. The result is `NULL` if the comparison doesn't return `FALSE` for any row and it returns `NULL` for at least one row.

<table>
<thead>
<tr>
<th>expression</th>
<th><code>operator</code> <code>ALL</code> (subquery)</th>
</tr>
</thead>
</table>

`NOT IN` is equivalent to `<> ALL`. As with `EXISTS`, you can't assume that the subquery will evaluate completely.

### 14.4.3.13 Identifier functions

Identifier functions that information about the instance and session.

**SYS_GUID**

The `SYS_GUID` function generates and returns a globally unique identifier. The identifier takes the form of 16 bytes of `RAW` data. The `SYS_GUID` function is based on the `uuid-ossp` module to generate universally unique identifiers. The signature is:

```
SYS_GUID()
```

**Example**

This example adds a column to the table `EMP`, inserts a unique identifier, and returns a 16-byte `RAW` value:

```
edb=# CREATE TABLE EMP(C1 RAW (16) DEFAULT SYS_GUID() PRIMARY KEY, C2 INT);
CREATE TABLE
edb=# INSERT INTO EMP(C2) VALUES (1);
INSERT 0 1
edb=# SELECT * FROM EMP;
```

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>----------------</td>
<td>----</td>
</tr>
</tbody>
</table>
The `USERENV` function retrieves information about the current session. The signature is:

```
USERENV(<parameter>)
```

The `parameter` specifies a value to return from the current session. The table shows the possible `parameter` values.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ISDBA</code></td>
<td>Returns <code>TRUE</code> if the current user has DBA privileges, otherwise <code>FALSE</code>.</td>
</tr>
<tr>
<td><code>LANGUAGE</code></td>
<td>The language, territory, and character set of the current session in the following format: <code>language_territory.characterset</code>.</td>
</tr>
<tr>
<td><code>LANG</code></td>
<td>The ISO abbreviation for the language name, a short name for the existing <code>LANGUAGE</code> parameter.</td>
</tr>
<tr>
<td><code>SID</code></td>
<td>The current session identifier.</td>
</tr>
<tr>
<td><code>TERMINAL</code></td>
<td>The current session's operating system terminal identifier.</td>
</tr>
</tbody>
</table>

**Examples**

This example returns the `ISDBA` parameter of the current session:

```
edb=# SELECT USERENV('ISDBA') FROM DUAL;
__OUTPUT__
userenv
---------
TRUE
(1 row)
```

This example returns the `LANG` parameter of the current session:

```
edb=# SELECT USERENV('LANG') FROM DUAL;
__OUTPUT__
userenv
---------
en
(1 row)
```

This example returns the `LANGUAGE` parameter of the current session:

```
edb=# SELECT USERENV('LANGUAGE') FROM DUAL;
__OUTPUT__
userenv
--------------
English_USA.UTF-8
(1 row)
```

This example returns the `TERMINAL` identifier:

```
edb=# SELECT USERENV('TERMINAL') FROM DUAL;
userenv
-------
[local]
(1 row)
```

This example returns the `SID` number of the current session:

```
edb=# SELECT USERENV('SID') FROM DUAL;
userenv
-------
56867
(1 row)
```
SYS_CONTEXT

The `SYS_CONTEXT` function returns the value of a `parameter` associated with the context `namespace` in the current session. The signature is:

```
SYS_CONTEXT(<namespace>, <parameter>)
```

Or

```
SYS_CONTEXT(<userenv>, <parameter>)
```

Parameters

- **namespace**
  - `namespace` can be any named context. `USERENV` is a built-in context that shows information about the current session.

- **parameter**
  - The `parameter` is a defined attribute of a namespace. The following table lists predefined attributes of the `USERENV` namespace.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ISDBA</code></td>
<td>Returns <code>TRUE</code> if the current user has DBA privileges, otherwise <code>FALSE</code>.</td>
</tr>
<tr>
<td><code>LANGUAGE</code></td>
<td>The language, territory, and character set of the current session in the following format: <code>language_territory.characterset</code></td>
</tr>
<tr>
<td><code>LANG</code></td>
<td>The ISO abbreviation for the language name, a short name for the existing <code>LANGUAGE</code> parameter.</td>
</tr>
<tr>
<td><code>SID</code></td>
<td>The current session identifier.</td>
</tr>
<tr>
<td><code>TERMINAL</code></td>
<td>The current session's operating system terminal identifier.</td>
</tr>
</tbody>
</table>

Examples

In these examples, the built-in `USERENV` namespace is used with the `SYS_CONTEXT` function.

This example returns the `ISDBA` parameter of the current session:

```
edb=# SELECT SYS_CONTEXT('USERENV','ISDBA') AS ISDBA FROM DUAL;

    isdb
    -------
     TRUE
(1 row)
```

This example returns the `LANG` parameter of the current session:

```
edb=# SELECT SYS_CONTEXT('USERENV','LANG') AS LANG FROM DUAL;

    lang
    ------
    en
(1 row)
```

This example returns the `LANGUAGE` parameter of the current session:

```
edb=# SELECT SYS_CONTEXT('USERENV','LANGUAGE') AS LANGUAGE FROM DUAL;

    language
         ------------
         English_USA.UTF-8
(1 row)
```

This example returns the `TERMINAL` identifier:

```
edb=# SELECT SYS_CONTEXT('USERENV','TERMINAL') AS TERMINAL FROM DUAL;

    terminal
    ------------

(1 row)
```
This example returns the SID number of the current session:

```
EDB# SELECT SYS_CONTEXT('USERENV','SID') AS SID FROM DUAL;
```

```
sid
-------
56867
(1 row)
```

14.4.3.14 Bitwise functions

The bitwise functions include `BITAND` and `BITOR`.

**BITAND**

The `BITAND` function performs a bitwise AND operation and returns a value based on the data type of input argument. The signature is:

```
BITAND(<expr1>, <expr2>)
```

The input data type of `expr1` and `expr2` is a NUMBER on which the bitwise AND operation is performed. The `BITAND` compares each bit of `expr1` value to the corresponding bit of `expr2` using a bitwise AND operation. It returns the value based on the data type of the input argument. If either argument to `BITAND` is NULL, the result is a NULL value.

**Return types**

The `BITAND` function returns the same value as the data type of the input argument.

**Examples**

This example performs an AND operation on the numbers 10 and 11.

```
EDB# SELECT BITAND(10,11) FROM DUAL;
```

```
bitand
-------
 10
(1 row)
```

**BITOR**

The `BITOR` function performs a bitwise OR operation and returns a value based on the data type of the input argument. The signature is:

```
BITOR(<expr1>, <expr2>)
```

The `BITOR` compares each bit of `expr1` value to the corresponding bit of `expr2` using a bitwise OR operation. It returns the value based on the data type of the input argument.

**Return types**

The `BITOR` function returns the same value as the data type of the input argument. If all the values of a column are NULL, the `BITOR` returns NULL.

**Examples**

This example performs an OR operation on the numbers 10 and 11.
BITWISE OPERATORS

The bitwise operators perform the same operation as `BITAND` and `BITOR`. The bitwise operators are:

- `&`: (Bitwise AND) Return the result of a bitwise AND operation performed on the input type.
- `|`: (Bitwise OR) Return the result of a bitwise OR operation performed on the input type.

Examples

This example uses the `&` operator and returns the output as `BITAND` of the input:

```
edb=# SELECT 10&11 FROM DUAL;
```

```
+------------------+
<table>
<thead>
<tr>
<th>column</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

This example uses the `|` operator and returns the output as `BITOR` of the input:

```
edb=# SELECT 10|11 FROM DUAL;
```

```
+------------------+
<table>
<thead>
<tr>
<th>column</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

14.4.3.15 DUMP function

The DUMP function returns a VARCHAR2 value that contains the datatype code, length in bytes, and internal representation of the expression.

```
DUMP(expr[, <return_fmt> [, <start_position> [, <length> ] ] ] )
```

Where `return_fmt` specifies the format of the any of these return values:

- `8` returns the result in octal notation.
- `10` returns the result in decimal notation.
- `16` returns the result in hexadecimal notation.
- `17` returns each byte printed as a character only if it can be interpreted as a printable character in the character set of the compiler.

The arguments `start_position` and `length` combine to determine the portion of the internal representation to return. The default is to return the entire internal representation in decimal notation.

If `expr` is null, then the function returns `NULL`.

Examples

```
SELECT dump('abc', 10);
```

```
+------------------+
<table>
<thead>
<tr>
<th>dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typ=25 Len=3: 97,98,99</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```
14.4.3.16  TO_MULTI_BYTE function

`TO_MULTI_BYTE` function returns char with all its single-byte characters converted to their corresponding multibyte characters. Char can be of data type `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2`. The value returned is in the same data type as char.

Any single-byte characters in char with no multibyte equivalents appear in the output string as single-byte characters. This function applies if your database character set contains single-byte and multibyte characters.

Examples

```sql
SELECT to_multi_byte('ABC&123') FROM dual;

<table>
<thead>
<tr>
<th>to_multi_byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC&amp;123</td>
</tr>
</tbody>
</table>

SELECT octet_length('A') FROM dual;

<table>
<thead>
<tr>
<th>octet_length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

SELECT to_timestamp('11-06-2021 12:45:24','MM-DD-YYYY HH:MI:SS'), 1008 FROM dual;

```

SELECT octet_length(to_timestamp('11-06-2021 12:45:24','MM-DD-YYYY HH:MI:SS'), 1008) FROM dual;

<table>
<thead>
<tr>
<th>octet_length</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
</tr>
</tbody>
</table>
```
14.4.3.17  TO_SINGLE_BYTE function

TO_SINGLE_BYTE returns char with all its multibyte characters converted to their corresponding single-byte characters. Char can be of data type CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The value returned is in the same data type as char.

Any multibyte characters in char that have no single-byte equivalents appear in the output as multibyte characters. This function applies if your database character set contains single-byte and multibyte characters.

Examples

```
SELECT ascii(to_multi_byte('A')) FROM dual;

<table>
<thead>
<tr>
<th>ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

SELECT to_hex(ascii(to_multi_byte('A'))) FROM dual;

<table>
<thead>
<tr>
<th>to_hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff21</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```

```
SELECT ascii(to_single_byte('')) FROM dual;

<table>
<thead>
<tr>
<th>ascii</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>

SELECT to_hex(ascii(to_single_byte(''))) FROM dual;

<table>
<thead>
<tr>
<th>to_hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
</tr>
<tr>
<td>(1 row)</td>
</tr>
</tbody>
</table>
```
```sql
SELECT TO_SINGLE_BYTE(CHR(65313)) FROM dual;
```

<table>
<thead>
<tr>
<th>to_single_byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

(1 row)