EDB Postgres Advanced Server

Version 11

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Built at 2024-05-17T20:29:25

1 EDB Postgres Advanced Server 19
2 EDB Postgres Advanced Server Release Notes 19
2.1 EDB Postgres Advanced Server 11.22.34 release notes 20
2.2 EDB Postgres Advanced Server 11.21.33 release notes 20
2.3 Version 11.21.32 20
2.4 Version 11.20.31 21
2.5 Version 11.19.30 21
2.6 Version 11.18.29 22
2.7 Version 11.17.28 22
2.8 Version 11.16.26 22
2.9 Version 11.15.25 23
2.10 Version 11.14.24 23
2.11 Version 11.13.23 24
2.12 Version 11.12.22 24
2.13 Version 11.12.21 24
2.14 Version 11.11.20 25
2.15 Version 11.10.19 25
2.16 Version 11.9.17 26
2.17 Version 11.9.16 26
2.18 Version 11.8.15 27
2.19 Version 11.7.14 28
2.20 Version 11.6.13 28
2.21 Version 11.5.12 29
2.22 Version 11.4.11 30
2.23 Version 11.3.10 31
2.24 Version 11.2.9 31
2.25 Version 11.1.7 32
3 Supported Platforms 38
4 Limitations 38
5 Installing EDB Postgres Advanced Server 39
5.1 Installing EDB Postgres Advanced Server on Linux x86 (amd64) 40
5.1.1 Installing EDB Postgres Advanced Server on RHEL 9 or OL 9 x86_64 41
5.1.2 Installing EDB Postgres Advanced Server on RHEL 8 or OL 8 x86_64 43
5.1.3 Installing EDB Postgres Advanced Server on AlmaLinux 9 or Rocky Linux 9 x86_64 46
5.1.4 Installing EDB Postgres Advanced Server on AlmaLinux 8 or Rocky Linux 8 x86_64 48
5.1.5 Installing EDB Postgres Advanced Server on RHEL 7 or OL 7 x86_64 51
5.1.6 Installing EDB Postgres Advanced Server on CentOS 7 x86_64 53
5.1.7 Installing EDB Postgres Advanced Server on SLES 15 x86_64 56
5.1.8 Installing EDB Postgres Advanced Server on SLES 12 x86_64 58
5.1.9 Installing EDB Postgres Advanced Server on Ubuntu 22.04 x86_64 61
5.1.10 Installing EDB Postgres Advanced Server on Ubuntu 20.04 x86_64 63
5.1.11 Installing EDB Postgres Advanced Server on Debian 11 x86_64 65
5.1.12 Installing EDB Postgres Advanced Server on Debian 10 x86_64 67
5.2 Installing EDB Postgres Advanced Server on Linux IBM Power (ppc64le) 69
5.2.1 Installing EDB Postgres Advanced Server on RHEL 9 ppc64le 70
5.2.2 Installing EDB Postgres Advanced Server on RHEL 8 ppc64le 73
5.2.3 Installing EDB Postgres Advanced Server on SLES 15 ppc64le 75
5.2.4 Installing EDB Postgres Advanced Server on SLES 12 ppc64le 78

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

5.2.5 Installing EDB Postgres Advanced Server on RHEL 7 ppc64le 80
5.3 Installing EDB Postgres Advanced Server on Windows 83
5.3.1 Requirements Overview 83
5.3.2 Installing Advanced Server with the Interactive Installer 84
5.3.2.1 Performing a Graphical Installation on Windows 85
5.3.2.2 Invoking the Graphical Installer from the Command Line 96
5.3.2.2.1 Performing an Unattended Installation 96
5.3.2.2.2 Performing an Installation with Limited Privileges 97
5.3.2.2.3 Reference - Command Line Options 101
5.3.2.3 Using StackBuilder Plus 105
5.4 Linux installation details 109
5.4.1 Default component locations 109
5.4.2 Available native packages 111
5.4.3 Managing an Advanced Server Installation 117
5.5 Windows installation details 126
5.5.1 Connecting to Advanced Server with the pgAdmin 4 Client 126
5.5.2 Managing an Advanced Server Installation 128
5.5.2.1 Using the Windows Services Applet 129
5.5.2.2 Using pg_ctl to Control Advanced Server 129
5.5.2.3 Controlling Server Startup Behavior on Windows 130
5.5.3 Configuring Advanced Server 131
5.5.3.1 Setting Advanced Server Environment Variables 131
5.5.3.2 Connecting to Advanced Server with psql 132
6 Upgrading EDB Postgres Advanced Server 133
6.1 Upgrade Considerations 133
6.2 Upgrading an Installation With pg_upgrade 133
6.2.1 Performing an Upgrade - Overview 134
6.2.1.1 Linking versus Copying 134
6.2.2 Invoking pg_upgrade 135
6.2.2.1 Command Line Options - Reference 136
6.2.3 Upgrading to Advanced Server 11 - Step-by-Step 138
6.2.4 Upgrading a pgAgent Installation 143
6.2.5 pg_upgrade Troubleshooting 144
6.2.6 Reverting to the Old Cluster 145
6.3 Performing a Minor Version Update of an RPM Installation 145
6.4 Using StackBuilder Plus to Perform a Minor Version Update 145
6.5 Migration to Version 11 150
7 Uninstalling EDB Postgres Advanced Server 152
7.1 Uninstalling Advanced Server on Linux 152
7.2 Uninstalling Advanced Server on Windows 155
8 Troubleshooting an installation 156
8.1 Linux installation troubleshooting 156
8.2 Windows installation troubleshooting 157
9 Database Compatibility for Oracle Developers Guide 157
9.1 Configuration Parameters Compatible with Oracle Databases 158
9.1.1 edb_redwood_date 158
9.1.2 edb_redwood_raw_names 159
9.1.3 edb_redwood_strings 160

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

9.1.4 edb_stmt_level_tx 161
9.1.5 oracle_home 162
9.2 About the Examples Used in this Guide 162
9.3 SQL Tutorial 163
9.3.1 Sample Database 163
9.3.1.1 Sample Database Installation 164
9.3.1.2 Sample Database Description 164
9.3.2 Creating a New Table 174
9.3.3 Populating a Table With Rows 174
9.3.4 Querying a Table 175
9.3.5 Joins Between Tables 177
9.3.6 Aggregate Functions 179
9.3.7 Updates 181
9.3.8 Deletions 181
9.3.9 The SQL Language 182
9.4 Advanced Concepts 182
9.4.1 Views 182
9.4.2 Foreign Keys 183
9.4.3 The ROWNUM Pseudo-Column 184
9.4.4 Synonyms 185
9.4.5 Hierarchical Queries 188
9.4.5.1 Defining the Parent/Child Relationship 188
9.4.5.2 Selecting the Root Nodes 189
9.4.5.3 Organization Tree in the Sample Application 189
9.4.5.4 Node Level 190
9.4.5.5 Ordering the Siblings 191
9.4.5.6 Retrieving the Root Node with CONNECT_BY_ROOT 192
9.4.5.7 Retrieving a Path with SYS_CONNECT_BY_PATH 196
9.4.6 Multidimensional Analysis 197
9.4.6.1 ROLLUP Extension 198
9.4.6.2 CUBE Extension 201
9.4.6.3 GROUPING SETS Extension 204
9.4.6.4 GROUPING Function 209
9.4.6.5 GROUPING_ID Function 211
9.5 Profile Management 213
9.5.1 Creating a New Profile 214
9.5.1.1 Creating a Password Function 216
9.5.2 Altering a Profile 218
9.5.3 Dropping a Profile 219
9.5.4 Associating a Profile with an Existing Role 220
9.5.5 Unlocking a Locked Account 221
9.5.6 Creating a New Role Associated with a Profile 222
9.5.7 Backing up Profile Management Functions 224
9.6 Optimizer Hints 224
9.6.1 Default Optimization Modes 225
9.6.2 Access Method Hints 227
9.6.3 Specifying a Join Order 230
9.6.4 Joining Relations Hints 231

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

9.6.5 Global Hints 233
9.6.6 Using the APPEND Optimizer Hint 235
9.6.7 Parallelism Hints 236
9.6.8 Conflicting Hints 239
9.7 dblink_ora 240
9.7.1 dblink_ora Functions and Procedures 241
9.7.1.1 dblink_ora_connect() 241
9.7.1.2 dblink_ora_status() 242
9.7.1.3 dblink_ora_disconnect() 242
9.7.1.4 dblink_ora_record() 242
9.7.1.5 dblink_ora_call() 243
9.7.1.6 dblink_ora_exec() 243
9.7.1.7 dblink_ora_copy() 243
9.7.2 Calling dblink_ora Functions 244
9.8 Open Client Library 245
9.9 Oracle Catalog Views 245
9.10 Tools and Utilities 245
9.11 ECPGPlus 246
9.12 System Catalog Tables 246
10 Database Compatibility for Oracle Developers Reference Guide 246
10.1 The SQL Language 247
10.1.1 SQL Syntax 247
10.1.1.1 Lexical Structure 247
10.1.1.2 Identifiers and Key Words 248
10.1.1.3 Constants 248
10.1.1.4 Comments 250
10.1.2 Data Types 250
10.1.2.1 Numeric Types 250
10.1.2.2 Character Types 252
10.1.2.3 Binary Data 253
10.1.2.4 Date/Time Types 254
10.1.2.5 Boolean Types 257
10.1.2.6 XML Type 258
10.1.3 Functions and Operators 258
10.1.3.1 Logical Operators 258
10.1.3.2 Comparison Operators 259
10.1.3.3 Mathematical Functions and Operators 260
10.1.3.4 String Functions and Operators 262
10.1.3.5 Pattern Matching String Functions 268
10.1.3.6 Pattern Matching Using the LIKE Operator 272
10.1.3.7 Data Type Formatting Functions 273
10.1.3.8 Date/Time Functions and Operators 278
10.1.3.9 Sequence Manipulation Functions 292
10.1.3.10 Conditional Expressions 292
10.1.3.11 Aggregate Functions 296
10.1.3.12 Subquery Expressions 297
10.2 System Catalog Tables 299
11 Database Compatibility for Oracle Developers Built-in Package Guide 303

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.1 Packages 303
11.1.1 Package Components 303
11.1.2 Creating Packages 310
11.1.3 Referencing a Package 312
11.1.4 Using Packages With User Defined Types 313
11.1.5 Dropping a Package 316
11.2 Built-In Packages 316
11.2.1 DBMS_ALERT 316
11.2.2 DBMS_AQ 323
11.2.2.1 ENQUEUE 325
11.2.2.2 DEQUEUE 327
11.2.2.3 REGISTER 330
11.2.2.4 UNREGISTER 331
11.2.3 DBMS_AQADM 332
11.2.3.1 ALTER_QUEUE 334
11.2.3.2 ALTER_QUEUE_TABLE 334
11.2.3.3 CREATE_QUEUE 335
11.2.3.4 CREATE_QUEUE_TABLE 336
11.2.3.5 DROP_QUEUE 339
11.2.3.6 DROP_QUEUE_TABLE 339
11.2.3.7 PURGE_QUEUE_TABLE 340
11.2.3.8 START_QUEUE 341
11.2.3.9 STOP_QUEUE 341
11.2.4 DBMS_CRYPTO 342
11.2.4.1 DECRYPT 343
11.2.4.2 ENCRYPT 345
11.2.4.3 HASH 346
11.2.4.4 MAC 347
11.2.4.5 RANDOMBYTES 348
11.2.4.6 RANDOMINTEGER 349
11.2.4.7 RANDOMNUMBER 349
11.2.5 DBMS_JOB 349
11.2.5.1 BROKEN 351
11.2.5.2 CHANGE 351
11.2.5.3 INTERVAL 352
11.2.5.4 NEXT_DATE 353
11.2.5.5 REMOVE 353
11.2.5.6 RUN 354
11.2.5.7 SUBMIT 354
11.2.5.8 WHAT 355
11.2.6 DBMS_LOB 356
11.2.6.1 APPEND 357
11.2.6.2 COMPARE 357
11.2.6.3 CONVERTTOBLOB 358
11.2.6.4 CONVERTTOCLOB 359
11.2.6.5 COPY 360
11.2.6.6 ERASE 361
11.2.6.7 GET_STORAGE_LIMIT 361

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.6.8 GETLENGTH 362
11.2.6.9 INSTR 362
11.2.6.10 READ 363
11.2.6.11 SUBSTR 363
11.2.6.12 TRIM 364
11.2.6.13 WRITE 364
11.2.6.14 WRITEAPPEND 365
11.2.7 DBMS_LOCK 365
11.2.8 DBMS_MVIEW 366
11.2.8.1 GET_MV_DEPENDENCIES 366
11.2.8.2 REFRESH 367
11.2.8.3 REFRESH_ALL_MVIEWS 369
11.2.8.4 REFRESH_DEPENDENT 370
11.2.9 DBMS_OUTPUT 371
11.2.10 DBMS_PIPE 379
11.2.10.1 CREATE_PIPE 380
11.2.10.2 NEXT_ITEM_TYPE 380
11.2.10.3 PACK_MESSAGE 382
11.2.10.4 PURGE 383
11.2.10.5 RECEIVE_MESSAGE 384
11.2.10.6 REMOVE_PIPE 385
11.2.10.7 RESET_BUFFER 386
11.2.10.8 SEND_MESSAGE 387
11.2.10.9 UNIQUE_SESSION_NAME 387
11.2.10.10 UNPACK_MESSAGE 388
11.2.10.11 Comprehensive Example 388
11.2.11 DBMS_PROFILER 391
11.2.12 DBMS_RANDOM 410
11.2.13 DBMS_REDACT 414
11.2.14 DBMS_RLS 431
11.2.15 DBMS_SCHEDULER 441
11.2.15.1 Using Calendar Syntax to Specify a Repeating Interval 442
11.2.15.2 CREATE_JOB 443
11.2.15.3 CREATE_PROGRAM 446
11.2.15.4 CREATE_SCHEDULE 447
11.2.15.5 DEFINE_PROGRAM_ARGUMENT 448
11.2.15.6 DISABLE 449
11.2.15.7 DROP_JOB 450
11.2.15.8 DROP_PROGRAM 451
11.2.15.9 DROP_PROGRAM_ARGUMENT 451
11.2.15.10 DROP_SCHEDULE 452
11.2.15.11 ENABLE 453
11.2.15.12 EVALUATE_CALENDAR_STRING 453
11.2.15.13 RUN_JOB 454
11.2.15.14 SET_JOB_ARGUMENT_VALUE 455
11.2.16 DBMS_SESSION 456
11.2.17 DBMS_SQL 456
11.2.17.1 BIND_VARIABLE 458

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.17.2 BIND_VARIABLE_CHAR 459
11.2.17.3 BIND_VARIABLE_RAW 459
11.2.17.4 CLOSE_CURSOR 460
11.2.17.5 COLUMN_VALUE 460
11.2.17.6 COLUMN_VALUE_CHAR 462
11.2.17.7 COLUMN_VALUE_RAW 462
11.2.17.8 DEFINE_COLUMN 463
11.2.17.9 DEFINE_COLUMN_CHAR 464
11.2.17.10 DEFINE_COLUMN_RAW 465
11.2.17.11 DESCRIBE_COLUMNS 465
11.2.17.12 EXECUTE 466
11.2.17.13 EXECUTE_AND_FETCH 467
11.2.17.14 FETCH_ROWS 468
11.2.17.15 IS_OPEN 470
11.2.17.16 LAST_ROW_COUNT 470
11.2.17.17 OPEN_CURSOR 471
11.2.17.18 PARSE 472
11.2.18 DBMS_UTILITY 473
11.2.19 UTL_ENCODE 487
11.2.19.1 BASE64_DECODE 488
11.2.19.2 BASE64_ENCODE 489
11.2.19.3 MIMEHEADER_DECODE 490
11.2.19.4 MIMEHEADER_ENCODE 490
11.2.19.5 QUOTED_PRINTABLE_DECODE 491
11.2.19.6 QUOTED_PRINTABLE_ENCODE 492
11.2.19.7 TEXT_DECODE 493
11.2.19.8 TEXT_ENCODE 494
11.2.19.9 UUDECODE 494
11.2.19.10 UUENCODE 496
11.2.20 UTL_FILE 497
11.2.21 UTL_HTTP 512
11.2.22 UTL_MAIL 530
11.2.23 UTL_RAW 534
11.2.24 UTL_SMTP 539
11.2.25 UTL_URL 548
12 Database Compatibility for Oracle Developers Tools and Utilities Guide 550
12.1 EDB*Loader 551
12.2 EDB*Wrap 579
12.3 Dynamic Runtime Instrumentation Tools Architecture (DRITA) 583
13 Database Compatibility for Oracle Developers Catalog Views Guide 611
13.1 ALL_ALL_TABLES 611
13.2 ALL_CONS_COLUMNS 611
13.3 ALL_CONSTRAINTS 612
13.4 ALL_DB_LINKS 612
13.5 ALL_DIRECTORIES 613
13.6 ALL_IND_COLUMNS 613
13.7 ALL_INDEXES 613
13.8 ALL_JOBS 614

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.9 ALL_OBJECTS 614
13.10 ALL_PART_KEY_COLUMNS 615
13.11 ALL_PART_TABLES 615
13.12 ALL_POLICIES 616
13.13 ALL_QUEUES 617
13.14 ALL_QUEUE_TABLES 618
13.15 ALL_SEQUENCES 618
13.16 ALL_SOURCE 619
13.17 ALL_SUBPART_KEY_COLUMNS 619
13.18 ALL_SYNONYMS 619
13.19 ALL_TAB_COLUMNS 620
13.20 ALL_TAB_PARTITIONS 620
13.21 ALL_TAB_SUBPARTITIONS 621
13.22 ALL_TABLES 622
13.23 ALL_TRIGGERS 622
13.24 ALL_TYPES 623
13.25 ALL_USERS 623
13.26 ALL_VIEW_COLUMNS 624
13.27 ALL_VIEWS 624
13.28 DBA_ALL_TABLES 625
13.29 DBA_CONS_COLUMNS 625
13.30 DBA_CONSTRAINTS 625
13.31 DBA_DB_LINKS 626
13.32 DBA_DIRECTORIES 626
13.33 DBA_IND_COLUMNS 627
13.34 DBA_INDEXES 627
13.35 DBA_JOBS 628
13.36 DBA_OBJECTS 628
13.37 DBA_PART_KEY_COLUMNS 629
13.38 DBA_PART_TABLES 629
13.39 DBA_POLICIES 630
13.40 DBA_PROFILES 631
13.41 DBA_QUEUES 631
13.42 DBA_QUEUE_TABLES 632
13.43 DBA_ROLE_PRIVS 632
13.44 DBA_ROLES 633
13.45 DBA_SEQUENCES 633
13.46 DBA_SOURCE 633
13.47 DBA_SUBPART_KEY_COLUMNS 634
13.48 DBA_SYNONYMS 634
13.49 DBA_TAB_COLUMNS 634
13.50 DBA_TAB_PARTITIONS 635
13.51 DBA_TAB_SUBPARTITIONS 636
13.52 DBA_TABLES 637
13.53 DBA_TRIGGERS 637
13.54 DBA_TYPES 638
13.55 DBA_USERS 638
13.56 DBA_VIEW_COLUMNS 639

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.57 DBA_VIEWS 640
13.58 USER_ALL_TABLES 640
13.59 USER_CONS_COLUMNS 640
13.60 USER_CONSTRAINTS 641
13.61 USER_DB_LINKS 642
13.62 USER_IND_COLUMNS 642
13.63 USER_INDEXES 642
13.64 USER_JOBS 643
13.65 USER_OBJECTS 643
13.66 USER_PART_KEY_COLUMNS 644
13.67 USER_PART_TABLES 644
13.68 USER_POLICIES 645
13.69 USER_QUEUES 646
13.70 USER_QUEUE_TABLES 646
13.71 USER_ROLE_PRIVS 647
13.72 USER_SEQUENCES 647
13.73 USER_SOURCE 648
13.74 USER_SUBPART_KEY_COLUMNS 648
13.75 USER_SYNONYMS 648
13.76 USER_TAB_COLUMNS 649
13.77 USER_TAB_PARTITIONS 649
13.78 USER_TAB_SUBPARTITIONS 650
13.79 USER_TABLES 651
13.80 USER_TRIGGERS 652
13.81 USER_TYPES 652
13.82 USER_USERS 652
13.83 USER_VIEW_COLUMNS 653
13.84 USER_VIEWS 654
13.85 V$VERSION 654
13.86 PRODUCT_COMPONENT_VERSION 654
14 Database Compatibility Stored Procedural Language Guide 655
14.1 Basic SPL Elements 655
14.1.1 Case Sensitivity 655
14.1.2 Identifiers 655
14.1.3 Qualifiers 656
14.1.4 Constants 656
14.1.5 User-Defined PL/SQL Subtypes 656
14.1.6 Character Set 658
14.2 SPL Programs 659
14.2.1 SPL Block Structure 659
14.2.2 Anonymous Blocks 661
14.2.3 Procedures Overview 661
14.2.3.1 Creating a Procedure 661
14.2.3.2 Calling a Procedure 664
14.2.3.3 Deleting a Procedure 665
14.2.4 Functions Overview 666
14.2.4.1 Creating a Function 666
14.2.4.2 Calling a Function 669

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.2.4.3 Deleting a Function 670
14.2.5 Procedure and Function Parameters 671
14.2.5.1 Positional vs. Named Parameter Notation 672
14.2.5.2 Parameter Modes 674
14.2.5.3 Using Default Values in Parameters 675
14.2.6 Subprograms – Subprocedures and Subfunctions 676
14.2.6.1 Creating a Subprocedure 676
14.2.6.2 Creating a Subfunction 678
14.2.6.3 Block Relationships 680
14.2.6.4 Invoking Subprograms 681
14.2.6.5 Using Forward Declarations 687
14.2.6.6 Overloading Subprograms 688
14.2.6.7 Accessing Subprogram Variables 691
14.2.7 Compilation Errors in Procedures and Functions 698
14.2.8 Program Security 699
14.2.8.1 EXECUTE Privilege 699
14.2.8.2 Database Object Name Resolution 700
14.2.8.3 Database Object Privileges 700
14.2.8.4 Definer’s vs. Invokers Rights 700
14.2.8.5 Security Example 701
14.3 Variable Declarations 706
14.3.1 Declaring a Variable 707
14.3.2 Using %TYPE in Variable Declarations 707
14.3.3 Using %ROWTYPE in Record Declarations 709
14.3.4 User-Defined Record Types and Record Variables 710
14.4 Basic Statements 712
14.4.1 Assignment 712
14.4.2 DELETE 713
14.4.3 INSERT 714
14.4.4 NULL 715
14.4.5 Using the RETURNING INTO Clause 716
14.4.6 SELECT INTO 718
14.4.7 UPDATE 719
14.4.8 Obtaining the Result Status 720
14.5 Control Structures 721
14.5.1 IF Statement 721
14.5.1.1 IF-THEN 721
14.5.1.2 IF-THEN-ELSE 722
14.5.1.3 IF-THEN-ELSE IF 723
14.5.1.4 IF-THEN-ELSIF-ELSE 725
14.5.2 RETURN Statement 726
14.5.3 GOTO Statement 726
14.5.4 CASE Expression 728
14.5.4.1 Selector CASE Expression 728
14.5.4.2 Searched CASE Expression 729
14.5.5 CASE Statement 730
14.5.5.1 Selector CASE Statement 730
14.5.5.2 Searched CASE Statement 732

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.5.6 Loops 733
14.5.6.1 LOOP 733
14.5.6.2 EXIT 733
14.5.6.3 CONTINUE 734
14.5.6.4 WHILE 735
14.5.6.5 FOR (integer variant) 736
14.5.7 Exception Handling 736
14.5.8 User-defined Exceptions 738
14.5.9 PRAGMA EXCEPTION_INIT 739
14.5.10 RAISE_APPLICATION_ERROR 741
14.6 Transaction Control 742
14.6.1 COMMIT 742
14.6.2 ROLLBACK 744
14.6.3 PRAGMA AUTONOMOUS_TRANSACTION 746
14.7 Dynamic SQL 753
14.8 Static Cursors 755
14.8.1 Declaring a Cursor 755
14.8.2 Opening a Cursor 755
14.8.3 Fetching Rows From a Cursor 756
14.8.4 Closing a Cursor 757
14.8.5 Using %ROWTYPE With Cursors 758
14.8.6 Cursor Attributes 758
14.8.6.1 %ISOPEN 759
14.8.6.2 %FOUND 759
14.8.6.3 %NOTFOUND 760
14.8.6.4 %ROWCOUNT 761
14.8.6.5 Summary of Cursor States and Attributes 762
14.8.7 Cursor FOR Loop 763
14.8.8 Parameterized Cursors 764
14.9 REF CURSORs and Cursor Variables 764
14.9.1 REF CURSOR Overview 764
14.9.2 Declaring a Cursor Variable 765
14.9.2.1 Declaring a SYS_REFCURSOR Cursor Variable 765
14.9.2.2 Declaring a User Defined REF CURSOR Type Variable 765
14.9.3 Opening a Cursor Variable 766
14.9.4 Fetching Rows From a Cursor Variable 766
14.9.5 Closing a Cursor Variable 766
14.9.6 Usage Restrictions 767
14.9.7 Examples 768
14.9.7.1 Returning a REF CURSOR From a Function 768
14.9.7.2 Modularizing Cursor Operations 769
14.9.8 Dynamic Queries With REF CURSORs 771
14.10 Collections 773
14.10.1 Associative Arrays 774
14.10.2 Nested Tables 777
14.10.3 Varrays 780
14.11 Collection Methods 782
14.11.1 COUNT 782

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.11.2 DELETE 782
14.11.3 EXISTS 783
14.11.4 EXTEND 784
14.11.5 FIRST 786
14.11.6 LAST 787
14.11.7 LIMIT 787
14.11.8 NEXT 787
14.11.9 PRIOR 788
14.11.10 TRIM 788
14.12 Working with Collections 789
14.12.1 TABLE() 790
14.12.2 Using the MULTISET UNION Operator 790
14.12.3 Using the FORALL Statement 792
14.12.4 Using the BULK COLLECT Clause 794
14.12.4.1 SELECT BULK COLLECT 794
14.12.4.2 FETCH BULK COLLECT 796
14.12.4.3 EXECUTE IMMEDIATE BULK COLLECT 796
14.12.4.4 RETURNING BULK COLLECT 797
14.12.5 Errors and Messages 799
14.13 Triggers 799
14.13.1 Overview 800
14.13.2 Types of Triggers 800
14.13.3 Creating Triggers 800
14.13.4 Trigger Variables 802
14.13.5 Transactions and Exceptions 803
14.13.6 Trigger Examples 803
14.13.6.1 Before Statement-Level Trigger 803
14.13.6.2 After Statement-Level Trigger 804
14.13.6.3 Before Row-Level Trigger 805
14.13.6.4 After Row-Level Trigger 805
14.14 Packages 807
14.15 Object Types and Objects 808
14.15.1 Basic Object Concepts 808
14.15.1.1 Attributes 808
14.15.1.2 Methods 809
14.15.1.3 Overloading Methods 809
14.15.2 Object Type Components 809
14.15.2.1 Object Type Specification Syntax 809
14.15.2.2 Object Type Body Syntax 812
14.15.3 Creating Object Types 813
14.15.3.1 Member Methods 814
14.15.3.2 Static Methods 815
14.15.3.3 Constructor Methods 816
14.15.4 Creating Object Instances 817
14.15.5 Referencing an Object 818
14.15.6 Dropping an Object Type 820
15 Database Compatibility for Oracle Developers SQL Guide 820
15.1 ALTER INDEX 821

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

15.2 ALTER PROCEDURE 822
15.3 ALTER PROFILE 823
15.4 ALTER QUEUE 825
15.5 ALTER QUEUE TABLE 828
15.6 ALTER ROLE... IDENTIFIED BY 828
15.7 ALTER ROLE - Managing Database Link and DBMS_RLS Privileges 829
15.8 ALTER SEQUENCE 831
15.9 ALTER SESSION 832
15.10 ALTER SYNONYM 834
15.11 ALTER TABLE 835
15.12 ALTER TABLESPACE 837
15.13 ALTER USER... IDENTIFIED BY 838
15.14 ALTER USER|ROLE... PROFILE MANAGEMENT CLAUSES 839
15.15 CALL 841
15.16 COMMENT 841
15.17 COMMIT 842
15.18 CREATE DATABASE 843
15.19 CREATE PUBLIC DATABASE LINK 844
15.20 CREATE DIRECTORY 854
15.21 CREATE FUNCTION 856
15.22 CREATE INDEX 861
15.23 CREATE MATERIALIZED VIEW 862
15.24 CREATE PACKAGE 864
15.25 CREATE PACKAGE BODY 866
15.26 CREATE PROCEDURE 869
15.27 CREATE PROFILE 875
15.28 CREATE QUEUE 877
15.29 CREATE QUEUE TABLE 879
15.30 CREATE ROLE 881
15.31 CREATE SCHEMA 882
15.32 CREATE SEQUENCE 883
15.33 CREATE SYNONYM 885
15.34 CREATE TABLE 886
15.35 CREATE TABLE AS 892
15.36 CREATE TRIGGER 893
15.37 CREATE TYPE 895
15.38 CREATE TYPE BODY 902
15.39 CREATE USER 904
15.40 CREATE USER|ROLE... PROFILE MANAGEMENT CLAUSES 905
15.41 CREATE VIEW 907
15.42 DELETE 908
15.43 DROP DATABASE LINK 909
15.44 DROP DIRECTORY 910
15.45 DROP FUNCTION 911
15.46 DROP INDEX 912
15.47 DROP PACKAGE 913
15.48 DROP PROCEDURE 913
15.49 DROP PROFILE 914

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

15.50 DROP QUEUE 915
15.51 DROP QUEUE TABLE 916
15.52 DROP SYNONYM 917
15.53 DROP ROLE 918
15.54 DROP SEQUENCE 919
15.55 DROP TABLE 919
15.56 DROP TABLESPACE 920
15.57 DROP TRIGGER 921
15.58 DROP TYPE 922
15.59 DROP USER 922
15.60 DROP VIEW 923
15.61 EXEC 924
15.62 GRANT 925
15.63 INSERT 930
15.64 LOCK 932
15.65 REVOKE 933
15.66 ROLLBACK 936
15.67 ROLLBACK TO SAVEPOINT 937
15.68 SAVEPOINT 937
15.69 SELECT 938
15.70 SET CONSTRAINTS 947
15.71 SET ROLE 948
15.72 SET TRANSACTION 949
15.73 TRUNCATE 950
15.74 UPDATE 950
16 Database Compatibility Table Partitioning Guide 952
16.1 Selecting a Partition Type 953
16.2 Using Partition Pruning 954
16.2.1 Example - Partition Pruning 956
16.3 Partitioning Commands Compatible with Oracle Databases 958
16.3.1 CREATE TABLE...PARTITION BY 958
16.3.1.1 Example - PARTITION BY LIST 961
16.3.1.2 Example - PARTITION BY RANGE 962
16.3.1.3 Example - PARTITION BY HASH 963
16.3.1.4 Example - PARTITION BY RANGE, SUBPARTITION BY LIST 963
16.3.2 ALTER TABLE...ADD PARTITION 965
16.3.2.1 Example - Adding a Partition to a LIST Partitioned Table 966
16.3.2.2 Example - Adding a Partition to a RANGE Partitioned Table 967
16.3.3 ALTER TABLE...ADD SUBPARTITION 968
16.3.3.1 Example - Adding a Subpartition to a LIST/RANGE Partitioned Table 969
16.3.3.2 Example - Adding a Subpartition to a RANGE/LIST Partitioned Table 971
16.3.4 ALTER TABLE...SPLIT PARTITION 972
16.3.4.1 Example - Splitting a LIST Partition 973
16.3.4.2 Example - Splitting a RANGE Partition 975
16.3.5 ALTER TABLE...SPLIT SUBPARTITION 977
16.3.5.1 Example - Splitting a LIST Subpartition 978
16.3.5.2 Example - Splitting a RANGE Subpartition 980
16.3.6 ALTER TABLE...EXCHANGE PARTITION 983

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

16.3.6.1 Example - Exchanging a Table for a Partition 984
16.3.7 ALTER TABLE...MOVE PARTITION 986
16.3.7.1 Example - Moving a Partition to a Different Tablespace 987
16.3.8 ALTER TABLE...RENAME PARTITION 988
16.3.8.1 Example - Renaming a Partition 989
16.3.9 ALTER TABLE...DROP PARTITION 990
16.3.9.1 Example - Deleting a Partition 990
16.3.10 ALTER TABLE...DROP SUBPARTITION 991
16.3.10.1 Example - Deleting a Subpartition 991
16.3.11 TRUNCATE TABLE 992
16.3.11.1 Example - Emptying a Table 993
16.3.12 ALTER TABLE...TRUNCATE PARTITION 994
16.3.12.1 Example - Emptying a Partition 995
16.3.13 ALTER TABLE...TRUNCATE SUBPARTITION 996
16.3.13.1 Example - Emptying a Subpartition 997
16.3.14 DROP TABLE 999
16.4 Handling Stray Values in a LIST or RANGE Partitioned Table 1000
16.5 Specifying Multiple Partitioning Keys in a RANGE Partitioned Table 1004
16.6 Retrieving Information about a Partitioned Table 1005
16.6.1 Table Partitioning Views - Reference 1006
16.6.1.1 ALL_PART_TABLES 1006
16.6.1.2 ALL_TAB_PARTITIONS 1007
16.6.1.3 ALL_TAB_SUBPARTITIONS 1008
16.6.1.4 ALL_PART_KEY_COLUMNS 1009
16.6.1.5 ALL_SUBPART_KEY_COLUMNS 1009
17 ECPGPlus Guide 1010
17.1 ECPGPlus - Overview 1010
17.2 Using Embedded SQL 1016
17.3 Using Descriptors 1022
17.4 Building and Executing Dynamic SQL Statements 1031
17.5 Error Handling 1045
17.6 Reference 1049
18 EDB Postgres Language Pack Guide 1079
18.1 Supported Database Server Versions 1079
18.2 Installing and Configuring Language Pack 1080
18.3 Using the Procedural Languages 1084
18.4 Uninstalling Language Pack 1087
19 EDB Postgres Advanced Server Security Features Guide 1087
19.1 Protecting Against SQL Injection Attacks 1087
19.1.1 SQL/Protect Overview 1087
19.1.2 Configuring SQL/Protect 1090
19.1.3 Common Maintenance Operations 1098
19.1.4 Backing Up and Restoring a SQL/Protect Database 1102
19.2 Virtual Private Database 1107
19.3 sslutils 1107
19.4 Data Redaction 1109
20 EDB Postgres Advanced Server Guide 1118
20.1 Introduction 1118

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

20.1.1 Conventions Used in this Guide 1118
20.1.2 About the Examples Used in this Guide 1119
20.2 Enhanced Compatibility Features 1127
20.3 Database Administration 1132
20.3.1 Configuration Parameters 1133
20.3.1.1 Setting Configuration Parameters 1133
20.3.1.2 Summary of Configuration Parameters 1135
20.3.1.3 Configuration Parameters by Functionality 1148
20.3.1.3.1 Top Performance Related Parameters 1148
20.3.1.3.1.1 shared_buffers 1148
20.3.1.3.1.2 work_mem 1149
20.3.1.3.1.3 maintenance_work_mem 1149
20.3.1.3.1.4 wal_buffers 1150
20.3.1.3.1.5 checkpoint_segments 1150
20.3.1.3.1.6 checkpoint_completion_target 1150
20.3.1.3.1.7 checkpoint_timeout 1151
20.3.1.3.1.8 max_wal_size 1151
20.3.1.3.1.9 min_wal_size 1152
20.3.1.3.1.10 bgwriter_delay 1152
20.3.1.3.1.11 seq_page_cost 1152
20.3.1.3.1.12 random_page_cost 1153
20.3.1.3.1.13 effective_cache_size 1153
20.3.1.3.1.14 synchronous_commit 1154
20.3.1.3.1.15 edb_max_spins_per_delay 1154
20.3.1.3.1.16 pg_prewarm.autoprewarm 1155
20.3.1.3.1.17 pg_prewarm.autoprewarm_interval 1155
20.3.1.3.2 Resource Usage / Memory 1156
20.3.1.3.3 Resource Usage / EDB Resource Manager 1157
20.3.1.3.4 Query Tuning 1158
20.3.1.3.5 Query Tuning / Planner Method Configuration 1158
20.3.1.3.6 Reporting and Logging / What to Log 1160
20.3.1.3.7 Auditing Settings 1161
20.3.1.3.7.1 edb_audit 1161
20.3.1.3.7.2 edb_audit_directory 1162
20.3.1.3.7.3 edb_audit_filename 1162
20.3.1.3.7.4 edb_audit_rotation_day 1162
20.3.1.3.7.5 edb_audit_rotation_size 1163
20.3.1.3.7.6 edb_audit_rotation_seconds 1163
20.3.1.3.7.7 edb_audit_connect 1163
20.3.1.3.7.8 edb_audit_disconnect 1164
20.3.1.3.7.9 edb_audit_statement 1164
20.3.1.3.7.10 edb_audit_tag 1165
20.3.1.3.7.11 edb_audit_destination 1165
20.3.1.3.7.12 edb_log_every_bulk_value 1165
20.3.1.3.8 Client Connection Defaults / Locale and Formatting 1166
20.3.1.3.9 Client Connection Defaults / Statement Behavior 1166
20.3.1.3.10 Client Connection Defaults / Other Defaults 1167
20.3.1.3.11 Compatibility Options 1168

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

20.3.1.3.12 Customized Options 1175
20.3.1.3.13 Ungrouped 1183
20.3.2 Index Advisor 1185
20.3.2.1 Index Advisor Components 1186
20.3.2.2 Index Advisor Configuration 1187
20.3.2.3 Using Index Advisor 1188
20.3.2.4 Reviewing the Index Advisor Recommendations 1191
20.3.2.5 Index Advisor Limitations 1195
20.3.3 SQL Profiler 1195
20.3.4 pgsnmpd 1196
20.3.5 EDB Audit Logging 1198
20.3.5.1 Audit Logging Configuration Parameters 1199
20.3.5.2 Selecting SQL Statements to Audit 1200
20.3.5.3 Enabling Audit Logging 1212
20.3.5.4 Audit Log File 1216
20.3.5.5 Using Error Codes to Filter Audit Logs 1220
20.3.5.6 Using Command Tags to Filter Audit Logs 1221
20.3.5.7 Redacting Passwords from Audit Logs 1222
20.3.6 Unicode Collation Algorithm 1222
20.4 EDB Resource Manager 1232
20.5 libpq C Library 1251
20.6 Debugger 1259
20.7 Performance Analysis and Tuning 1275
20.8 EDB Clone Schema 1284
20.9 PL/Java 1303
Installing PL/Java on Linux 1304
Installing PL/Java on Windows 1304
Using PL/Java 1305
20.10 Enhanced SQL and Other Miscellaneous Features 1307
20.11 Advanced Server Keywords 1312
20.12 System Catalog Tables 1312

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

1 EDB Postgres Advanced Server

With EDB Postgres Advanced Server, EnterpriseDB leads as the only worldwide company to deliver innovative and low cost open-source-derived database solutions with commercial quality, ease of use, compatibility, scalability, and performance for small or large-scale enterprises.

2 EDB Postgres Advanced Server Release Notes

EDB Postgres Advanced Server 11 is built on the open source PostgreSQL 11. EDB Postgres Advanced Server 11 adds a number of new features including:

- Data redaction
- Autonomous transactions
- Performance diagnostics
- Various performance improvements to OCI dblink

The EDB Postgres Advanced Server (Advanced Server) documentation describes the latest version of Advanced Server 11 including minor releases and patches. The release notes in this section provide information on what was new in each release.

Version	Release Date	Upstream Merge
11.22.34	09 Nov 2023	11.22
11.21.33	25 Sep 2023	
11.21.32	21 Aug 2023	11.21
11.20.31	11 May 2023	11.20
11.19.30	10 Feb 2023	11.19
11.18.29	10 Nov 2022	11.18
11.17.28	11 Aug 2022	11.17
11.16.26	12 May 2022	11.16
11.15.25	10 Feb 2022	11.15
11.14.24	11 Nov 2021	11.14
11.13.23	08 Sep 2021	11.13
11.12.22	05 May 2021	11.12
11.12.21	15 Apr 2021	11.12
11.11.20	12 Feb 2021	11.11
11.10.19	20 Nov 2020	11.10
11.9.17	18 Aug 2020	11.9
11.9.16	17 Aug 2020	11.9
11.8.15	18 May 2020	11.8
11.7.14	14 Feb 2020	11.7
11.6.13	19 Nov 2019	11.6
11.5.12	26 Aug 2019	11.5
11.4.11	25 Jun 2019	11.4
11.3.10	13 May 2019	11.3
11.2.9	22 Feb 2019	11.2
11.1.7	28 Nov 2018	11.1

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

2.1 EDB Postgres Advanced Server 11.22.34 release notes

Released: 09 Nov 2023

EDB Postgres Advanced Server 11.22.34 includes the following enhancements and bug fixes:

Type	Description	Addresses
Bug fix	Fixed a high memory usage issue due to the re-transformation of a DECODE expression.	
Bug fix	Fixed split partition failure issue due to case-sensitive partition name in `edb_partition`.	#97109

2.2 EDB Postgres Advanced Server 11.21.33 release notes

Released: 22 Sep 2023

EDB Postgres Advanced Server 11.21.33 includes the following enhancements and bug fixes:

Type	Description	Addresses
Bug fix	Hexadecimal values are now allowed as record and field delimiter in EDB*Loader.	#91832
Bug fix	Fixed memory leak in sub-transaction with usage of EXCEPTION in package.	#94255
Bug fix	Fixed the cache lookup error for event triggers with proper initialization of the variables in SPL.	

2.3 Version 11.21.32

Released: 21 Aug 2023

Updated: 30 Aug 2023

> **Upgrading**
>
> After you upgrade to this version of EDB Postgres Advanced Server, you need to run `edb_sqlpatch` on all your databases to complete the upgrade. This application checks that your databases system objects are up to date with this version. See the EDB SQL Patch documentation for more information on how to deploy this tool.

> **After applying patches**
>
> Users making use of the UTL_MAIL package now require EXECUTE permission on the UTL_SMTP and UTL_TCP packages in addition to EXECUTE permission on UTL_MAIL.
>
> Users making use of the UTL_SMTP package now require EXECUTE permission on the UTL_TCP packages in addition to EXECUTE permission on UTL_SMTP.

EDB Postgres Advanced Server 11.21.32 includes the following enhancements and bug fixes:

Type	Description	Addresses

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Description	Addresses
Security fix	EDB Postgres Advanced Server (EPAS) SECURITY DEFINER functions and procedures may be hijacked via search_path.	CVE-2023-41117
Security fix	EDB Postgres Advanced Server (EPAS) dbms_aq helper function may run arbitrary SQL as a superuser.	CVE-2023-41119
Security fix	EDB Postgres Advanced Server (EPAS) permissions bypass via accesshistory()	CVE-2023-41113
Security fix	EDB Postgres Advanced Server (EPAS) UTL_FILE permission bypass	CVE-2023-41118
Security fix	EDB Postgres Advanced Server (EPAS) permission bypass for materialized views	CVE-2023-41116
Security fix	EDB Postgres Advanced Server (EPAS) authenticated users may fetch any URL	CVE-2023-41114
Security fix	EDB Postgres Advanced Server (EPAS) permission bypass for large objects	CVE-2023-41115
Security fix	EDB Postgres Advanced Server (EPAS) DBMS_PROFILER data may be removed without permission	CVE-2023-41120
Bug fix	Allowed subtypes in INDEX BY clause of the packaged collection.	#1371
Bug fix	Fixed %type resolution when pointing to a packaged type field.	#1243

> **Addresses**
>
> Entries in the Addresses column are either CVE numbers or, if preceded by #, a customer case number.

2.4 Version 11.20.31

EDB Postgres Advanced Server 11.20.31 includes the following enhancements:

Type	Description	Category
Upstream merge	Merged with community PostgreSQL 11.20.31. See the community Release Notes for details.	
Enhancement	SQL Profiler and Index Advisor are now extensions and can be downloaded from EDB Repos.	
Bug fix	Fixed an issue in which "PASSWORD EXPIRE AT" was dumped when the password status wasn't expired. This fix prevents marking the user account as expired after an upgrade.	Profile
Bug fix	Fixed the password profile behavior after the password grace time has changed.	
Bug fix	Fixed unexpected error for `edb_enable_pruning` parameter. [Support ticket: #89863]	
Bug fix	Fixed an issue when a user enters `Ctrl-c` (SIGINT) to cancel the load in EDB*Loader. [Support ticket: #88734]	
Bug fix	Set correct object descriptions for redaction policy to make pg_dump work cleanly with `--clean` and `--if-exists` options.	
Bug fix	Fixed pg_dump to dump password verify function for the user profile.	
Bug fix	Fixed assertion failure while terminating the process within the autonomous transaction.	
Bug fix	Fixed corner-case uninitialized-variable issues in SPL.	

2.5 Version 11.19.30

EDB Postgres Advanced Server 11.19.30 includes the following enhancements:

Type	Description
Upstream merge	Merged with community PostgreSQL 11.19.30. See the community Release Notes for details.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Description
Bug fix	Fix possible server crash with autonomous transaction and temp table. [Support tickets: #89602 and #82695]
Bug fix	Allow the password expiry feature to work with SCRAM authentication. [Support ticket: #89340]
Bug fix	Fix Bulk API for update on an invisible tuple. [Support ticket: #86900]

2.6 Version 11.18.29

EDB Postgres Advanced Server 11.18.29 includes the following enhancements:

Type	Description	ID	Category
Upstream merge	Merged with community PostgreSQL 11.18.29. See the community Release Notes for details.		
Enhancement	Enhance edb_filter_log.redact_password_commands so that passwords used in the CREATE/ALTER ROLE, CREATE/ALTER USER, and CREATE/ALTER GROUP syntax will get redacted when using optional parameters. [Support Ticket: #86366]	DB-1984	
Enhancement	Improve an error when you alter the table column dependent on the package variable. [Support Ticket: #85027]	DB-1950	

2.7 Version 11.17.28

EDB Postgres Advanced Server 11.17.28 includes the following bug fixes:

Type	Description	ID	Category
Upstream merge	Merged with community PostgreSQL 11.17. See the community Release Notes for details.		
Bug fix	For the re-entrant block, initialize all variables of type collection declared in that block to NULL to avoid incorrect results when the block is executed in a loop. [Support Ticket: #83856]	DB-1870	
Bug fix	Disable index_advisor if it is a physical replica. [Support Ticket: #83312]	DB-1819	
Bug fix	Fix unexpected failure when the PLpgSQL procedure with OUT parameters is called twice. [Support Ticket: #83289]	DB-1818	
Bug fix	Fix unexpected behavior of autonomous transactions with explicit COMMIT command in SPL block. [Support Ticket: #82695]	DB-1787	
Bug fix	Correct check to forbid UPDATE of declarative partitioned tables.	DB-237	edbldr
Bug fix	Restore the portal-level snapshot after the procedure COMMIT/ROLLBACK in CALL statement execution in PL to avoid a server crash.	DB-1820	

2.8 Version 11.16.26

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

EDB Postgres Advanced Server 11.16.26 includes the following bug fixes:

Type	Description	ID	Category
Upstream merge	Merged with community PostgreSQL 11.16. See the community Release Notes for details.		
Bug fix	Add syntax check for TABLE(...) clause. [Support Ticket: #81624]	DB-1752	
Bug fix	Add an appropriate cast for const TIMESTAMP and TIMESTAMPTZ to the query pushed to the remote servers. [Support Ticket: #72928]	DB-1493	oci_dblink
Bug fix	Do not push expressions to remote servers involving INTERVAL data type since representation in EPAS/PostgreSQL is different.	DB-1493	oci_dblink
Bug fix	Fix NLS_TIMESTAMP_TZ_FORMAT setting in the remote connection so that timestamptz data gets the correct representation.	DB-1493	oci_dblink
Bug fix	Add logic of encoding conversion at the time of writing the data on the remote servers.	DB-181	dblink_ora
Bug fix	Force NO SCROLL for SPL's implicit cursors [Support Ticket: #79231].	DB-1700	
Bug fix	Fix possible server crash in the partition-wise join when PARALLEL hint is used [Support Ticket: #81246].	DB-1741	

2.9 Version 11.15.25

EDB Postgres Advanced Server 11.15.25 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.15. See the community Release Notes for details.		
Bug Fix	Fix encoding conversion at the time of writing data to a remote server. [Support Ticket: #76625]	DB-1617	dblink_oci
Bug Fix	Add pg_dump support for security-relevant columns of edb_policy.	DB-1438	
Bug Fix	Prevent possible crash after implicit rollback handling Bind and Describe messages.	DB-1451	

2.10 Version 11.14.24

EDB Postgres Advanced Server 11.14.24 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with communuity PostgreSQL 11.14. See the community Release Notes for details.		
Bug Fix	Obey the AM meridian indicator correctly in `to_timestamp()`. [Support Ticket: #74035]	DB-1497	
Bug Fix	Prevent possible crash after implicit rollback handling `Parse` protocol message. [Support Ticket: #72626]	DB-1449	
Bug Fix	Fix possible server crash when the package is dropped from another session	DB-1403	SPL
Bug Fix	Populate the event type for missing node type.	DB-1184	edb_audit
Bug Fix	Fix server crash when the package is re-compiled in the same session. [Support Ticket: #1181417]	DB-1038	SPL

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

2.11 Version 11.13.23

Type	Description	ID
Upstream Merge	Merged with communuity PostgreSQL 11.13. See the community Release Notes for details.	

2.12 Version 11.12.22

EDB Postgres Advanced Server 11.12.22 includes the following bug fixes:

Type	Description	ID
Upstream Merge	Merged with communuity PostgreSQL 11.12. See the community Release Notes for details.	
Bug Fix	Fix `pg_upgrade` to allow the system catalog composite type used in user tables.	DB-1237
Bug Fix	Fix possible misbehavior when aborting an autonomous transaction and also fix interaction of autonomous transactions with `edb_stmt_level_tx=on`.	DB-1034

2.13 Version 11.12.21

EDB Postgres Advanced Server 11.12.21 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with communuity PostgreSQL PostgreSQL 11.12. See the community Release Notes for details.		
Bug Fix	Prevent some uses of `COMMIT` and `ROLLBACK` in stored procedures that were unsafe and could cause crashes. [Support Ticket: #1234722]	DB-1183	
Bug Fix	Prevent `index_advisor` from interfering with materialized view creation. [Support Ticket: #1189859]	DB-1154	
Bug Fix	Support password redaction in `edb_filter_log` with the extended protocol, specially used with connectors. [Support Ticket: #1234131]	DB-1139	
Bug Fix	Correct `QUEUE` object handling in `EVENT TRIGGER`.	DB-1129	
Bug Fix	Correct `REDACTION COLUMN` object handling in `EVENT TRIGGER`.	DB-1129	
Bug Fix	Fix `pg_upgrade` to not fail when a custom configuration file directory is used. [Support Ticket: #1200560]	DB-1084	
Bug Fix	Refrain from dropping trigger on parent table through partitioning dependency. [Support Ticket: #1187215]	DB-1063	
Bug Fix	Free temporary memory to avoid PGA memory accumulation and exceeds errors. [Support Ticket: #1129386]	DB-1061	dblink_ora
Bug Fix	Fix possible server crash with partition-wish join push-down code path.	DB-1042	edb_dblink_oci
Bug Fix	Fix incorrect error message in `edbldr`. [Support Ticket: #1104048]	DB-826	
Bug Fix	Allow partition creation with `ROWIDS` option.	DB-731	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

2.14 Version 11.11.20

EDB Postgres Advanced Server 11.11.20 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.11. See the community Release Notes for details.		
Bug Fix	Use correct relation information while loading into multiple tables with a single control file to avoid unexpected behavior. [Support Ticket: #1165964]	DB-973	edbldr
Bug Fix	For nested subprocedures, verify that a set-returning function is called in a valid place or not.	DB-946	
Bug Fix	Skip SQL Protect-related files when running `pg_checksums` or `pg_verify_checksums`. [Support Ticket: #1140841]	DB-919	
Bug Fix	Forbid `CONNECT_BY_ROOT` and `SYS_CONNECT_BY_PATH` in join expressions.	DB-914	
Bug Fix	Correct start position handling with multibyte encodings in `INSTR`. [Support Ticket: #1133262]	DB-911	
Bug Fix	Remove obsolete function overloading check condition in `pg_dump`. [Support Ticket: #1133344]	DB-892	
Bug Fix	Fix handling of whitespaces when the delimiter is whitespace. [Support Ticket: #1060286]	DB-739	edbldr

2.15 Version 11.10.19

EDB Postgres Advanced Server 11.10.19 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.10. See the community Release Notes for details.		
Bug Fix	Fixed crash with `COPY FROM` and/or foreign partition routing operations performed on `libpq_dblink` and `oci_dblink` FDW table.	DB-804	
Bug Fix	Don't emit useless messages about GSS or SSL negotiation failures. [Support Tickets: #1048322, #1048143, #1018659]	DB-770	
Bug Fix	We create this system column for all the foreign tables. However, only `oci_dblink` and `libpq_dblink` FDWs use it and the code assumes that this column is used with these two FDWs only. That is incorrect. This commit fixes the assertion failure when system column `__remote_rowid_` is selected.	DB-749	
Bug Fix	Fixed handling of multi-character record separators in edbldr. [Support Ticket: #1051362]	DB-743	
Bug Fix	Fixed group estimate for the remote relation in `oci_dblink`.	DB-701	
Bug Fix	In non-Pro*C mode, evaluate `EXEC SQL IFDEF/IFNDEF` expression irrespective of the enclosed 'C' preprocessor directives and emit all 'C' preprocessor directives as-is to the output file. [Support Ticket: #1007795]	DB-551	ecpg
Bug Fix	Fix row incompatibility issue when `SUBTYPE` is used in composite type.	DB-107	SPL

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

2.16 Version 11.9.17

EDB Postgres Advanced Server 11.9.17 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.9. See the community Release Notes for details.		
Bug Fix	In the case of an error coming from `standard_planner`, global pointer is not getting cleaned up and results in a server crash. Commit fix dangling pointer de-reference for global index candidates. [Support Ticket: #1026986]	DB-682	
Bug Fix	Fix minor problems with non-exclusive backup cleanup. [Support Ticket: #1009383]	DB-597	
Bug Fix	Fix to follow the same order to compute the hash values of partition key in which it's defined. [Support Ticket: #1010975]	DB-537	edb_enable_pruning
Bug Fix	Make the scope of an autonomous transaction include the exception block. [Support Ticket: #977822]	DB-371	
Bug Fix	Fix FF mode in Redwood `to_char()` datetime related functions	DB-640	
Bug Fix	Protect (timestamp + number) and (timestamp - number) against overflow.	DB-639	
Bug Fix	Disallow using `NULLIF` as SQL expression evaluation is not supported in the direct load.	DB-477	edbldr
Bug Fix	Fix assorted issues where ZONED data input length is less than the precision.	DB-301	edbldr
Bug Fix	Commit move the schema grant statement from `edb-sys.sql` to code in order to fix the `pg_upgrade` for `--no-redwood-compat mode`.	RM43953	

2.17 Version 11.9.16

EDB Postgres Advanced Server 11.9.16 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.9. See the community Release Notes for details.		
Bug Fix	In the case of an error coming from `standard_planner`, global pointer is not getting cleaned up and results in a server crash. Commit fix dangling pointer de-reference for global index candidates. [Support Ticket: #1026986]	DB-682	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Description	ID	Category
Bug Fix	Fix to follow the same order to compute the hash values of partition key in which it's defined. [Support Ticket: #1010975]	DB-537	edb_enable_pruning
Bug Fix	Make the scope of an autonomous transaction include the exception block. [Support Ticket: #977822]	DB-371	
Bug Fix	Fix FF mode in Redwood `to_char()` datetime related functions	DB-640	
Bug Fix	Protect (timestamp + number) and (timestamp - number) against overflow.	DB-639	
Bug Fix	Disallow using `NULLIF` as SQL expression evaluation is not supported in the direct load.	DB-477	edbldr
Bug Fix	Fix assorted issues where ZONED data input length is less than the precision.	DB-301	edbldr
Bug Fix	Fix assorted issues where ZONED data input length is less than the precision.	DB-301	edbldr
Bug Fix	Commit move the schema grant statement from `edb-sys.sql` to code in order to fix the `pg_upgrade` for `--no-redwood-compat mode`.	RM43953	

2.18 Version 11.8.15

EDB Postgres Advanced Server 11.8.15 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.8. See the community Release Notes for details.		
Bug Fix	Add GUC to control the scan type for the remote statement. New GUC `edb_dblink_oci.rescan = {serializable | scroll}` allows a user to choose the scrollable vs non-scrollable cursor. [Support Ticket: #947738]	DB-380	dblink_oci
Bug Fix	Do not push `ROWNUM` to the child scan/join targets paths and disable partition-wise aggregate when query has `ROWNUM`.	DB-187	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

2.19 Version 11.7.14

EDB Postgres Advanced Server 11.7.14 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.7. See the community Release Notes for details.		
Bug Fix	Suppressed error detail when password expiration is set to infinity while using password profiles.	RM44176	
Bug Fix	Allowed altering only its own password when expired while using password profiles.	RM44175/ DB-228	
Bug Fix	Disabled partition pruning while fetching a cursor rowtype. [Support Ticket: #950257]	RM44174	
Bug Fix	Fixed execution of empty statements via expired accounts. [Support Ticket: #942512]	RM44155	
Bug Fix	Skipped expression simplification while fetching cursor rowtype. [Support Ticket: #950257]	RM44151	
Bug Fix	Fixed `DROP TRIGGER` by name behavior for statement-level triggers.	RM44141	
Bug Fix	Fixed `makeConst()` call arguments to order correctly in `parse_spl_var()` .	RM44130	
Bug Fix	Fixed TAP testcase failure.	RM44129	
Bug Fix	Changed datatype from int to double to avoid wrap-around in password profile.	RM44118	
Bug Fix	Fixed handling of subtypes in the extended protocol.	RM44123	
Bug Fix	Fixed `CREATE TRIGGER .. AUTHORIZATION` behavior to avoid potential `pg_upgrade` failures.	RM44109	
Bug Fix	Cleared `rolpasswordsetat` when a user is renamed.	RM44108	
Bug Fix	Don't try to prune a relation when an `OUTER JOIN` is present to avoid possible projection of incorrect results.	RM44049	
Bug Fix	Fixed server crash while loading data into partition table having foreign key with `CLOBBER_CACHE_ALWAYS` build.	RM44145	
Bug Fix	Fixed server crash while loading data into table through edbldr with `CLOBBER_CACHE_ALWAYS` build.	RM44146	

2.20 Version 11.6.13

EDB Postgres Advanced Server 11.6.13 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.6. See the community Release Notes for details.		
Bug Fix	Apply user default privileges with CREATE FOREIGN TABLE. [Support Ticket: #919364]	RM44104	
Bug Fix	Fix data loss when executing query using oci link and small temporary tablespace at redwood side. [Support Ticket: #864275]	RM43979	oci_db link
Bug Fix	Avoid WAL-logging bulkload information when the system is in recovery mode. [Support Ticket: #915987]	RM44112	
Bug Fix	Fix libpq bulk API to get into ready for new query state after the query execution fail. [Support Ticket: #755248]	RM43965	
Bug Fix	Fix failure to attach ROWNUM qual to the correct part of the query. [Support Ticket: #812764]	RM43787	
Bug Fix	Rectify an assumption while dumping queue callback actions.	RM44110	dbms_aq

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Description	ID	Category
Bug Fix	Replaced `’` with `'` as the former special character may not be converted to some other locale like zh_CN resulting in an error.	RM44122	
Bug Fix	Fix cache flush hazard with the use of already released system cache.	RM41952	
Bug Fix	Changed testcase wordings from ‘postgres’ to ‘edb-postgres’ to start the advanced server.	RM44117	
Bug Fix	Fix restrict `FREEZE` option for partition table like `COPY`.	RM44079	edbldr
Bug Fix	Fix 004_logrotate TAP test failure related to update of current_logfiles.	RM44083	pg_ctl
Bug Fix	Testcase to verify server is not crashing with `oci` subquery.	RM44097	
Bug Fix	Fix DROP TRIGGER by name with the partitioned table.	RM44010	
Bug Fix	Fixed restricted use of `LEVEL` with `PRIOR` in `CONNECT BY`.	RM43998	
Bug Fix	Record owner dependency for redwood-style trigger functions.	RM44044	
Bug Fix	Fix OOM while inserting huge data into multiple tables.	RM43985	edbldr
Bug Fix	Correct the method used to release tuple descriptor.	RM44060	
Bug Fix	Fixed re-establishment of ocidblink connections after server or user mapping changes.	RM43963	
Bug Fix	Fix `ALTER SERVER` for `connstr` option.	RM44058	ocilink
Bug Fix	Fix identity column with not null constraint.	RM44016	edbldr

2.21 Version 11.5.12

EDB Postgres Advanced Server 11.5.12 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.5. See the community Release Notes for details.		
Bug Fix	Adjust the `CALL` statement to account for CallStmt rule. [Support Ticket: #872479]	RM44027	ecpg
Bug Fix	Fix to minimize the memory consumption for partition table select. [Support Ticket: #861800]	RM43945	dblink_oci
Bug Fix	Convert `RAW` to `VARCHAR` before assigning to `VARCHAR` variable. [Support Ticket: #865427]	RM43983	
Bug Fix	Fix `to_char()` for formats like spell mode, "IWW", "SPTH", "DDD", "D (day of the week)" etc. to make `to_char()` more redwood and PG compatible. [Support Ticket: #865427]	RM43873	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Description	ID	Category
Bug Fix	Fix buffer overflow issue in CALL statement handling.	RM44055	ecpg
Bug Fix	Prevent parallel index build under autonomous transaction.	RM44023	
Bug Fix	Fix `WARNING` for not owning proper lock on relation during resource cleanup.	RM44030	edbldr
Bug Fix	Fix test_decoding slot test.	RM44026	
Bug Fix	Add support for `edb_filter_log`.	RM44025	
Bug Fix	Don't allow password and obfuscated_password options together.	RM43962	dblink_oci
Bug Fix	Fixed error "unexpected varattno 2 in expression to be mapped" in subpartition when `edb_enable_pruning` is ON.	RM44020	
Bug Fix	Fix `INSERT` mode for partitioning table.	RM44011	edbldr
Bug Fix	Fix concurrency problem in autonomous transactions.	RM43824	
Bug Fix	Fix `__remote_rowid_` attribute values correctly.	RM44002	

2.22 Version 11.4.11

EDB Postgres Advanced Server 11.4.11 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.4. See the community Release Notes for details.		
Bug Fix	Don't mark portal as FAILED when executing `SPL ROLLBACK`. [Support Ticket: #870716]	RM43994	SPL
Bug Fix	Fix `REASSIGN OWNED BY` for `dbms_aq` objects.	RM43959	
Bug Fix	Suppress line numbers (#line directive) with `-l` option.	RM43689	ecpg
Bug Fix	Add missing `sepgsql` checks for namespace lookups	RM43055	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Description	ID	Category
Bug Fix	Reject non-`QT_NORMAL SELECT` statements in `SPI_is_cursor_plan()`.	RM43970	
Bug Fix	Throw a user-friendly error when package type has dropped attributes.	RM43938	
Bug Fix	Cloneschema fails when applying FK constraints on the target if rows are constantly being inserted in the source. [Support Ticket: #860472]	DI-166	

2.23 Version 11.3.10

EDB Postgres Advanced Server 11.3.10 includes the following bug fixes:

Type	Description	ID	Category
Upstream Merge	Merged with community PostgreSQL 11.3. See the community Release Notes for details.		
Bug Fix	ROMNUM with partitioning table by making sure rownum filter gets applied over Append node.	RM43887	
Bug Fix	REASSIGN/DROP OWNED BY for synonym and directory objects. [Support Ticket: #855677]	RM43930	
Bug Fix	Add support for synonym resolution to the type. [Support Ticket: #712701]	RM43047	
Bug Fix	Fix for `dbms_scheduler` to set the job next run correctly. [Support Ticket: #851452]	RM43917	
Bug Fix	`ALTER USER MAPPING` to treat `obfuscated_password` and `password` as same option. [Support Ticket: #847443]	RM43903	
Bug Fix	Fix an issue with addition of timestamp and negative numbers. [Support Ticket: #853879]	RM43923	
Bug Fix	Use `localVarCxt` to store trigger variable values.	RM43956	SPL
Bug Fix	Serialize tuple and index updates while exchanging statistics for partition during exchange partition. [Support Ticket: #864189]	RM43952	
Bug Fix	Fix for server crash with `edb_enable_pruning` is true. [Support Ticket: #853578]	RM43939	
Bug Fix	Fixed 'C' multiline macro handling.	RM43920	ecpg
Bug Fix	Fix for `pg_dump` / `restore` for `VIEW`s having `DECODE` expression by displaying implicit cast while deparsing `DECODE` expressions. [Support Ticket: #851791]	RM43915	
Bug Fix	Fix server crash with `ASSERT` statement.	RM43940	SPL
Bug Fix	Preserve the shared dependencies on package in case of replace.	RM41964	
Bug Fix	Quote nested sub-procedure name while deparsing it into a `CREATE` statement.	RM41007	
Bug Fix	Qualify type name while deparsing package variable.	RM41958	

2.24 Version 11.2.9

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

EDB Postgres Advanced Server 11.2.9 includes the following bug fixes:

Type	Description	ID
Upstream Merge	Merged with community PostgreSQL 11.2. See the community Release Notes for details.	
Bug Fix	Fix possible server crash in `bytea_smaller()` and `bytea_larger()`. [Support Ticket: #833954]	RM43888
Bug Fix	Allow `COMMIT` / `ROLLBACK` in any function-like expressions invoked via `CALL`.	RM43886
Bug Fix	Fix possible `pg_upgrade` failure for databases containing `DBMS_AQ` queue tables.	RM41964
Bug Fix	Fix possible server crash when attempting to use named-parameter syntax to call a function whose arguments are unnamed.	RM41971
Bug Fix	Fix failure to dump triggers on child partitioned tables when upgrading from a version prior to v10.	RM43593
Bug Fix	Fix incorrect handling of `IYYY` mask when `to_char()` is used. [Support Ticket: #823847]	RM43843
Bug Fix	Fix wrong answer when `to_date()` is used with the RR mask for with or during a year ending in 00. [Support Ticket: #822321]	RM43832
Bug Fix	Fix possible cache lookup failure when accessing the definition of an object type from which an attribute has been dropped.	RM43739
Bug Fix	Don't permit modifications to `aq_administrator_role`, since it is a special role used by the database system. [Support Ticket: #731149]	RM43281
Bug Fix	Fix locking bug when renaming a redaction policy.	
Bug Fix	Fix failure to apply insert performance optimization when inserting into a partitioned table.	
Bug Fix	Fix `CREATE ROLE` to honor `NOSUPERUSER` even when `IDENTIFIED BY` is specified. [Support Ticket: #818138]	RM43809

2.25 Version 11.1.7

New features, enhancements, bug fixes, and other changes in EDB Postgres Advanced Server 11 include:

Type	Category	Description
Upstream Merge		Merged with community PostgreSQL 11. See the community Release Notes for details.
Feature	General Functionality	Advanced Server no longer creates the `dbo` system catalog in its databases.
Feature	General Functionality	Advanced Server now supports data redaction. Data redaction is a technique for limiting the exposure of sensitive data to certain users. Data redaction results in the alteration of the displayed data to such users. This is accomplished by defining redaction policies for tables with sensitive data.
Feature	General Functionality	You can use the `edb_filter_log.redact_password_commands` extension to instruct the server to redact stored passwords from the log file.
Feature	General Functionality	Advanced Server now supports EDB wait states, which is a background worker that probes each running session at a regular interval collecting information such as the database connection, the logged in user, the running query, and the wait events.
Feature	General Functionality	Advanced Server now permits an infinite number of tries for custom plan generation if you set the `edb_custom_plan_tries` parameter to `-1`.
Feature	General Functionality	The output format of the `version()` function has been changed to appear more consistent with the PostgreSQL community version output.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Category	Description
Feature	General Functionality	Advanced Server now supports SPL standalone procedure overloading. Note that this feature is not compatible with Oracle databases.
Feature	General Functionality	Advanced Server now supports the `PRAGMA AUTONOMOUS_TRANSACTION` directive within any SPL block to provide the autonomous transaction capability.
Feature	General Functionality	Advanced server now offers performance improvements to libpq dblink and the OCI.
Enhancement	Partitioning	Allow faster partition elimination during query processing; this speeds access to partitioned tables with many partitions.
Enhancement	Partitioning	Allow partition elimination during query execution.
Enhancement	Partitioning	Allow the creation of partitions based on hashing a key.
Enhancement	Partitioning	Allow updated rows to automatically move to new partitions based on the new row contents.
Enhancement	Partitioning	Allow partitioned tables to have a default partition.
Enhancement	Partitioning	Allow `UNIQUE` indexes on partitioned tables if the partition key guarantees uniqueness.
Enhancement	Partitioning	Allow indexes on a partitioned table to be automatically created in any child partitions. The new command `ALTER INDEX ATTACH PARTITION` allows indexes to be attached to partitions. This does not behave as a global index since the contents are private to each index. `WARN WHEN USING AN EXISTING INDEX?`
Enhancement	Partitioning	Allow foreign keys on partitioned tables.
Enhancement	Partitioning	Allow `INSERT`, `UPDATE`, and `COPY` on partitioned tables to properly route rows to foreign partitions. This is supported by `postgres_fdw` foreign tables.
Enhancement	Partitioning	Allow `FOR EACH ROW` triggers on partitioned tables.
Enhancement	Partitioning	Allow equality joins between partitioned tables with identically partitioned child tables to join the child tables directly.
Enhancement	Partitioning	Perform aggregation on each partition, and then merge the results.
Enhancement	Partitioning	Allow `postgres_fdw` to push down aggregates to foreign tables that are partitions.
Enhancement	Parallel Queries	Allow B-tree indexes to be built in parallel.
Enhancement	Parallel Queries	Allow hash joins to be performed in parallel using a shared hash table.
Enhancement	Parallel Queries	Allow `UNION` to run each `SELECT` in parallel if the individual SELECTs cannot be parallelized.
Enhancement	Parallel Queries	Allow partition scans to more efficiently use parallel workers.
Enhancement	Parallel Queries	Allow `LIMIT` to be passed to parallel workers.
Enhancement	Parallel Queries	Allow single-evaluation queriea (for example, `WHERE` clause aggregate queries) and functions in the target list to be parallelized.
Enhancement	Parallel Queries	Add server option `parallel_leader_participation` to control if the leader executes subplans.
Enhancement	Parallel Queries	Allow parallelization of commands `CREATE TABLE .. AS`, `SELECT INTO`, and `CREATE MATERIALIZED VIEW`.
Enhancement	Parallel Queries	Improve performance of sequential scans with many parallel workers.
Enhancement	Parallel Queries	Add reporting of parallel worker sort activity to `EXPLAIN`.
Enhancement	Indexes	Allow indexes to `INCLUDE` columns that are not part of the unique constraint but are available for index-only scans. This is also useful for including columns that don't have B-tree support.
Enhancement	Indexes	Remember the highest B-tree index page to optimize future monotonically increasing index additions.
Enhancement	Indexes	Allow entire hash index pages to be scanned. Previously for each hash index entry, we need to refind the scan position within the page. This cuts down on lock/unlock traffic.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Category	Description
Enhancement	Indexes	Add predicate locking for hash, GiST and GIN indexes.
Enhancement	Indexes	Allow heap-only-tuple (HOT) updates for expression indexes when the values of the expressions are unchanged.
Enhancement	SP-Gist	Add `TEXT` prefix operator ^@ which is supported by SP-GiST.
Enhancement	SP-Gist	Allow polygons to be indexed with SP-GiST.
Enhancement	SP-Gist	Allow SP-GiST to use lossy representation of leaf keys.
Enhancement	Optimizer	Improve the selection of the optimizer statistics' most-common-values.
Enhancement	Optimizer	Improve selectivity estimates for `>=` and `<=` when the constants are not common values.
Enhancement	Optimizer	Optimize `var = var` to `var IS NOT NULL` where equivalent.
Enhancement	Optimizer	Improve row count optimizer estimates for `EXISTS` and `NOT EXISTS` queries.
Enhancement	Optimizer	Add optimizer selectivity costs for `HAVING` clauses.
Enhancement	Performance	Add Just-in-Time (JIT) compilation of some parts of query plans to improve execution speed.
Enhancement	Performance	Allow bitmap scans to perform index-only scans when possible.
Enhancement	Performance	Update the free space map during vacuum.
Enhancement	Performance	Allow vacuum to avoid unnecessary index scans.
Enhancement	Performance	Improve performance of committing multiple concurrent transactions.
Enhancement	Performance	Reduce memory usage for queries using set-returning functions in their target lists.
Enhancement	Performance	Allow `postgres_fdw` to push UPDATEs and DELETEs using joins to foreign servers.
Enhancement	Monitoring	Show memory usage in `log_statement_stats`, `log_parser_stats`, `log_planner_stats`, `log_executor_stats`.
Enhancement	Monitoring	Add `pg_stat_activity.backend_type` now shows the type of background worker.
Enhancement	Monitoring	Add `bgw_type` to the background worker C structure. This is displayed to the user in `pg_stat_activity.backend_type` and ps output.
Enhancement	Monitoring	Have `log_autovacuum_min_duration` log skipped tables that are concurrently being dropped.
Enhancement	Information Schema	Add `information_schema` columns related to table constraints and triggers.
Enhancement	Authentication	Allow the server to specify more complex LDAP specifications in search+bind mode.
Enhancement	Authentication	Allow LDAP authentication to use ldaps.
Enhancement	Authentication	Improve LDAP logging of errors.
Enhancement	Permissions	Add default roles which control file system access.
Enhancement	Permissions	Allow access to file system functions to be controlled by `GRANT/REVOKE` permissions, rather than superuser checks.
Enhancement	Permissions	Use `GRANT/REVOKE` to control access to `lo_import()` and `lo_export()`.
Enhancement	Permissions	Compile-time option `ALLOW_DANGEROUS_LO_FUNCTIONS` has been removed.
Enhancement	Permissions	Use viewowner not session owner when preventing non-password access to `postgres_fdw` tables.
Enhancement	Permissions	Fix invalid locking permission check in `SELECT FOR UPDATE` on views.
Enhancement	Server Configuration	Add server setting `ssl_passphrase_command` to allow supplying of the passphrase for SSL key files.
Enhancement	Server Configuration	Add storage parameter `toast_tuple_target` to control the minimum length before `TOAST` storage will be considered for new rows.
Enhancement	Server Configuration	Allow server options related to memory and file sizes to be specified as a number of bytes.
Enhancement	Write-Ahead Log (WAL)	Allow the WAL file size to be set via `initdb`.
Enhancement	Write-Ahead Log (WAL)	Do not retain WAL that spans two checkpoints.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Category	Description
Enhancement	Write-Ahead Log (WAL)	Fill the unused portion of force-switched WAL segment files with zeros for improved compressibility.
Enhancement	Base Backup and Streaming Replication	Replicate `TRUNCATE` activity when using logical replication.
Enhancement	Base Backup and Streaming Replication	Pass prepared transaction information to logical replication subscribers.
Enhancement	Base Backup and Streaming Replication	Exclude unlogged tables, temporary tables, and `pg_internal.init` files from streaming base backups.
Enhancement	Base Backup and Streaming Replication	Allow heap pages checksums to be checked during streaming base backup.
Enhancement	Base Backup and Streaming Replication	Allow replication slots to be advanced programmatically rather than be consumed by subscribers.
Enhancement	Base Backup and Streaming Replication	Add timeline information to the `backup_label` file.
Enhancement	Base Backup and Streaming Replication	Add host and port connection information to the `pg_stat_wal_receiver` system view.
Enhancement	Window Functions	Add window function features to complete SQL:2011 compliance.
Enhancement	Utility Commands	Allow `ALTER TABLE` to add a column with a non-null default without a table rewrite.
Enhancement	Utility Commands	Allow views to be locked by locking the underlying tables.
Enhancement	Utility Commands	Allow `ALTER INDEX` to set statistics-gathering targets for expression indexes.
Enhancement	Utility Commands	In psql, \d+ now shows the statistics target for indexes.
Enhancement	Utility Commands	Allow multiple tables to be specified in one `VACUUM` or `ANALYZE` command. Also, if any table mentioned in VACUUM uses a column list, then the `ANALYZE` keyword must be supplied; previously, `ANALYZE` was implied in such cases.
Enhancement	Utility Commands	Add parenthesized options syntax to `ANALYZE`. This is similar to the syntax supported by `VACUUM`.
Enhancement	Utility Commands	Add `CREATE AGGREGATE` option to specify the behavior of the aggregate finalization function.
Enhancement	Data Types	Allow the creation of arrays of domains.
Enhancement	Data Types	Support domains over composite types.
Enhancement	Data Types	Add casts from jsonb scalars to numeric and boolean data types.
Enhancement	Functions	Add SHA-2 family of hash functions; specifically, `sha224()`, `sha256()`, `sha384()`, `sha512()` were added.
Enhancement	Functions	Add support for 64-bit non-cryptographic hash functions.
Enhancement	Functions	Allow `to_char()` and `to_timestamp()` to specify the time zone's hours and minutes from UTC.
Enhancement	Functions	Improve the speed of aggregate computations.
Enhancement	Functions	Add text search function `websearch_to_tsquery()` that supports a query syntax similar to that used by web search engines.
Enhancement	Functions	Add function `json(b)_to_tsvector()` to create a text search query for matching JSON/JSONB values.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Category	Description
Enhancement	Server-Side Languages	Add SQL-level procedures, which can start and commit their own transactions. They are created with the new `CREATE PROCEDURE` command and invoked via `CALL`. The new `ALTER/DROP ROUTINE` commands allows altering/dropping of procedures, functions, and aggregates.
Enhancement	Server-Side Languages	Add transaction control to PL/pgSQL, PL/Perl, PL/Python, PL/Tcl, and SPI server-side languages.
Enhancement	Server-Side Languages	Add the ability to define PL/pgSQL record types as not null, constant, or with initial values.
Enhancement	Server-Side Languages	Allow PL/pgSQL to handle changes to composite types (e.g. record, row) that happen between the first and later function executions in the same session. Previously such circumstances generated errors.
Enhancement	Server-Side Languages	Add extension `jsonb_plpython` to transform JSONB to/from PL/Python types.
Enhancement	Server-Side Languages	Add extension `jsonb_plperl` to transform JSONB to/from PL/Perl types.
Enhancement	Client Interfaces	Change libpq to disable compression by default.
Enhancement	Client Interfaces	Add `DO CONTINUE` action to the `ECPG WHENEVER` statement.
Enhancement	Client Interfaces	Add ecpg mode to enable Oracle Pro*C handling of char arrays. This mode is enabled with -C.
Enhancement	psql	Add psql command `\gdesc` to display the column names and types of the query output.
Enhancement	psql	Add psql variables to report query activity and errors.
Enhancement	psql	Allow psql to test for the existence of a variable.
Enhancement	psql	Add `PSQL_PAGER` to control psql's pager.
Enhancement	psql	Have psql `\d+` always show the partition information.
Enhancement	psql	Have psql report the proper user name before the password prompt.
Enhancement	psql	Allow quit and exit to exit psql when used in an empty buffer.
Enhancement	psql	Have psql hint at using control-D when `\q` is entered alone on a line but ignored.
Enhancement	psql	Improve tab-completion for `ALTER INDEX RESET/SET`.
Enhancement	psql	Add infrastructure to allow psql to customize tab completion queries based on the server version. Previously tab completion queries could fail.
Enhancement	pgbench	Add pgbench expressions support for NULLs, booleans, and some functions and operators.
Enhancement	pgbench	Add `\if` conditional support to pgbench.
Enhancement	pgbench	Allow the use of non-ASCII characters in pgbench variable names.
Enhancement	pgbench	Add pgbench option `--init-steps` to control the initialization steps performed.
Enhancement	pgbench	Add an approximated Zipfian-distributed random generator to pgbench.
Enhancement	pgbench	Allow the random seed to be set in pgbench.
Enhancement	pgbench	Allow pgbench to do exponentiation with `pow()` and `power()`.
Enhancement	pgbench	Add hashing functions to pgbench.
Enhancement	pgbench	Make pgbench statistics more accurate when using `--latency-limit` and `--rate`.
Enhancement	Server Applications	Add an option to `pg_basebackup` that creates a named replication slot. The option `--create-slot` creates the named replication slot `(--slot)` when the WAL streaming method `(--wal-method=stream)` is used.
Enhancement	Server Applications	Allow `initdb` to set group read access to the data directory.
Enhancement	Server Applications	Add `pg_verify_checksums` tool to verify database checksums while offline.
Enhancement	Server Applications	Allow `pg_resetwal` to change the WAL segment size via `--wal-segsize`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Category	Description
Enhancement	Server Applications	Add long options to `pg_resetwal` and `pg_controldata`.
Enhancement	Server Applications	Add `pg_receivewal` option `--no-sync` to prevent synchronous WAL writes, for testing.
Enhancement	Server Applications	Add `pg_receivewal` option `--endpos` to specify when WAL receiving should stop.
Enhancement	Server Applications	Allow `pg_ctl` to send the `SIGKILL` signal to processes.
Enhancement	Server Applications	Reduce the number of files copied by `pg_rewind`.
Enhancement	Server Applications	Prevent `pg_rewind` from running as root.
Enhancement	pg_dump, pg_dumpall, pg_restore	Add `pg_dumpall` option `--encoding` to control encoding.
Enhancement	pg_dump, pg_dumpall, pg_restore	Add `pg_dump` option `--load-via-partition-root` to force loading of data into the partition's root table rather than the original partitions.
Enhancement	pg_dump, pg_dumpall, pg_restore	Add an option to suppress dumping and restoring comments.
Enhancement	Source Code	Add support for large pages on Windows.
Enhancement	Source Code	Add support for ARMv8 hardware CRC calculations.
Enhancement	Source Code	Convert documentation to DocBook XML.
Enhancement	Source Code	Use `stdbool.h` to define type bool on platforms where it's suitable.
Enhancement	Source Code	Add ability to use channel binding when using SCRAM authentication.
Enhancement	Source Code	Overhaul the way system tables are defined for bootstrap use.
Enhancement	Source Code	Allow background workers to attach to databases that normally disallow connections.
Enhancement	Source Code	Speed up lookups of built-in function names matching OIDs.
Enhancement	Source Code	Speed up construction of query results.
Enhancement	Source Code	Improve access speed to system caches.
Enhancement	Source Code	Add a generational memory allocator which is optimized for serial allocation/deallocation.
Enhancement	Source Code	Make the computation of system column `pg_class.reltuples` consistent.
Enhancement	Source Code	Update to use perltidy version.
Enhancement	Additional Modules	Allow extension `pg_prewarm` to restore the previous shared buffer contents on startup.
Enhancement	Additional Modules	Add `pg_trgm` function `strict_word_similarity()` to compute the similarity of whole words.
Enhancement	Additional Modules	Allow creation of indexes on citext extension columns that can be used by `LIKE` comparisons.
Enhancement	Additional Modules	Allow `btree_gin` to index bool, bpchar, name and uuid data types.
Enhancement	Additional Modules	Allow cube and seg extensions using GiST indexes to perform index-only scans.
Enhancement	Additional Modules	Allow retrieval of negative cube coordinates using the `~>` operator.
Enhancement	Additional Modules	Add Vietnamese letter detection to the unaccent extension.
Enhancement	Additional Modules	Enhance amcheck to check that each heap tuple has an index entry.
Enhancement	Additional Modules	Have adminpack use the new default file system access roles.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type	Category	Description
Enhancement	Additional Modules	Increase `pg_stat_statement` 's query id to 64 bits.
Enhancement	Additional Modules	Install `errcodes.txt` to provide access to the error codes reported by PostgreSQL.
Enhancement	Additional Modules	Prevent extensions from creating custom server variables that take a quoted list of values.
Enhancement	Additional Modules	Removed `contrib/start-scripts/osx` .
Enhancement	Additional Modules	Removed the chkpass extension.

Deprecated Features

The PL/Java package is deprecated in EDB Postgres Advanced Server 11 and will be unavailable in EDB Postgres Advanced Server 12.

Advanced Server no longer supports the Infinite Cache feature. All related components have been removed such as the functions, scripts, configuration parameters, and columns from statistical tables and views.

Component Certification

The following components are included in the EDB Postgres Advanced Server v12 release:

- Procedural Language Packs – PL/Perl 5.26, PL/Python 3.7, PL/TCL 8.6
- CloneSchema 1.10
- EDB_Wait_States 1.0 (formerly known as Edb-Modules)
- Parallel Clone 1.5
- pgAgent 4.15
- Slony 2.2.8
- Connectors JDBC 42.2.8, ODBC 12.00.0000 .NET 4.0.10.1, OCI 11.0.3.1
- pgAdmin 4.15
- pgBouncer 1.12.0
- pgPool-II & pgPool-IIExtensions 4.0.6
- MTK 53.0.0
- EDBPlus 38.0.0
- PostGIS-JDBC 2.2.1

3 Supported Platforms

EDB Postgres Advanced Server v11 installers support Linux and Windows server platforms.

See Platform Compatibility for additional information about supported platforms.

4 Limitations

The following limitations apply:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- EDB recommends you don't store the `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 on RHEL or CentOS x86 only.

5 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

- 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
- 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Windows

- Windows Server 2019

5.1 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
- Debian 11
- Debian 10

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

5.1.1 Installing EDB Postgres Advanced Server on RHEL 9 or OL 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 https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
  ```

- 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

```
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 11, the package name is `edb-as11-server`.

To install an individual component:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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/as11`.

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-11-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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
psql (11.x.x, server 11.x.x)
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)
```

5.1.2 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:

  ```
  dnf repolist | grep enterprisedb
  ```

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

  To set up the EDB repository:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
```

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

```
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 11, the package name is `edb-as11-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/as11`.

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-11-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/as11/bin/edb-as-11-setup initdb
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
sudo systemctl start edb-as-11
```

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 (11.x.x, server 11.x.x)
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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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)
```

5.1.3 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 crb
  ```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 11, the package name is `edb-as11-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/as11`.

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-11-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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

```
psql (11.x.x, server 11.x.x)
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)
```

5.1.4 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 11, the package name is `edb-as11-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/as11`.

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-11-setup` script creates a cluster in Oracle-compatible mode with the `edb`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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 (11.x.x, server 11.x.x)
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');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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)
```

5.1.5 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:

  ```
  sudo yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
  ```

- Enable additional repositories to resolve dependencies:

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
"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 11, the package name is `edb-as11-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/as11`.

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-11-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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 (11.x.x, server 11.x.x)
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)
```

5.1.6 Installing EDB Postgres Advanced Server on CentOS 7 x86_64

Prerequisites

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
  ```

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 11, the package name is `edb-as11-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/as11`.

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-11-setup` script creates a cluster in Oracle-compatible mode with the `edb`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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 (11.x.x, server 11.x.x)
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');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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)
```

5.1.7 Installing EDB Postgres Advanced Server on SLES 15 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/15.4/x86_64
  ```

- Refresh the metadata:

  ```
  sudo zypper refresh
  ```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 11, the package name would be `edb-as11-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-11-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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

```
psql (11.x.x, server 11.x.x)
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)
```

5.1.8 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 11, the package name would be `edb-as11-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-11-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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 (11.x.x, server 11.x.x)
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');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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)
```

5.1.9 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:

  ```
  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 11, the package name is `edb-as11-server`.

To install an individual component:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 (11.x.x, server 11.x.x)
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));
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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)
```

5.1.10 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:

  ```
  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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 11, the package name is `edb-as11-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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
psql (11.x.x, server 11.x.x)
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)
```

5.1.11 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 11, the package name is `edb-as11-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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

```
psql (11.x.x, server 11.x.x)
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)
```

5.1.12 Installing EDB Postgres Advanced Server on Debian 10 x86_64

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 11, the package name is `edb-as11-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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 (11.x.x, server 11.x.x)
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)
```

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

Red Hat Enterprise Linux (RHEL)

- RHEL 9
- RHEL 8
- RHEL 7

SUSE Linux Enterprise (SLES)

- SLES 15
- SLES 12

5.2.1 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:

  ```
  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 https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
  ```

- Refresh the cache:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
sudo dnf makecache
```

- 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

```
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 11, the package name is `edb-as11-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/as11`.

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-11-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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 (11.x.x, server 11.x.x)
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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
20      | RESEARCH   | DALLAS
(2 rows)
```

5.2.2 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:

  ```
  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 https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
  ```

- Refresh the cache:

  ```
  sudo dnf makecache
  ```

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

  ```
  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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 11, the package name is `edb-as11-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/as11`.

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-11-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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE DATABASE
hr;
```

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

```
psql (11.x.x, server 11.x.x)
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)
```

5.2.3 Installing EDB Postgres Advanced Server on SLES 15 ppc64le

Prerequisites

Before you begin the installation process:

- Set up the EDB repository.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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/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 11, the package name would be `edb-as11-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-11-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/as11/bin/edb-as-11-setup initdb
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
sudo systemctl start edb-as-11
```

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 (11.x.x, server 11.x.x)
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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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)
```

5.2.4 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
  ```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 11, the package name would be `edb-as11-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-11-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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE DATABASE
```

Connect to the `hr` database inside psql:

```
\c hr
```

```
psql (11.x.x, server 11.x.x)
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)
```

5.2.5 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

- Refresh the cache:

```
sudo dnf makecache
```

- 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 11, the package name is `edb-as11-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/as11`.

Initial configuration

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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-11-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/as11/bin/edb-as-11-setup initdb

sudo systemctl start edb-as-11
```

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 (11.x.x, server 11.x.x)
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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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)
```

5.3 Installing EDB Postgres Advanced Server on Windows

This section provides comprehensive information about installing EDB Postgres Advanced Server (Advanced Server), including:

- Software prerequisites for Advanced Server 11 installation on Windows.
- Graphical installation options available through the interactive setup wizard on Windows.

5.3.1 Requirements Overview

For information about the platforms and versions supported by Advanced Server, see Platform Compatibility

Limitation

- The `data` directory of a production database should not be stored on an NFS file system.

Windows Installation Prerequisites

User Privileges

To perform an Advanced Server installation on a Windows system, you must have administrator privileges. If you are installing Advanced Server on a Windows system that is configured with `User Account Control` enabled, you can assume sufficient privileges to invoke the graphical installer by right clicking on the name of the installer and selecting `Run as administrator` from the context menu.

Windows-specific Software Requirements

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

You should apply Windows operating system updates before invoking the Advanced Server installer. If (during the installation process) the installer encounters errors, exit the installation, and ensure that your version of Windows is up-to-date before restarting the installer.

Migration Toolkit or EDB*Plus Installation Pre-requisites

Before using StackBuilder Plus to install Migration Toolkit or EDB*Plus, you must first install Java (version 1.8 or later). If you are using Windows, Java installers and instructions are available online at:

http://www.java.com/en/download/manual.jsp

5.3.2 Installing Advanced Server with the Interactive Installer

You can use the Advanced Server interactive installer to install Advanced Server on Windows. The interactive installer is available from the EnterpriseDB website at:

https://www.enterprisedb.com/advanced-downloads

You can invoke the graphical installer in different installation modes to perform an Advanced Server installation:

- For information about using the graphical installer, seePerforming a Graphical Installation on Windows.
- For information about performing an unattended installation, see Performing an Unattended Installation.
- For information about performing an installation with limited privileges, see Performing an Installation with Limited Privileges.
- For information about the command line options you can include when invoking the installer, seeReference - Command Line Options.

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 alternate location for the temporary files by modifying the `TEMP` environment variable.

If invoking the installer from the command line, you can set the value of the variable on the command line. Use the command:

```
SET TEMP=temp_file_location
```

Where `temp_file_location` specifies the alternate location for the temporary files and must match the permissions with the `TEMP` environment variable.

> **Note**
>
> If you are invoking the installer to perform a system upgrade, the installer will preserve the configuration options specified during the previous installation.

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:

```
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 will supersede the value if specified in `INITDBOPTS`.

For more information about using `initdb` cluster configuration options, see the PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/11/static/app-initdb.html

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

In addition to the cluster configuration options documented in the PostgreSQL core documentation, 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 the 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.

5.3.2.1 Performing a Graphical Installation on Windows

A graphical installation is a quick and easy way to install Advanced Server 11 on a Windows system. Use the wizard's dialogs to specify information about your system and system usage; when you have completed the dialogs, the installer performs an installation based on the selections made during the setup process.

To invoke the wizard, you must have administrator privileges. Assume administrator privileges, and double-click the `edb-as11-server-11.x.x-x-windows-x64` executable file.

> **Note**
>
> To install Advanced Server on some versions of Windows, you may be required to right click on the file icon and select `Run as Administrator` from the context menu to invoke the installer with `Administrator` privileges.

When the `Language Selection` popup opens, select an installation language and click `OK` to continue to the `Setup` window.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

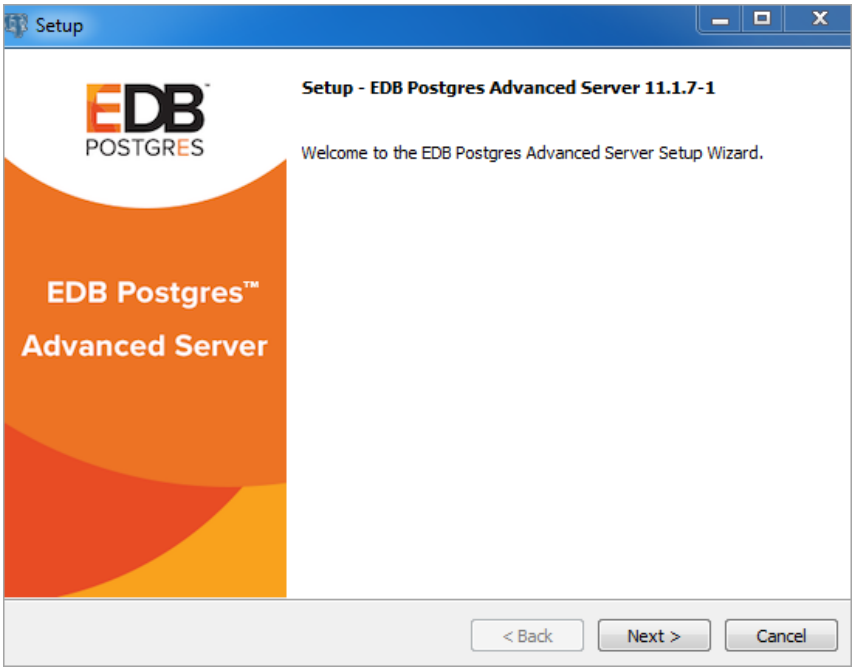


Fig. 1: The Advanced Server installer Welcome window

Click `Next` to continue.

The EnterpriseDB `License Agreement` window opens.

Fig. 2: The EnterpriseDB License Agreement

Carefully review the license agreement before highlighting the appropriate radio button; click `Next` to continue.

The `Installation Directory` window opens.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

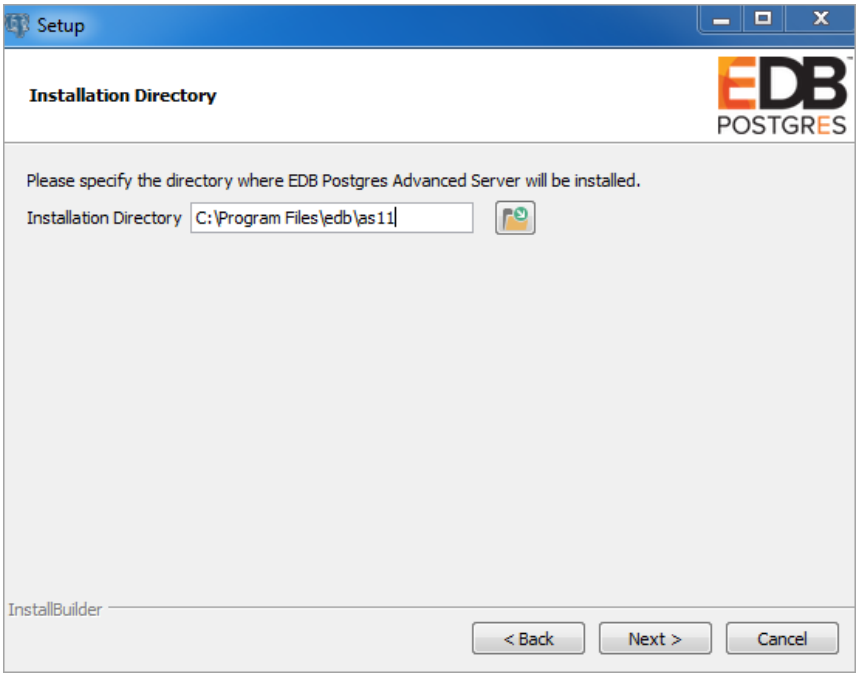


Fig. 3: The Installation Directory window

By default, the Advanced Server installation directory is:

```
C:\Program Files\edb\as11
```

You can accept the default installation location, and click `Next` to continue, or optionally click the `File Browser` icon to open the `Browse For Folder` dialog to choose an alternate installation directory.

> **Note**
>
> The `data` directory of a production database should not be stored on an NFS file system.

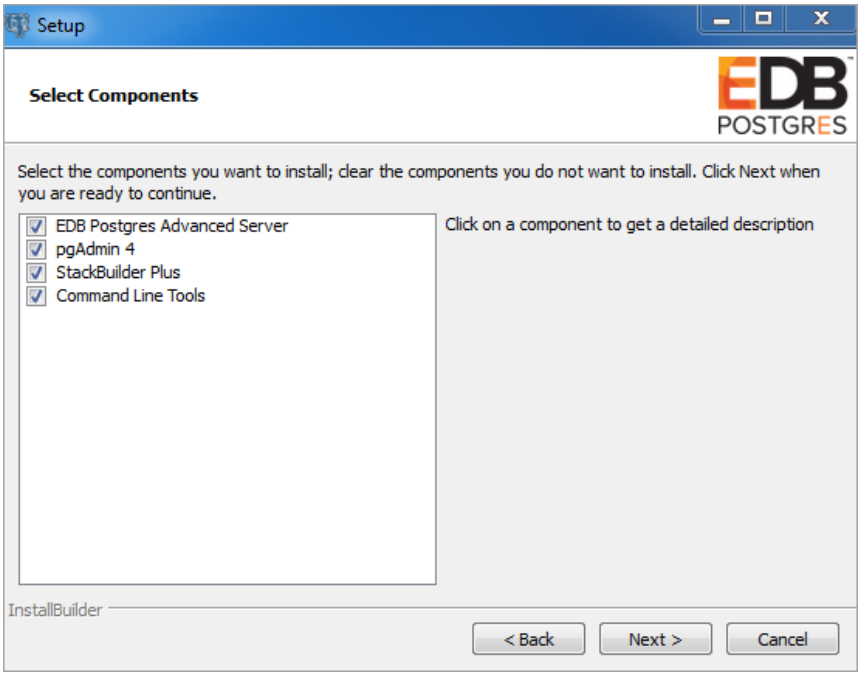


Fig. 4: The Select Components window

The `Select Components` window contains a list of optional components that you can install with the Advanced Server `Setup` wizard. You can omit a

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

module from the Advanced Server installation by deselecting the box next to the components name.

The `Setup` wizard can install the following components while installing Advanced Server 11:

EDB Postgres Advanced Server

Select the `EDB Postgres Advanced Server` option to install Advanced Server 11.

pgAdmin 4

Select the `pgAdmin 4` option to install the pgAdmin 4 client. pgAdmin 4 provides a powerful graphical interface for database management and monitoring.

StackBuilder Plus

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 Advanced Server setup and installation completes. See Using StackBuilder Plus for more information about StackBuilder Plus.

Command Line Tools

The `Command Line Tools` option installs command line tools and supporting client libraries including:

- libpq
- psql
- EDB*Loader
- ecpgPlus
- pg_basebackup, pg_dump, and pg_restore
- pg_bench and more.

> **Note**
>
> The `Command Line Tools` are required if you are installing Advanced Server or pgAdmin 4.

After selecting the components you wish to install, click `Next` to open the `Additional Directories` window.

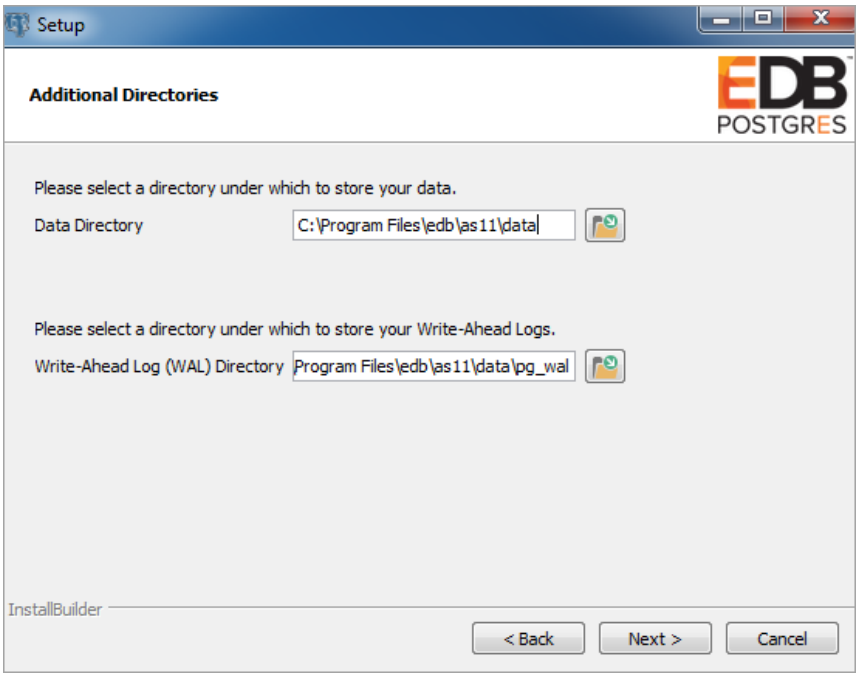


Fig. 5: The Additional Directories window

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

By default, the Advanced Server `data` files are saved to:

```
C:\Program Files\edb\as11\data
```

The default location of the Advanced Server `Write-Ahead Log (WAL) Directory` is:

```
C:\Program Files\edb\as11\data\pg_wal
```

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`, Advanced Server writes contents of the write-ahead log to disk.

Accept the default file locations, or use the `File Browser` icon to select an alternate location; click `Next` to continue to the `Advanced Server Dialect` window.

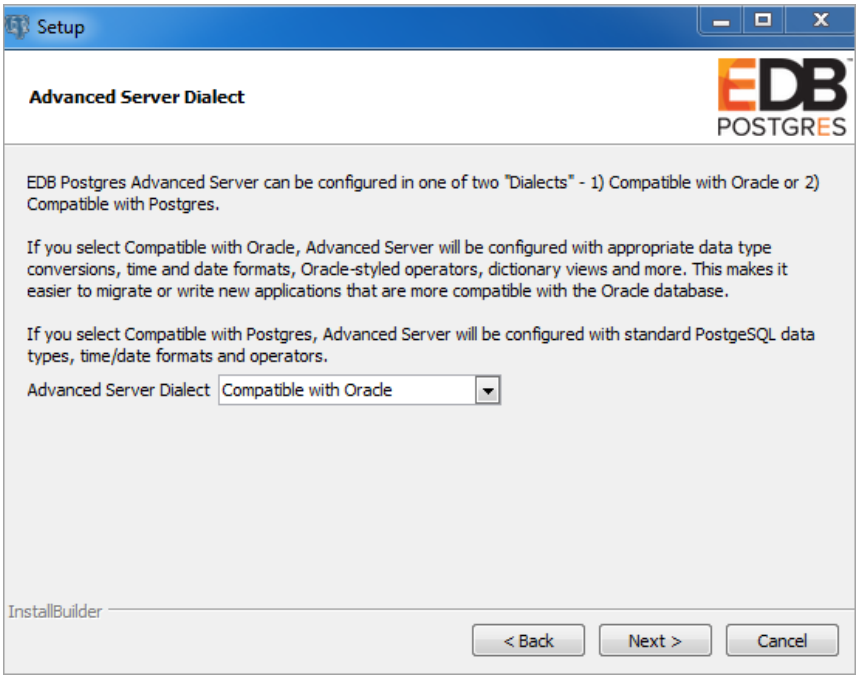


Fig. 6: The Advanced Server Dialect window

Use the drop-down listbox on the `Advanced Server Dialect` window to choose a server dialect. The server dialect specifies the compatibility features supported by Advanced Server.

By default, Advanced Server installs in `Compatible with Oracle` mode; you can choose between `Compatible with Oracle` and `Compatible with PostgreSQL` installation modes.

Compatible with Oracle

If you select `Compatible with Oracle`, the installation will include the following features:

- Data dictionary views that is 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:

Package	Functionality compatible with Oracle Databases

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Package	Functionality compatible with Oracle Databases
`dbms_alert`	Provides the capability to register for, send, and receive alerts.
`dbms_aq`	Provides queueing functionality for Advanced Server.
`dbms_aqadm`	Provides supporting functionality for dbms_aq.
`dbms_crypto`	Provides a way to encrypt or decrypt RAW, BLOB or CLOB data.
`dbms_job`	Implements job-scheduling functionality.
`dbms_lob`	Provides the ability to manage large objects.
`dbms_lock`	Provides support for the `DBMS_LOCK.SLEEP` procedure.
`dbms_mview`	Provides a way to manage and refresh materialized views.
`dbms_output`	Provides the ability to display a message on the client.
`dbms_pipe`	Provides the ability to send a message from one session and read it in another session.
`dbms_profiler`	Collects and stores performance data about PL/pgSQL and SPL statements.
`dbms_random`	Provides a way to generate random numbers.
`dbms_rls`	Implements row level security.
`dbms_scheduler`	Provides a way to create and manage Oracle-style jobs.
`dbms_session`	A partial implementation that provides support for `DBMS_SESSION.SET_ROLE`.
`dbms_sql`	Implements use of Dynamic SQL
`dbms_utility`	Provides a collection of misc functions and procedures.
`utl_encode`	Provides a way to encode or decode data.
`utl_file`	Provides a way for a function, procedure or anonymous block to interact with files stored in the server's file system.
`utl_http`	Provides a way to use HTTP or HTTPS to retrieve information found at a URL.
`utl_mail`	Provides a simplified interface for sending email and attachments.
`utl_raw`	Provides a way to manipulate or retrieve the length of raw data types.
`utl_smtp`	Implements smtp email functions.
`utl_url`	Provides a way to escape illegal and reserved characters in a URL.

This is not a comprehensive list of the compatibility features for Oracle included when Advanced Server is installed in `Compatible with Oracle` mode; for more information, see the *Database Compatibility for Oracle Developers Built-in Package Guide* available from the EnterpriseDB website at:

https://www.enterprisedb.com/docs

If you choose to install in `Compatible with Oracle` mode, the Advanced Server superuser name is `enterprisedb`.

Compatible with PostgreSQL

If you select `Compatible with PostgreSQL`, Advanced Server will exhibit compatibility with PostgreSQL version 11. If you choose to install in `Compatible with PostgreSQL` mode, the default Advanced Server superuser name is `postgres`.

For detailed information about PostgreSQL functionality, visit the official PostgreSQL website at:

http://www.postgresql.org

After specifying a configuration mode, click `Next` to continue to the `Password` window.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

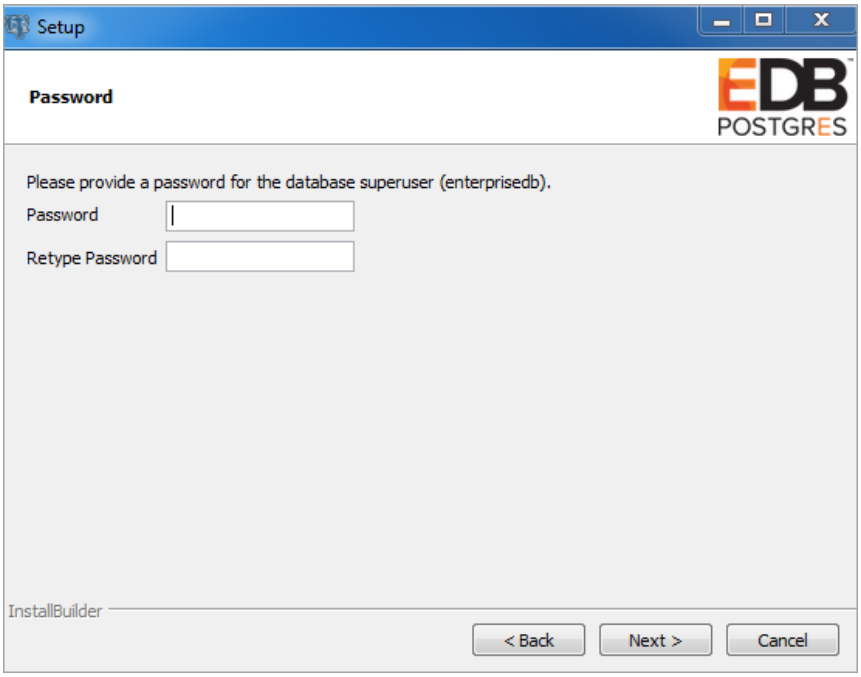


Fig. 7: The Password window

Advanced Server uses the password specified on the `Password` window for the database superuser. The specified password must conform to any security policies existing on the Advanced Server host.

After you enter a password in the `Password` field, confirm the password in the `Retype Password` field, and click `Next` to continue.

The `Additional Configuration` window opens.

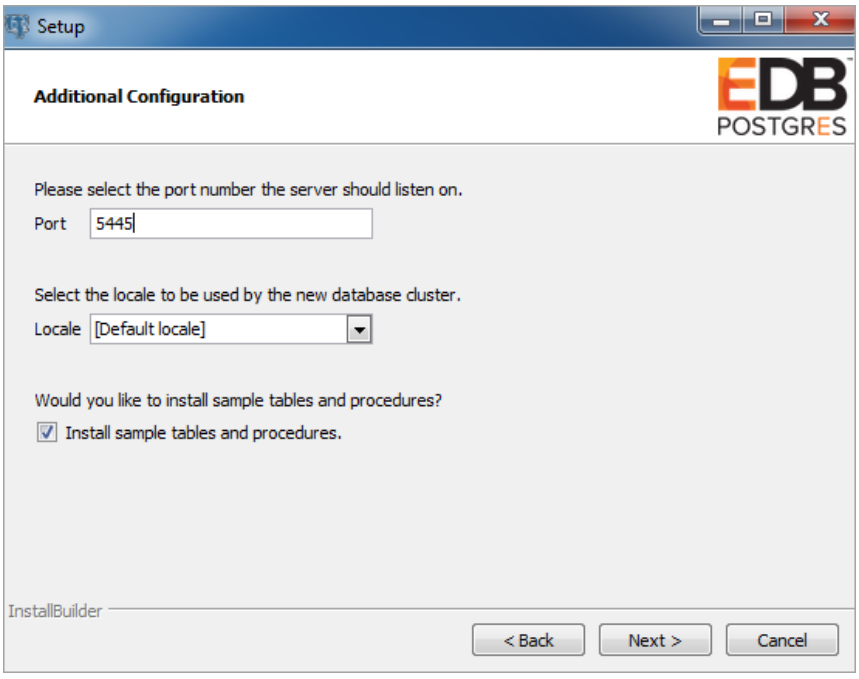


Fig. 8: The Additional Configuration window

Use the fields on the `Additional Configuration` window to specify installation details:

- Use the `Port` field to specify the port number that Advanced Server should listen to for connection requests from client applications. The default is `5444`.
- If the `Locale` field is set to `[Default locale]`, Advanced Server uses the system locale as the working locale. Use the drop-down listbox next to `Locale` to specify an alternate locale for Advanced Server.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- By default, the `Setup` wizard installs corresponding sample data for the server dialect specified by the compatibility mode `(Oracle` or `PostgreSQL)`. Clear the check box next to `Install sample tables and procedures` if you do not wish to have sample data installed.

After verifying the information on the `Additional Configuration` window, click `Next` to open the `Dynatune Dynamic Tuning: Server Utilization` window.

The graphical `Setup` wizard facilitates performance tuning via the Dynatune Dynamic Tuning feature. Dynatune functionality allows Advanced Server to make optimal usage of the system resources available on the host machine on which it is installed.

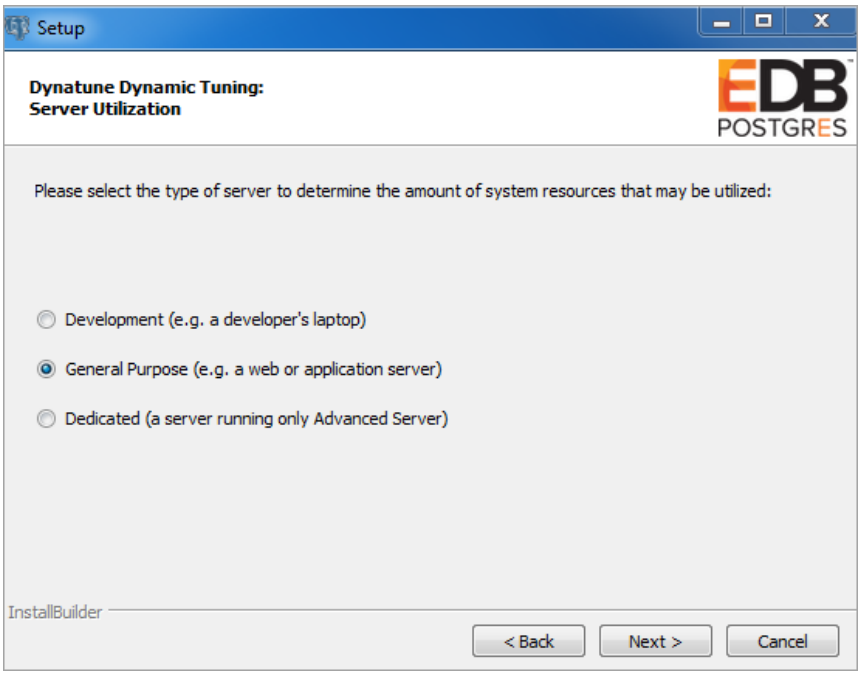


Fig. 9: The Dynatune Dynamic Tuning: Server Utilization window

The `edb_dynatune` configuration parameter determines how Advanced Server allocates system resources. Use the radio buttons on the `Server Utilization` window to set the initial value of the `edb_dynatune` configuration parameter:

- Select `Development` to set the value of `edb_dynatune` to `33`. A low value dedicates the least amount of the host machine's resources to the database server. This is a good choice for a development machine.
- Select `General Purpose` to set the value of `edb_dynatune` to `66`. A mid-range value dedicates a moderate amount of system resources to the database server. This would be a good setting for an application server with a fixed number of applications running on the same host as Advanced Server.
- Select `Dedicated` to set the value of `edb_dynatune` to `100`. A high value dedicates most of the system resources to the database server. This is a good choice for a dedicated server host.

After 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 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 click `Next` to continue to the `Dynatune Dynamic Tuning: Workload Profile` window.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

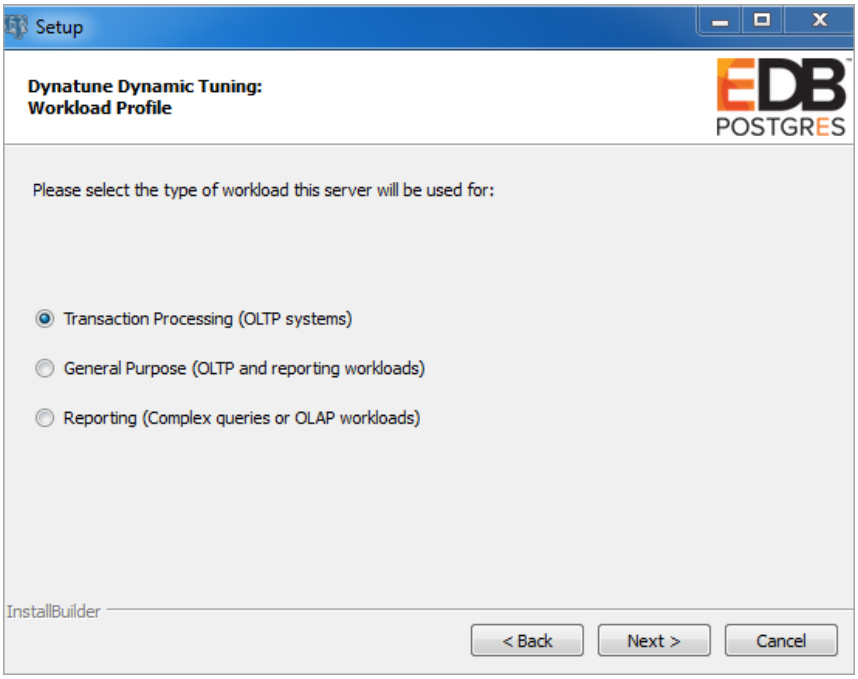


Fig. 10: The Dynatune Dynamic Tuning: Workload Profile window

Use the radio buttons 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 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 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 *EDB Postgres Advanced Server Guide* available from the EnterpriseDB website at:

https://www.enterprisedb.com/docs

Click `Next` to continue. The `Update Notification Service` window opens.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

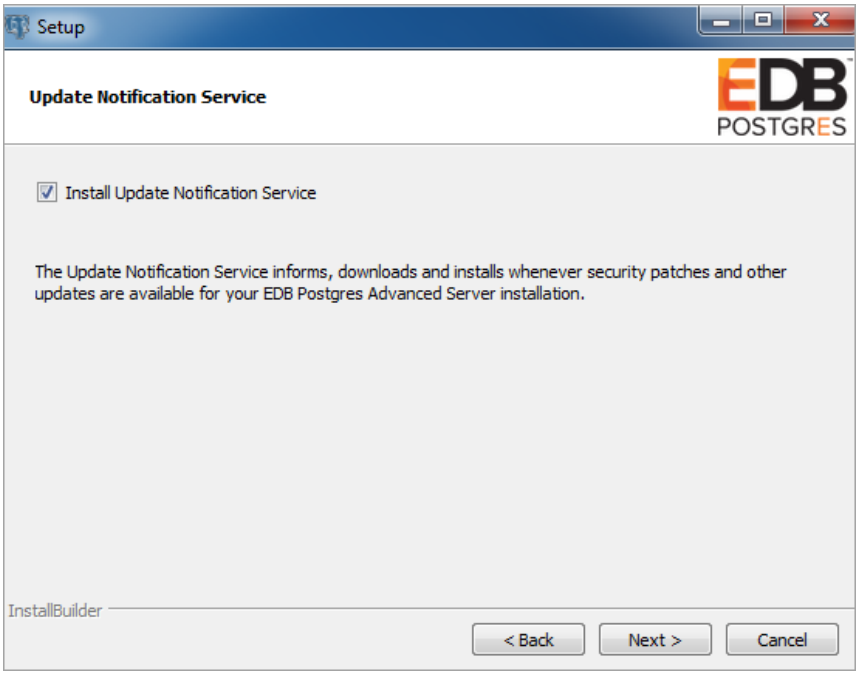


Fig. 11: The Update Notification Service window

When enabled, the update notification service notifies you of any new updates and security patches available for your installation of Advanced Server.

By default, Advanced Server is configured to start the service when the system boots; clear the `Install Update Notification Service` check box, or accept the default, and click `Next` to continue.

The `Pre Installation Summary` opens.

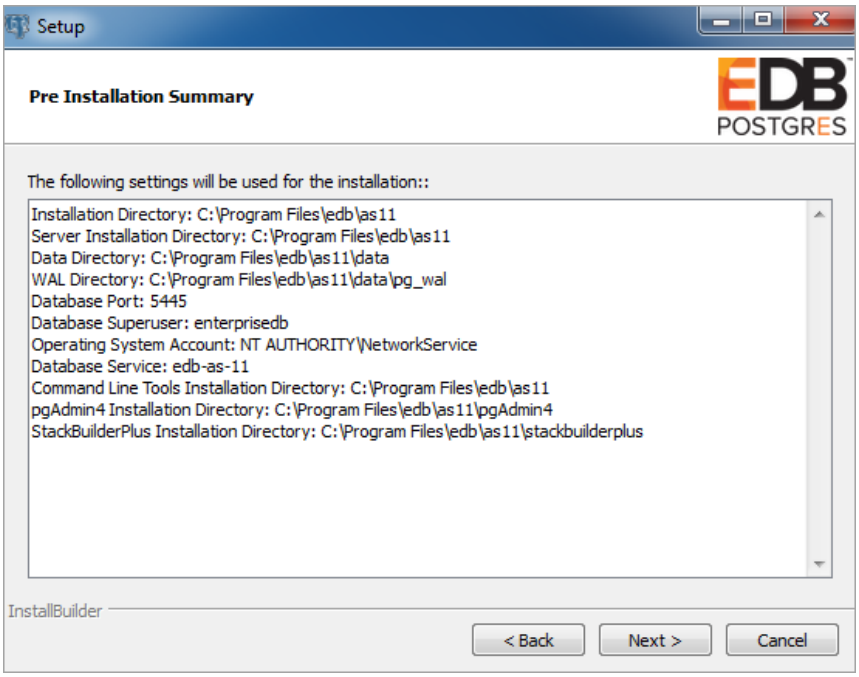


Fig. 12: The Pre Installation Summary

The `Pre Installation Summary` provides an overview of the options specified during the `Setup` process. Review the options before clicking `Next`; click `Back` to navigate back through the dialogs and update any options.

The `Ready to Install` window confirms that the installer has the information it needs about your configuration preferences to install Advanced Server. Click `Next` to continue.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

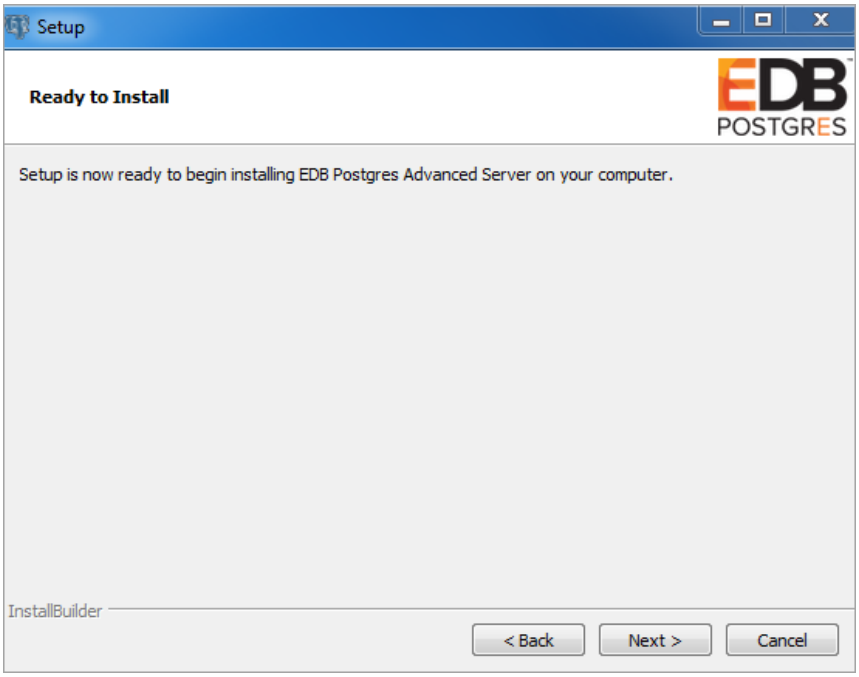


Fig. 13: The Ready to Install window

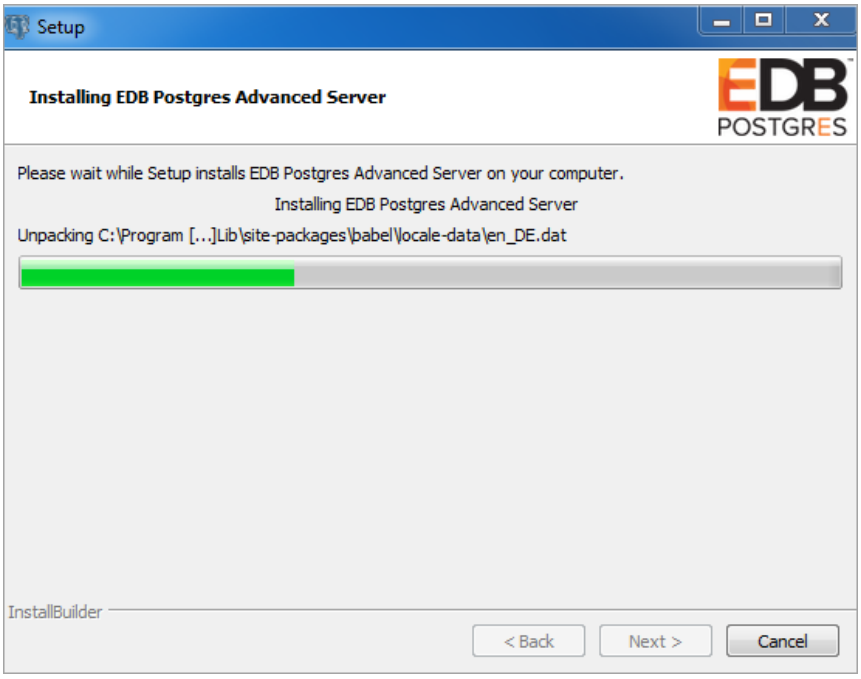


Fig. 14: Installing Advanced Server

As each supporting module is unpacked and installed, the module's installation is confirmed with a progress bar.

Before the `Setup` wizard completes the Advanced Server installation, it offers to `Launch StackBuilder Plus at exit?`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

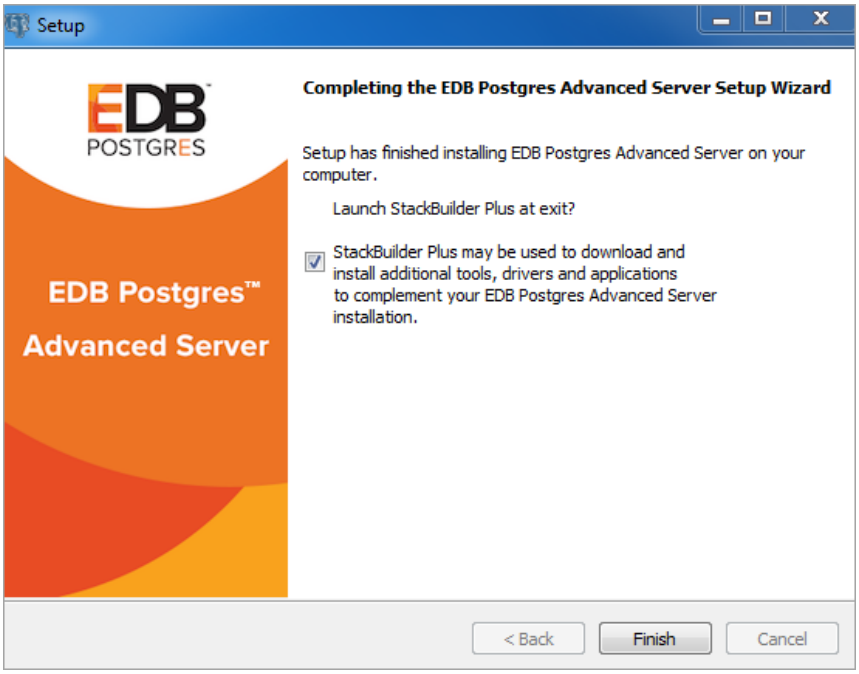


Fig. 15: The Setup wizard offers to Launch StackBuilder Plus at exit

You can clear the `StackBuilder Plus` check box and click `Finish` to complete the Advanced Server installation, or accept the default and proceed to StackBuilder Plus.

EDB Postgres StackBuilder Plus is included with the installation of 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 Advanced Server setup and installation completes. See Using StackBuilder Plus for more information about StackBuilder Plus.

5.3.2.2 Invoking the Graphical Installer from the Command Line

The command line options of the Advanced Server installer offer functionality for Windows systems that reside in situations where a graphical installation may 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, seeReference - Command Line Options.

> **Note**
>
> If you are invoking the installer from the command line to perform a system upgrade, the installer will ignore command line options, and preserve the configuration of the previous installation.

5.3.2.2.1 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)

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- parameters specified in an option file
- Advanced Server installation defaults

You can embed the non-interactive 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 Advanced Server using the `--mode unattended` option. If you are using the `--mode unattended` option to install 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-as11-server-11.x.x-x-windows-x64.exe --mode unattended --superpassword
database_superuser_password --servicepassword system_password
```

When invoking the installer, include the `--servicepassword` option to specify an operating system password for the user installing 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)`.

5.3.2.2.2 Performing an Installation with Limited Privileges

To perform an abbreviated installation of 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 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 the `initdb` and `pg_ctl` commands, see the PostgreSQL Core Documentation at:

https://www.postgresql.org/docs/11/static/app-initdb.html

https://www.postgresql.org/docs/11/static/app-pg-ctl.html

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 Advanced Server installation does not create menu shortcuts or provide access to EDB Postgres StackBuilder Plus, and no modifications are made to registry files.

To perform a limited installation and generate an installation script, download and unpack the Advanced Server installer. Navigate into the directory that contains the installer, and invoke the installer with the command:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb-as11-server-11.x.x-x-windows.exe --extract-only yes
```

A dialog opens, prompting you to choose an installation language. Select a language for the installation from the drop-down listbox, and click `OK` to continue. The `Setup Wizard` opens.

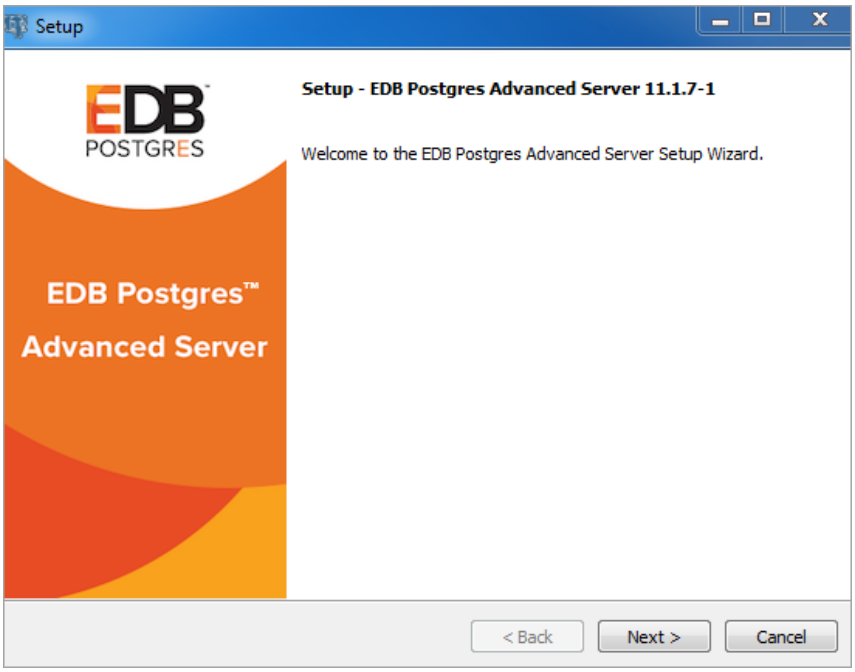


Fig. 16: The Welcome window

Click `Next` to continue.

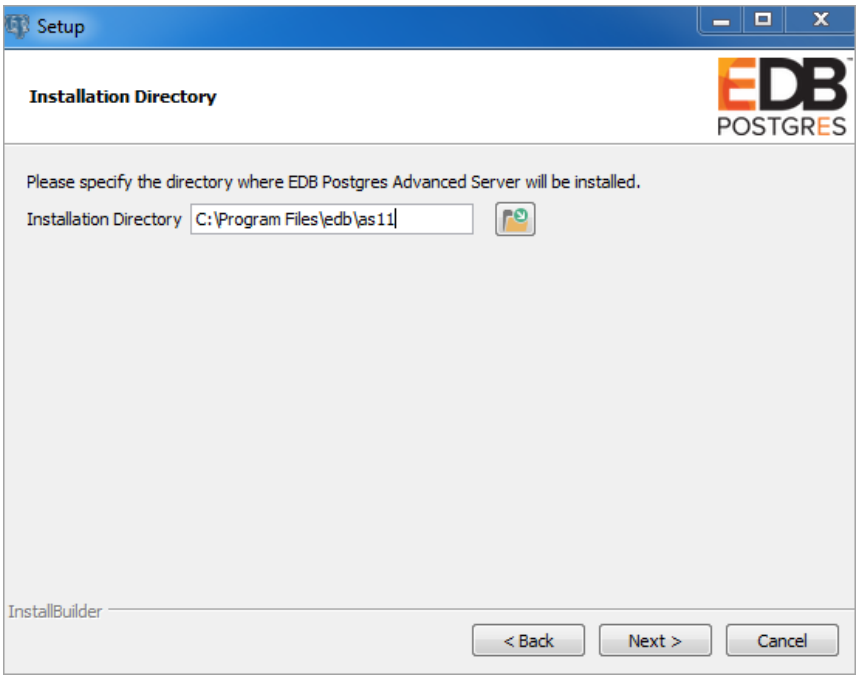


Fig. 17: Specify an installation directory

On Windows, the default Advanced Server installation directory is:

```
C:\Program Files\edb\as11
```

You can accept the default installation location and click `Next` to continue to the `Ready to Install` window, or optionally click the `File`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`Browser` icon to choose an alternate installation directory.

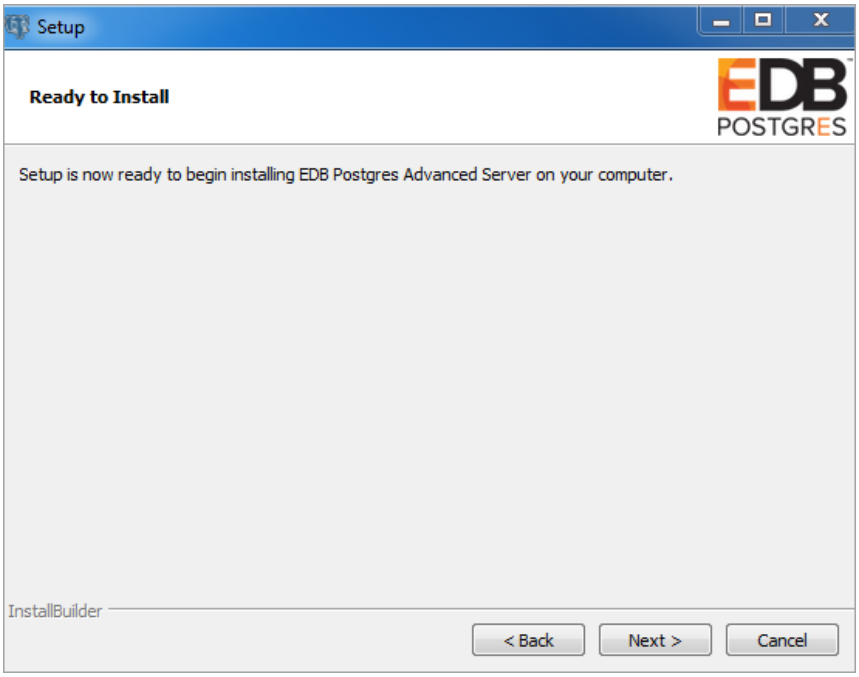


Fig. 18: The Setup wizard is ready to install Advanced Server

Click `Next` to proceed with the Advanced Server installation. During the installation, progress bars and popups mark the installation progress. The installer notifies you when the installation is complete.

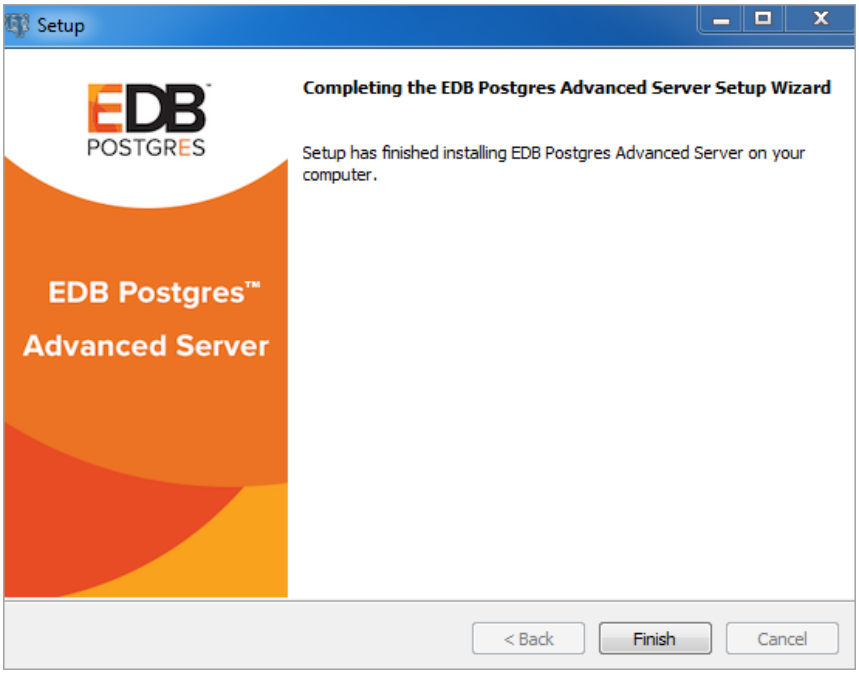


Fig. 19: The Advanced Server installation is complete

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
```

To execute the installation script, open a command line and assume administrative privileges. Navigate to the directory that contains the script, and execute the command:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
cscript runAsAdmin.vbs
```

The installation script executes at the command line, prompting you for installation configuration information. The default configuration value is displayed in square braces immediately before each prompt; update the default value or press `Enter` to accept the default value and continue.

Example

The following dialog is an example of a scripted installation. The actual installation dialog may vary, and will reflect the options specified during the installation.

Specify the installation directory is the directory where Advanced Server is installed:

```
Please enter the installation directory [ C:\Program Files\edb\as11 ] :
```

Specify the directory in which Advanced Server data will be stored:

```
Please enter the data directory path: [C:\Program Files\edb\as11\data ] :
```

Specify the WAL directory (where the write-ahead log will be written):

```
Please enter the Write-Ahead Log (WAL) directory path: [C:\Program Files\edb\as11\data\pg_wal ] :
```

The database mode specifies the database dialect with which the Advanced Server installation is compatible. The optional values are `oracle` or `postgresql`.

```
Please enter database mode: [ oracle ] :
```

Compatible with Oracle Mode

Specify `oracle` mode to include the following functionality:

- Data dictionary views and data type conversions compatible with Oracle databases.
- Date values displayed in a format compatible with Oracle syntax.
- 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 Oracle built-in packages.

If you choose to install in `Compatible with Oracle` mode, the Advanced Server superuser name is `enterprisedb`.

Compatible with PostgreSQL Mode

Specify `postgresql` to install Advanced Server with complete compatibility with Postgres version 11.

For more information about PostgreSQL functionality, see the PostgreSQL Core Documentation available at:

https://www.enterprisedb.com/docs

If you choose to install in `Compatible with PostgreSQL` mode, the Advanced Server superuser name is `postgres`.

Specify a port number for the Advanced Server listener to listen on:

```
NOTE: We will not be able to examine, if port is currently used by other application.
Please enter port: [ 5444 ] :
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Specify a locale for the Advanced Server installation. If you accept the `DEFAULT` value, the locale defaults to the locale of the host system.

```
Please enter the locale: [ DEFAULT ] :
```

You can optionally install sample tables and procedures. Press `Return`, or enter `Y` to accept the default and install the sample tables and procedures; enter an `n` and press `Return` to skip this step.

```
Install sample tables and procedures? (Y/n): [ Y ] :
```

Specify a password for the database superuser. By default, the database superuser is named `enterprisedb`.

```
Please enter the password for the SuperUser(enterprisedb): [ ] :
```

Specify a password for the service account user.

```
Please enter the password for the ServiceAccount(enterprisedb): [ ] :
```

The server utilization value is used as an initial value for the `edb_dynatune` configuration parameter. `edb_dynatune` determines how Advanced Server allocates system resources.

- A low value dedicates the least amount of the host machine's resources to the database server; a low value is a good choice for a development machine.
- A mid-range value dedicates a moderate amount of system resources to the database server. A mid-range value is a good setting for an application server with a fixed number of applications running on the same host as Advanced Server.
- A high value dedicates most of the system resources to the database server. This is a good choice for a dedicated server host.

Specify a value between `1` and `100`:

```
Please enter the Server Utilization: [ 66 ] :
```

After 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 Advanced Server installation. After editing the `postgresql.conf` file, you must restart the server for the changes to take effect.

The workload profile value is used as an initial value for the `edb_dynatune_profile` configuration parameter. `edb_dynatune_profile` controls performance-tuning based on the type of work that the server performs.

- Specify `oltp` if the server will be supporting heavy online transaction workloads.
- Specify `mixed` if the server will provide a mix of transaction processing and data reporting.
- Specify `reporting` if the database server will be used for heavy data reporting.

Specify a value between `1` and `100`:

```
Please enter the Workload Profile: [ oltp ] :
```

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 Advanced Server installation, and restarting the server.

After processing, the Advanced Server installation is complete.

5.3.2.2.3 Reference - Command Line Options

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

You can optionally include the following parameters for an Advanced Server installation on the command line, or in a configuration file when invoking the Advanced Server installer.

`--create_samples { yes | no }`

Use the `--create_samples` option to specify whether the installer should create the sample tables and procedures for the database dialect specified with the `--databasemode` parameter. The default is `yes`.

`--databasemode { oracle | postgresql }`

Use the `--databasemode` parameter to specify a database dialect. The default is `oracle`.

`--datadir data_directory`

Use the `--datadir` parameter to specify a location for the cluster's data directory. `data_directory` is the name of the directory; include the complete path to the desired directory.

`--debuglevel { 0 | 1 | 2 | 3 | 4 }`

Use the `--debuglevel` parameter to set 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`

Use the `--debugtrace` parameter to troubleshoot installation problems. `debug_log` is the name of the file that contains troubleshooting details.

`--disable-components component_list`

Use the `--disable-components` parameter to specify a list of 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 11.

`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 instructs the installer to grant permission to the user specified by the `--serviceaccount` option to access the 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**
>
> Specification of this option is valid only when installing on Windows. The `--enable_acledit 1` option should be specified when a `discretionary access control list` (DACL) needs to be set for allowing access to objects on which Advanced Server is to be installed. See the following for information on a DACL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa446597(v=vs.85).aspx

In order to perform future operations such as upgrading Advanced Server, access to the `data` directory must exist for the service account user specified by the `--serviceaccount` option. By specifying the `--enable_acledit 1` option, access to the `data` directory by the service account user is provided.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`--enable-components component_list`

Although this option is listed when you run the installer with the `--help` option, the `--enable-components` parameter has absolutely no effect on which components are installed. All components will be installed regardless of what is specified in `component_list`. In order to install only specific selected components, you must use the `--disable-components` parameter previously described in this section to list the components you do not want to install.

`--extract-only { yes | no }`

Include the `--extract-only` parameter to indicate that the installer should extract the Advanced Server binaries without performing a complete installation. Superuser privileges are not required for the `--extract-only` option. The default value is `no`.

`--help`

Include the `--help` parameter to view a list of the optional parameters.

`--installer-language { en | ja | zh_CN | zh_TW | ko }`

Use the `--installer-language` parameter to specify an installation language for Advanced Server. The default is `en`.

`en` specifies English.

`ja` specifies Japanese

`zh_CN` specifies Chinese Simplified.

`zh_TW` specifies Traditional Chinese.

`ko` specifies Korean.

`--install_runtimes { yes | no }`

Include `--install_runtimes` to specify whether the installer should install the Microsoft Visual C++ runtime libraries. Default is `yes`.

`--locale locale`

Specifies the locale for the Advanced Server cluster. By default, the installer will use to the locale detected by `initdb`.

`--mode { unattended }`

Use the `--mode unattended` parameter to specify that the installer should perform an installation that requires no user input during the installation process.

`--optionfile config_file`

Use the `--optionfile` parameter to specify 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/as11.x`

Use the `--prefix` parameter to specify an installation directory for Advanced Server. The installer will append a version-specific sub-directory (that is, `as11`) to the specified directory. The default installation directory is:

```
C:\Program Files\edb\as11
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`--serverport port_number`

Use the `--serverport` parameter to specify a listener port number for Advanced Server.

If you are installing Advanced Server in unattended mode, and do not specify a value using the `--serverport` parameter, the installer will use port `5444`, or the first available port after port `5444` as the default listener port.

`--server_utilization {33 | 66 | 100}`

Use the `--server_utilization` parameter to specify a value for the `edb_dynatune` configuration parameter. The `edb_dynatune` configuration parameter determines how 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 mid-range value dedicates a moderate amount of system resources to the database server. The default value is 66.
- A value of `100` is appropriate for a host machine that is dedicated to running 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 Advanced Server installation. After editing the `postgresql.conf` file, you must restart the server for the changes to take effect.

`--serviceaccount user_account_name`

Use the `--serviceaccount` parameter to specify 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 set to `postgres`.

Please note that for security reasons, the `--serviceaccount` parameter must specify the name of an account that does not hold administrator privileges.

If you specify both the `--serviceaccount` option and the `--enable_acledit 1` option when invoking the installer, the database service and pgAgent will use the same service account, thereby having the required permissions to access the Advanced Server binaries and `data` directory.

> **Note**
>
> - If you are installing on a Windows host, see the `--enable_acledit` option in this section for additional information. Specification of the `-- enable_acledit` option is permitted only when installing on Windows.
> - If you do not include the `--serviceaccount` option when invoking the installer, the `NetworkService` account will own the database service, and the pgAgent service will be owned by either `enterprisedb` or `postgres` (depending on the installation mode).

`--servicename service_name`

Use the `--servicename` parameter to specify the name of the Advanced Server service. The default is `edb-as-11`.

`--servicepassword user_password`

Use `--servicepassword` to specify the OS system password. If unspecified, the value of `--servicepassword` defaults to the value of `--superpassword`.

`--superaccount super_user_name`

Use the `--superaccount` parameter to specify the user name of the database superuser.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- 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 set to `postgres`.

`--superpassword superuser_password`

Use `--superpassword` to specify the database superuser password. If you are installing in non-interactive mode, `--superpassword` defaults to `enterprisedb`.

`--unattendedmodeui { none | minimal | minimalWithDialogs }`

Use the `--unattendedmodeui` parameter to specify installer behavior during an unattended installation.

Include `--unattendedmodeui none` to specify that the installer should not display progress bars during the Advanced Server installation.

Include `--unattendedmodeui minimal` to specify that the installer should display progress bars during the installation process. This is the default behavior.

Include `--unattendedmodeui minimalWithDialogs` to specify that the installer should display progress bars and report any errors encountered during the installation process (in additional dialogs).

`--version`

Include the `--version` parameter to retrieve version information about the installer:

```
EDB Postgres Advanced Server EDB Postgres Advanced Server 11 --- Built on 2018-03-15 00:04:00 IB: 15.11.1-201511121057
```

`--workload_profile {oltp | mixed | reporting}`

Use the `--workload_profile` parameter to specify 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 Advanced Server installation will be used to support heavy online transaction processing workloads.
- The default value is `oltp`.
- Specify `mixed` if Advanced Server will provide a mix of transaction processing and data reporting.
- Specify `reporting` if Advanced Server will be 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 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 the *EDB Postgres Advanced Server Guide* available at:

https://www.enterprisedb.com/docs

5.3.2.3 Using StackBuilder Plus

The StackBuilder Plus utility provides a graphical interface that simplifies the process of updating, downloading, and installing modules that complement your Advanced Server installation. When you install a module with StackBuilder Plus, StackBuilder Plus automatically resolves any software dependencies.

You can invoke StackBuilder Plus at any time after the installation has completed by selecting the `StackBuilder Plus` menu option from the `Apps` menu. Enter your system password (if prompted), and the StackBuilder Plus welcome window opens.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

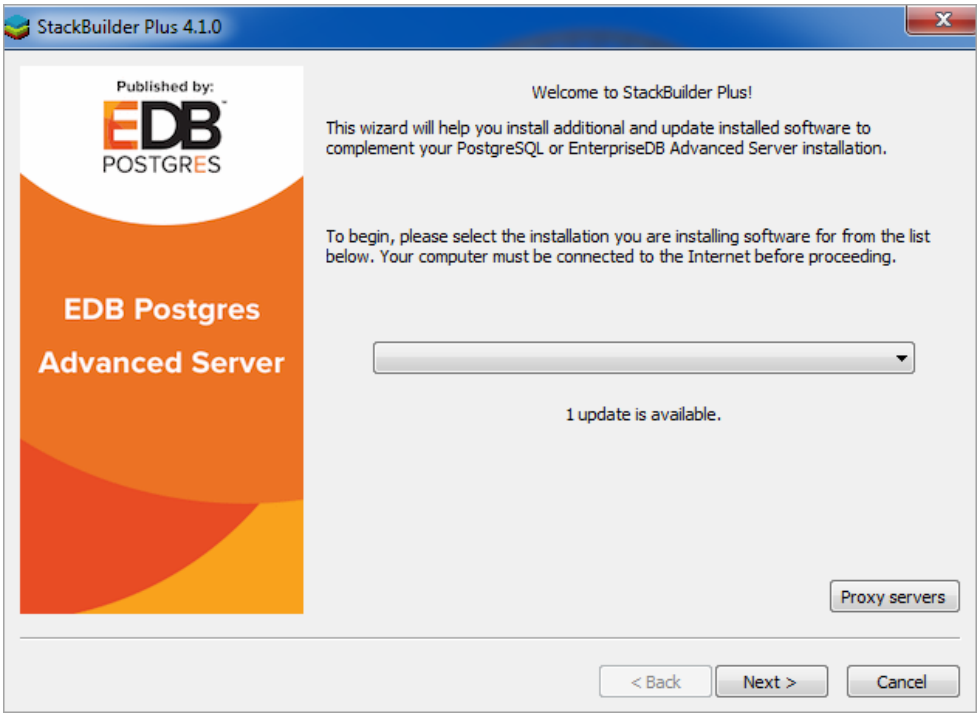


Fig. 20: The StackBuilder Plus welcome window

Use the drop-down listbox on the welcome window to select your Advanced Server installation.

StackBuilder Plus requires Internet access; if your installation of Advanced Server resides 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 via an HTTP proxy and the FTP proxy information is not used.

If the selected Advanced Server installation has restricted Internet access, use the `Proxy Servers` on the `Welcome` window to open the `Proxy servers` dialog.

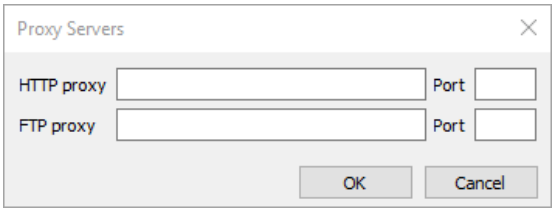


Fig. 21: The Proxy Servers dialog

Enter the IP address and port number of the proxy server in the `HTTP proxy` on the `Proxy Servers` dialog. Currently, all StackBuilder Plus modules are distributed via HTTP proxy (FTP proxy information is ignored). Click `OK` to continue.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

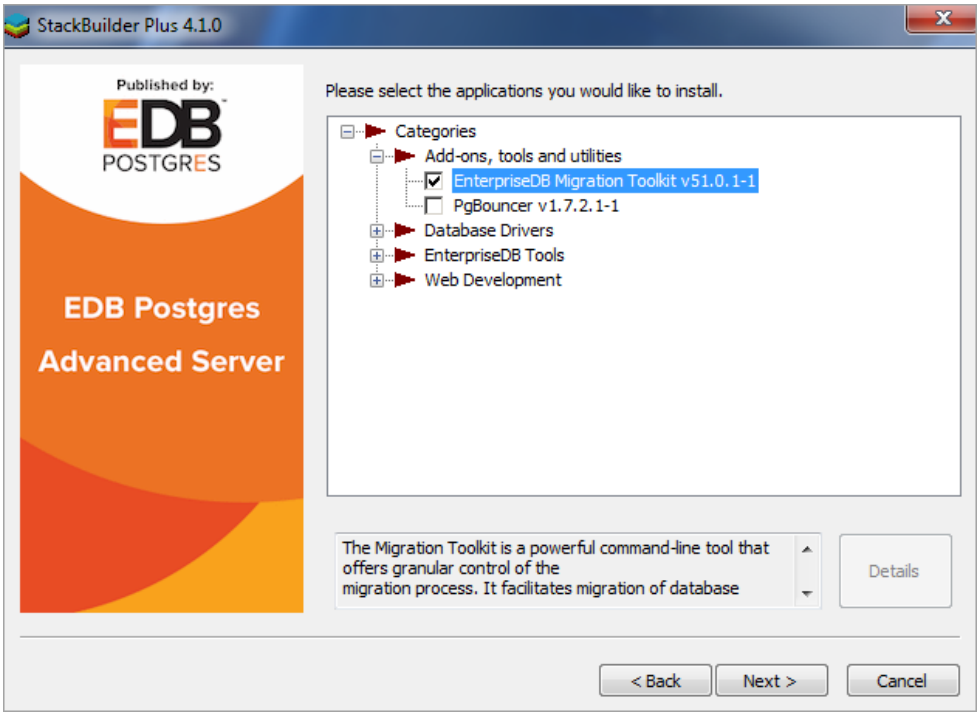


Fig. 22: The StackBuilder Plus module selection window

The tree control on the StackBuilder Plus module selection window displays a node for each module category.

Expand a module, and highlight a component name in the tree control to review a detailed description of the component. To add one or more components to the installation or to upgrade a component, check the box to the left of a module name and click `Next`.

StackBuilder Plus confirms the packages selected.

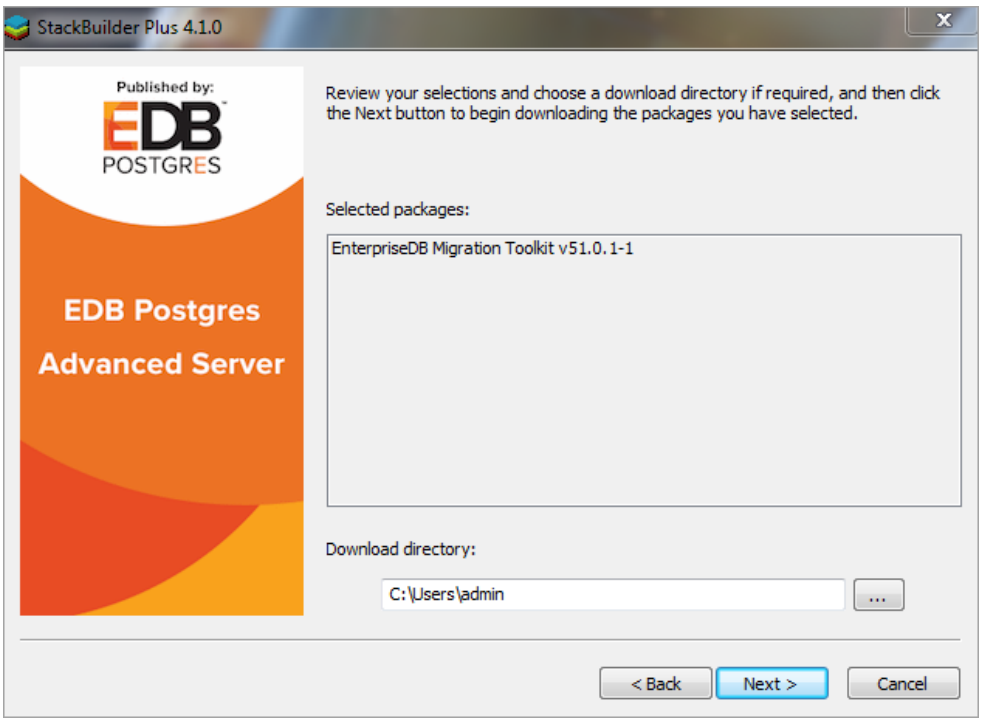


Fig. 23: A summary window displays a list of selected packages

Use the browse icon (...) to the right of the `Download directory` field to open a file selector, and choose an alternate location to store the downloaded installers. Click `Next` to connect to the server and download the required installation files.

When the download completes, a window opens that confirms the installation files have been downloaded and are ready for installation.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

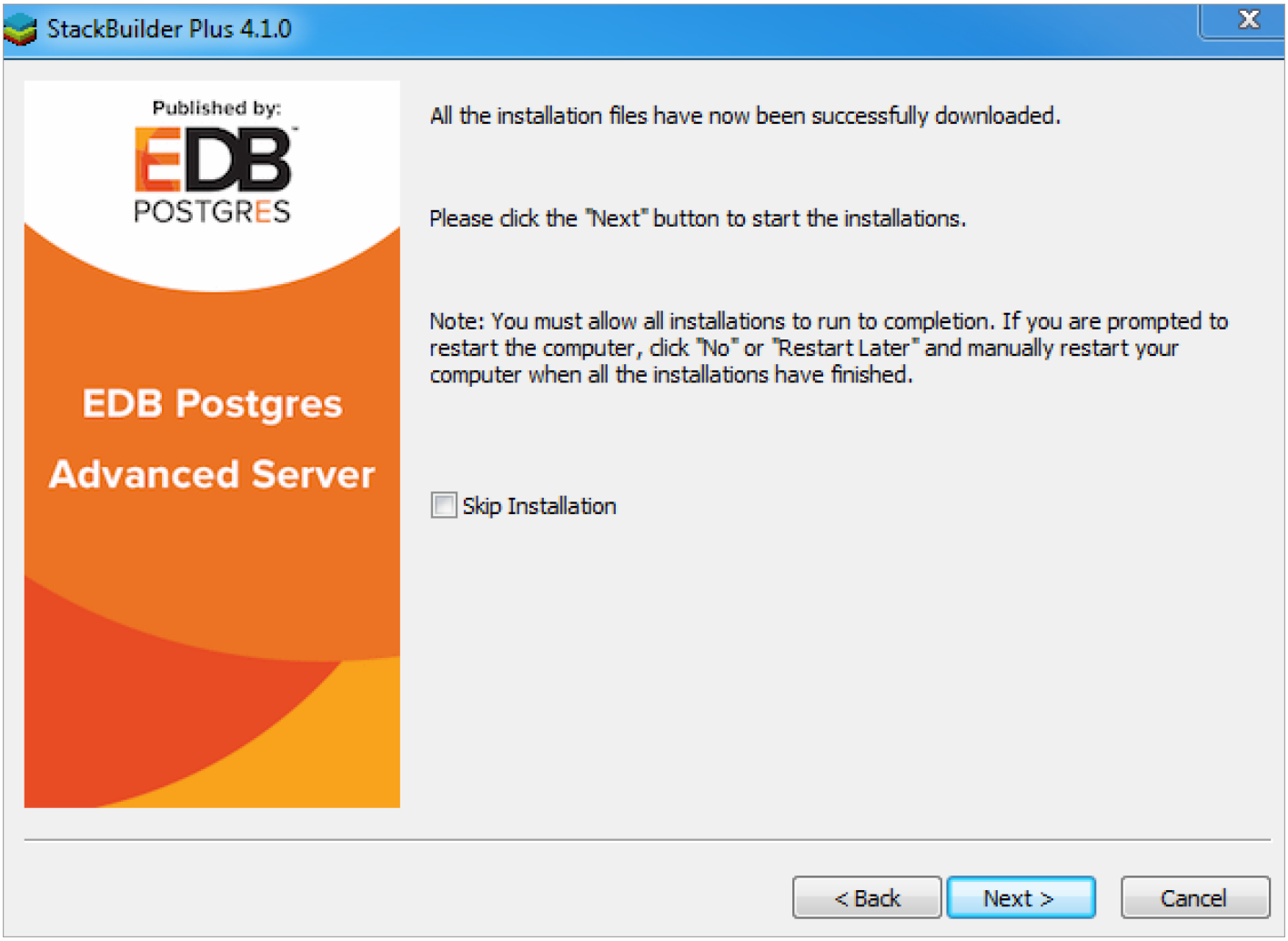


Fig. 24: Confirmation that the download process is complete

You can check the box next to `Skip Installation`, and select `Next` to exit StackBuilder Plus without installing the downloaded files, or leave the box unchecked and click `Next` to start the installation process.

Each downloaded installer has different requirements. As the installers execute, they may prompt you to confirm acceptance of license agreements, to enter passwords, and provide configuration information.

During the installation process, you may 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, reboot the system to apply all of the updates.

You may occasionally encounter packages that don't install successfully. If a package fails to install, StackBuilder Plus will alert you to the installation error with a popup dialog, and write a message to the log file at `%TEMP%`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

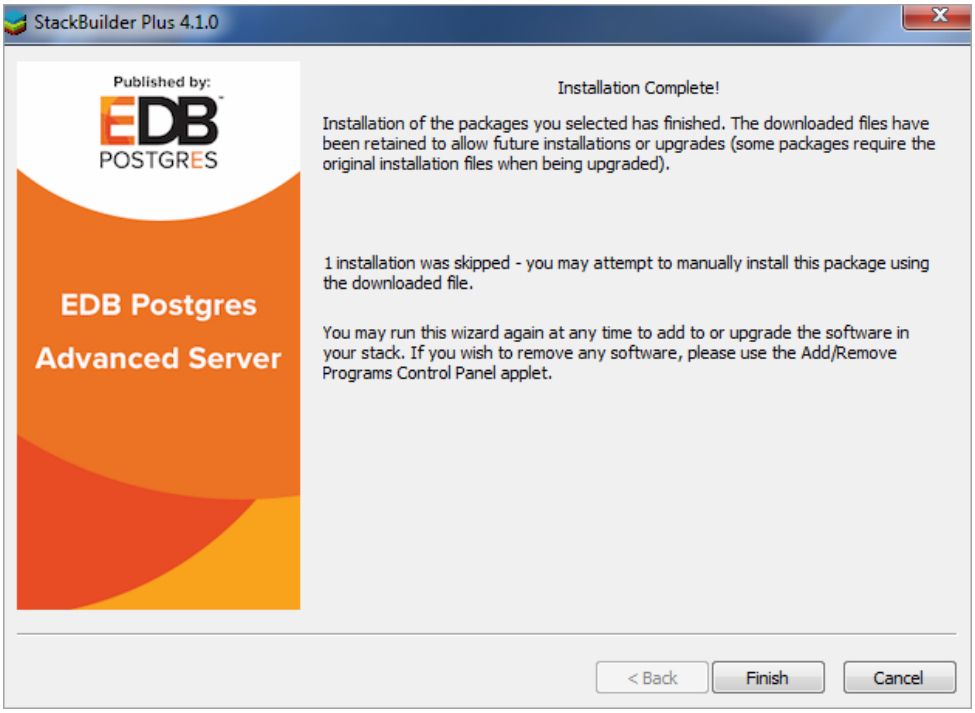


Fig. 25: StackBuilder Plus confirms the completed installation

When the installation is complete, StackBuilder Plus will alert you to the success or failure of the installations of the requested packages. If you were prompted by an installer to restart your computer, reboot now.

5.4 Linux installation details

If you need access to the EDB Postgres Advanced Server components after installation, see Default component locations.

For information on available native packages from the EDB repository, see Available native packages.

After installing, there are various procedures you may need for Managing the Advanced Server installation.

5.4.1 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 below:

Component	Location

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Component	Location
Executables	`/usr/edb/as11/bin`
Libraries	`/usr/edb/as11/lib`
Cluster configuration files	`/var/lib/edb/as11`
Documentation	`/usr/edb/as11/share/doc`
Contrib	`/usr/edb/as11/share/contrib`
Data	`/var/lib/edb/as11/data`
Logs	`/var/log/edb/as11`
Lock files	`/var/lock/edb/as11`
Backup area	`/var/lib/edb/as11/backups`
Templates	`/usr/edb/as11/share`
Procedural Languages	`/usr/edb/as11/lib or /usr/edb/as11/lib64`
Development Headers	`/usr/edb/as11/include`
Shared data	`/usr/edb/as11/share`
Regression tests	`/usr/edb/as11/lib/pgxs/src/test/regress`
SGML Documentation	`/usr/edb/as11/share/doc`

Debian/Ubuntu Locations

The Debian package manager places EDB Postgres Advanced Server and supporting components in the directories listed in the following table:

Component	Location
Server	`/usr/lib/edb-as/11/`
Data and Configuration Directory	`/var/lib/edb-as/11/main` `/etc/edb-as/11/main/`
pgAgent	`/usr/lib/edb-as/11`
Pgpool	`/usr/edb/pgpool4.1/`
Postgis	`/usr/lib/edb-as/11/`
PGSNMPD	`/usr/lib/edb-as/11`
Slony Replication	`/usr/lib/edb-as/11`
pgBouncer	`/usr/edb/pgbouncer1.11/`
pgBouncer Configuration Files	`/etc/edb/pgbouncer1.11/pgbouncer.ini` `/etc/edb/pgbouncer1.11/userlist.txt`
SQL-Profiler	`/usr/lib/edb-as/11/lib`
SQL-Protect	`/usr/lib/edb-as/11/lib`
SSLUTILS	`/usr/lib/edb-as/11/lib`
PL-PERL	`/usr/lib/edb-as/11/lib`
PL-PYTHON	`/usr/lib/edb-as/11/lib`
PLTCL	`/usr/lib/edb-as/11/lib`
EFM	`/usr/edb/efm-4.1/`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Component	Location
JDBC	`/usr/edb/jdbc`
MTK	`/usr/edb/migrationtoolkit/`

5.4.2 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. Open a command line, assume superuser privileges, and enter:

On RHEL or CentOS 7:

```
yum search <package>
```

On RHEL or Rocky Linux or AlmaLinux 8:

```
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.

Package name	Package installs
edb-as11-server	This package installs core components of the EDB Postgres Advanced Server database server.
edb-as11-server-client	Client programs and utilities that you can use to access and manage EDB Postgres Advanced Server.
edb-as11-server-contrib	Installs contributed tools and utilities that are distributed with EDB Postgres Advanced Server. Files for these modules are installed in: Documentation: `/usr/edb/as11/share/doc` Loadable modules: `/usr/edb/as11/lib` Binaries: `/usr/edb/as11/bin`
edb-as11-server-core	Includes the programs needed to create the core functionality behind the EDB Postgres Advanced Server database.
edb-as11-server-devel	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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Package name	Package installs
edb-as11-server-docs	Installs the readme file.
edb-as11-server-edb-modules	Installs supporting modules for EDB Postgres Advanced Server
edb-as11-server-indexadvisor	Installs EDB Postgres Advanced Server's Index Advisor feature. The Index Advisor utility helps determine which columns you should index to improve performance in a given workload.
edb-as11-server-libs	Provides the essential shared libraries for any EDB Postgres Advanced Server client program or interface.
edb-as11-server-llvmjit	This package contains support for Just in Time (JIT) compiling parts of EDBAS queries.
edb-as11-server-pldebugger	This package implements an API for debugging PL/pgSQL functions on EDB Postgres Advanced Server.
edb-as11-server-plperl	Installs the PL/Perl procedural language for EDB Postgres Advanced Server. The `edb-as11-server-plperl` package is dependent on the platform-supplied version of Perl.
edb-as11-server-plpython3	Installs the PL/Python procedural language for EDB Postgres Advanced Server. The PL/Python2 support is no longer available from EDB Postgres Advanced Server version 11 onwards.
edb-as11-server-pltcl	Installs the PL/Tcl procedural language for EDB Postgres Advanced Server. The `edb-as11-pltcl` package is dependent on the platform-supplied version of TCL.
edb-as11-server-sqlprofiler	This package installs EDB Postgres Advanced Server's SQL Profiler feature. SQL Profiler helps identify and optimize SQL code.
edb-as11-server-sqlprotect	This package installs EDB Postgres Advanced Server's SQL Protect feature. SQL Protect provides protection against SQL injection attacks.
edb-as11-server-sslutils	This package installs functionality that provides SSL support.
edb-as11-server-cloneschema	This package installs the EDB Clone Schema extension. For more information about EDB Clone Schema, see the*EDB Postgres Advanced Server Guide*.
edb-as11-server-parallel-clone	This package installs functionality that supports the EDB Clone Schema extension.
edb-as11-pgagent	Installs pgAgent; pgAgent is a job scheduler for EDB Postgres Advanced Server. Before installing this package, you must install EPEL; for detailed information about installing EPEL, see Troubleshooting an installation.
edb-as11-edbplus	The `edb-edbplus` package contains the files required to install the EDB*Plus command line client. EDB*Plus commands are compatible with Oracle's SQL*Plus.
edb-as11-pgsnmpd	SNMP (Simple Network Management Protocol) is a protocol that allows you to supervise an apparatus connected to the network.
edb-as11-pgpool41-extensions	This package creates pgPool extensions required by the server for use with pgpool.
edb-as11-postgis3	Installs POSTGIS meta RPMs.
edb-as11-postgis3-core	This package provides support for geographic objects to the PostgreSQL object-relational database. In effect, PostGIS "spatially enables" 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.
edb-as11-postgis3-docs	This package installs pdf documentation of PostGIS.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Package name	Package installs
edb-as11-postgis-jdbc	This package installs the essential jdbc driver for PostGIS.
edb-as11-postgis3-utils	This package installs the utilities for PostGIS.
edb-as11-postgis3-gui	This package provides a GUI for PostGIS.
edb-as11-slony-replication	Installs the meta RPM for Slony-I.
edb-as11-slony-replication-core	Slony-I builds a primary-standby system that includes all features and capabilities needed to replicate large databases to a reasonably limited number of standby systems.
edb-as11-slony-replication-docs	This package contains the Slony project documentation (in pdf form).
edb-as11-slony-replication-tools	This package contains the Slony altperl tools and utilities that are useful when deploying Slony replication environments. Before installing this package, you must install EPEL; for detailed information about installing EPEL, see Troubleshooting anstallation Troubleshooting.
edb-as11-libicu	These packages contain supporting library files.

The following table lists the packages for EDB Postgres Advanced Server 11 supporting components.

Package name	Package installs
edb-pgpool41	This package 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 is not supported on Windows systems.
edb-jdbc	The `edb-jdbc` package includes the .jar files needed for Java programs to access an EDB Postgres Advanced Server database.
edb-migrationtoolkit	The `edb-migrationtoolkit` package installs Migration Toolkit, facilitating migration to an EDB Postgres Advanced Server database from Oracle, PostgreSQL, MySQL, Sybase and SQL Server.
edb-oci	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.
edb-oci-devel	This package installs the OCI include files; install this package if you are developing C/C++ applications that require these files.
edb-odbc	This package installs the driver needed for applications to access an EDB Postgres Advanced Server system via ODBC.
edb-odbc-devel	This package installs the ODBC include files; install this package if you are developing C/C++ applications that require these files.
edb-pgbouncer111	This package contains PgBouncer (a lightweight connection pooler). This package requires the libevent package.
ppas-xdb	This package contains the xDB installer; xDB provides asynchronous cross-database replication.
ppas-xdb-console	This package provides support for xDB.
ppas-xdb-libs	This package provides support for xDB.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Package name	Package installs
ppas-xdb-publisher	This package provides support for xDB.
ppas-xdb-subscriber	This package provides support for xDB.
edb-pem	The `edb-pem` package installs Management Tool that efficiently manages, monitor, and tune large Postgres deployments from a single remote GUI console.
edb-pem-agent	This package is an agent component of Postgres Enterprise Manager.
edb-pem-docs	This package contains documentation for various languages, which are in HTML format.
edb-pem-server	This package contains server components of Postgres Enterprise Manager.
edb-pgadmin4	This package is a management tool for PostgreSQL capable of hosting the Python application and presenting it to the user as a desktop application.
edb-pgadmin4-desktop-common	This package installs the desktop components of pgAdmin4 for all window managers.
edb-pgadmin4-desktop-gnome	This package installs the gnome desktop components of pgAdmin4
edb-pgadmin4-docs	This package contains documentation of pgAdmin4.
edb-pgadmin4-web	This package contains the required files to run pgAdmin4 as a web application.
edb-efm40	This package 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.
edb-rs	This package is a java-based replication framework that provides asynchronous replication across Postgres and EPAS database servers. It supports primary-standby, primary-primary, and hybrid configurations.
edb-rs-client	This package is a java-based command-line tool that is used to configure and operate a replication network via different commands by interacting with the EPRS server.
edb-rs-datavalidator	This package is 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, EPAS, Oracle, and MS SQL Server.
edb-rs-libs	This package contains certain libraries that are commonly used by ERPS Server, EPRS Client, and Monitoring modules.
edb-rs-monitor	This package is a java-based application that provides monitoring capabilities to ensure a smooth functioning of the EPRS replication cluster.
edb-rs-server	This package is a java-based replication framework that provides asynchronous replication across Postgres and EPAS database servers. It supports primary-standby, primary-primary, and hybrid configurations.
edb-bart	This package installs the Backup and Recovery Tool (BART) to support online backup and recovery across local and remote PostgreSQL and EDB EDB Postgres Advanced Servers.
libevent-edb	This package contains supporting library files.
libiconv-edb	This package contains supporting library files.
libevent-edb-devel	This package contains supporting library files.

Debian/Ubuntu Packages

EDB Postgres Advanced Server Debian packages

The table that follows 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. Open a command line, assume superuser privileges, and enter:

```
apt list edb*
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> **Note**
>
> The available package list is subject to change.

Package name	Package installs
edb-as11-server	Installs core components of the EDB Postgres Advanced Server database server.
edb-as11-server-client	Includes client programs and utilities that you can use to access and manage EDB Postgres Advanced Server.
edb-as11-server-core	Includes the programs needed to create the core functionality behind the EDB Postgres Advanced Server database.
edb-as11-server-dev	The `edb-as11-server-dev` package 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.
edb-as11-server-doc	Installs the readme file.
edb-as11-server-edb-modules	Installs supporting modules for EDB Postgres Advanced Server.
edb-as11-server-indexadvisor	Installs EDB Postgres Advanced Server's Index Advisor feature. The Index Advisor utility helps determine which columns you should index to improve performance in a given workload.
edb-as11-server-pldebugger	This package implements an API for debugging PL/pgSQL functions on EDB Postgres Advanced Server.
edb-as11-server-plpython3	Installs the PL/Python procedural language for EDB Postgres Advanced Server. PL/Python2 support is no longer available from EDB Postgres Advanced Server version 11 onwards.
edb-as11-server-pltcl	Installs the PL/Tcl procedural language for EDB Postgres Advanced Server. The `edb-as11-pltcl` package is dependent on the platform-supplied version of TCL.
edb-as11-server-sqlprofiler	This package installs EDB Postgres Advanced Server's SQL Profiler feature. SQL Profiler helps identify and optimize SQL code.
edb-as11-server-sqlprotect	This package installs EDB Postgres Advanced Server's SQL Protect feature. SQL Protect provides protection against SQL injection attacks.
edb-as11-server-sslutils	This package installs functionality that provides SSL support.
edb-as11-server-cloneschema	This package installs the EDB Clone Schema extension. For more information about EDB Clone Schema, see the *EDB Postgres Advanced Server Guide*.
edb-as11-server-parallel-clone	This package installs functionality that supports the EDB Clone Schema extension.
edb-as11-edbplus	The `edb-edbplus` package contains the files required to install the EDB*Plus command line client. EDB*Plus commands are compatible with Oracle's SQL*Plus.
edb-as11-pgsnmpd	SNMP (Simple Network Management Protocol) is a protocol that allows you to supervise an apparatus connected to the network.
edb-as11-pgadmin4	pgAdmin 4 provides a graphical management interface for EDB Postgres Advanced Server and PostgreSQL databases.
edb-as11-pgadmin-apache	Apache support module for pgAdmin 4.
edb-as11-pgadmin4-common	pgAdmin 4 supporting files.
edb-as11-pgadmin4-doc	pgAdmin 4 documentation module.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Package name	Package installs
edb-as11-pgpool41-extensions	This package creates pgPool extensions required by the server.
edb-as11-postgis3	This package installs POSTGIS support for geospatial data.
edb-as11-postgis3-scripts	This package installs POSTGIS support for geospatial data.
edb-as11-postgis3-doc	This package provides support for POSTGIS.
edb-as11-postgis3-gui	This package provides a GUI for POSTGIS.
edb-as11-postgis-jdbc	This package provides support for POSTGIS.
edb-as11-postgis-scripts	This package provides support for POSTGIS.
edb-as11-pgagent	This package installs pgAgent; pgAgent is a job scheduler for EDB Postgres Advanced Server. Before installing this package, you must install EPEL; for detailed information about installing EPEL, see Troubleshooting an installation.
edb-as11-slony-replication	This package installs the meta RPM for Slony-I.
edb-as11-slony-replication-core	This package contains core portions of Slony-I to build a primary-standby system that includes all features and capabilities needed to replicate large databases to a reasonably limited number of standby systems.
edb-as11-slony-replication-docs	This package contains the Slony project documentation (in pdf form).
edb-as11-slony-replication-tools	This package contains the Slony altperl tools and utilities that are useful when deploying Slony replication environments. Before installing this package, you must install EPEL; for detailed information about installing EPEL, see Troubleshooting an installation.
edb-as11-hdfs-fdw	The Hadoop Data Adapter allows you to query and join data from Hadoop environments with your Postgres or EDB Postgres Advanced Server instances. It is YARN Ready certified with HortonWorks, and provides optimizations for performance with predicate pushdown support.
edb-as11-hdfs-fdw-doc	Documentation for the Hadoop Data Adapter.
edb-as11-mongo-fdw	This EDB EDB Postgres Advanced Server extension implements a Foreign Data Wrapper for MongoDB.
edb-as11-mongo-fdw-doc	Documentation for the Foreign Data Wrapper for MongoDB.
edb-as11-mysql-fdw	This EDB EDB Postgres Advanced Server extension implements a Foreign Data Wrapper for MySQL.
edb-pgpool41	This package 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 is not supported on Windows systems.
edb-jdbc	The `edb-jdbc` package includes the .jar files needed for Java programs to access an EDB Postgres Advanced Server database.
edb-migrationtoolkit	The `edb-migrationtoolkit` package installs Migration Toolkit, facilitating migration to an EDB Postgres Advanced Server database from Oracle, PostgreSQL, MySQL, Sybase and SQL Server.
edb-pgbouncer111	This package contains PgBouncer (a lightweight connection pooler). This package requires the libevent package.
edb-efm40	This package 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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:

```
yum upgrade edb-repo
```

- On RHEL or Rocky Linux or AlmaLinux 8:

```
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` command perform an update only between minor releases; to update between major releases, you must use `pg_upgrade`.

For more information about using yum commands and options, enter `yum --help` on your command line.

For more information about using `dnf` commands and options, visit:

https://docs.fedoraproject.org/en-US/quick-docs/dnf/

5.4.3 Managing an Advanced Server Installation

Unless otherwise noted, the commands and paths noted in the following section assume that you have performed an installation using the native packages.

Starting and Stopping Advanced Server and Supporting Components

A service is a program that runs in the background and requires no user interaction (in fact, a service provides no user interface); a service can be configured 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 Advanced Server supporting components are services.

The following table lists the names of the services that control Advanced Server and services that control Advanced Server supporting components:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Advanced Server Component Name	Linux Service Name	Debian Service Name
Advanced Server	edb-as-11	edb-as@11-main
pgAgent	edb-pgagent-11	edb-as11-pgagent
PgBouncer	edb-pgbouncer-1.14	edb-pgbouncer114
pgPool-II	edb-pgpool-4.1	edb-pgpool41
Slony	edb-slony-replication-11	edb-as11-slony-replication
EFM	edb-efm-4.0	edb-efm-4.0

You can use the Linux command line to control Advanced Server's database server and the services of Advanced Server's supporting components. The commands that control the 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 Advanced Server resides on RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x, you must use the `systemctl` command to control the Advanced Server service and supporting components.

The `systemctl` command must be in your search path and must be invoked with superuser privileges. To use the command, open a command line, and 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 Ubuntu 18.04

By default, a server is running with the peer or md5 permission on a Debian or Ubuntu host. You can change the `auth_type` from `pg_hba.conf` file available under:

```
vi /etc/edb-as/11/main/pg_hba.conf
```

Please note, all the server configuration files are available under `/etc/edb-as/11/main`.

If your installation of Advanced Server resides on version 18.04 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/bin/epas_ctlcluster 11 main status
  ```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Stop a service:

```
/usr/bin/epas_ctlcluster 11 main stop
```

- Restart a service:

```
/usr/bin/epas_ctlcluster 11 main restart
```

- Reload a service:

```
/usr/bin/epas_ctlcluster 11 main reload
```

- Control the component services:

```
systemctl restart edb-as@11-main
```

Using pg_ctl to Control Advanced Server

You can use the `pg_ctl` utility to control an Advanced Server service from the command line on any platform. `pg_ctl` allows you to start, stop, or restart the 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 Advanced Server, and issue the command:

```
./bin/pg_ctl -D <data_directory> <action>
```

`data_directory` is the location of the data controlled by the 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` sends 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 official PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/11/static/app-pg-ctl.html

Choosing Between pg_ctl and the service Command

You can use the `pg_ctl` utility to manage the status of an Advanced Server cluster, but it is important to note that `pg_ctl` does not alert the operating system service controller to changes in the status of a server, so it is beneficial to use the `service` command whenever possible.

Configuring Component Services to AutoStart at System Reboot

After installing, configuring, and starting the services of Advanced Server supporting components on a Linux system, you must manually configure your system to autostart the service when your system reboots. To configure a service to autostart on a Linux system, open a command line, assume superuser privileges, and enter the following command.

On a Redhat-compatible Linux system, enter:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
/sbin/chkconfig <service_name> on
```

Where `service_name` specifies the name of the service.

Connecting to Advanced Server with psql

`psql` is a command line client application that allows you to execute SQL commands and view the results. To open the psql client, the client must be in your search path. The executable resides in the `bin` directory, under your Advanced Server installation.

Use the following command and options to start the psql client:

```
psql -d edb -U enterprisedb
```

Where:

`-d` specifies the database to which psql will connect.

`-U` specifies the identity of the database user that will be used for the session.

For more information about using the command line client, see the PostgreSQL Core Documentation at:

https://www.postgresql.org/docs/11/static/app-psql.html

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. After installing 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.

To review the `initdb` documentation, visit:

https://www.postgresql.org/docs/11/static/app-initdb.html

After specifying any options in the service configuration file, you can create the database cluster and start the service; these 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 RHEL/Rocky Linux/AlmaLinux 8.x system, with the options specified in the service configuration file, assume the identity of the operating system superuser:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
su - root
```

To initialize a cluster with the non-default values, you can use the `PGSETUP_INITDB_OPTIONS` environment variable by invoking the `edb-as-11-setup` cluster initialization script that resides under `EPAS_Home/bin`.

To invoke `initdb` export the `PGSETUP_INITDB_OPTIONS` environment variable with the following command:

```
PGSETUP_INITDB_OPTIONS="-E UTF-8" /usr/edb/as11/bin/edb-as-11-setup initdb
```

After creating the cluster, use `systemctl` to start, stop, or restart the service:

```
systemctl { start | stop | restart } edb-as-11
```

On Ubuntu 18.04

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 11 main2
```

To start a new cluster, use the following command:

```
/usr/bin/epas_ctlcluster 11 main2 start
```

To list all the available clusters, use the following command:

```
/usr/bin/epas_lsclusters
```

> **Note**
>
> The data directory is created under `/var/lib/edb-as/11/main2` and configuration directory is created under `/etc/edb-as/11/main/`.

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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Package	Functionality compatible with Oracle databases
`dbms_alert`	Provides the capability to register for, send, and receive alerts.
`dbms_job`	Provides the capability to create, schedule, and manage jobs.
`dbms_lob`	Provides the capability to manage on large objects.
`dbms_output`	Provides the capability to send messages to a message buffer or get messages from the message buffer.
`dbms_pipe`	Provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.
`dbms_rls`	Enables the implementation of Virtual Private Database on certain EDB Postgres Advanced Server database objects.
`dbms_sql`	Provides an application interface to the EDB dynamic SQL functionality.
`dbms_utility`	Provides various utility programs.
`dbms_aqadm`	Provides supporting procedures for Advanced Queueing functionality.
`dbms_aq`	Provides message queueing and processing for EDB Postgres Advanced Server.
`dbms_profiler`	Collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a performance profiling session.
`dbms_random`	Provides a number of methods to generate random values.
`dbms_redact`	Enables redacting or masking data that's returned by a query.
`dbms_lock`	Provides support for the `DBMS_LOCK.SLEEP` procedure.
`dbms_scheduler`	Provides a way to create and manage jobs, programs, and job schedules.
`dbms_crypto`	Provides functions and procedures to encrypt or decrypt RAW, BLOB or CLOB data. You can also use `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 `DBMS_SESSION.SET_ROLE` procedure.
`utl_encode`	Provides a way to encode and decode data.
`utl_http`	Provides a way to use the HTTP or HTTPS protocol to retrieve information found at an URL.
`utl_file`	Provides the capability to read from and write to files on the operating system's file system.
`utl_smtp`	Provides the capability to send e-mails over the Simple Mail Transfer Protocol (SMTP).
`utl_mail`	Provides the capability to manage email.
`utl_url`	Provides a way to escape illegal and reserved characters in a URL.
`utl_raw`	Provides a way to manipulate or retrieve the length of raw data types.

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, enabling 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 package:

Package	Functionality noncompatible with Oracle databases
`dbms_aqadm`	Provides supporting procedures for Advanced Queueing functionality.
`dbms_aq`	Provides message queueing and processing for EDB Postgres Advanced Server.
`edb_bulkload`	Provides direct/conventional data loading capability when loading huge amount of data into a database.
`edb_gen`	Provides miscellaneous packages to run built-in packages.
`edb_objects`	Provides Oracle-compatible objects such as packages and procedures.
`waitstates`	Provides monitor session blocking.
`edb_dblink_libpq`	Provides link to foreign databases via libpq.
`edb_dblink_oci`	Provides link to foreign databases via OCI.
`snap_tables`	Creates tables to hold wait information. Included with DRITA scripts.
`snap_functions`	Creates functions to return a list of snap ids and the time the snapshot was taken. Included with DRITA scripts.
`sys_stats`	Provides OS performance statistics.

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, that is, an empty string (''), following 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 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.

Modifying the Data Directory Location on RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

On a RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x host, the unit file is named `edb-as-11.service` and resides in `/usr/lib/systemd/system`. The unit file contains references to the location of the Advanced Server `data` directory. You should avoid making any modifications directly to the unit file because it may be overwritten during package upgrades.

By default, data files reside under `/var/lib/edb/as11/data` directory. To use a data directory that resides in a non-default location, perform the following steps:

- Create a copy of the unit file under the `/etc` directory:

```
cp /usr/lib/systemd/system/edb-as-11.service /etc/systemd/system/
```

- After copying the unit file to the new location, modify the service file `/etc/systemd/system/edb-as-11.service` with your editor of choice, correcting any required paths.

- Use the following command to reload `systemd`, updating the modified service scripts:

```
systemctl daemon-reload
```

- Start the Advanced Server service with the following command:

```
systemctl start edb-as-11
```

Starting Multiple Postmasters with Different Clusters

You can configure Advanced Server to use multiple postmasters, each with its own database cluster. The steps required are version specific to the Linux host.

On RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x

The `edb-as11-server-core` RPM for version 7.x | 8.x contains a unit file that starts the Advanced Server instance. The file allows you to start multiple services, with unique `data` directories and that monitor different ports. You must have `root` access to invoke or modify the script.

The example that follows creates an 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 is not already used in `/etc/systemd/system`.

```
cp /usr/lib/systemd/system/edb-as-11.service /etc/systemd/system/secondary-edb-as-11.service
```

- Edit the file, changing `PGDATA` to point to the new `data` directory that you will create the cluster against.

- Create the target `PGDATA` with user `enterprisedb`.

- Run `initdb`, specifying the setup script:

```
/usr/edb/as11/bin/edb-as-11-setup initdb secondary-edb-as-11
```

- Edit the `postgresql.conf` file for the new instance, specifying the port, the IP address, TCP/IP settings, etc.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Make sure that new cluster runs after a reboot:

```
systemctl enable secondary-edb-as-11
```

- Start the second cluster with the following command:

```
systemctl start secondary-edb-as-11
```

Creating an Advanced Server Repository on an Isolated Network

You can create a local repository to act as a host for the Advanced Server RPM packages if the server on which you wish to install Advanced Server (or supporting components) cannot directly access the EnterpriseDB repository. Please note that this is a high-level listing of the steps requires; you will need to modify the process for your individual network.

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/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 EnterpriseDB repository at https://repos.enterprisedb.com/.

- Sync the RPM packages and create the repository.

```
reposync -r edbas11 -p /srv/repos
createrepo /srv/repos
```

- Install your preferred webserver on the host that will act 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:

```
[edbas11]
name=EnterpriseDB Advanced Server 11
baseurl=https://yum.your_domain.com/edbas11
enabled=1
gpgcheck=0
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

After specifying the location and connection information for your local repository, you can use `yum` or `dnf` commands to install Advanced Server and its supporting components on the isolated servers. For example:

- On RHEL or CentOS 7:

```
yum -y install edb-as11-server
```

- On RHEL or Rocky Linux/AlmaLinux 8:

```
dnf -y install edb-as11-server
```

For more information about creating a local `yum` repository, visit:

https://wiki.centos.org/HowTos/CreateLocalRepos

5.5 Windows installation details

This section provides detailed information about:

- Connecting to Advanced Server with the pgAdmin 4 Client
- Managing an Advanced Server installation
- Configuring Advanced Server

5.5.1 Connecting to Advanced Server with the pgAdmin 4 Client

pgAdmin 4 provides an interactive graphical interface that you can use to manage your database and database objects. Easy-to-use dialogs 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 pgAdmin4 from the `EDB Postgres` menu. The client opens in your default browser.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

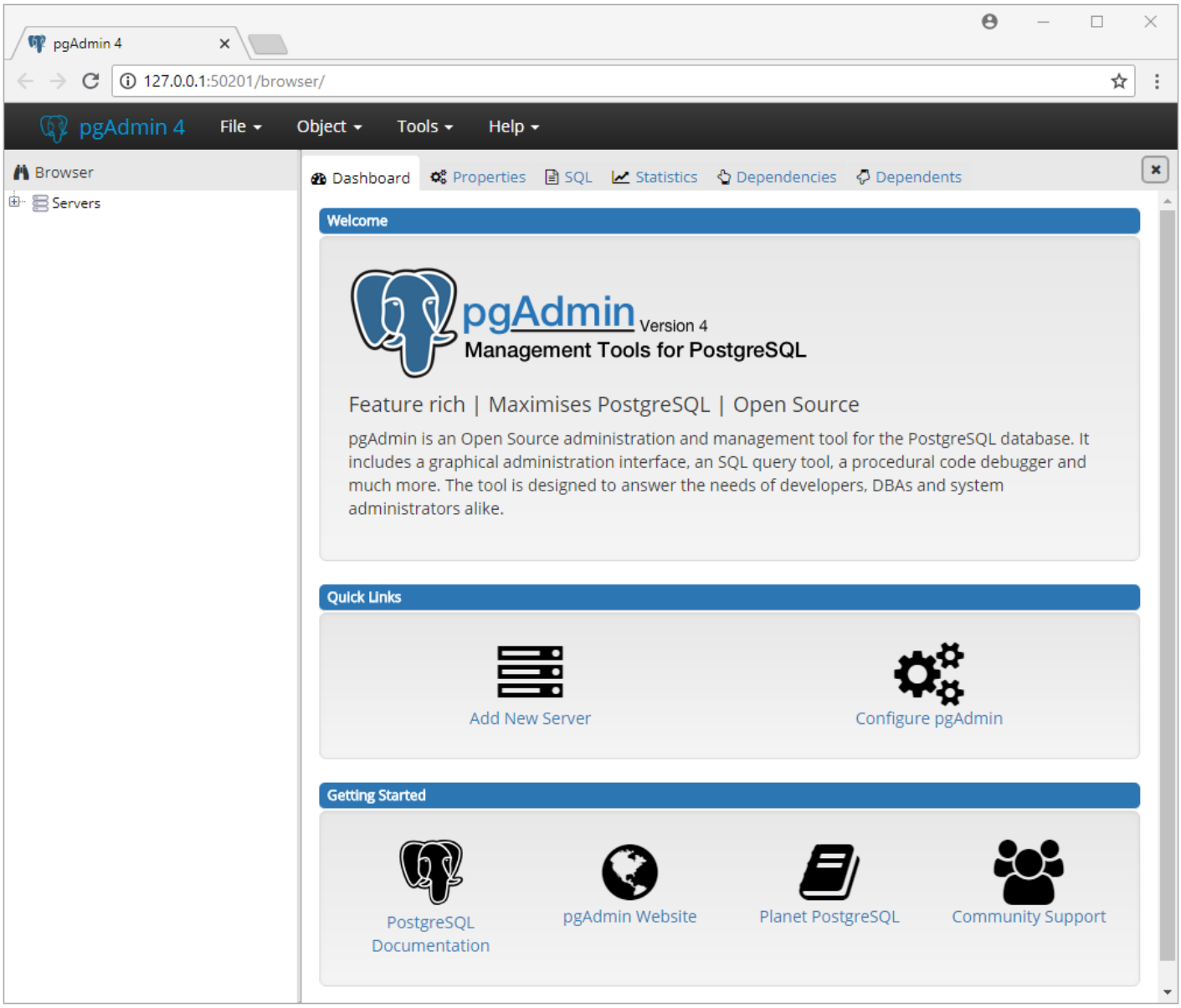


Fig. 1: The pgAdmin 4 client Dashboard

To connect to the Advanced Server database server, expand the `Servers` node of the `Browser` tree control, and right click on the `EDB Postgres Advanced Server` node. When the context menu opens, select `Connect Server`. The `Connect to Server` dialog opens.

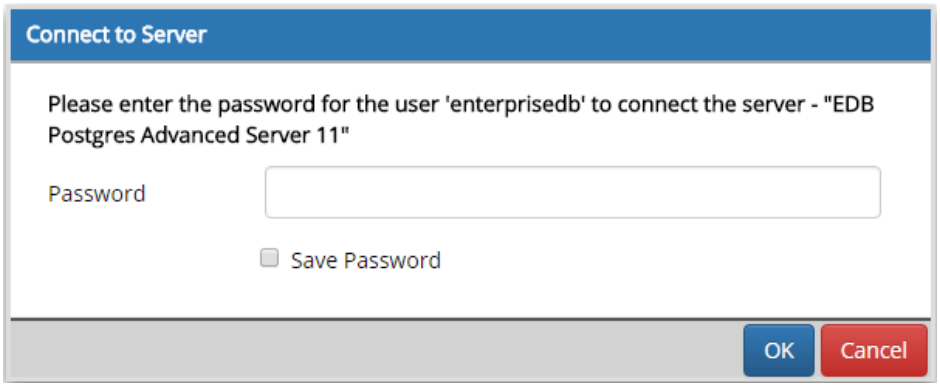


Fig. 2: The Connect to Server dialog

Provide the password associated with the database superuser in the `Password` field, and click `OK` to connect.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

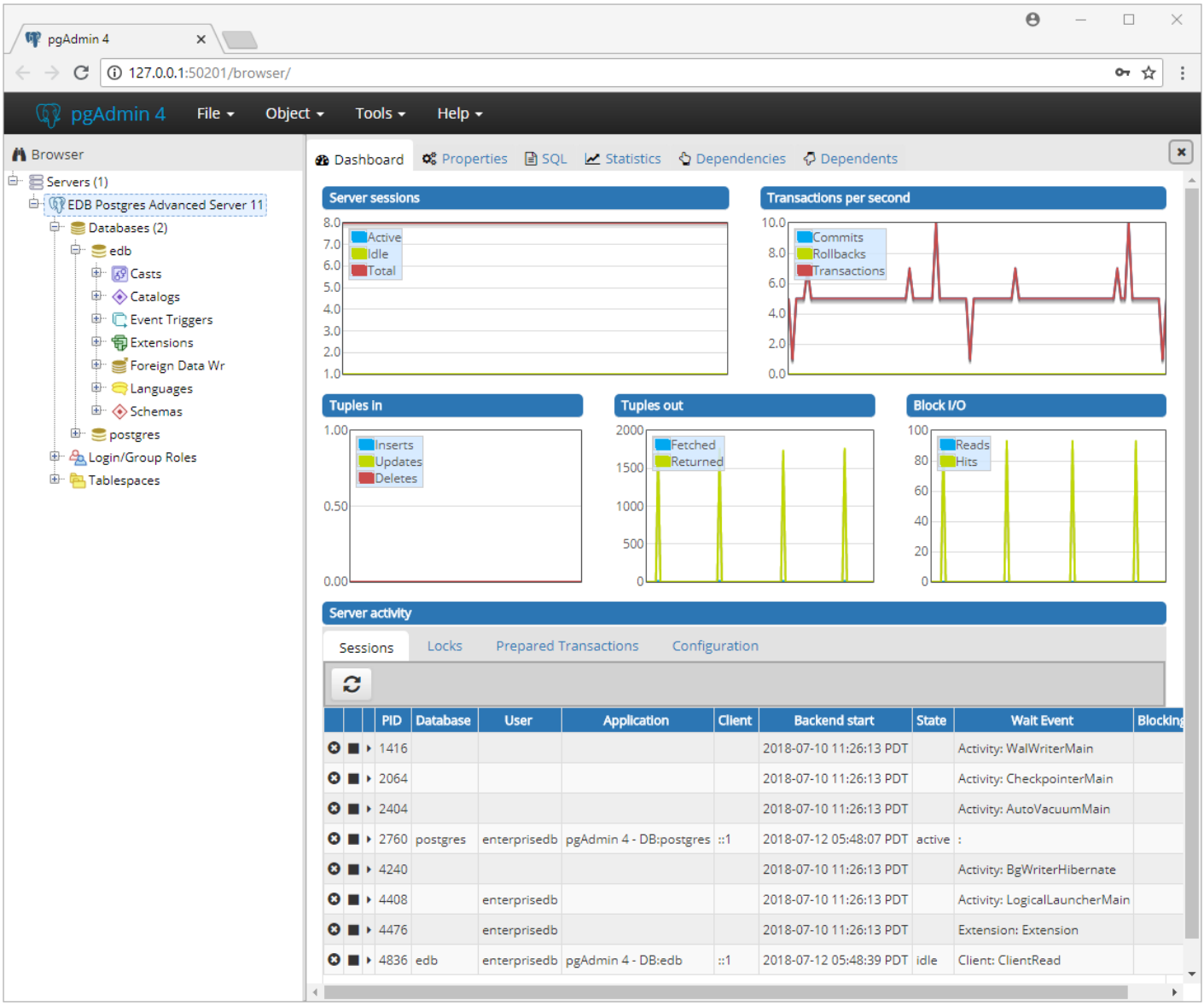


Fig. 3: Connecting to an Advanced Server database

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` drop-down menu to access the online help files.

5.5.2 Managing an Advanced Server Installation

Unless otherwise noted, the commands and paths noted in the following section assume that you have performed an installation with the interactive installer.

Starting and Stopping Advanced Server and Supporting Components

A service is a program that runs in the background and requires no user interaction (in fact, a service provides no user interface); a service can be configured 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 Advanced Server supporting components are services.

The following table lists the names of the services that control Advanced Server and services that control Advanced Server supporting components:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Advanced Server Component Name	Windows Service Name
Advanced Server	edb-as-11
pgAgent	EDB Postgres Advanced Server 11 Scheduling Agent
PgBouncer	edb-pgbouncer-1.7
Slony	edb-slony-replication-11

You can use the command line or the Windows Services applet to control Advanced Server's database server and the services of Advanced Server's supporting components on a Windows host.

5.5.2.1 Using the Windows Services Applet

The Windows operating system includes a graphical service controller that offers control of Advanced Server and the services associated with Advanced Server components. The `Services` utility can be accessed through the `Administrative Tools` section of the Windows `Control Panel`.

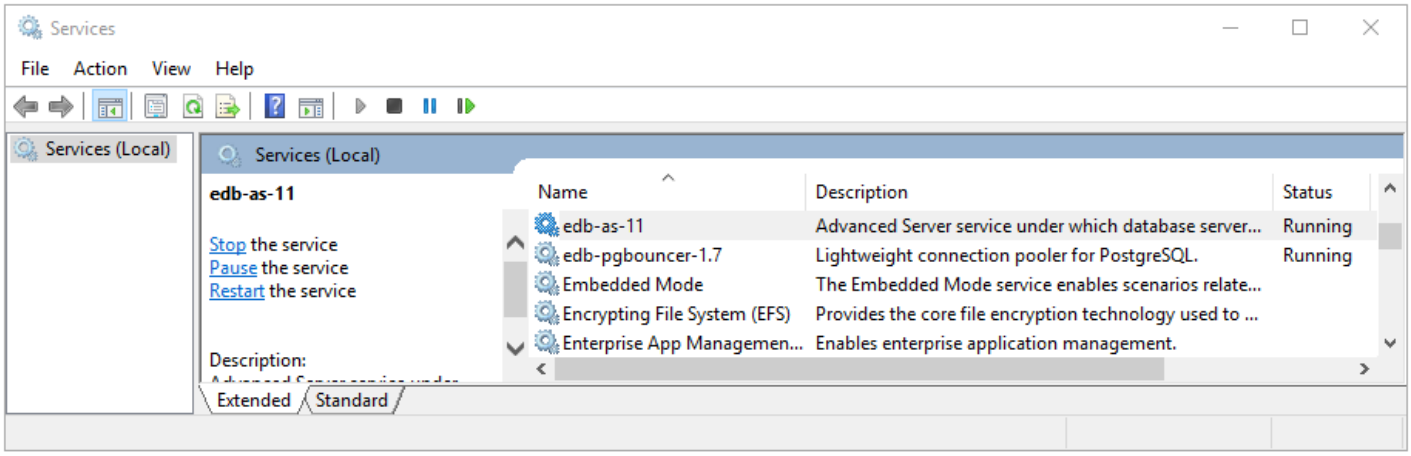


Fig. 1: The Advanced Server service in the Windows Services window

The `Services` window displays an alphabetized list of services; the `edb-as-11` service controls Advanced Server.

- Use the `Stop the service` option to stop the instance of Advanced Server. Please note that any user (or client application) connected to the Advanced Server instance will be abruptly disconnected if you stop the service.
- Use the `Start the service` option to start the Advanced Server service.
- Use the `Pause the service` option to tell Advanced Server to reload the server configuration parameters without disrupting user sessions for many of the configuration parameters. See Configuring Advanced Server for more information about the parameters that can be updated with a server reload.
- Use the `Restart the service` option to stop and then start the Advanced Server. Please note that any user sessions will be terminated when you stop the service. This option is useful to reset server parameters that only take effect on server start.

5.5.2.2 Using pg_ctl to Control Advanced Server

You can use the `pg_ctl` utility to control an Advanced Server service from the command line on any platform. `pg_ctl` allows you to start, stop, or restart the 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 Advanced Server, and issue the command:

```
./bin/pg_ctl -D data_directory action
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`data_directory`

`data_directory` is the location of the data controlled by the Advanced Server cluster.

`action`

`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` sends 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 official PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/11/static/app-pg-ctl.html

5.5.2.3 Controlling Server Startup Behavior on Windows

You can use the Windows Services utility to control the startup behavior of the server. Right click on the name of the service you wish to update, and select `Properties` from the context menu to open the `Properties` dialog.

Use the drop-down listbox in the `Startup type` field to specify how the Advanced Server service will behave when the host starts.

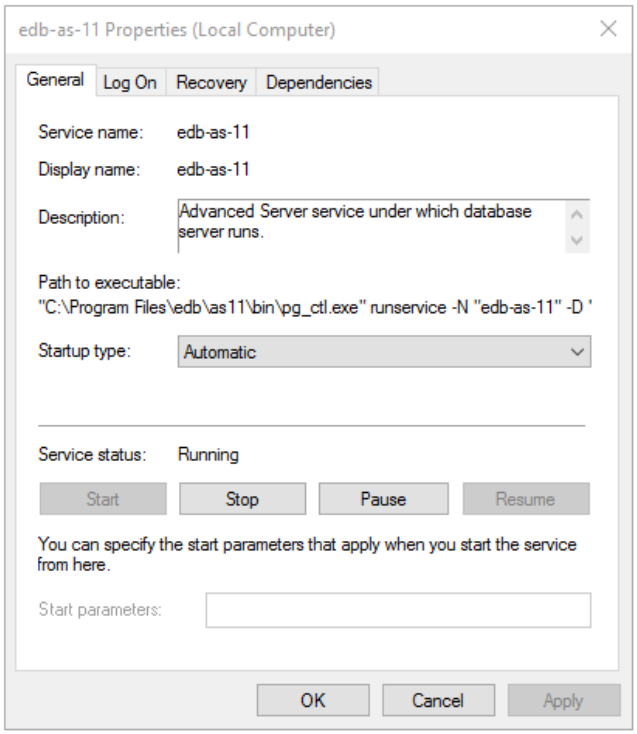


Fig. 2: Specifying Advanced Server's startup behavior

- Specify `Automatic (Delayed Start)` to instruct the service controller to start after boot.
- Specify `Automatic` to instruct the service controller to start and stop the server whenever the system starts or stops.
- Specify `Manual` to instruct the service controller that the server must be started manually.
- Specify `Disabled` to instruct the service controller to disable the service; after disabling the service, you must stop the service or restart the server to make the change take effect. Once disabled, the server's status cannot be changed until `Startup type` is reset to `Automatic (Delayed`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`Start)`, `Automatic`, or `Manual`.

5.5.3 Configuring Advanced Server

You can easily update parameters that determine the behavior of Advanced Server and supporting components by modifying the following configuration files:

- The `postgresql.conf` file determines the initial values of 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 Advanced Server identities (roles) when using `ident`-based authentication.

For more information about Modifying the postgresql.conf file and Modifying the pg_hba.conf file, see the *EDB Postgres Advanced Server Guide* available from the EDB website at:

https://www.enterprisedb.com/docs

You can use your editor of choice to open a configuration file, or on Windows navigate through the `EDB Postgres` menu to open a file.

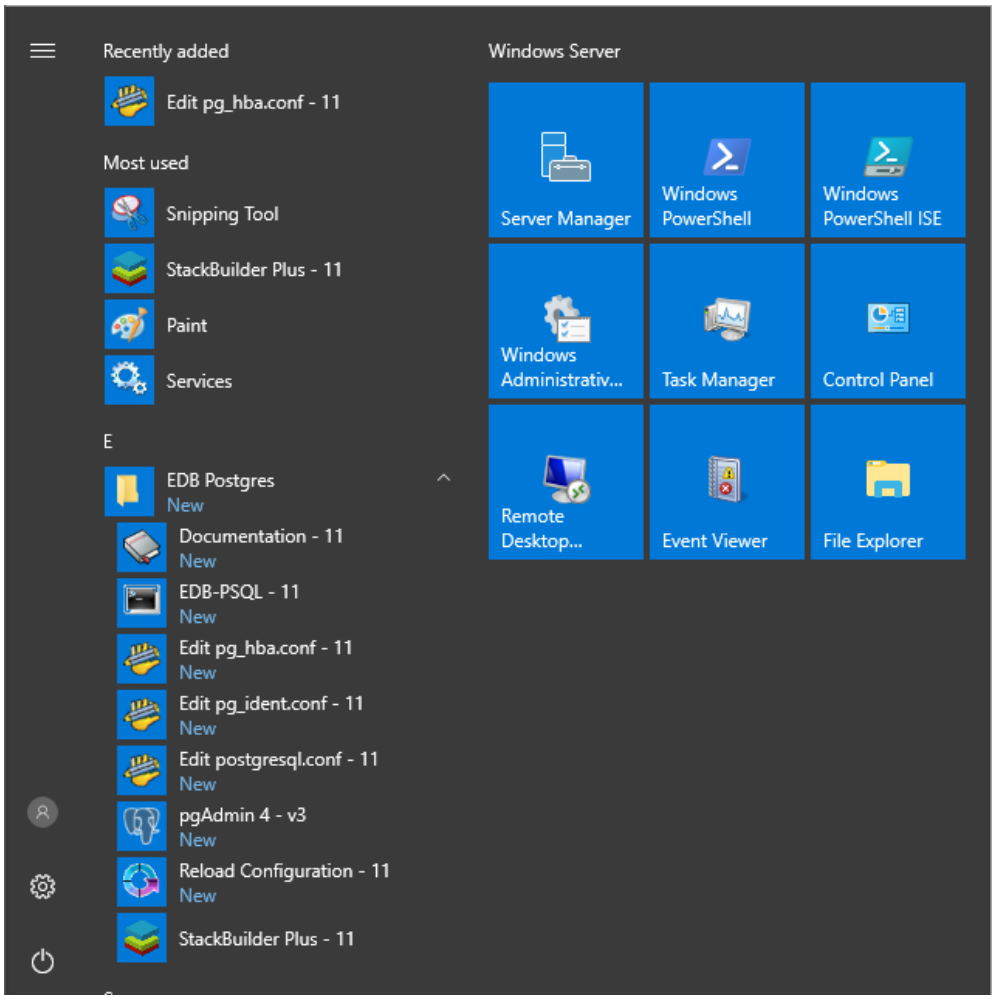


Fig. 1: Accessing the configuration files through the Windows system menu

5.5.3.1 Setting Advanced Server Environment Variables

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 may wish to invoke `pgplus_env.bat` from your system-wide shell startup script, so that environment variables are automatically defined for each shell session.

The `pgplus_env` script is created during the Advanced Server installation process and reflects the choices made during installation. To invoke the script, open a command line and enter:

```
C:\Program Files\edb\as11\pgplus_env.bat
```

As the `pgplus_env.bat` script executes, it sets the following environment variables:

```
PATH="C:\Program Files\edb\as11\bin";%PATH%
EDBHOME=C:\Program Files\edb\as11
PGDATA=C:\Program Files\edb\as11\data
PGDATABASE=edb
REM @SET PGUSER=enterprisedb
PGPORT=5444
PGLOCALEDIR=C:\Program Files\edb\as11\share\locale
```

If you have used an installer created by EnterpriseDB to install PostgreSQL, the `pg_env` script performs the same function:

```
C:\Progra~1\PostgreSQL\11\pg_env.bat
```

As the `pg_env.bat` script executes on PostgreSQL, it sets the following environment variables:

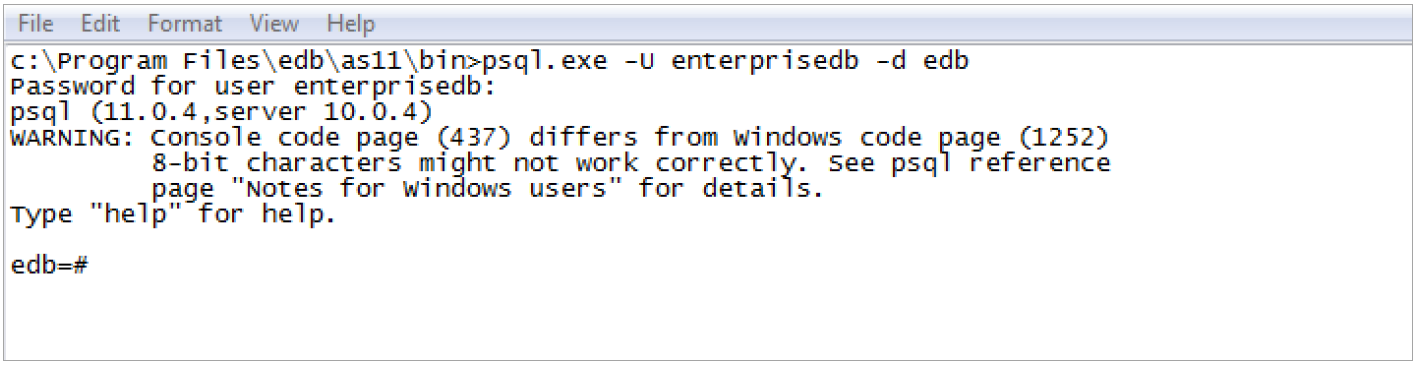
```
PATH="C:\Program Files\PostgreSQL\11\bin";%PATH%
PGDATA=C:\Program Files\PostgreSQL\11\data
PGDATABASE=postgres
PGUSER=postgres
PGPORT=5432
PGLOCALEDIR=C:\Program Files\PostgreSQL\11\share\locale
```

5.5.3.2 Connecting to Advanced Server with psql

`psql` is a command line client application that allows you to execute SQL commands and view the results. To open the `psql` client, the client must be in your search path. The executable resides in the `bin` directory, under your Advanced Server installation.

Use the following command and options to start the `psql` client:

```
psql -d edb -U enterprisedb
```



Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Fig. 2: Connecting to the server

Where:

`-d` specifies the database to which `psql` will connect.

`-U` specifies the identity of the database user that will be used for the session.

If you have performed an installation with the interactive installer, you can access the `psql` client by selecting `EDB-PSQL` from the `EDB Postgres` menu. When the client opens, provide connection information for your session.

For more information about using the command line client, see the PostgreSQL Core Documentation at:

https://www.postgresql.org/docs/11/static/app-psql.html

6 Upgrading EDB Postgres Advanced Server

This section explains how to upgrade EDB Postgres Advanced Server (Advanced Server). It includes detailed information about using:

- `pg_upgrade` to upgrade from an earlier version of Advanced Server to Advanced Server 11.
- `yum` to perform a minor version upgrade on a Linux host.
- `StackBuilder Plus` to perform a minor version upgrade on a Windows host.

6.1 Upgrade Considerations

The following limitations apply to upgrading EDB Postgres Advanced Server:

- The `pg_upgrade` utility cannot upgrade a partitioned table if a foreign key refers to the partitioned table.
- If you are upgrading from the version 9.4 server or a lower version of Advanced Server, and you use partitioned tables that include a `SUBPARTITION BY` clause, you must use `pg_dump` and `pg_restore` to upgrade an existing Advanced Server installation to a later version of Advanced Server. To upgrade, you must:
    1. Use `pg_dump` to preserve the content of the subpartitioned table.
    2. Drop the table from the Advanced Server 9.4 database or a lower version of Advanced Server database.
    3. Use `pg_upgrade` to upgrade the rest of the Advanced Server database to a more recent version.
    4. Use `pg_restore` to restore the subpartitioned table to the latest upgraded Advanced Server database.
- If you perform an upgrade of the Advanced Server installation, you must rebuild any hash-partitioned table on the upgraded server.

6.2 Upgrading an Installation With pg_upgrade

While minor upgrades between versions are fairly simple and require only the installation of new executables, past major version upgrades has been both expensive and time consuming. `pg_upgrade` facilitates migration between any version of Advanced Server (version 9.0 or later), and any subsequent release of Advanced Server that is supported on the same platform.

Without `pg_upgrade`, to migrate from an earlier version of Advanced Server to Advanced Server 11, you must export all of your data using `pg_dump`, install the new release, run `initdb` to create a new cluster, and then import your old data.

pg_upgrade can reduce both the amount of time required and the disk space required for many major-version upgrades.

The `pg_upgrade` utility performs an in-place transfer of existing data between Advanced Server and any subsequent version.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 Advanced Server with `pg_upgrade`, both versions must share a common binary representation for each data type. Therefore, you cannot use `pg_upgrade` to migrate from a 32-bit to a 64-bit Linux platform.

Before performing a version upgrade, `pg_upgrade` will verify that the two clusters (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 involves a change in the binary representation of data types, `pg_upgrade` will be unable to perform the upgrade; to upgrade, you will have to `pg_dump` the old data and then import that data into the new cluster.

The `pg_upgrade` executable is distributed with Advanced Server 11, and is installed as part of the `Database Server` component; no additional installation or configuration steps are required.

6.2.1 Performing an Upgrade - Overview

To upgrade an earlier version of Advanced Server to the current version, you must:

- Install the current version of Advanced Server. The new installation must contain the same supporting server components as the old installation.
- Empty the target database or create a new target cluster with `initdb`.
- Place the `pg_hba.conf` file for both databases in `trust` authentication mode (to avoid authentication conflicts).
- Shut down the old and new Advanced Server services.
- 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 `$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 a later step to recreate all user-defined objects in the new cluster.

Note that the script produced by `pg_dumpall` recreates only user-defined objects and not system-defined objects. The new cluster will *already* contain the system-defined objects created by the latest version of 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.

`pg_upgrade` runs the `pg_dumpall` script against the new cluster to create (empty) database objects of the same shape and type as those found in the old cluster. Then, `pg_upgrade` links or copies each table and index from the old cluster to the new cluster.

If you are upgrading to Advanced Server 11 and have installed the `edb_dblink_oci` or `edb_dblink_libpq` extension, you must 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 have completed upgrading, you can use the `CREATE EXTENSION` command to add the current versions of the extensions to your installation.

6.2.1.1 Linking versus Copying

When invoking `pg_upgrade`, you can use a command-line option to specify whether `pg_upgrade` should *copy* or *link* each table and index in the old cluster to the new cluster.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Linking is much faster because `pg_upgrade` simply creates a second name (a hard link) for each file in the cluster; linking also requires no extra workspace because `pg_upgrade` does not make a copy of the original data. When linking the old cluster and the new cluster, the old and new clusters share the data; note that after starting the new cluster, your data can no longer be used with the previous version of Advanced Server.

If you choose to copy data from the old cluster to the new cluster, `pg_upgrade` will still reduce the amount of 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 simply copied from the old cluster to the new cluster. Finally, using a `dump/restore` procedure to upgrade requires a great deal of workspace to hold the intermediate text-based dump of all of your data, while `pg_upgrade` requires very little extra workspace.

Data that is stored in user-defined tablespaces is not 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, 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.

6.2.2 Invoking pg_upgrade

When invoking `pg_upgrade`, you must specify the location of the old and new cluster's `PGDATA` and executable `(/bin)` directories, as well as the name of the Advanced Server superuser, and the ports on which the installations are listening. A typical call to invoke `pg_upgrade` to migrate from Advanced Server 10 to Advanced Server 11 takes the form:

```
pg_upgrade
--old-datadir <path_to_10_data_directory>
--new-datadir <path_to_11_data_directory>
--user <superuser_name>
--old-bindir <path_to_10_bin_directory>
--new-bindir <path_to_11_bin_directory>
--old-port <10_port> --new-port <11_port>
```

Where:

`--old-datadir path_to_10_data_directory`

Use the `--old-datadir` option to specify the complete path to the `data` directory within the Advanced Server 10 installation.

`--new-datadir path_to_11_data_directory`

Use the `--new-datadir` option to specify the complete path to the `data` directory within the Advanced Server 11 installation.

`--username superuser_name`

Include the `--username` option to specify the name of the Advanced Server superuser. The superuser name should be the same in both versions of Advanced Server. By default, when Advanced Server is installed in Oracle mode, the superuser is named `enterprisedb`. If installed in PostgreSQL mode, the superuser is named `postgres`.

If the Advanced Server superuser name is not the same in both clusters, the clusters will not pass the `pg_upgrade` consistency check.

`--old-bindir path_to_10_bin_directory`

Use the `--old-bindir` option to specify the complete path to the `bin` directory in the Advanced Server 10 installation.

`--new-bindir path_to_11_bin_directory`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Use the `--new-bindir` option to specify the complete path to the `bin` directory in the Advanced Server 11 installation.

`--old-port 10_port`

Include the `--old-port` option to specify the port on which Advanced Server 10 listens for connections.

`--new-port 11_port`

Include the `--new-port` option to specify the port on which Advanced Server 11 listens for connections.

6.2.2.1 Command Line Options - Reference

`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`
`--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`
`--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`
`--check`

Include the `-c` or `--check` keyword to specify that `pg_upgrade` should perform a consistency check on the old and new cluster without performing a version upgrade.

`-d path_to_old_data_directory`
`--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`
`--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 is stored in user-defined tablespaces is not 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`
`--jobs`

Include the `-j` or `--jobs` keyword to specify the number of simultaneous processes or threads to use during the upgrade.

`-k`
`--link`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

```
-o options
--old-options options
```

Use the `-o` or `--old-options` keyword to specify options that will be passed to the old `postgres` command. Enclose options in single or double quotes to ensure that they are passed as a group.

```
-O options
--new-options options
```

Use the `-O` or `--new-options` keyword to specify options to be passed to the new `postgres` command. Enclose options in single or double quotes to ensure that they are passed as a group.

```
-p old_port_number
--old-port old_port_number
```

Include the `-p` or `--old-port` keyword to specify the port number of the Advanced Server installation that you are upgrading.

```
-P new_port_number
--new-port new_port_number
```

Include the `-P` or `--new-port` keyword to specify the port number of the new Advanced Server installation.

> **Note**
>
> If the original Advanced Server installation is using port number `5444` when you invoke the Advanced Server 11 installer, the installer will recommend using listener port `5445` for the new installation of Advanced Server.

```
-r
--retain
```

During the upgrade process, `pg_upgrade` creates four append-only log files; when the upgrade is completed, `pg_upgrade` deletes these files. Include the `-r` or `--retain` option to specify that the server should retain the `pg_upgrade` log files.

```
-U user_name
--username user_name
```

Include the `-U` or `--username` keyword to specify the name of the Advanced Server database superuser. The same superuser must exist in both clusters.

```
-v
--verbose
```

Include the `-v` or `--verbose` keyword to enable verbose output during the upgrade process.

```
-V
--version
```

Use the `-V` or `--version` keyword to display version information for `pg_upgrade`.

```
-?
-h
--help
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Use `-?, -h,` or `--help` options to display `pg_upgrade` help information.

6.2.3 Upgrading to Advanced Server 11 - Step-by-Step

You can use `pg_upgrade` to upgrade from an existing installation of Advanced Server into the cluster built by the Advanced Server 11 installer or into an alternate cluster created using the `initdb` command. In this section, we will provide the details of upgrading into the cluster provided by the installer.

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 Advanced Server 11 installer, but you can omit Step 2 `(Empty the edb database)`, and substitute the location of the alternate cluster when specifying a target cluster for the upgrade.

If a problem occurs during the upgrade process, you can revert to the previous version. SeeReverting to the old cluster Section for detailed information about this process.

You must be an operating system superuser or Windows Administrator to perform an Advanced Server upgrade.

Step 1 - Install the New Server

Install Advanced Server 11, specifying the same non-server components that were installed during the previous 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 Advanced Server 11. If you have installed Advanced Server in PostgreSQL mode, the installer creates a single database named `postgres`; if you have installed Advanced Server in Oracle mode, it 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, navigate through the `Control Panel` to the `Services` manager; highlight 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:

```
service edb-pgagent-11 stop
```

to stop the pgAgent service.

After stopping any services that are currently connected to 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 `template1` database.

```
CREATE DATABASE <database_name>;
```

Step 3 - Set both servers in trust mode

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

During the upgrade process, `pg_upgrade` will connect 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, navigate through the `Start` menu to the `EDB Postgres` menu; to the `Advanced Server` menu, and open the `Expert Configuration` menu; select the `Edit pg_hba.conf` menu option to open the `pg_hba.conf` file.

You must allow trust authentication for the previous Advanced Server installation, and Advanced Server 11 servers. Edit the `pg_hba.conf` file for both installations of Advanced Server as shown in the following figure.

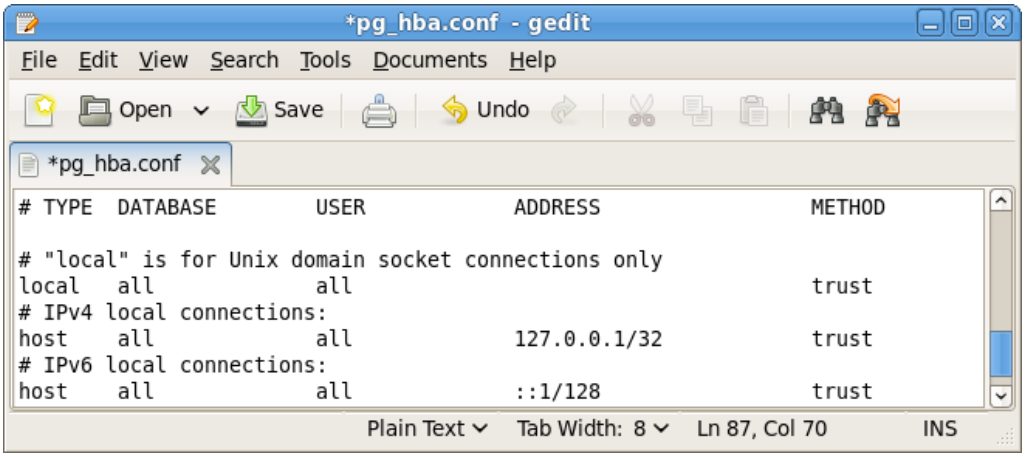


Fig. 1: Configuring Advanced Server to use trust authentication

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, available at:

https://www.postgresql.org/docs/11/static/libpq-pgpass.html

Step 4 - Stop All Component Services and Servers

Before you invoke `pg_upgrade`, you must stop any services that belong to the original Advanced Server installation, Advanced Server 11, or the supporting components. This ensures that a service will not attempt to access either cluster during the upgrade process.

The services that are most likely to be running in your installation are:

Service:	On Linux:	On Windows:
EnterpriseDB Postgres Advanced Server 9.6	edb-as-9.6	edb-as-9.6
EnterpriseDB Postgres Advanced Server 10	edb-as-10	edb-as-10
EnterpriseDB Postgres Advanced Server 11	edb-as-11	edb-as-11
Advanced Server 9.6 Scheduling Agent (pgAgent)	edb-pgagent-9.6	EnterpriseDB Postgres Advanced Server 9.6 Scheduling Agent
Infinite Cache 9.6	edb-icache	N/A
Infinite Cache 10	edb-icache	N/A
PgBouncer	Pgbouncer	Pgbouncer
PgBouncer 1.6	ppas-pgbouncer-1.6 or ppas-pgbouncer16	ppas-pgbouncer-1.6
PgBouncer 1.7	edb-pgbouncer-1.7	edb-pgbouncer-1.7
PgPool	ppas-pgpool	N/A
PgPool 3.4	ppas-pgpool-3.4 or ppas-pgpool34	N/A
pgPool-II	edb-pgpool-3.5	N/A

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Service:	On Linux:	On Windows:
Slony 9.6	edb-slony-replication-9.6	edb-slony-replication-9.6
xDB Subscription Server	edb-xdbsubserver-90	Subscription Service 90
xDB Subscription Server	edb-xdbsubserver-91	Subscription Service 91
EDB Replication Server v6.x	edb-xdbpubserver	Publication Service for xDB Replication Server
EDB Subscription Server v6.x	edb-xdbsubserver	Subscription Service for xDB Replication Server

To stop a service on Windows:

Open the `Services` applet; highlight each 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 are using Linux, assume the identity of the Advanced Server cluster owner. (The following example assumes Advanced Server was installed in the default, compatibility with Oracle database mode, thus assigning `enterprisedb` as the cluster owner. If installed in compatibility with PostgreSQL database mode, `postgres` is the cluster owner.)

```
su - enterprisedb
```

Enter the Advanced Server cluster owner password if prompted. Then, set the path to include the location of the `pg_upgrade` executable:

```
export PATH=$PATH:/usr/edb/as11/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 above commands, navigate to a directory in which the `enterprisedb` user has sufficient privileges to write a file.

```
cd /tmp
```

Proceed to Step 6.

Step 5 For Windows only - Assume the identity of the cluster owner

If you are using Windows, open a terminal window, assume the identity of the 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 Advanced Server, then `service_account_user` is the Advanced Server cluster owner and is the user to be given with the `RUNAS` command.

```
RUNAS /USER:service_account_user "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\edb\as11\bin
```

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 above commands, navigate to a directory in which the service account user has sufficient privileges to write a file.

```
cd %TEMP%
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Proceed to Step 6.

If the `--serviceaccount` parameter was omitted during the initial installation of Advanced Server, then the default owner of the Advanced Server service and the database cluster is `NT AUTHORITY\NetworkService`.

When `NT AUTHORITY\NetworkService` is the service account user, the `RUNAS` command may not be usable as it prompts for a password and the `NT AUTHORITY\NetworkService` account is not assigned a password. Thus, there is typically a failure with an error message such as, “Unable to acquire user password”.

Under this circumstance a Windows utility program named `PsExec` must be used to run `CMD.EXE` as the service account `NT AUTHORITY\NetworkService`.

The `PsExec` program must be obtained by downloading `PsTools`, which is available at the following site:

https://technet.microsoft.com/en-us/sysinternals/bb897553.aspx

You can then use the following command to run `CMD.EXE` as `NT AUTHORITY\NetworkService`, and then set the path to the `pg_upgrade` executable.

```
psexec.exe -u "NT AUTHORITY\NetworkService" CMD.EXE
SET PATH=%PATH%;C:\Program Files\edb\as11\bin
```

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 above commands, navigate to a directory in which the service account user has sufficient privileges to write a file.

```
cd %TEMP%
```

Proceed with Step 6.

Step 6 - Perform a consistency check

Before attempting an upgrade, perform a consistency check to assure that the old and new clusters are compatible and properly configured. Include the `--check` option to instruct `pg_upgrade` to perform the consistency check.

The following example demonstrates invoking `pg_upgrade` to perform a consistency check on Linux:

```
pg_upgrade -d /var/lib/edb/as10/data
-D /var/lib/edb/as11/data -U enterprisedb
-b /usr/edb/as10/bin -B /usr/edb/as11/bin -p 5444 -P 5445 --check
```

If the command is successful, it will return `*Clusters are compatible*`.

If you are using Windows, you must quote any directory names that contain a space:

```
pg_upgrade.exe
-d "C:\Program Files\ PostgresPlus\10AS\data"
-D "C:\Program Files\edb\as11\data" -U enterprisedb
-b "C:\Program Files\PostgresPlus\10AS\bin"
-B "C:\Program Files\edb\as11\bin" -p 5444 -P 5445 --check
```

During the consistency checking process, `pg_upgrade` will log any discrepancies that it finds to a file located in the directory from which `pg_upgrade` was invoked. When the consistency check completes, review the file to identify any missing components or upgrade conflicts. You must resolve any conflicts before invoking `pg_upgrade` to perform a version upgrade.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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, you must restart the Advanced Server 11 service. After restarting the service, open StackBuilder Plus by navigating through the `Start` menu to the `Advanced Server 11` menu, and selecting `StackBuilder Plus`. Follow the onscreen advice of the StackBuilder Plus wizard to download and install the missing components.

When `pg_upgrade` has confirmed 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 Advanced Server.

On Linux:

```
pg_upgrade -d /var/lib/edb/as10/data
-D /var/lib/edb/as11/data -U enterprisedb
-b /usr/edb/as10/bin -B /usr/edb/as11/bin -p 5444 -P 5445
```

On Windows:

```
pg_upgrade.exe -d "C:\Program Files\PostgresPlus\10AS\data"
-D "C:\Program Files\edb\as11\data" -U enterprisedb
-b "C:\Program Files\PostgresPlus\10AS\bin"
-B "C:\Program Files\edb\as11\bin" -p 5444 -P 5445
```

`pg_upgrade` will display the progress of the upgrade onscreen:

```
$ pg_upgrade -d /var/lib/edb/as10/data -D /var/lib/edb/as11/data -U
enterprisedb -b /usr/edb/as10/bin -B /usr/edb/as11/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.

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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 may generate SQL scripts that handle special circumstances that it has encountered during your upgrade. For example, if the old cluster contains large objects, you may 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` will alert you to any script that you may be required 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 is generally unsafe to access tables referenced in rebuild scripts until the rebuild scripts have completed; accessing the tables could yield incorrect results or poor performance. Tables not referenced in rebuild scripts can be accessed immediately.

If `pg_upgrade` fails to complete the upgrade process, the old cluster will be 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 may have been moved before the upgrade failed.

After performing these steps, resolve any upgrade conflicts encountered before attempting the upgrade again.

When the upgrade is complete, `pg_upgrade` may also recommend vacuuming the new cluster, and will provide a script that allows you to delete the old cluster.

> **Note**
>
> Before removing the old cluster, ensure that the cluster has been upgraded as expected, and that you have preserved a backup of the cluster in case you need to revert to a previous version.

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 (located in the `pg_tblspc` directory under your cluster's `data` directory) to point to the files.

6.2.4 Upgrading a pgAgent Installation

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

If your existing Advanced Server installation uses pgAgent, you can use a script provided with the Advanced Server 11 installer to update pgAgent. The script is named `dbms_job.upgrade.script.sql`, and is located in the `/share/contrib/` directory under your Advanced Server installation.

If you are using `pg_upgrade` to upgrade your installation, you should:

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 will convert the installation to an extension.

6.2.5 pg_upgrade Troubleshooting

The troubleshooting tips in this section address problems you may 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, please stop all 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, please 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 is not empty, please empty the new cluster. The target cluster may not contain any user-defined databases.

Upgrade Error - Failed to load library

If the original Advanced Server cluster included libraries that are not included in the Advanced Server 11 cluster, `pg_upgrade` will alert 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 are not part of a major component upgrade, perform the following steps:

1. Restart the Advanced Server service.

   Use StackBuilder Plus to download and install the missing module. Then:

2. Stop the Advanced Server service.

3. Resume the upgrade process: invoke `pg_upgrade` to perform consistency checking.

4. When you have resolved any remaining problems noted in the consistency checks, invoke `pg_upgrade` to perform the data migration from the old cluster to the new cluster.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

6.2.6 Reverting to the Old Cluster

The method used 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 has not been performed, and no modifications have been made to the old cluster; you can re-use 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 have started the server that is servicing the new cluster, the new server has written to those shared files and it is unsafe to use the old cluster.
- If you ran `pg_upgrade` without the `--link` specification or have not started the new server, the old cluster is unchanged, except that the `.old` suffix has been appended to the `$PGDATA/global/pg_control` and tablespace directories.
- To reuse the old cluster, delete the tablespace directories created by the new cluster and remove the `.old` suffix from `$PGDATA/global/pg_control` and the old cluster tablespace directory names and restart the server that services the old cluster.

6.3 Performing a Minor Version Update of an RPM Installation

If you used an RPM package to install 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 wish to search for updates. Please note that you can include wild-card 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 wish to update. Include wild-card 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 will only perform an update between minor releases; to update between major releases, you must use `pg_upgrade`.

For more information about using yum commands and options, enter `yum --help` on your command line, or visit:

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/ch-yum.html

6.4 Using StackBuilder Plus to Perform a Minor Version Update

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 Advanced Server installation. When you install a module with StackBuilder Plus, StackBuilder Plus automatically resolves any software dependencies.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

You can invoke StackBuilder Plus at any time after the installation has completed by selecting the `StackBuilder Plus` menu option from the `Apps` menu. Enter your system password (if prompted), and the StackBuilder Plus welcome window opens.

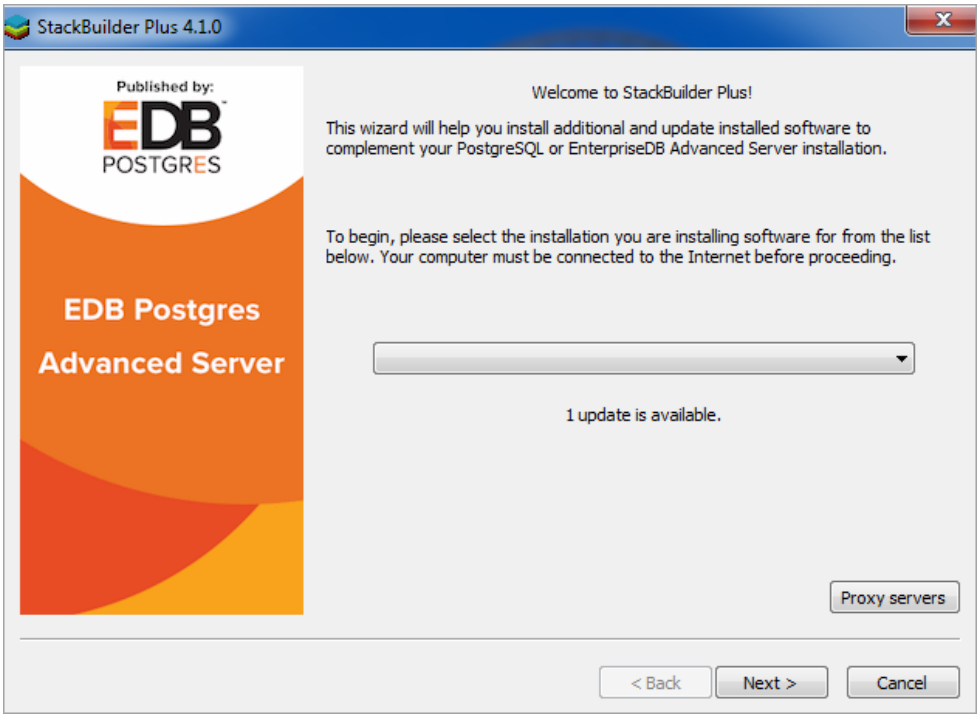


Fig. 1: The StackBuilder Plus welcome window

Use the drop-down listbox on the welcome window to select your Advanced Server installation.

StackBuilder Plus requires Internet access; if your installation of Advanced Server resides 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 via an HTTP proxy and the FTP proxy information is not used.

If the selected Advanced Server installation has restricted Internet access, use the `Proxy Servers` on the `Welcome` window to open the `Proxy servers` dialog (shown in the following figure).

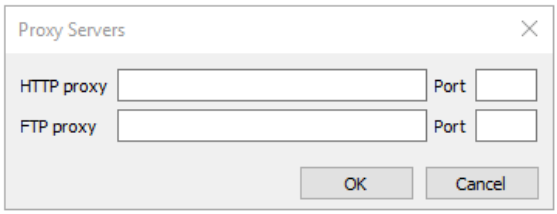


Fig. 2: The Proxy servers dialog

Enter the IP address and port number of the proxy server in the `HTTP proxy` on the `Proxy servers` dialog. Currently, all StackBuilder Plus modules are distributed via HTTP proxy (FTP proxy information is ignored). Click `OK` to continue.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

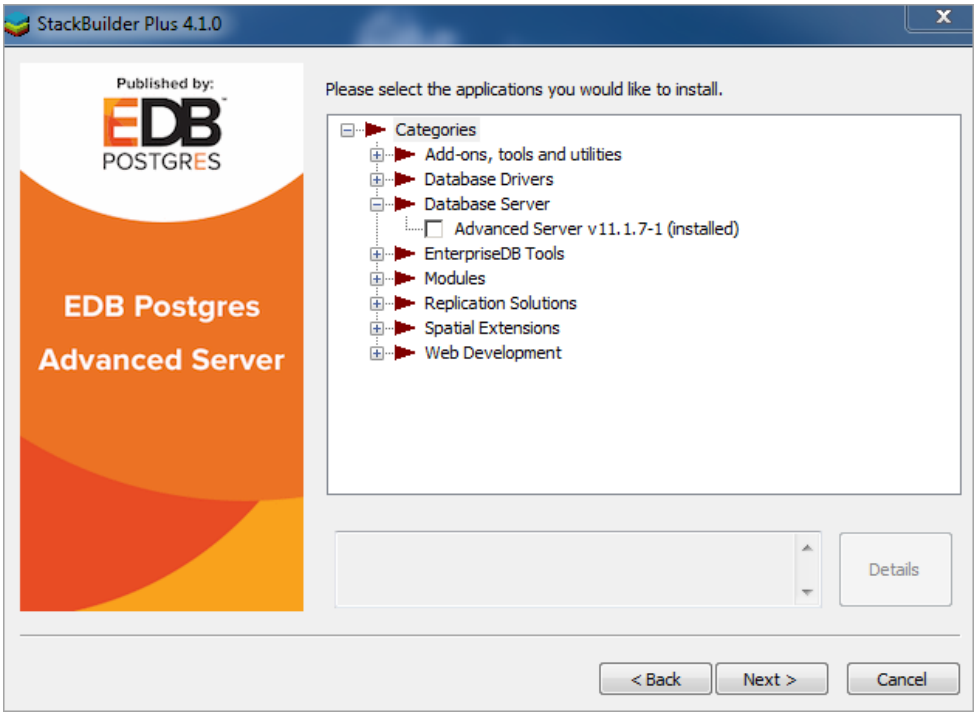


Fig. 3: The StackBuilder Plus module selection window

The tree control on the StackBuilder Plus module selection window (shown in the figure) displays a node for each module category. To perform an Advanced Server update, expand the Database Server module in the tree control and check the box to the left of the Advanced Server upgraded version. Then, click Next.

If prompted, enter your email address and password on the StackBuilder Plus registration window.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

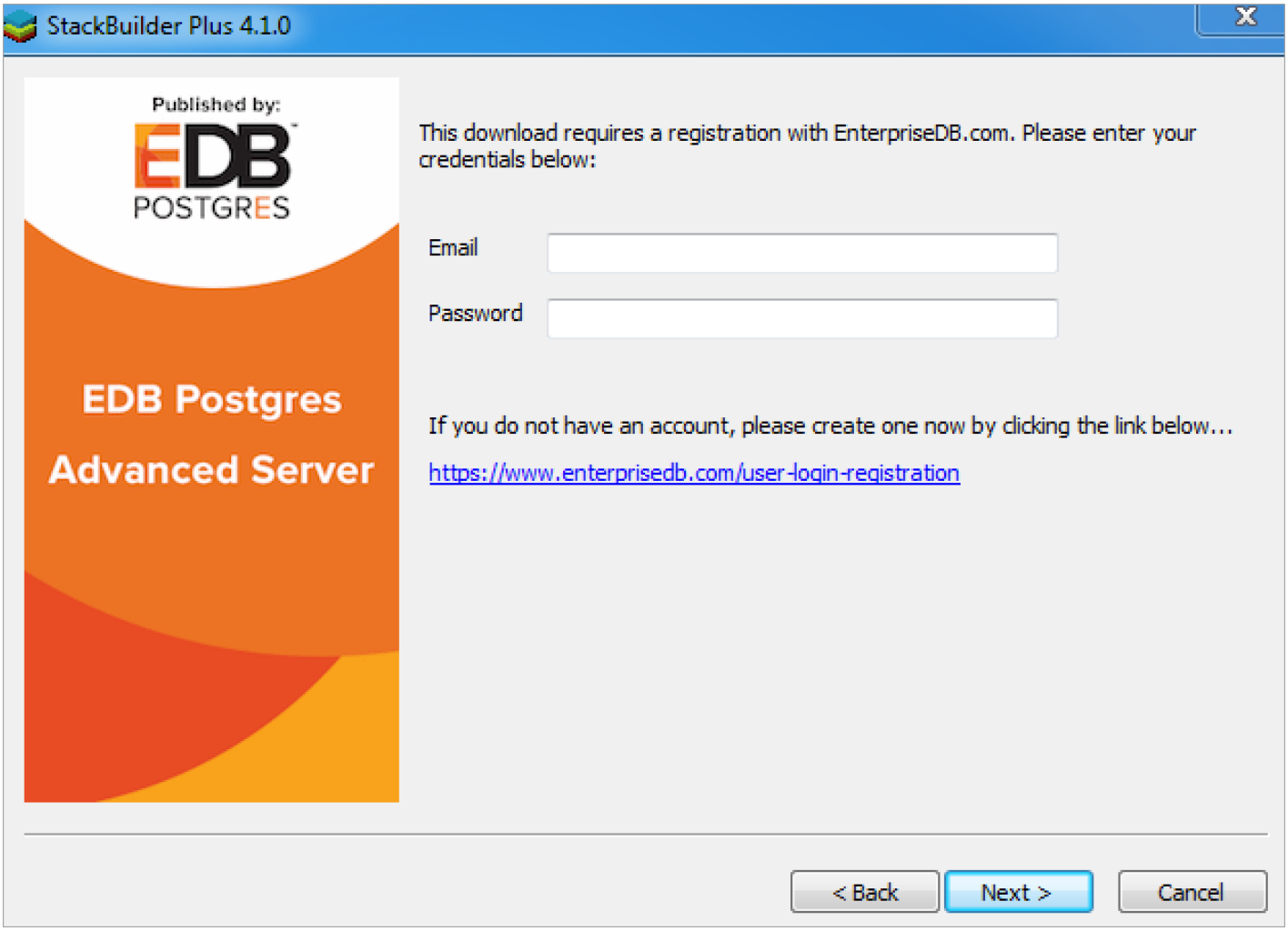


Fig. 4: The StackBuilder Plus registration window.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

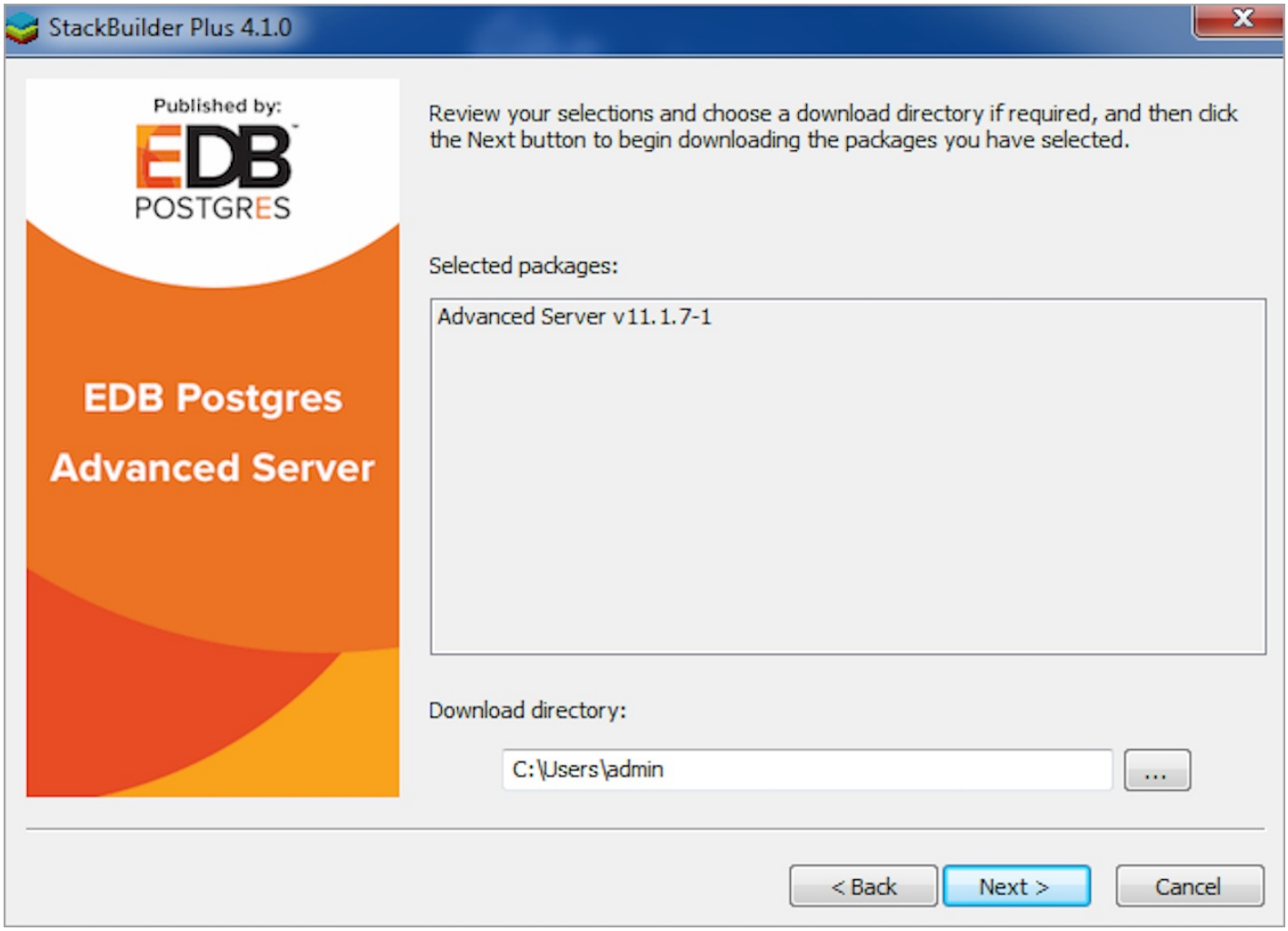
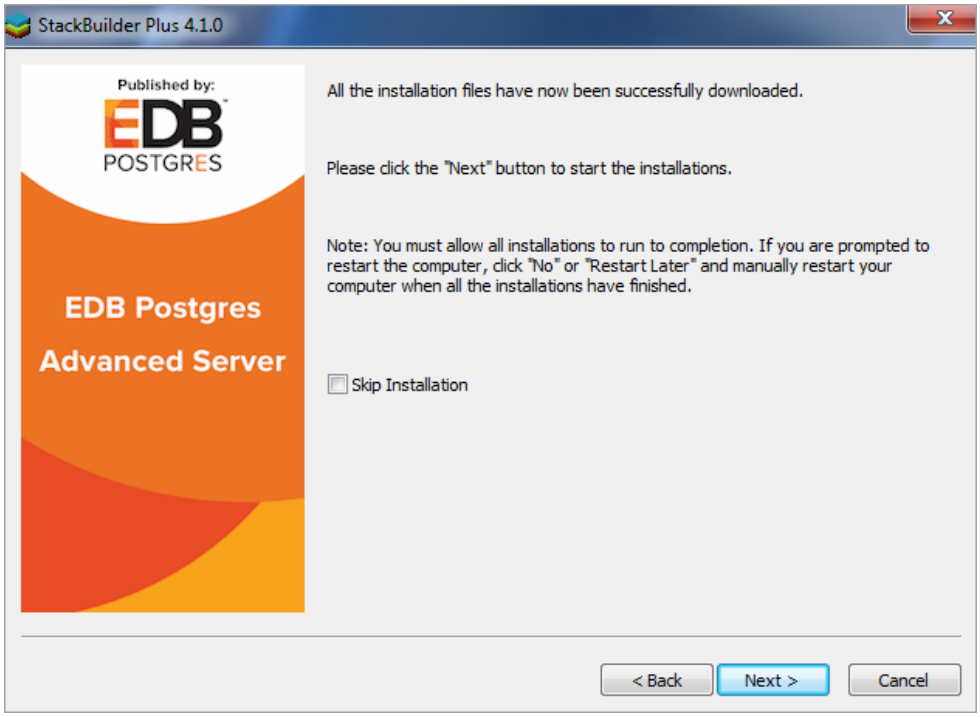


Fig. 5: A summary window displays a list of selected packages

StackBuilder Plus confirms the packages selected. The `Selected packages` dialog will display the name and version of the installer; click `Next` to continue.

When the download completes, a window opens that confirms the installation files have been downloaded and are ready for installation.



Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Fig. 6: Confirmation that the download process is complete

You can check the box next to `Skip Installation`, and select `Next` to exit StackBuilder Plus without installing the downloaded files, or leave the box unchecked and click `Next` to start the installation process.

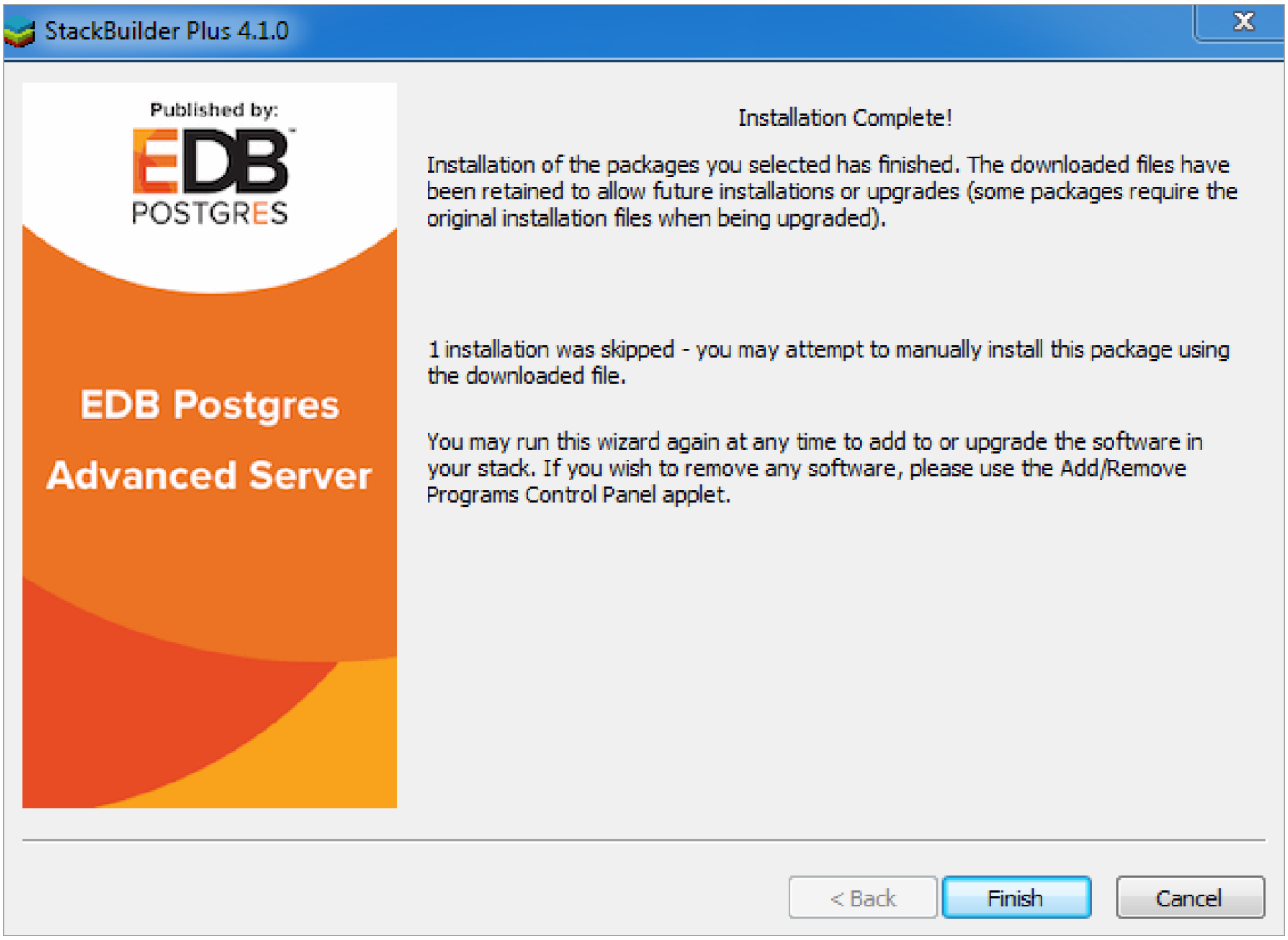


Fig. 7: StackBuilder Plus confirms the completed installation

When the upgrade is complete, StackBuilder Plus will alert you to the success or failure of the installation of the requested package. If you were prompted by an installer to restart your computer, reboot now.

> **Note**
>
> If the update fails to install, StackBuilder Plus will alert you to the installation error with a popup dialog and write a message to the log file at `%TEMP%`.

6.5 Migration to Version 11

A dump/restore using `pg_dumpall` or `pg_upgrade` or logical replication is required for migrating data from any previous release. See Upgrading an Installation With pg_upgrade for information on migrating to new major releases.

Version 11 contains a number of changes that may affect compatibility with previous releases. Listed is the following incompatibilities:

- Make `pg_dump` dump the properties of a database, not just its contents.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Previously, attributes of the database itself, such as database-level `GRANT/REVOKE` permissions and `ALTER DATABASE SET` variable settings, were only dumped by `pg_dumpall`. Now `pg_dump --create` and `pg_restore --create` will restore these database properties in addition to the objects within the database. `pg_dumpall -g` now only dumps role- and tablespace-related attributes. `pg_dumpall`'s complete output (without `-g`) is unchanged.

`pg_dump` and `pg_restore`, without `--create`, no longer dump/restore database-level comments and security labels; those are now treated as properties of the database.

`pg_dumpall`'s output script will now always create databases with their original locale and encoding, and hence will fail if the locale or encoding name is unknown to the destination system. Previously, `CREATE DATABASE` would be emitted without these specifications if the database locale and encoding matched the old cluster's defaults.

`pg_dumpall --clean` now restores the original locale and encoding settings of the `postgres` and `template1` databases, as well as those of user-created databases.

- Consider syntactic form when disambiguating function versus column references.

  When `x` is a table name or composite column, PostgreSQL has traditionally considered the syntactic forms `f(x)` and `x.f` to be equivalent, allowing tricks such as writing a function and then using it as though it were a computed-on- demand column. However, if both interpretations are feasible, the column interpretation was always chosen, leading to surprising results if the user intended the function interpretation. Now, if there is ambiguity, the interpretation that matches the syntactic form is chosen.

- Fully enforce uniqueness of table and domain constraint names.

  PostgreSQL expects the names of a table's constraints to be distinct, and likewise for the names of a domain's constraints. However, there was not rigid enforcement of this, and previously there were corner cases where duplicate names could be created.

- Make `power(numeric, numeric)` and `power(float8, float8)` handle `NaN` inputs according to the POSIX standard.

  POSIX says that `NaN ^ 0 = 1` and `1 ^ NaN = 1`, but all other cases with `NaN` input(s) should return `NaN`.`power(numeric, numeric)` just returned `NaN` in all such cases; now it honors the two exceptions. `power(float8, float8)` followed the standard if the C library does; but on some old Unix platforms the library doesn't, and there were also problems on some versions of Windows.

- Prevent `to_number()` from consuming characters when the template separator does not match.

  Specifically, `SELECT to_number('1234', '9,999')` used to return `134`. It will now return `1234`. `L` and `TH` now only consume characters that are not digits, positive/negative signs, decimal points, or commas.

- Fix `to_date(), to_number(),` and `to_timestamp()` to skip a character for each template character.

  Previously, they skipped one `byte` for each byte of template character, resulting in strange behavior if either string contained multibyte characters.

- Adjust the handling of backslashes inside double-quotes in template strings for `to_char()`, `to_number()`, and `to_timestamp()`.

  Such a backslash now escapes the character after it, particularly a double-quote or another backslash.

- Correctly handle relative path expressions in `xmltable(), xpath()`, and other XML-handling functions.

  Per the SQL standard, relative paths start from the document node of the XML input document, not the root node as these functions previously did.

- In the extended query protocol, make `statement_timeout` apply to each Execute message separately, not to all commands before Sync.

- Remove the `relhaspkey` column from system catalog `pg_class`. Applications needing to check for a primary key should consult `pg_index`.

- Replace system catalog `pg_proc's proisagg` and `proiswindow` columns with `prokind`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

This new column more clearly distinguishes functions, procedures, aggregates, and window functions.

- Correct information schema column `tables.table_type` to return `FOREIGN` instead of `FOREIGN TABLE`.

  This new output matches the SQL standard.

- Change the ps process display labels for background workers to match the `pg_stat_activity.backend_type` labels.

- Cause large object permission checks to happen during large object open, `lo_open()`, not when a read or write is attempted.

  If write access is requested and not available, an error will now be thrown even if the large object is never written to.

- Prevent non-superusers from reindexing shared catalogs.

  Previously, database owners were also allowed to do this, but now it is considered outside the bounds of their privileges.

- Remove deprecated `adminpack` functions `pg_file_read(), pg_file_length()`, and `pg_logfile_rotate()`.

  Equivalent functionality is now present in the core backend. Existing `adminpack` installs will continue to have access to these functions until they are updated via `ALTER EXTENSION ... UPDATE`.

- Honor the capitalization of double-quoted command options.

  Previously, option names in certain SQL commands were forcibly lower-cased even if entered with double quotes; thus for example `"FillFactor"` would be accepted as an index storage option, though properly its name is lower-case. Such cases will now generate an error.

- Remove server parameter `replacement_sort_tuples`.

  Replacement sorts were determined to be no longer useful.

- Remove `WITH` clause in `CREATE FUNCTION`.

  PostgreSQL has long supported a more standard-compliant syntax for this capability.

- In PL/pgSQL trigger functions, the `OLD` and `NEW` variables now read as `NULL` when not assigned.

  Previously, references to these variables could be parsed but not executed.

- The `SELECT DISTINCT...ORDER BY` clause of the `SELECT DISTINCT` query behavior differs after upgrade.

  If `SELECT DISTINCT` is specified 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 (applicable when upgrading from version 9.6 to any higher version of Advanced Server).

7 Uninstalling EDB Postgres Advanced Server

This section provides detailed instructions for uninstalling Advanced Server on Linux and Windows.

7.1 Uninstalling Advanced Server on Linux

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Note that after uninstalling Advanced Server, the cluster data files remain intact and the service user persists. You may 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. Note that removing a package does not damage the Advanced Server `data` directory.

Include the `-e` option when invoking the `rpm` command to remove an installed package; the command syntax is:

```
rpm -e <package_name>
```

Where `package_name` is the name of the package that you would like 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, open a terminal window, assume superuser privileges, and enter the 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 would like to remove.

`yum` and RPM will not remove a package that is required by another package. If you attempt to remove a package that satisfies a package dependency, `yum` or RPM will provide a warning.

> **Note**
>
> In RHEL or Rocky Linux or AlmaLinux 8, removing a package also removes all its dependencies that are not required by other packages. To override this default behavior of RHEL or Rocky Linux or AlmaLinux 8, you must disable the `clean_requirements_on_remove` parameter in the `/etc/yum.conf` file.

To uninstall Advanced Server and its dependent packages; use the following command:

- On RHEL or CentOS 7:

  ```
  yum remove edb-as11-server*
  ```

- On RHEL or Rocky Linux or AlmaLinux 8:

  ```
  dnf remove edb-as11-server*
  ```

Uninstalling on Debian or Ubuntu

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- To uninstall Advanced Server, invoke the following command. The configuration files and data directory remains intact.

```
apt-get remove edb-as11-server*
```

- To uninstall Advanced Server, configuration files, and data directory, invoke the following command:

```
apt-get purge edb-as11-server*
```

Uninstalling on SLES

To uninstall Advanced Server, assume the identity of a root user and invoke the following command:

```
zypper remove edb-as11-server
```

Updating an RPM Installation

If you have an existing 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:

```
yum upgrade edb-repo
```

- On RHEL or Rocky Linux/AlmaLinux 8:

```
dnf upgrade edb-repo
```

`yum` or `dnf` will update 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/AlmaLinux 8:

```
dnf upgrade edb*
```

> **Note**
>
> The `yum upgrade` or `dnf upgrade` command will only perform an update between minor releases; to update between major releases, you must use `pg_upgrade`.

For more information about using yum commands and options, enter `yum --help` on your command line.

For more information about using `dnf` commands and options, visit:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

https://docs.fedoraproject.org/en-US/quick-docs/dnf/

Updating Components on a SLES Host:

To update components installed with zypper, use the `zypper update` command.

7.2 Uninstalling Advanced Server on Windows

Note that after uninstalling Advanced Server, the cluster data files remain intact and the service user persists. You may manually remove the cluster data and service user from the system.

Using Advanced Server Uninstallers at the Command Line

The Advanced Server interactive installer creates an uninstaller that you can use to remove Advanced Server or components that reside on a Windows host. The uninstaller is created in `C:\Program Files\edb\as11`. To open the uninstaller, assume superuser privileges, navigate into the directory that contains the uninstaller, and enter:

```
uninstall-edb-as11-server.exe
```

The uninstaller opens.

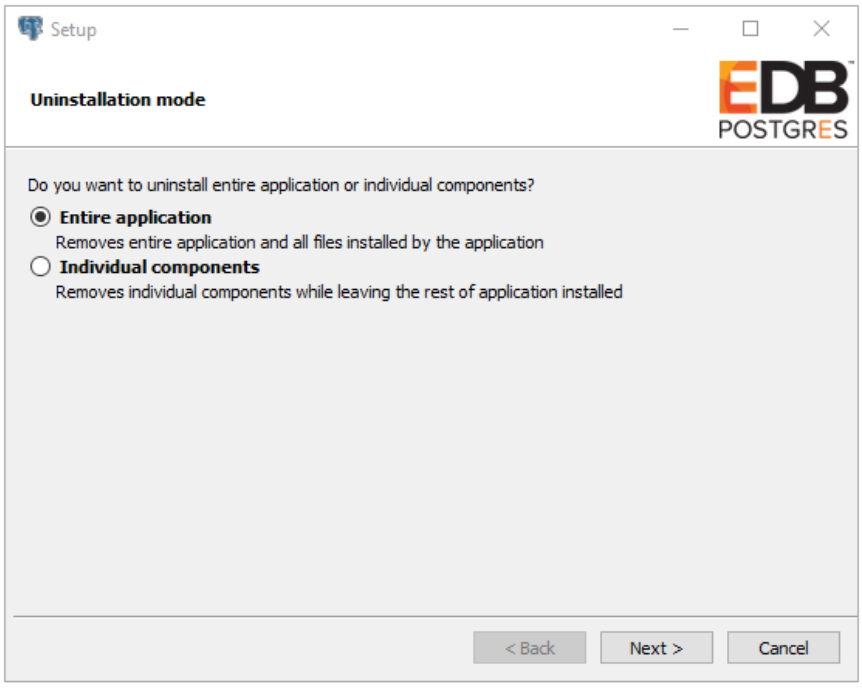


Fig. 1: The Advanced Server uninstaller

You can remove the `Entire application` (the default), or select the radio button next to `Individual components` to select components for removal; if you select `Individual components`, a dialog will open, prompting you to select the components you wish to remove. After making your selection, click `Next`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

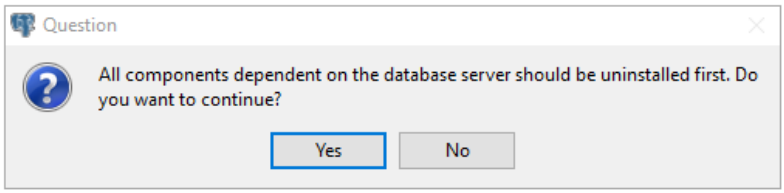


Fig. 2: Acknowledge that dependent components are removed first

If you have elected to remove components that are dependent on Advanced Server, those components will be removed first; click `Yes` to acknowledge that you wish to continue.

Progress bars are displayed as the software is removed. When the uninstallation is complete, an `Info` dialog opens to confirm that Advanced Server (and/or its components) has been removed.

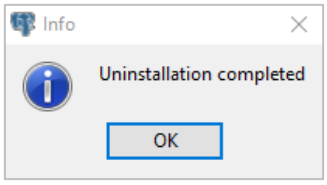


Fig. 3: The uninstallation is complete

8 Troubleshooting an installation

This section provides guidance for troubleshooting an Advanced Server installation on Linux and Windows.

8.1 Linux installation troubleshooting

Difficulty Displaying Java-based Applications

If you encounter difficulty displaying Java-based server features (controls or text not being displayed correctly, or blank windows), upgrading to the latest `libxcb-xlib` libraries should correct the problem on most distributions. Please visit the following link for other possible work-arounds:

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6532373

The Installation Fails to Complete Due to Existing data Directory Contents

If an installation fails to complete due to an existing content in the data directory, the server will write an error message to the server logs:

```
A data directory is neither empty, or a recognisable data directory.
```

If you encounter a similar message, you should confirm that the data directory is empty; the presence of files (including the system-generated `lost+found` folder) will prevent the installation from completing. Either remove the files from the data directory, or specify a different location for the data directory before re-invoking the installer to complete the installation.

Difficulty Installing the EPEL Release Package

If you encounter difficulty when installing the `EPEL` release package, you can use the following command to install the `epel-release` package on RHEL/CentOS 7.x or RHEL/Rocky Linux/AlmaLinux 8.x:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm

dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
```

Please note that you may need to enable the `[extras]` repository definition in the `CentOS-Base.repo` file (located in `/etc/yum.repos.d`). If `yum` cannot access a repository that contains `epel-release`, you will get an error message:

```
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 install `EPEL` with the command:

```
yum -y install epel-release

dnf -y install epel-release
```

8.2 Windows installation troubleshooting

Difficulty Displaying Java-based Applications

If you encounter difficulty displaying Java-based server features (controls or text not being displayed correctly, or blank windows), upgrading to the latest `libxcb-xlib` libraries should correct the problem on most distributions. Please visit the following link for other possible work-arounds:

http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6532373

- -mode unattended Authentication Errors

Authentication errors from component modules during unattended installations may indicate that the specified values of `--servicepassword` or `--superpassword` may be incorrect.

Errors During an 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, re-invoke the Advanced Server installer.

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 will write an error message to the server logs:

`A data directory is neither empty, or a recognisable data directory`.

If you encounter a similar message, you should confirm that the data directory is empty; the presence of files (including the system-generated `lost+found` folder) will prevent the installation from completing. Either remove the files from the data directory, or specify a different location for the data directory before re-invoking the installer to complete the installation.

9 Database Compatibility for Oracle Developers Guide

Database Compatibility for Oracle means that an application runs in an Oracle environment as well as in the EDB Postgres Advanced Server (Advanced Server) environment with minimal or no changes to the application code. Developing an application that is compatible with Oracle databases in the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Advanced Server requires special attention to which features are used in the construction of the application. For example, developing a compatible application means choosing compatible:

- System and built-in functions for use in SQL statements and procedural logic.
- 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 catalog views that are compatible with Oracle's data dictionary

For detailed information about the compatible SQL syntax, data types, and views, see SQL reference.

The compatibility offered by the procedures and functions that are part of the Built-in packages is documented in Built-in packages.

For information about using the compatible tools and utilities (EDB*Plus, EDB*Loader, DRITA, and EDB*Wrap) that are included with an EDB Postgres Advanced Server installation, see Tools, utilities, and components.

For applications written using the Oracle Call Interface (OCI), EDB's Open Client Library (OCL) provides interoperability with these applications. For detailed information about using the Open Client Library, see EDB OCL Connector.

9.1 Configuration Parameters Compatible with Oracle Databases

EDB Postgres Advanced Server supports the development and execution of applications compatible with PostgreSQL and Oracle. Some system behaviors can be altered to act in a more PostgreSQL or in a more Oracle compliant manner; these behaviors are controlled by configuration parameters. Modifying the parameters in the `postgresql.conf` file changes the behavior for all databases in the cluster, while a user or group can `SET` the parameter value on the command line, effecting only their session. These parameters are:

- `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 Advanced Server to the correct Oracle installation directory. See oracle_home.

9.1.1 edb_redwood_date

When `DATE` appears as the data type of a column in the commands, it is translated to `TIMESTAMP` at the time the table definition is stored in the data base if the configuration parameter `edb_redwood_date` is set to `TRUE`. Thus, a time component will also be stored in the column along 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 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 such as 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, it is always internally translated to a `TIMESTAMP` and thus, can handle a time component if present.

See the *Database Compatibility for Oracle Developers Reference Guide* for more information about date/time data types.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

9.1.2 edb_redwood_raw_names

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, user names, etc. appear in uppercase letters when viewed from Oracle catalogs (for a complete list of supported catalog views, see the *Database Compatibility for Oracle Developers Catalog Views Guide*). 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 they are stored in the PostgreSQL system catalogs when viewed from the Oracle catalogs. Thus, names created without enclosing quotation marks appear in lowercase as expected in PostgreSQL. Names created with enclosing quotation marks appear exactly 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;
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 uppercase except for the `Mixed_Case` name, which appears as created and also with enclosing quotation marks.

```
edb=> SELECT * FROM USER_TABLES;
 schema_name  |  table_name  | tablespace_name | status | temporary
--------------+--------------+-----------------+--------+-----------
 REDUSER      | ALL_LOWER    |                 | VALID  | N
 REDUSER      | ALL_UPPER    |                 | VALID  | N
 REDUSER      | "Mixed_Case" |                 | VALID  | N
(3 rows)
```

When viewed with `edb_redwood_raw_names` set to `TRUE`, the names appear in lowercase except for the `Mixed_Case` name, which appears as created, but now without the enclosing quotation marks.

```
edb=> SET edb_redwood_raw_names TO true;
SET
edb=> SELECT * FROM USER_TABLES;
 schema_name  | table_name | tablespace_name | status | temporary
--------------+------------+-----------------+--------+-----------
 reduser      | all_lower  |                 | VALID  | N
 reduser      | all_upper  |                 | VALID  | N
 reduser      | Mixed_Case |                 | VALID  | N
(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';
 schemaname  | tablename  | tableowner
-------------+------------+------------
 reduser     | all_lower  | reduser
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 reduser     | all_upper  | reduser
 reduser     | Mixed_Case | reduser
(3 rows)
```

9.1.3 edb_redwood_strings

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 aforementioned 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.

The following example illustrates the difference.

The sample application introduced in the next section contains a table of employees. This table has a column named `comm` that is 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        2,975.00
MARTIN       1,250.00   1,400.00
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
BLAKE        2,850.00
CLARK        2,450.00
SCOTT        3,000.00
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)
```

9.1.4 edb_stmt_level_tx

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 have not yet been 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 be started.

If `edb_stmt_level_tx` is set to `TRUE`, then an exception will not automatically roll back prior uncommitted database updates, emulating the Oracle behavior. If `edb_stmt_level_tx` is set to `FALSE`, then an exception will roll back uncommitted database updates.

> **Note**
>
> Use `edb_stmt_level_tx` set to `TRUE` only when absolutely necessary, as this may cause a negative performance impact.

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. Note that in PSQL, the command `\set AUTOCOMMIT off` must be issued, otherwise every statement commits automatically defeating the purpose of this demonstration of the effect of `edb_stmt_level_tx`.

```
 \set AUTOCOMMIT off
SET edb_stmt_level_tx TO off;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
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 > 9000;

empno  | ename | deptno
-------+-------+--------
(0 rows)
```

In the following example, with `edb_stmt_level_tx` set to `TRUE`, the first `INSERT` command has not been rolled back after the error on the second `INSERT` command. At this point, the first `INSERT` command can either be committed or rolled back.

```
 \set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
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 > 9000;

empno  | ename | deptno
-------+-------+--------
9001   | JONES |     40
(1 row)

COMMIT;
```

A `ROLLBACK` command could have been issued instead of the `COMMIT` command in which case the insert of employee number `9001` would have been rolled back as well.

9.1.5 oracle_home

Before creating a link to an Oracle server, you must direct 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 will override the `LD_LIBRARY_PATH` environment variable in Linux and `PATH` environment variable in Windows.

> **Note**
>
> The `oracle_home` configuration parameter should provide the correct path to the Oracle client i.e. `OCI` library.

To set the `oracle_home` configuration parameter in the `postgresql.conf` file, edit the file, adding the following line:

```
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`.

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.

9.2 About the Examples Used in this Guide

The examples shown in this guide are illustrated using the PSQL program. The prompt that normally appears when using PSQL is omitted in these examples to provide extra clarity for the point being demonstrated.

```
Examples and output from examples are shown in fixed-width, white font on a dark background.
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Also note the following points:

- During installation of the EDB Postgres Advanced Server the selection for configuration and defaults compatible with Oracle databases must be chosen in order to reproduce the same results as the examples shown in this guide. A default compatible configuration can be verified by issuing the following commands in PSQL and obtaining the same results as shown below.

```
SHOW edb_redwood_date;

 edb_redwood_date
------------------
 on

SHOW datestyle;

 DateStyle
--------------
 Redwood, DMY

SHOW edb_redwood_strings;

 edb_redwood_strings
---------------------
 on
```

- The examples use the sample tables, `dept`, `emp`, and `jobhist`, created and loaded when Advanced Server is installed. The `emp` table is installed with triggers that must be disabled in order to reproduce the same results as shown in this guide. Log onto Advanced Server as the `enterprisedb` superuser and disable the triggers by issuing the following command.

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

The triggers on the `emp` table can later be re-activated with the following command.

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

9.3 SQL Tutorial

Advanced Server is a *relational database management system* (RDBMS). That means it is a system for managing data stored in *relations*. A relation is essentially a mathematical term for a *table*. The notion of storing data in tables is so commonplace today that it might seem inherently obvious, but there are a number of other ways of organizing databases. Files and directories on Unix-like operating systems form an example of a hierarchical database. A more modern development is the object-oriented database.

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 is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

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

9.3.1 Sample Database

Throughout this documentation we will be working with a sample database to help explain some basic to advanced level database concepts.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

9.3.1.1 Sample Database Installation

When Advanced Server is installed a sample database named, `edb`, is automatically created. This sample database contains the tables and programs used throughout this document by executing the script, `edb-sample.sql`, located in the `/usr/edb/as11/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 will be 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:

```
SHOW SEARCH_PATH;
```

Altering the search path can be done using commands in PSQL.

9.3.1.2 Sample Database Description

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, etc. 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

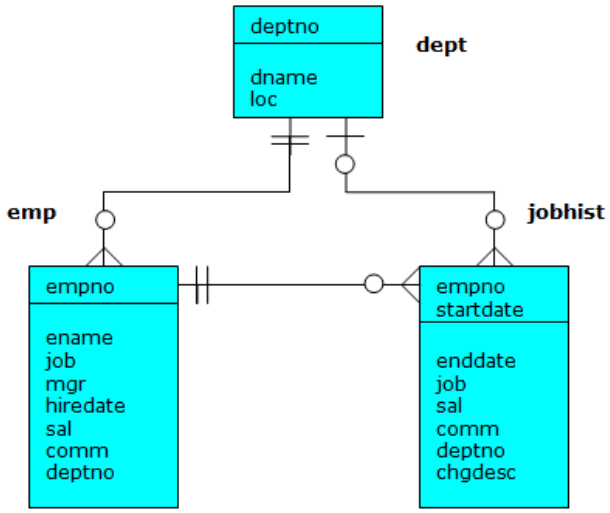


Fig. 1: Sample Database Tables

The following is the `edb-sample.sql` script.

```
--
--  Script that creates the 'sample' tables, views, procedures,
--  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           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(10),
   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)
);
--
--  Create the 'jobhist' table
--
CREATE TABLE jobhist (
   empno            NUMBER(4) NOT NULL,
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
   startdate       DATE NOT NULL,
   enddate         DATE,
   job             VARCHAR2(9),
   sal             NUMBER(7,2),
   comm            NUMBER(7,2),
   deptno          NUMBER(2),
   chgdesc         VARCHAR2(80),
   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';
--
--  Sequence to generate values for function 'new_empno'.
--
CREATE SEQUENCE next_empno START WITH 8000 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;
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,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'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,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',1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,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',800,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,'New Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,'New Hire');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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',1250,1400,30,'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,'New Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-88','CLERK',1000,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-89','CLERK',1040,NULL,20,'Raise');
INSERT INTO jobhist VALUES (7788,'05-MAY-90',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',1500,0,30,'New Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-83','CLERK',950,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7900,'15-JAN-83',NULL,'CLERK',950,NULL,30,'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,'NewHire');
--
--  Populate statistics table and view (pg_statistic/pg_stats)
--
ANALYZE dept;
ANALYZE emp;
ANALYZE jobhist;
--
--  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(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;
/
--
--  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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
       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);
       DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
       DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
/
--
--  Procedure that queries the 'emp' table based on
--  department number and employee number or name.  Returns
--  employee number and name as IN OUT parameters and job,
--  hire date, and salary as OUT parameters.
--
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(10);
   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);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
   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 VALUES (v_empno, p_ename, 'CLERK', 7782,
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
       TRUNC(SYSDATE), 950.00, NULL, p_deptno);
   SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
       v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
       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', 7698,
       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:';
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
       RAISE INFO '%', SQLERRM;
       RAISE INFO 'The following is SQLSTATE:';
       RAISE INFO '%', 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 ';
   ELSIF UPDATING THEN
       v_action := ' updated employee(s) on ';
   ELSIF DELETING THEN
       v_action := ' deleted 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
   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;
END;
/
--
--  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
   --  number and returns the corresponding department name.
   --
   FUNCTION get_dept_name (
       p_deptno        IN NUMBER
   ) RETURN VARCHAR2
   IS
       v_dname         VARCHAR2(14);
   BEGIN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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.
--
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
)
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
)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
   AS
   BEGIN
       DELETE FROM emp WHERE empno = p_empno;
   END;
END;
/
COMMIT;
```

9.3.2 Creating a New Table

A new table is created 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 just the minimal information needed to define a table.

```
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 into PSQL with line breaks. PSQL will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) may be used freely in SQL commands. That means you can type the command aligned differently than the above, or even all on one line. Two dashes ("--") introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

`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.

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 any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

9.3.3 Populating a Table With Rows

The `INSERT` statement is used to populate a table with rows:

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

Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes ('), as in the example. The `DATE` type is actually quite flexible in what it accepts, but for this tutorial we will stick to 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 wish or even omit some columns, e.g., if the commission is unknown:

```
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.

9.3.4 Querying a Table

To retrieve data from a table, the table is *queried*. An SQL `SELECT` statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and 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.

```
SELECT * FROM emp;
```

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

```
 empno| ename |  job     | mgr  |      hiredate       | sal     | comm  | deptno
------+-------+----------+------+---------------------+---------+-------+-------
7369 | SMITH | CLERK    | 7902 | 17-DEC-80 00:00:00 | 800.00  |       | 20
7499 | ALLEN | SALESMAN | 7698 | 20-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 | 02-APR-81 00:00:00 | 2975.00 |       | 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 | 2850.00 |       | 30
7782 | CLARK | MANAGER  | 7839 | 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 | 7698 | 08-SEP-81 00:00:00 | 1500.00 | 0.00  | 30
7876 | ADAMS | CLERK    | 7788 | 23-MAY-87 00:00:00 | 1100.00 |       | 20
7900 | JAMES | CLERK    | 7698 | 03-DEC-81 00:00:00 |  950.00 |       | 30
7902 | FORD  | ANALYST  | 7566 | 03-DEC-81 00:00:00 | 3000.00 |       | 20
7934 | MILLER| CLERK    | 7782 | 23-JAN-82 00:00:00 | 1300.00 |       | 10
(14 rows)
```

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

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

 ename   |   sal   | yearly_salary | deptno
---------+---------+---------------+--------
 SMITH   |  800.00 |      19200.00 |     20
 ALLEN   | 1600.00 |      38400.00 |     30
 WARD    | 1250.00 |      30000.00 |     30
 JONES   | 2975.00 |      71400.00 |     20
 MARTIN  | 1250.00 |      30000.00 |     30
 BLAKE   | 2850.00 |      68400.00 |     30
 CLARK   | 2450.00 |      58800.00 |     10
 SCOTT   | 3000.00 |      72000.00 |     20
 KING    | 5000.00 |     120000.00 |     10
 TURNER  | 1500.00 |      36000.00 |     30
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 ADAMS   | 1100.00 |      26400.00 |     20
 JAMES   |  950.00 |      22800.00 |     30
 FORD    | 3000.00 |      72000.00 |     20
 MILLER  | 1300.00 |      31200.00 |     10
(14 rows)
```

Notice how the `AS` clause is used to re-label the output column. (The `AS` clause is optional.)

A query can be qualified by adding a `WHERE` clause that specifies which rows are wanted. The `WHERE` clause contains a Boolean (truth value) expression, and 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:

```
SELECT ename, sal, deptno FROM emp WHERE deptno = 20 AND sal > 1000;

 ename |   sal   | deptno
-------+---------+--------
 JONES | 2975.00 |     20
 SCOTT | 3000.00 |     20
 ADAMS | 1100.00 |     20
 FORD  | 3000.00 |     20
(4 rows)
```

You can request that the results of a query be returned in sorted order:

```
SELECT ename, sal, deptno FROM emp ORDER BY ename;

 ename  |   sal   | deptno
--------+---------+--------
 ADAMS  | 1100.00 |     20
 ALLEN  | 1600.00 |     30
 BLAKE  | 2850.00 |     30
 CLARK  | 2450.00 |     10
 FORD   | 3000.00 |     20
 JAMES  |  950.00 |     30
 JONES  | 2975.00 |     20
 KING   | 5000.00 |     10
 MARTIN | 1250.00 |     30
 MILLER | 1300.00 |     10
 SCOTT  | 3000.00 |     20
 SMITH  |  800.00 |     20
 TURNER | 1500.00 |     30
 WARD   | 1250.00 |     30
(14 rows)
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT job FROM emp;

   job
-----------
 ANALYST
 CLERK
 MANAGER
 PRESIDENT
 SALESMAN
(5 rows)
```

The following section shows how to obtain rows from more than one table in a single query.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

9.3.5 Joins Between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being 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. For example, say you wish to list all the employee records together with the name and location of the associated department. To do that, we need to compare the `deptno` column of each row of the `emp` table with the `deptno` column of all rows in the `dept` table, and select the pairs of rows where these values match. This would be accomplished by the following query:

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

 ename  |   sal   | deptno |   dname    |   loc
--------+---------+--------+------------+----------
 MILLER | 1300.00 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 2450.00 |     10 | ACCOUNTING | NEW YORK
 KING   | 5000.00 |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 3000.00 |     20 | RESEARCH   | DALLAS
 JONES  | 2975.00 |     20 | RESEARCH   | DALLAS
 SMITH  |  800.00 |     20 | RESEARCH   | DALLAS
 ADAMS  | 1100.00 |     20 | RESEARCH   | DALLAS
 FORD   | 3000.00 |     20 | RESEARCH   | DALLAS
 WARD   | 1250.00 |     30 | SALES      | CHICAGO
 TURNER | 1500.00 |     30 | SALES      | CHICAGO
 ALLEN  | 1600.00 |     30 | SALES      | CHICAGO
 BLAKE  | 2850.00 |     30 | SALES      | CHICAGO
 MARTIN | 1250.00 |     30 | SALES      | CHICAGO
 JAMES  |  950.00 |     30 | SALES      | CHICAGO
(14 rows)
```

Observe two things about the result set:

- There is no result row for department 40. This is because there is no matching entry in the `emp` table for department 40, so the join ignores the unmatched rows in the `dept` table. Shortly we will see how this can be fixed.
- It is more desirable to list the output columns qualified by table name rather than using * or leaving out the qualification as follows:

```
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 automatically found out which table they belong to, but it is good style to fully qualify column names in join queries:

Join queries of the kind seen thus far can also be written in this alternative form:

```
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 as the one above, but we show it here to help you understand the following topics.

You will notice that in all the above results for joins no employees were returned that belonged to department 40 and as a consequence, the record for department 40 never appears. Now we will figure out how we can get the department 40 record in the results despite the fact that there are no matching employees. What we want the query to do is to scan the `dept` table and for each row to find the matching `emp` row. If no matching row is found we want some “empty” values to be substituted for the `emp` table’s columns. This kind of query is called an *outer join*. (The joins we have seen so far are *inner joins*.) The command looks like this:

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 ename  |   sal   | deptno |   dname    |   loc
--------+---------+--------+------------+----------
 MILLER | 1300.00 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 2450.00 |     10 | ACCOUNTING | NEW YORK
 KING   | 5000.00 |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 3000.00 |     20 | RESEARCH   | DALLAS
 JONES  | 2975.00 |     20 | RESEARCH   | DALLAS
 SMITH  |  800.00 |     20 | RESEARCH   | DALLAS
 ADAMS  | 1100.00 |     20 | RESEARCH   | DALLAS
 FORD   | 3000.00 |     20 | RESEARCH   | DALLAS
 WARD   | 1250.00 |     30 | SALES      | CHICAGO
 TURNER | 1500.00 |     30 | SALES      | CHICAGO
 ALLEN  | 1600.00 |     30 | SALES      | CHICAGO
 BLAKE  | 2850.00 |     30 | SALES      | CHICAGO
 MARTIN | 1250.00 |     30 | SALES      | CHICAGO
 JAMES  |  950.00 |     30 | SALES      | CHICAGO
        |         |     40 | OPERATIONS | BOSTON
(15 rows)
```

This query is called a *left outer join* because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have 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 within the `WHERE` clause. The outer join operator is placed after the column name of the table for which null values should be substituted for unmatched rows. So for all the rows in the `dept` table that have no matching rows in the `emp` table, Advanced Server returns null for any select list expressions containing columns of `emp` . Hence the above example could be rewritten as:

```
 SELECT emp.ename, emp.sal, dept.deptno, dept.dname, dept.loc FROM dept, emp
 WHERE emp.deptno(+) = dept.deptno;

 ename  |   sal   | deptno |   dname    |   loc
--------+---------+--------+------------+----------
 MILLER | 1300.00 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 2450.00 |     10 | ACCOUNTING | NEW YORK
 KING   | 5000.00 |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 3000.00 |     20 | RESEARCH   | DALLAS
 JONES  | 2975.00 |     20 | RESEARCH   | DALLAS
 SMITH  |  800.00 |     20 | RESEARCH   | DALLAS
 ADAMS  | 1100.00 |     20 | RESEARCH   | DALLAS
 FORD   | 3000.00 |     20 | RESEARCH   | DALLAS
 WARD   | 1250.00 |     30 | SALES      | CHICAGO
 TURNER | 1500.00 |     30 | SALES      | CHICAGO
 ALLEN  | 1600.00 |     30 | SALES      | CHICAGO
 BLAKE  | 2850.00 |     30 | SALES      | CHICAGO
 MARTIN | 1250.00 |     30 | SALES      | CHICAGO
 JAMES  |  950.00 |     30 | SALES      | CHICAGO
        |         |     40 | OPERATIONS | BOSTON
(15 rows)
```

We can also join a table against itself. This is called a *self join.* As an example, suppose we wish to find the name of each employee along with the name of that employee's manager. So we need to compare the `mgr` column of each `emp` row to the `empno` column of all other `emp` rows.

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

  Employees and their Managers
------------------------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 FORD works for JONES
 SCOTT works for JONES
 WARD works for BLAKE
 TURNER works for BLAKE
 MARTIN works for BLAKE
 JAMES works for BLAKE
 ALLEN works for BLAKE
 MILLER works for CLARK
 ADAMS works for SCOTT
 CLARK works for KING
 BLAKE works for KING
 JONES works for KING
 SMITH works for FORD
(13 rows)
```

Here, the `emp` table has been re-labeled as `e1` to represent the employee row in the select list and in the join condition, and also as `e2` to represent the matching employee row acting as manager in the select list and in the join condition. These kinds of aliases can be used in other queries to save some typing, for example:

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

 ename  | mgr  | deptno |   dname    |   loc
--------+------+--------+------------+----------
 MILLER | 7782 |     10 | ACCOUNTING | NEW YORK
 CLARK  | 7839 |     10 | ACCOUNTING | NEW YORK
 KING   |      |     10 | ACCOUNTING | NEW YORK
 SCOTT  | 7566 |     20 | RESEARCH   | DALLAS
 JONES  | 7839 |     20 | RESEARCH   | DALLAS
 SMITH  | 7902 |     20 | RESEARCH   | DALLAS
 ADAMS  | 7788 |     20 | RESEARCH   | DALLAS
 FORD   | 7566 |     20 | RESEARCH   | DALLAS
 WARD   | 7698 |     30 | SALES      | CHICAGO
 TURNER | 7698 |     30 | SALES      | CHICAGO
 ALLEN  | 7698 |     30 | SALES      | CHICAGO
 BLAKE  | 7839 |     30 | SALES      | CHICAGO
 MARTIN | 7698 |     30 | SALES      | CHICAGO
 JAMES  | 7698 |     30 | SALES      | CHICAGO
(14 rows)
```

This style of abbreviating will be encountered quite frequently.

9.3.6 Aggregate Functions

Like most other relational database products, Advanced Server supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the `COUNT`, `SUM`, `AVG` (average), `MAX` (maximum), and `MIN` (minimum) over a set of rows.

As an example, the highest and lowest salaries can be found with the following query:

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

 highest_salary | lowest_salary
----------------+---------------
        5000.00 |        800.00
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

If we wanted to find the employee with the largest salary, we may be tempted to try:

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

ERROR:  aggregates not allowed in WHERE clause
```

This does not work because the aggregate function, `MAX`, cannot be used in the `WHERE` clause. This restriction exists because the `WHERE` clause determines the rows that will go into the aggregation stage so it has to be evaluated before aggregate functions are computed. However, the query can be restated to accomplish the intended result by using a *subquery*:

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

 ename
-------
 KING
(1 row)
```

The subquery is an independent computation that obtains its own result separately from the outer query.

Aggregates are also very 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;

 deptno |   max
--------+---------
     10 | 5000.00
     20 | 3000.00
     30 | 2850.00
(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;

 deptno |   max
--------+---------
     10 | 5000.00
     20 | 3000.00
(2 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;

 deptno |   max
--------+---------
     20 | 3000.00
(1 row)
```

There is 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 have been grouped and aggregate functions have been computed for each group.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

9.3.7 Updates

The column values of existing rows can be changed 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:

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

 ename |   sal
-------+---------
 JONES | 2975.00
 BLAKE | 2850.00
 CLARK | 2450.00
(3 rows)

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

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

 ename |   sal
-------+---------
 JONES | 3272.50
 BLAKE | 3135.00
 CLARK | 2695.00
(3 rows)
```

9.3.8 Deletions

Rows can be removed 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`.

```
SELECT ename, deptno FROM emp;

 ename  | deptno
--------+--------
 SMITH  |     20
 ALLEN  |     30
 WARD   |     30
 JONES  |     20
 MARTIN |     30
 BLAKE  |     30
 CLARK  |     10
 SCOTT  |     20
 KING   |     10
 TURNER |     30
 ADAMS  |     20
 JAMES  |     30
 FORD   |     20
 MILLER |     10
(14 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DELETE FROM emp WHERE deptno = 20;

SELECT ename, deptno FROM emp;
 ename  | deptno
--------+--------
 ALLEN  |     30
 WARD   |     30
 MARTIN |     30
 BLAKE  |     30
 CLARK  |     10
 KING   |     10
 TURNER |     30
 JAMES  |     30
 MILLER |     10
(9 rows)
```

Be extremely careful of giving a `DELETE` command without a `WHERE` clause such as the following:

```
DELETE FROM tablename;
```

This statement will remove all rows from the given table, leaving it completely empty. The system will not request confirmation before doing this.

9.3.9 The SQL Language

Advanced Server supports SQL language that is compatible with Oracle syntax as well as syntax and commands for extended functionality (functionality that does not provide database compatibility for Oracle or support Oracle-styled applications).

The *Database Compatibility for Oracle Developers SQL Guide* provides detailed information about:

- Compatible SQL syntax and language elements
- Data types
- Supported SQL command syntax

To review a copy of the guide, visit the Advanced Server website at:

https://www.enterprisedb.com/docs

9.4 Advanced Concepts

The previous section discussed the basics of using SQL to store and access your data in Advanced Server. This section discusses more advanced SQL features that may simplify management and prevent loss or corruption of your data.

9.4.1 Views

Consider the following `SELECT` command.

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 ename  |   sal   | yearly_salary | deptno
--------+---------+---------------+--------
 SMITH  |  800.00 |      19200.00 |     20
 ALLEN  | 1600.00 |      38400.00 |     30
 WARD   | 1250.00 |      30000.00 |     30
 JONES  | 2975.00 |      71400.00 |     20
 MARTIN | 1250.00 |      30000.00 |     30
 BLAKE  | 2850.00 |      68400.00 |     30
 CLARK  | 2450.00 |      58800.00 |     10
 SCOTT  | 3000.00 |      72000.00 |     20
 KING   | 5000.00 |     120000.00 |     10
 TURNER | 1500.00 |      36000.00 |     30
 ADAMS  | 1100.00 |      26400.00 |     20
 JAMES  |  950.00 |      22800.00 |     30
 FORD   | 3000.00 |      72000.00 |     20
 MILLER | 1300.00 |      31200.00 |     10
(14 rows)
```

If this is a query that is used repeatedly, a shorthand method of reusing this query without re-typing the entire `SELECT` command each time is to create a *view* as shown below.

```
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.

```
SELECT * FROM employee_pay;

 ename  |   sal   | yearly_salary | deptno
--------+---------+---------------+--------
 SMITH  |  800.00 |      19200.00 |     20
 ALLEN  | 1600.00 |      38400.00 |     30
 WARD   | 1250.00 |      30000.00 |     30
 JONES  | 2975.00 |      71400.00 |     20
 MARTIN | 1250.00 |      30000.00 |     30
 BLAKE  | 2850.00 |      68400.00 |     30
 CLARK  | 2450.00 |      58800.00 |     10
 SCOTT  | 3000.00 |      72000.00 |     20
 KING   | 5000.00 |     120000.00 |     10
 TURNER | 1500.00 |      36000.00 |     30
 ADAMS  | 1100.00 |      26400.00 |     20
 JAMES  |  950.00 |      22800.00 |     30
 FORD   | 3000.00 |      72000.00 |     20
 MILLER | 1300.00 |      31200.00 |     10
(14 rows)
```

Making liberal use of views is a key aspect of good SQL database design. Views provide a consistent interface that encapsulate details of the structure of your tables which may change as your application evolves.

Views can be used in almost any place a real table can be used. Building views upon other views is not uncommon.

9.4.2 Foreign Keys

Suppose you want to make sure all employees belong to a valid department. This is called maintaining the *referential integrity* of your data. In simplistic

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

database systems this would be implemented (if at all) 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 very inconvenient. Advanced Server can make it easier for you.

A modified version of the `emp` table presented in Creating a New Table is shown in this section 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 an attempt is made to issue the following `INSERT` command in the sample `emp` table, the foreign key constraint, `emp_ref_dept_fk`, ensures that department `50` exists in the `dept` table. Since it does not, 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".
```

The behavior of foreign keys can be finely tuned to your application. Making correct use of foreign keys will definitely improve the quality of your database applications, so you are strongly encouraged to learn more about them.

9.4.3 The ROWNUM Pseudo-Column

`ROWNUM` is a pseudo-column that is 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 will have `ROWNUM` of `1`; the second row will have `ROWNUM` of `2` and so on.

This feature can be used to limit the number of rows retrieved by a query. This is demonstrated in the following example:

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

 empno | ename |   job
-------+-------+----------
 7369  | SMITH | CLERK
 7499  | ALLEN | SALESMAN
 7521  | WARD  | SALESMAN
 7566  | JONES | MANAGER
(4 rows)
```

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, but the `ROWNUM` values may not necessarily be in ascending order as shown in the following example:

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

 rownum | empno | ename |   job
--------+-------+-------+----------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
      2 |  7499 | ALLEN | SALESMAN
      4 |  7566 | JONES | MANAGER
      1 |  7369 | SMITH | CLERK
      3 |  7521 | WARD  | SALESMAN
(4 rows)
```

The following example shows how a sequence number can be added to every row in the `jobhist` table. First a new column named, `seqno`, is added to the table and then `seqno` is set 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;

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

9.4.4 Synonyms

A *synonym* is an identifier that can be used to reference another database object in a SQL statement. A synonym is useful in cases where a database object would normally require 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.

Advanced Server supports synonyms for:

- tables
- views
- materialized views
- sequences
- procedures
- functions
- types
- objects that are accessible through a database link
- other synonyms

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Neither the referenced schema or referenced object must exist at the time that you create the synonym; a synonym may refer to a non-existent object or schema. A synonym will become invalid if you drop the referenced object or schema. You must explicitly drop a synonym to remove it.

As with any other schema object, 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 will resolve 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 Advanced Server executes an SQL command, the privileges of the current user are checked against the synonym's underlying database object; if the user does not have the proper permissions for that object, the SQL command will fail.

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`

`syn_name` is the name of the synonym. A synonym name must be unique within a schema.

`schema`

`schema` specifies the name of the schema that the synonym resides in. If you do not specify a schema name, the synonym is created in the first existing schema in your search path.

`object_name`

`object_name` specifies the name of the object.

`object_schema`

`object_schema` specifies the name of the schema that the object resides in.

`dblink_name`

`dblink_name` specifies the name of the database link through which a target object may be accessed.

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 just a shorthand way to write:

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

The following example creates a synonym named `personnel` that refers to the `enterprisedb.emp` table.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

Unless the synonym is schema qualified in the `CREATE SYNONYM` command, it will be 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)
```

In our example, if a schema named `development` does not exist, the synonym will be created in the schema named `accounting`.

Now, the `emp` table in the `enterprisedb` schema can be referenced 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);

SELECT * FROM personnel;

 empno | ename   |  job     | mgr |    hiredate      | sal    | comm  |deptno
-------+---------+----------+-----+------------------+--------+-------+------
 7369  | SMITH   | CLERK    |7902 |17-DEC-80 00:00:00| 800.00 |       |  20
 7499  | ALLEN   | SALESMAN |7698 |20-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 |02-APR-81 00:00:00| 2975.00|       |  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| 2850.00|       |  30
 7782  | CLARK   | MANAGER  |7839 |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 |7698 |08-SEP-81 00:00:00| 1500.00| 0.00  |  30
 7876  | ADAMS   | CLERK    |7788 |23-MAY-87 00:00:00| 1100.00|       |  20
 7900  | JAMES   | CLERK    |7698 |03-DEC-81 00:00:00| 950.00 |       |  30
 7902  | FORD    | ANALYST  |7566 |03-DEC-81 00:00:00| 3000.00|       |  20
 7934  | MILLER  | CLERK    |7782 |23-JAN-82 00:00:00| 1300.00|       |  10
 8142  | ANDERSON| CLERK    |7902 |17-DEC-06 00:00:00| 1300.00|       |  20
(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 within a schema.

`schema`

`schema` specifies the name of the schema in which the synonym resides.

Like any other object that can be schema-qualified, you may have two synonyms with the same name in your search path. To disambiguate the name of the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

synonym that you are dropping, include a schema name. Unless a synonym is schema qualified in the `DROP SYNONYM` command, 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>;
```

The following example drops the synonym, `personnel`:

```
DROP SYNONYM personnel;
```

9.4.5 Hierarchical Queries

A *hierarchical query* is a type of query that returns the rows of the result set in a hierarchical order based upon data forming a parent-child relationship. A hierarchy is typically represented by an inverted tree structure. The tree is comprised of interconnected *nodes*. Each node may 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 - e.g., the trivial case where the tree is comprised of a single node. In this case it 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**
>
> It is possible that a single, given row may appear in more than one tree and thus appear more than once in the result set.

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. The context of where the `CONNECT BY` clause and its associated optional clauses appear in the `SELECT` command is shown below.

```
SELECT <select_list> FROM <table_expression> [ WHERE ...]
  [ START WITH <start_expression> ]
    CONNECT BY { PRIOR <parent_expr> = <child_expr> |
      <child_expr> = PRIOR <parent_expr> }
  [ ORDER SIBLINGS BY <column1> [ ASC | DESC ]
      [, <column2> [ ASC | DESC ] ] ...
  [ GROUP BY ...]
  [ HAVING ...]
  [ <other> ...]
```

`select_list` is one or more expressions that comprise 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. The clauses pertinent to hierarchical queries, `START WITH, CONNECT BY`, and `ORDER SIBLINGS BY` are described in the following sections.

> **Note**
>
> At this time, Advanced Server does not support the use of `AND` (or other operators) in the `CONNECT BY` clause.

9.4.5.1 Defining the Parent/Child Relationship

For any given row, its parent and its children are determined by the `CONNECT BY` clause. The `CONNECT BY` clause must consist of two expressions

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

compared with the equals (=) operator. In addition, one of these two expressions must be preceded by the keyword, `PRIOR`.

For any given row, to determine its children:

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`.

By iteratively repeating this process treating each child node found in the prior steps as a parent, an inverted tree of nodes is constructed. The process is complete when the final set of child nodes has no children of their 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 - i.e., the rows that are the initial parent nodes upon which the algorithm described previously is to be applied. This is further explained in the following section.

9.4.5.2 Selecting the Root Nodes

The `START WITH` clause is used to determine the row(s) selected by `table_expression` that are to be used 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.

9.4.5.3 Organization Tree in the Sample Application

Consider the `emp` table of the sample application. The rows of the `emp` table form a hierarchy based upon 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 so he has no manager, therefore KING's `mgr` column is null. Also, it is possible for an employee to act as a manager for more than one employee. This relationship forms a typical, tree-structured, hierarchical organization chart as illustrated below.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

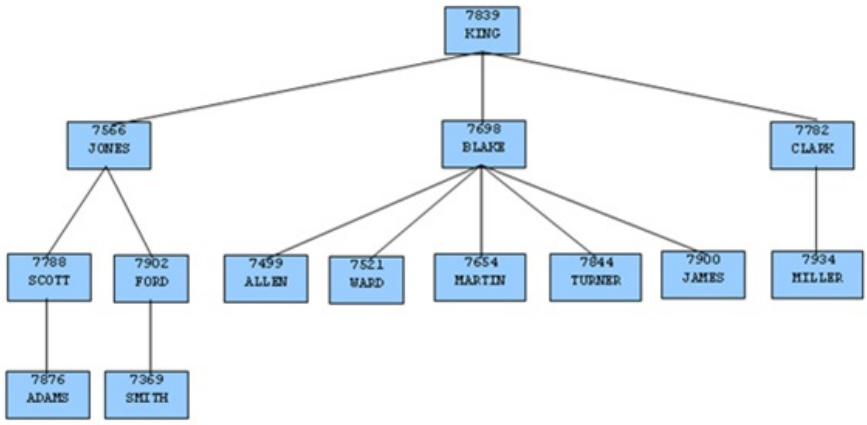


Fig. 1: Employee Organization Hierarchy

To form a hierarchical query based upon 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` which is true for `JONES, BLAKE`, and `CLARK` (these are 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 are `SCOTT` and `FORD` in this 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 shown below.

```
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. Below is the output from this query.

```
 ename  | empno | mgr
--------+-------+------
KING    |  7839 |
JONES   |  7566 | 7839
SCOTT   |  7788 | 7566
ADAMS   |  7876 | 7788
FORD    |  7902 | 7566
SMITH   |  7369 | 7902
BLAKE   |  7698 | 7839
ALLEN   |  7499 | 7698
WARD    |  7521 | 7698
MARTIN  |  7654 | 7698
TURNER  |  7844 | 7698
JAMES   |  7900 | 7698
CLARK   |  7782 | 7839
MILLER  |  7934 | 7782
(14 rows)
```

9.4.5.4 Node Level

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`LEVEL` is a pseudo-column that can be used 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.

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.

```
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 follows.

```
level |  employee   | empno | mgr
------+-------------+-------+------
   1  | KING        |  7839 |
   2  |   JONES     |  7566 | 7839
   3  |     SCOTT   |  7788 | 7566
   4  |       ADAMS |  7876 | 7788
   3  |     FORD    |  7902 | 7566
   4  |       SMITH |  7369 | 7902
   2  |   BLAKE     |  7698 | 7839
   3  |     ALLEN   |  7499 | 7698
   3  |     WARD    |  7521 | 7698
   3  |     MARTIN  |  7654 | 7698
   3  |     TURNER  |  7844 | 7698
   3  |     JAMES   |  7900 | 7698
   2  |   CLARK     |  7782 | 7839
   3  |     MILLER  |  7934 | 7782
(14 rows)
```

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.

9.4.5.5 Ordering the Siblings

The result set can be ordered so the siblings appear in ascending or descending order by selected column value(s) using the `ORDER SIBLINGS BY` clause. This is a special case of the `ORDER BY` clause that can be used only with hierarchical queries.

The previous query is further modified with the addition of `ORDER SIBLINGS BY` ename `ASC`.

```
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.

```
level |  employee   | empno | mgr
------+-------------+-------+------
  1   | KING        |  7839 |
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  2   |   BLAKE     |  7698 | 7839
  3   |     ALLEN   |  7499 | 7698
  3   |     JAMES   |  7900 | 7698
  3   |     MARTIN  |  7654 | 7698
  3   |     TURNER  |  7844 | 7698
  3   |     WARD    |  7521 | 7698
  2   |   CLARK     |  7782 | 7839
  3   |     MILLER  |  7934 | 7782
  2   |   JONES     |  7566 | 7839
  3   |     FORD    |  7902 | 7566
  4   |       SMITH |  7369 | 7902
  3   |     SCOTT   |  7788 | 7566
  4   |       ADAMS |  7876 | 7788
(14 rows)
```

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.

```
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 do not satisfy the `WHERE` clause have been eliminated from the output.

```
 level | employee  | empno | mgr
------+-----------+-------+------
  1   | BLAKE     |  7698 | 7839
  1   | CLARK     |  7782 | 7839
  2   |   MILLER  |  7934 | 7782
  1   | JONES     |  7566 | 7839
  3   |     SMITH |  7369 | 7902
  3   |     ADAMS |  7876 | 7788
(6 rows)
```

9.4.5.6 Retrieving the Root Node with CONNECT_BY_ROOT

`CONNECT_BY_ROOT` is a unary operator that can be used to qualify a column in order 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, which in the case of `CONNECT_BY_ROOT` , is the column name following the `CONNECT_BY_ROOT` keyword.

In the context of the `SELECT` list, the `CONNECT_BY_ROOT` operator is shown by the following.

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

The following are some points to note about the `CONNECT_BY_ROOT` operator.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- The `CONNECT_BY_ROOT` operator can be used in the `SELECT` list, the `WHERE` clause, the `GROUP BY` clause, the `HAVING` clause, the `ORDER BY` clause, and the `ORDER SIBLINGS BY` clause as long as the `SELECT` command is for a hierarchical query.
- The `CONNECT_BY_ROOT` operator cannot be used in the `CONNECT BY` clause or the `START WITH` clause of the hierarchical query.
- It is possible to apply `CONNECT_BY_ROOT` to an expression involving a column, but to do so, the expression must be enclosed within parentheses.

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`.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
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;
```

Note that 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.

```
 level | employee  | empno | mgr  | mgr empno | mgr ename
------+-----------+-------+------+-----------+-----------
   1   | BLAKE     |  7698 | 7839 |      7698 | BLAKE
   2   |   ALLEN   |  7499 | 7698 |      7698 | BLAKE
   2   |   JAMES   |  7900 | 7698 |      7698 | BLAKE
   2   |   MARTIN  |  7654 | 7698 |      7698 | BLAKE
   2   |   TURNER  |  7844 | 7698 |      7698 | BLAKE
   2   |   WARD    |  7521 | 7698 |      7698 | BLAKE
   1   | CLARK     |  7782 | 7839 |      7782 | CLARK
   2   |   MILLER  |  7934 | 7782 |      7782 | CLARK
   1   | JONES     |  7566 | 7839 |      7566 | JONES
   2   |   FORD    |  7902 | 7566 |      7566 | JONES
   3   |     SMITH |  7369 | 7902 |      7566 | JONES
   2   |   SCOTT   |  7788 | 7566 |      7566 | JONES
   3   |     ADAMS |  7876 | 7788 |      7566 | JONES
(13 rows)
```

The following is a similar query, but producing only one tree starting with the single, top-level, employee where the `mgr` column is null.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
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 particular query.

```
 level |  employee   | empno | mgr  | mgr empno | mgr ename
------+-------------+-------+------+-----------+-----------
    1 | KING        |  7839 |      |      7839 | KING
    2 |   BLAKE     |  7698 | 7839 |      7839 | KING
    3 |     ALLEN   |  7499 | 7698 |      7839 | KING
    3 |     JAMES   |  7900 | 7698 |      7839 | KING
    3 |     MARTIN  |  7654 | 7698 |      7839 | KING
    3 |     TURNER  |  7844 | 7698 |      7839 | KING
    3 |     WARD    |  7521 | 7698 |      7839 | KING
    2 |   CLARK     |  7782 | 7839 |      7839 | KING
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    3 |     MILLER   |  7934 | 7782 |      7839 | KING
    2 |   JONES      |  7566 | 7839 |      7839 | KING
    3 |     FORD     |  7902 | 7566 |      7839 | KING
    4 |       SMITH  |  7369 | 7902 |      7839 | KING
    3 |     SCOTT    |  7788 | 7566 |      7839 | KING
    4 |       ADAMS  |  7876 | 7788 |      7839 | KING
(14 rows)
```

By contrast, the following example omits the `START WITH` clause thereby resulting in fourteen trees.

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

The following is 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.

```
 level |  employee    | empno | mgr  | mgr empno | mgr ename
-------+--------------+-------+------+-----------+-----------
    1 | ADAMS        |  7876 | 7788 |      7876 | ADAMS
    1 | ALLEN        |  7499 | 7698 |      7499 | ALLEN
    1 | BLAKE        |  7698 | 7839 |      7698 | BLAKE
    2 |   ALLEN      |  7499 | 7698 |      7698 | BLAKE
    2 |   JAMES      |  7900 | 7698 |      7698 | BLAKE
    2 |   MARTIN     |  7654 | 7698 |      7698 | BLAKE
    2 |   TURNER     |  7844 | 7698 |      7698 | BLAKE
    2 |   WARD       |  7521 | 7698 |      7698 | BLAKE
    1 | CLARK        |  7782 | 7839 |      7782 | CLARK
    2 |   MILLER     |  7934 | 7782 |      7782 | CLARK
    1 | FORD         |  7902 | 7566 |      7902 | FORD
    2 |   SMITH      |  7369 | 7902 |      7902 | FORD
    1 | JAMES        |  7900 | 7698 |      7900 | JAMES
    1 | JONES        |  7566 | 7839 |      7566 | JONES
    2 |   FORD       |  7902 | 7566 |      7566 | JONES
    3 |     SMITH    |  7369 | 7902 |      7566 | JONES
    2 |   SCOTT      |  7788 | 7566 |      7566 | JONES
    3 |     ADAMS    |  7876 | 7788 |      7566 | JONES
    1 | KING         |  7839 |      |      7839 | KING
    2 |   BLAKE      |  7698 | 7839 |      7839 | KING
    3 |     ALLEN    |  7499 | 7698 |      7839 | KING
    3 |     JAMES    |  7900 | 7698 |      7839 | KING
    3 |     MARTIN   |  7654 | 7698 |      7839 | KING
    3 |     TURNER   |  7844 | 7698 |      7839 | KING
    3 |     WARD     |  7521 | 7698 |      7839 | KING
    2 |   CLARK      |  7782 | 7839 |      7839 | KING
    3 |     MILLER   |  7934 | 7782 |      7839 | KING
    2 |   JONES      |  7566 | 7839 |      7839 | KING
    3 |     FORD     |  7902 | 7566 |      7839 | KING
    4 |       SMITH  |  7369 | 7902 |      7839 | KING
    3 |     SCOTT    |  7788 | 7566 |      7839 | KING
    4 |       ADAMS  |  7876 | 7788 |      7839 | KING
    1 | MARTIN       |  7654 | 7698 |      7654 | MARTIN
    1 | MILLER       |  7934 | 7782 |      7934 | MILLER
    1 | SCOTT        |  7788 | 7566 |      7788 | SCOTT
    2 |   ADAMS      |  7876 | 7788 |      7788 | SCOTT
    1 | SMITH        |  7369 | 7902 |      7369 | SMITH
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    1 | TURNER      |  7844 | 7698 |      7844 | TURNER
    1 | WARD        |  7521 | 7698 |      7521 | WARD
(39 rows)
```

The following illustrates the unary operator effect of `CONNECT_BY_ROOT`. As shown in this example, when applied to an expression that is not enclosed in parentheses, the `CONNECT_BY_ROOT` operator affects only the term, `ename`, immediately following it. The subsequent concatenation of `|| ' manages ' || ename` is not 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;
```

The following is the output from the query. Note the values produced under the `top mgr/employee` column.

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

The following example uses the `CONNECT_BY_ROOT` operator on an expression enclosed in parentheses.

```
 SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
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;
```

The following is the output of the query. Note that the values of both `ename` and `empno` are affected by the `CONNECT_BY_ROOT` operator and as a result, return the values from the root node as shown under the `top mgr/empno` column.

```
 level | employee  | empno | mgr  |         top mgr/empno
------+-----------+-------+------+-------------------------------
    1 | BLAKE     |  7698 | 7839 | Manager BLAKE is emp # 7698
    2 |   ALLEN   |  7499 | 7698 | Manager BLAKE is emp # 7698
    2 |   JAMES   |  7900 | 7698 | Manager BLAKE is emp # 7698
    2 |   MARTIN  |  7654 | 7698 | Manager BLAKE is emp # 7698
    2 |   TURNER  |  7844 | 7698 | Manager BLAKE is emp # 7698
    2 |   WARD    |  7521 | 7698 | Manager BLAKE is emp # 7698
    1 | CLARK     |  7782 | 7839 | Manager CLARK is emp # 7782
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    2 |   MILLER  |  7934 | 7782 | Manager CLARK is emp # 7782
    1 | JONES     |  7566 | 7839 | Manager JONES is emp # 7566
    2 |   FORD    |  7902 | 7566 | Manager JONES is emp # 7566
    3 |     SMITH |  7369 | 7902 | Manager JONES is emp # 7566
    2 |   SCOTT   |  7788 | 7566 | Manager JONES is emp # 7566
    3 |     ADAMS |  7876 | 7788 | Manager JONES is emp # 7566
(13 rows)
```

9.4.5.7 Retrieving a Path with SYS_CONNECT_BY_PATH

`SYS_CONNECT_BY_PATH` is a function that works within a hierarchical query to retrieve the column values of a specified column that occur between the current node and the root node. The signature of the function is:

```
SYS_CONNECT_BY_PATH (<column>, <delimiter>)
```

The function takes two arguments:

`column` is the name of a column that resides within a table specified in the hierarchical query that is calling the function.

`delimiter` is the `varchar` value that separates each entry in the specified column.

The following example returns a list of employee names, and their managers; if the manager has a manager, that name is appended to the result:

```
edb=# SELECT level, ename , SYS_CONNECT_BY_PATH(ename, '/') managers
      FROM emp
      CONNECT BY PRIOR empno = mgr
      START WITH mgr IS NULL
      ORDER BY level, ename, managers;

 level | ename  |        managers
-------+--------+-------------------------
     1 | KING   | /KING
     2 | BLAKE  | /KING/BLAKE
     2 | CLARK  | /KING/CLARK
     2 | JONES  | /KING/JONES
     3 | ALLEN  | /KING/BLAKE/ALLEN
     3 | FORD   | /KING/JONES/FORD
     3 | JAMES  | /KING/BLAKE/JAMES
     3 | MARTIN | /KING/BLAKE/MARTIN
     3 | MILLER | /KING/CLARK/MILLER
     3 | SCOTT  | /KING/JONES/SCOTT
     3 | TURNER | /KING/BLAKE/TURNER
     3 | WARD   | /KING/BLAKE/WARD
     4 | ADAMS  | /KING/JONES/SCOTT/ADAMS
     4 | SMITH  | /KING/JONES/FORD/SMITH
(14 rows)
```

Within 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 Advanced Server implementation of `SYS_CONNECT_BY_PATH` does not support use of:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `SYS_CONNECT_BY_PATH` inside `CONNECT_BY_PATH`
- `SYS_CONNECT_BY_PATH` inside `SYS_CONNECT_BY_PATH`

9.4.6 Multidimensional Analysis

Multidimensional analysis refers to the process commonly used in data warehousing applications 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. The results associated with a particular set of dimensions are called *facts*. Facts are typically figures associated with product sales, profits, volumes, counts, etc.

In order to obtain these facts according to a set of dimensions in a relational database system, SQL aggregation is typically used. *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.

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, the `GROUPING` function and the `GROUPING_ID` function can be used in the `SELECT` list or the `HAVING` clause to aid with the interpretation of the results when these extensions are used.

> **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 SET loc = 'BOSTON' WHERE deptno = 20;
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 loc, dname, job, empno FROM emp e, dept d
WHERE e.deptno = d.deptno
ORDER BY 1, 2, 3, 4;

   loc    |   dname    |    job    | empno
----------+------------+-----------+-------
 BOSTON   | OPERATIONS | ANALYST   |  9002
 BOSTON   | OPERATIONS | CLERK     |  9001
 BOSTON   | OPERATIONS | MANAGER   |  9003
 BOSTON   | RESEARCH   | ANALYST   |  7788
 BOSTON   | RESEARCH   | ANALYST   |  7902
 BOSTON   | RESEARCH   | CLERK     |  7369
 BOSTON   | RESEARCH   | CLERK     |  7876
 BOSTON   | RESEARCH   | MANAGER   |  7566
 CHICAGO  | SALES      | CLERK     |  7900
 CHICAGO  | SALES      | MANAGER   |  7698
 CHICAGO  | SALES      | SALESMAN  |  7499
 CHICAGO  | SALES      | SALESMAN  |  7521
 CHICAGO  | SALES      | SALESMAN  |  7654
 CHICAGO  | SALES      | SALESMAN  |  7844
 NEW YORK | ACCOUNTING | CLERK     |  7934
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 NEW YORK | ACCOUNTING | MANAGER   |  7782
 NEW YORK | ACCOUNTING | PRESIDENT |  7839
(17 rows)
```

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.

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

```
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.

```
loc      |   dname    |    job    | employees
---------+------------+-----------+-----------
BOSTON   | OPERATIONS | ANALYST   |         1
BOSTON   | OPERATIONS | CLERK     |         1
BOSTON   | OPERATIONS | MANAGER   |         1
BOSTON   | RESEARCH   | ANALYST   |         2
BOSTON   | RESEARCH   | CLERK     |         2
BOSTON   | RESEARCH   | MANAGER   |         1
CHICAGO  | SALES      | CLERK     |         1
CHICAGO  | SALES      | MANAGER   |         1
CHICAGO  | SALES      | SALESMAN  |         4
NEW YORK | ACCOUNTING | CLERK     |         1
NEW YORK | ACCOUNTING | MANAGER   |         1
NEW YORK | ACCOUNTING | PRESIDENT |         1
(12 rows)
```

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 provides the ability to combine different types of groupings into a single result set.

The `GROUPING` and `GROUPING_ID` functions aid in the interpretation of the result set.

The additions provided by these extensions are discussed in more detail in the subsequent sections.

9.4.6.1 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 leftmost item in the list. Each successive item proceeding to the right moves down the hierarchy with the rightmost item being the lowest level.

The syntax for a single `ROLLUP` is as follows:

```
ROLLUP ( { <expr_1> | ( <expr_1a> [, <expr_1b> ] ...) }
  [, <expr_2> | ( <expr_2a> [, <expr_2b> ] ...) ] ...)
```

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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, within 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 within 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 also appear in the `ROLLUP expression_list` ; or they must be aggregate functions such as `COUNT, SUM, AVG, MIN,` or `MAX` ; or they must be constants or functions whose return values are independent of the individual rows in the group (for example, the `SYSDATE` function).

The `GROUP BY` clause may specify multiple `ROLLUP` extensions as well as multiple occurrences of other `GROUP BY` extensions and individual expressions.

The `ORDER BY` clause should be used if you want the output to display in a hierarchical or other meaningful structure. There is no guarantee on the order of the result set if no `ORDER BY` clause is specified.

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.

The following query produces a rollup based on a hierarchy of columns `loc, dname` , 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 following is the result of the query. There is 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` , and a grand total displayed on the last line.

```
loc      |   dname    |    job    | employees
---------+------------+-----------+-----------
BOSTON   | OPERATIONS | ANALYST   |         1
BOSTON   | OPERATIONS | CLERK     |         1
BOSTON   | OPERATIONS | MANAGER   |         1
BOSTON   | OPERATIONS |           |         3
BOSTON   | RESEARCH   | ANALYST   |         2
BOSTON   | RESEARCH   | CLERK     |         2
BOSTON   | RESEARCH   | MANAGER   |         1
BOSTON   | RESEARCH   |           |         5
BOSTON   |            |           |         8
CHICAGO  | SALES      | CLERK     |         1
CHICAGO  | SALES      | MANAGER   |         1
CHICAGO  | SALES      | SALESMAN  |         4
CHICAGO  | SALES      |           |         6
CHICAGO  |            |           |         6
NEW YORK | ACCOUNTING | CLERK     |         1
NEW YORK | ACCOUNTING | MANAGER   |         1
NEW YORK | ACCOUNTING | PRESIDENT |         1
NEW YORK | ACCOUNTING |           |         3
NEW YORK |            |           |         3
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
         |            |           |        17
(20 rows)
```

The following query shows the effect of combining items in the `ROLLUP` list within parenthesis.

```
 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;
```

In the output, note that there are no subtotals for `loc` and `dname` combinations as in the prior example.

```
 loc      |   dname    |    job    | employees
----------+------------+-----------+-----------
BOSTON    | OPERATIONS | ANALYST   |         1
BOSTON    | OPERATIONS | CLERK     |         1
BOSTON    | OPERATIONS | MANAGER   |         1
BOSTON    | RESEARCH   | ANALYST   |         2
BOSTON    | RESEARCH   | CLERK     |         2
BOSTON    | RESEARCH   | MANAGER   |         1
BOSTON    |            |           |         8
CHICAGO   | SALES      | CLERK     |         1
CHICAGO   | SALES      | MANAGER   |         1
CHICAGO   | SALES      | SALESMAN  |         4
CHICAGO   |            |           |         6
NEW YORK  | ACCOUNTING | CLERK     |         1
NEW YORK  | ACCOUNTING | MANAGER   |         1
NEW YORK  | ACCOUNTING | PRESIDENT |         1
NEW YORK  |            |           |         3
          |            |           |        17
(16 rows)
```

If the first two columns in the `ROLLUP` list are enclosed in parenthesis, the subtotal levels differ as well.

```
 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 is a subtotal for each unique `loc` and `dname` combination, but none for unique values of `loc` .

```
 loc      |   dname    |    job    | employees
----------+------------+-----------+-----------
BOSTON    | OPERATIONS | ANALYST   |         1
BOSTON    | OPERATIONS | CLERK     |         1
BOSTON    | OPERATIONS | MANAGER   |         1
BOSTON    | OPERATIONS |           |         3
BOSTON    | RESEARCH   | ANALYST   |         2
BOSTON    | RESEARCH   | CLERK     |         2
BOSTON    | RESEARCH   | MANAGER   |         1
BOSTON    | RESEARCH   |           |         5
CHICAGO   | SALES      | CLERK     |         1
CHICAGO   | SALES      | MANAGER   |         1
CHICAGO   | SALES      | SALESMAN  |         4
CHICAGO   | SALES      |           |         6
NEW YORK  | ACCOUNTING | CLERK     |         1
NEW YORK  | ACCOUNTING | MANAGER   |         1
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
NEW YORK | ACCOUNTING | PRESIDENT |         1
NEW YORK | ACCOUNTING |           |         3
         |            |           |        17
(17 rows)
```

9.4.6.2 CUBE Extension

The `CUBE` extension is similar to the `ROLLUP` extension. However, unlike `ROLLUP`, which 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.

The syntax for a single `CUBE` is as follows:

```
CUBE ( { <expr_1> | ( <expr_1a> [, <expr_1b> ] ...) }
  [, <expr_2> | ( <expr_2a> [, <expr_2b> ] ...) ] ...)
```

Each `expr` is an expression that determines the grouping of the result set. If enclosed within 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 is 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, and 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 `CUBE` extension specified within the context of the `GROUP BY` clause is shown by the following:

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

The items specified in `select_list` must also appear in the `CUBE expression_list`; or they must be aggregate functions such as `COUNT, SUM, AVG, MIN`, or `MAX`; or they must be constants or functions whose return values are independent of the individual rows in the group (for example, the `SYSDATE` function).

The `GROUP BY` clause may specify multiple `CUBE` extensions as well as multiple occurrences of other `GROUP BY` extensions and individual expressions.

The `ORDER BY` clause should be used if you want the output to display in a meaningful structure. There is no guarantee on the order of the result set if no `ORDER BY` clause is specified.

The number of grouping levels or totals is `2` raised to the power of *n* where *n* represents the number of items in the `CUBE` expression list. A parenthesized list counts as one item.

The following query produces a cube based on permutations of columns `loc, dname`, and `job`.

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
ORDER BY 1, 2, 3;
```

The following is the result of the query. There is a count of the number of employees for each combination of `loc, dname` , and `job` , as well as subtotals for each combination of `loc` and `dname` , for each combination of `loc` and `job` , for each combination of `dname` and `job` , for each unique value of `loc` , for each unique value of `dname` , for each unique value of `job` , and a grand total displayed on the last line.

```
 loc      |   dname    |    job    | employees
----------+------------+-----------+-----------
BOSTON    | OPERATIONS | ANALYST   |         1
BOSTON    | OPERATIONS | CLERK     |         1
BOSTON    | OPERATIONS | MANAGER   |         1
BOSTON    | OPERATIONS |           |         3
BOSTON    | RESEARCH   | ANALYST   |         2
BOSTON    | RESEARCH   | CLERK     |         2
BOSTON    | RESEARCH   | MANAGER   |         1
BOSTON    | RESEARCH   |           |         5
BOSTON    |            | ANALYST   |         3
BOSTON    |            | CLERK     |         3
BOSTON    |            | MANAGER   |         2
BOSTON    |            |           |         8
CHICAGO   | SALES      | CLERK     |         1
CHICAGO   | SALES      | MANAGER   |         1
CHICAGO   | SALES      | SALESMAN  |         4
CHICAGO   | SALES      |           |         6
CHICAGO   |            | CLERK     |         1
CHICAGO   |            | MANAGER   |         1
CHICAGO   |            | SALESMAN  |         4
CHICAGO   |            |           |         6
NEW YORK  | ACCOUNTING | CLERK     |         1
NEW YORK  | ACCOUNTING | MANAGER   |         1
NEW YORK  | ACCOUNTING | PRESIDENT |         1
NEW YORK  | ACCOUNTING |           |         3
NEW YORK  |            | CLERK     |         1
NEW YORK  |            | MANAGER   |         1
NEW YORK  |            | PRESIDENT |         1
NEW YORK  |            |           |         3
          | ACCOUNTING | CLERK     |         1
          | ACCOUNTING | MANAGER   |         1
          | ACCOUNTING | PRESIDENT |         1
          | ACCOUNTING |           |         3
          | OPERATIONS | ANALYST   |         1
          | OPERATIONS | CLERK     |         1
          | OPERATIONS | MANAGER   |         1
          | OPERATIONS |           |         3
          | RESEARCH   | ANALYST   |         2
          | RESEARCH   | CLERK     |         2
          | RESEARCH   | MANAGER   |         1
          | RESEARCH   |           |         5
          | SALES      | CLERK     |         1
          | SALES      | MANAGER   |         1
          | SALES      | SALESMAN  |         4
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
          |            |           |        17
(50 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following query shows the effect of combining items in the `CUBE` list within parenthesis.

```
 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;
```

In the output note that there are no subtotals for permutations involving `loc` and `dname` combinations, `loc` and `job` combinations, or for `dname` by itself, or for `job` by itself.

```
 loc      |   dname    |    job    | employees
----------+------------+-----------+-----------
BOSTON    | OPERATIONS | ANALYST   |         1
BOSTON    | OPERATIONS | CLERK     |         1
BOSTON    | OPERATIONS | MANAGER   |         1
BOSTON    | RESEARCH   | ANALYST   |         2
BOSTON    | RESEARCH   | CLERK     |         2
BOSTON    | RESEARCH   | MANAGER   |         1
BOSTON    |            |           |         8
CHICAGO   | SALES      | CLERK     |         1
CHICAGO   | SALES      | MANAGER   |         1
CHICAGO   | SALES      | SALESMAN  |         4
CHICAGO   |            |           |         6
NEW YORK  | ACCOUNTING | CLERK     |         1
NEW YORK  | ACCOUNTING | MANAGER   |         1
NEW YORK  | ACCOUNTING | PRESIDENT |         1
NEW YORK  |            |           |         3
          | ACCOUNTING | CLERK     |         1
          | ACCOUNTING | MANAGER   |         1
          | ACCOUNTING | PRESIDENT |         1
          | OPERATIONS | ANALYST   |         1
          | OPERATIONS | CLERK     |         1
          | OPERATIONS | MANAGER   |         1
          | RESEARCH   | ANALYST   |         2
          | RESEARCH   | CLERK     |         2
          | RESEARCH   | MANAGER   |         1
          | SALES      | CLERK     |         1
          | SALES      | MANAGER   |         1
          | SALES      | SALESMAN  |         4
          |            |           |        17
(28 rows)
```

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

```
 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` within each grouping of `loc` .

```
 loc      |   dname    |    job    | employees
----------+------------+-----------+-----------
BOSTON    | OPERATIONS | ANALYST   |         1
BOSTON    | OPERATIONS | CLERK     |         1
BOSTON    | OPERATIONS | MANAGER   |         1
BOSTON    | OPERATIONS |           |         3
BOSTON    | RESEARCH   | ANALYST   |         2
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

9.4.6.3 GROUPING SETS Extension

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

Note that a `UNION ALL` operation, and therefore the `GROUPING SETS` extension, do not eliminate duplicate rows from the result sets that are being combined together.

The syntax for a single `GROUPING SETS` extension is as follows:

```
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 within parenthesis, `ROLLUP` extensions, and `CUBE` extensions.

The `GROUPING SETS` extension is specified within the context of the `GROUP BY` clause as shown by the following:

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

The items specified in `select_list` must also appear in the `GROUPING SETS expression_list`; or they must be aggregate functions such as `COUNT, SUM, AVG, MIN`, or `MAX`; or they must be constants or functions whose return values are independent of the individual rows in the group (for example, the `SYSDATE` function).

The `GROUP BY` clause may specify multiple `GROUPING SETS` extensions as well as multiple occurrences of other `GROUP BY` extensions and individual expressions.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `ORDER BY` clause should be used if you want the output to display in a meaningful structure. There is no guarantee on the order of the result set if no `ORDER BY` clause is specified.

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

```
 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    | employees
----------+------------+-----------+-----------
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
(12 rows)
```

This is equivalent to the following query, which employs the use of the `UNION ALL` operator.

```
 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.

```
 loc      |   dname    |    job    | employees
----------+------------+-----------+-----------
BOSTON    |            |           |         8
CHICAGO   |            |           |         6
NEW YORK  |            |           |         3
          | ACCOUNTING |           |         3
          | OPERATIONS |           |         3
          | RESEARCH   |           |         5
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
         |            | MANAGER   |         4
         |            | PRESIDENT |         1
         |            | SALESMAN  |         4
(12 rows)
```

The following example shows how various types of `GROUP BY` extensions can be used together within a `GROUPING SETS` expression list.

```
 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.

```
 loc      |   dname    |    job    | employees
----------+------------+-----------+-----------
BOSTON    |            | ANALYST   |         3
BOSTON    |            | CLERK     |         3
BOSTON    |            | MANAGER   |         2
BOSTON    |            |           |         8
BOSTON    |            |           |         8
CHICAGO   |            | CLERK     |         1
CHICAGO   |            | MANAGER   |         1
CHICAGO   |            | SALESMAN  |         4
CHICAGO   |            |           |         6
CHICAGO   |            |           |         6
NEW YORK  |            | CLERK     |         1
NEW YORK  |            | MANAGER   |         1
NEW YORK  |            | PRESIDENT |         1
NEW YORK  |            |           |         3
NEW YORK  |            |           |         3
          | ACCOUNTING | CLERK     |         1
          | ACCOUNTING | MANAGER   |         1
          | ACCOUNTING | PRESIDENT |         1
          | ACCOUNTING |           |         3
          | OPERATIONS | ANALYST   |         1
          | OPERATIONS | CLERK     |         1
          | OPERATIONS | MANAGER   |         1
          | OPERATIONS |           |         3
          | RESEARCH   | ANALYST   |         2
          | RESEARCH   | CLERK     |         2
          | RESEARCH   | MANAGER   |         1
          | RESEARCH   |           |         5
          | SALES      | CLERK     |         1
          | SALES      | MANAGER   |         1
          | SALES      | SALESMAN  |         4
          | SALES      |           |         6
          |            | ANALYST   |         3
          |            | CLERK     |         5
          |            | MANAGER   |         4
          |            | PRESIDENT |         1
          |            | SALESMAN  |         4
          |            |           |        17
          |            |           |        17
(38 rows)
```

The output is basically a concatenation of the result sets that would be produced individually from `GROUP BY loc, GROUP BY ROLLUP (dname, job)`, and `GROUP BY CUBE (job, loc)`. These individual queries are shown by the following.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 following is the result set from the `GROUP BY loc` clause.

```
loc      | dname | job | employees
----------+-------+-----+-----------
BOSTON   |       |     |         8
CHICAGO  |       |     |         6
NEW YORK |       |     |         3
(3 rows)
```

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

```
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 following is the result set from the `GROUP BY ROLLUP (dname, job)` clause.

```
loc |   dname    |    job    | employees
----+------------+-----------+-----------
    | ACCOUNTING | CLERK     |         1
    | ACCOUNTING | MANAGER   |         1
    | ACCOUNTING | PRESIDENT |         1
    | ACCOUNTING |           |         3
    | OPERATIONS | ANALYST   |         1
    | OPERATIONS | CLERK     |         1
    | OPERATIONS | MANAGER   |         1
    | OPERATIONS |           |         3
    | RESEARCH   | ANALYST   |         2
    | RESEARCH   | CLERK     |         2
    | RESEARCH   | MANAGER   |         1
    | RESEARCH   |           |         5
    | SALES      | CLERK     |         1
    | SALES      | MANAGER   |         1
    | SALES      | SALESMAN  |         4
    | SALES      |           |         6
    |            |           |        17
(17 rows)
```

The following query uses the `GROUP BY CUBE (job, loc)` clause.

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

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

```
loc      | dname |    job    | employees
---------+-------+-----------+-----------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
BOSTON   |        | ANALYST   |         3
BOSTON   |        | CLERK     |         3
BOSTON   |        | MANAGER   |         2
BOSTON   |        |           |         8
CHICAGO  |        | CLERK     |         1
CHICAGO  |        | MANAGER   |         1
CHICAGO  |        | SALESMAN  |         4
CHICAGO  |        |           |         6
NEW YORK |        | CLERK     |         1
NEW YORK |        | MANAGER   |         1
NEW YORK |        | PRESIDENT |         1
NEW YORK |        |           |         3
         |        | ANALYST   |         3
         |        | CLERK     |         5
         |        | MANAGER   |         4
         |        | PRESIDENT |         1
         |        | SALESMAN  |         4
         |        |           |        17
(18 rows)
```

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

```
 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, 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 following is the output, which is the same as when the `GROUP BY GROUPING SETS (loc, ROLLUP (dname, job), CUBE (job, loc))` clause is used.

```
 loc      |   dname    |    job    | employees
----------+------------+-----------+-----------
BOSTON   |            | ANALYST   |         3
BOSTON   |            | CLERK     |         3
BOSTON   |            | MANAGER   |         2
BOSTON   |            |           |         8
BOSTON   |            |           |         8
CHICAGO  |            | CLERK     |         1
CHICAGO  |            | MANAGER   |         1
CHICAGO  |            | SALESMAN  |         4
CHICAGO  |            |           |         6
CHICAGO  |            |           |         6
NEW YORK |            | CLERK     |         1
NEW YORK |            | MANAGER   |         1
NEW YORK |            | PRESIDENT |         1
NEW YORK |            |           |         3
NEW YORK |            |           |         3
         | ACCOUNTING | CLERK     |         1
         | ACCOUNTING | MANAGER   |         1
         | ACCOUNTING | PRESIDENT |         1
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
         | ACCOUNTING |           |         3
         | OPERATIONS | ANALYST   |         1
         | OPERATIONS | CLERK     |         1
         | OPERATIONS | MANAGER   |         1
         | OPERATIONS |           |         3
         | RESEARCH   | ANALYST   |         2
         | RESEARCH   | CLERK     |         2
         | RESEARCH   | MANAGER   |         1
         | RESEARCH   |           |         5
         | SALES      | CLERK     |         1
         | SALES      | MANAGER   |         1
         | SALES      | SALESMAN  |         4
         | SALES      |           |         6
         |            | ANALYST   |         3
         |            | CLERK     |         5
         |            | MANAGER   |         4
         |            | PRESIDENT |         1
         |            | SALESMAN  |         4
         |            |           |        17
         |            |           |        17
(38 rows)
```

9.4.6.4 GROUPING Function

When using the `ROLLUP, CUBE, or GROUPING SETS` extensions to the `GROUP BY` clause, it may 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 `GROUPING` function provides a means of making this distinction.

The general syntax for use of the `GROUPING` function is shown by the following.

```
SELECT [ <expr> ...,] GROUPING( <col_expr> ) [, <expr> ] ...
FROM ...
GROUP BY [...,]
  { ROLLUP | CUBE | GROUPING SETS }( [...,] <col_expr>
  [, ...] ) [, ...]
```

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

The return value of the `GROUPING` function is either a 0 or 1. In the result set of a query, if the column expression specified in the `GROUPING` function is null because the row represents a subtotal over multiple values of that column then the `GROUPING` function returns a value of 1. If the row returns results based on a particular value of the column specified in the `GROUPING` function, then the `GROUPING` function returns a value of 0. In the latter case, the column can be null as well as non-null, but in any case, it is for a particular value of that column, not a subtotal across multiple values.

The following query shows how the return values of the `GROUPING` function correspond to the subtotal lines.

```
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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

In the three right-most columns displaying the output of the `GROUPING` functions, a value of 1 appears on a subtotal line wherever a subtotal is taken across values of the corresponding columns.

```
 loc      |   dname    |    job    | employees | gf_loc | gf_dname | gf_job
----------+------------+-----------+-----------+--------+----------+--------
BOSTON    | OPERATIONS | ANALYST   |         1 |      0 |        0 |      0
BOSTON    | OPERATIONS | CLERK     |         1 |      0 |        0 |      0
BOSTON    | OPERATIONS | MANAGER   |         1 |      0 |        0 |      0
BOSTON    | OPERATIONS |           |         3 |      0 |        0 |      1
BOSTON    | RESEARCH   | ANALYST   |         2 |      0 |        0 |      0
BOSTON    | RESEARCH   | CLERK     |         2 |      0 |        0 |      0
BOSTON    | RESEARCH   | MANAGER   |         1 |      0 |        0 |      0
BOSTON    | RESEARCH   |           |         5 |      0 |        0 |      1
BOSTON    |            |           |         8 |      0 |        1 |      1
CHICAGO   | SALES      | CLERK     |         1 |      0 |        0 |      0
CHICAGO   | SALES      | MANAGER   |         1 |      0 |        0 |      0
CHICAGO   | SALES      | SALESMAN  |         4 |      0 |        0 |      0
CHICAGO   | SALES      |           |         6 |      0 |        0 |      1
CHICAGO   |            |           |         6 |      0 |        1 |      1
NEW YORK  | ACCOUNTING | CLERK     |         1 |      0 |        0 |      0
NEW YORK  | ACCOUNTING | MANAGER   |         1 |      0 |        0 |      0
NEW YORK  | ACCOUNTING | PRESIDENT |         1 |      0 |        0 |      0
NEW YORK  | ACCOUNTING |           |         3 |      0 |        0 |      1
NEW YORK  |            |           |         3 |      0 |        1 |      1
          |            |           |        17 |      1 |        1 |      1
(20 rows)
```

These indicators can be used 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.

```
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:

```
 loc      |   dname    | job | employees | gf_loc | gf_dname | gf_job
----------+------------+-----+-----------+--------+----------+--------
BOSTON    | OPERATIONS |     |         3 |      0 |        0 |      1
BOSTON    | RESEARCH   |     |         5 |      0 |        0 |      1
CHICAGO   | SALES      |     |         6 |      0 |        0 |      1
NEW YORK  | ACCOUNTING |     |         3 |      0 |        0 |      1
(4 rows)
```

The `GROUPING` function can be used to distinguish a subtotal row from a base aggregate row or from certain subtotal rows where one of the items in the expression list returns null as a result of the column on which the expression is based being null for one or more rows in the table, as opposed to representing a subtotal over the column.

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

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

```
SELECT loc, job, COUNT(*) AS "employees",
  GROUPING(loc) AS "gf_loc",
  GROUPING(job) AS "gf_job"
FROM emp e, dept d
WHERE e.deptno = d.deptno AND loc = 'BOSTON'
GROUP BY CUBE (loc, job)
ORDER BY 1, 2;
```

Note that the output contains two rows containing `BOSTON` in the `loc` column and spaces in the `job` column (fourth and fifth entries in the table).

```
 loc    |   job   | employees | gf_loc | gf_job
--------+---------+-----------+--------+--------
BOSTON | ANALYST |         3 |      0 |      0
BOSTON | CLERK   |         3 |      0 |      0
BOSTON | MANAGER |         2 |      0 |      0
BOSTON |         |         1 |      0 |      0
BOSTON |         |         9 |      0 |      1
       | ANALYST |         3 |      1 |      0
       | CLERK   |         3 |      1 |      0
       | MANAGER |         2 |      1 |      0
       |         |         1 |      1 |      0
       |         |         9 |      1 |      1
(10 rows)
```

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

The fourth row where the `GROUPING` function on the `job` column as well as on the `loc` column returns 0 indicates 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.

Also note that 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 indicating 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.

9.4.6.5 GROUPING_ID Function

The `GROUPING_ID` function provides a simplification of the `GROUPING` function in order 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 or not a row is a subtotal over all values of the given column. Thus, multiple `GROUPING` functions may be required to interpret the level of subtotals for queries with multiple grouping columns.

The `GROUPING_ID` function accepts one or more column expressions that have been used in the `ROLLBACK, CUBE`, or `GROUPING SETS` extensions and returns a single integer that can be used to determine over which of these columns a subtotal has been aggregated.

The general syntax for use of the `GROUPING_ID` function is shown by the following.

```
SELECT [ <expr> ...,]
  GROUPING_ID( <col_expr_1> [, <col_expr_2> ] ... )
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  [, <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 1's and 0's that would be 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.

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`.

```
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`.

```
 loc      |   dname    | employees | gf_loc | gf_dname | gid
----------+------------+-----------+--------+----------+-----
BOSTON    | OPERATIONS |         3 |      0 |        0 |   0
BOSTON    | RESEARCH   |         5 |      0 |        0 |   0
CHICAGO   | SALES      |         6 |      0 |        0 |   0
NEW YORK  | ACCOUNTING |         3 |      0 |        0 |   0
BOSTON    |            |         8 |      0 |        1 |   1
CHICAGO   |            |         6 |      0 |        1 |   1
NEW YORK  |            |         3 |      0 |        1 |   1
          | ACCOUNTING |         3 |      1 |        0 |   2
          | OPERATIONS |         3 |      1 |        0 |   2
          | RESEARCH   |         5 |      1 |        0 |   2
          | SALES      |         6 |      1 |        0 |   2
          |            |        17 |      1 |        1 |   3
(12 rows)
```

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.

loc	dname	Bit Vector gf_loc gf_dname	GROUPING_ID gid
`BOSTON`	`OPERATIONS`	$0 * 2^1 + 0 * 2^0$	`0`
`BOSTON`	`null`	$0 * 2^1 + 1 * 2^0$	`1`
`null`	`ACCOUNTING`	$1 * 2^1 + 0 * 2^0$	`2`
`null`	`null`	$1 * 2^1 + 1 * 2^0$	`3`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

Aggregation by Column	Bit Vector		GROUPING_ID
	gf_loc	gf_dname	gid
`loc, dname`	`0`	`0`	`0`
`loc`	`0`	`1`	`1`
`dname`	`1`	`0`	`2`
`Grand Total`	`1`	`1`	`3`

So to display only those subtotals by `dname`, the following simplified query can be used with a `HAVING` clause based on the `GROUPING_ID` function.

```
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 following is the result of the query.

```
 loc |   dname    | employees | gf_loc | gf_dname | gid
-----+------------+-----------+--------+----------+-----
     | ACCOUNTING |         3 |      1 |        0 |   2
     | OPERATIONS |         3 |      1 |        0 |   2
     | RESEARCH   |         5 |      1 |        0 |   2
     | SALES      |         6 |      1 |        0 |   2
(4 rows)
```

9.5 Profile Management

Advanced Server allows a database superuser to create named *profiles*. 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 re-use

A profile is a named set of password attributes that allow you to easily manage a group of roles that share comparable authentication requirements. If the password requirements change, you can modify the profile to have the new requirements applied to each user that is 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 is associated with their login role. Profiles are shared by all databases within a cluster, but each cluster may have multiple profiles. A single user with access to multiple databases will use the same profile when connecting to each database within the cluster.

Advanced Server creates a profile named `default` that is associated with a new role when the role is created unless an alternate profile is specified. If you upgrade to Advanced Server from a previous server version, existing roles will automatically be assigned to the `default` profile. You cannot delete

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

the `default` profile.

The `default` profile specifies the following attributes:

`FAILED_LOGIN_ATTEMPTS`	`UNLIMITED`
`PASSWORD_LOCK_TIME`	`UNLIMITED`
`PASSWORD_LIFE_TIME`	`UNLIMITED`
`PASSWORD_GRACE_TIME`	`UNLIMITED`
`PASSWORD_REUSE_TIME`	`UNLIMITED`
`PASSWORD_REUSE_MAX`	`UNLIMITED`
`PASSWORD_VERIFY_FUNCTION`	`NULL`
`PASSWORD_ALLOW_HASHED`	`TRUE`

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.

9.5.1 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 Advanced Server.

Parameters:

`profile_name` specifies the name of the profile.

`parameter` specifies the attribute limited by the profile.

`value` specifies the parameter limit.

Advanced Server supports the `value` shown below for each `parameter`:

`FAILED_LOGIN_ATTEMPTS` specifies the number of failed login attempts that a user may make before the server locks the user 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 may 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 has been locked because of `FAILED_LOGIN_ATTEMPTS`. Supported values are:

- A `NUMERIC` value greater than 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_LOCK_TIME` specified in the `DEFAULT` profile.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `UNLIMITED` – the account is locked until it is manually unlocked by a database superuser.

`PASSWORD_LIFE_TIME` specifies the number of days that the current password may 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 that will pass after the password expires before connections by the role are rejected. If `PASSWORD_GRACE_TIME` is not specified, the password will expire on the day specified by the default value of `PASSWORD_GRACE_TIME`, and the user will not be allowed to execute any command until a new password is provided. Supported values are:

- A `NUMERIC` value greater than 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 does not 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 will be allowed to connect, but will not be allowed to execute any command until they update their expired password. Supported values are:

- A `NUMERIC` value greater than 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_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 re-using a password. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used 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 greater than 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_REUSE_TIME` specified in the `DEFAULT` profile.
- `UNLIMITED` – The password can be re-used without restrictions.

`PASSWORD_REUSE_MAX` specifies the number of password changes that must occur before a password can be reused. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used 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 or equal to `0`.
- `DEFAULT` - the value of `PASSWORD_REUSE_MAX` specified in the `DEFAULT` profile.
- `UNLIMITED` – The password can be re-used 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 to be allowed for use or not. 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 in order to be validated effectively, else an error will be 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` is not an Oracle-compatible parameter.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Use `DROP PROFILE` command to remove the profile.

Examples

The following command creates a profile named `acctg`. The profile specifies that if a user has not authenticated with the correct password in five attempts, the account will be 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 has not changed their password before the 90 days specified in the profile has passed, they will be issued a warning at login. After a grace period of 3 days, their account will not be allowed to invoke any commands until they change their password.

The following command creates a profile named `accts`. The profile specifies that a user cannot re-use a password within 180 days of the last use of the password, and must change their password at least 5 times before re-using 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 will verify that the password provided meets their standards for complexity:

```
CREATE PROFILE resources LIMIT
       PASSWORD_VERIFY_FUNCTION password_rules;
```

9.5.1.1 Creating a Password Function

When specifying `PASSWORD_VERIFY_FUNCTION`, you can provide a customized function that specifies the security rules that will be applied 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 may not 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `old_password` is the user's previous password. If you reference this parameter within your function:

When a database superuser changes their password, the third parameter will always be `NULL`.

When a user with the `CREATEROLE` attribute changes their password, the parameter will pass the previous password if the statement includes the `REPLACE` clause. Note that the `REPLACE` clause is optional syntax for a user with the `CREATEROLE` privilege.

When a user that is not a database superuser and does not have the `CREATEROLE` attribute changes their password, the third parameter will contain the previous password for the role.

The function returns a Boolean value. If the function returns true and does not 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 does not 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:

The following example creates a profile and a custom function; then, the function is associated with the profile. The following `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
  THEN
    raise_application_error(-20002, 'includes old password');
  END IF;

  RETURN true;
END;
```

The function first ensures that the password is at least 5 characters long, and then compares the new password to the old password. If the new password contains fewer than 5 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 5 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 will be lifted:

```
edb=# ALTER ROLE alice PASSWORD 'hey';
ALTER ROLE
```

9.5.2 Altering a Profile

Use the `ALTER PROFILE` command to modify a user-defined profile; Advanced Server supports two forms of the command:

```
ALTER PROFILE <profile_name> RENAME TO <new_name>;

ALTER PROFILE <profile_name>
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
      LIMIT {<parameter value>}[...];
```

Include the `LIMIT` clause and one or more space-delimited `parameter/value` pairs to specify the rules enforced by 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`:

```
ALTER PROFILE acctg_profile
       LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

`acctg_profile` will count failed connection attempts when a login role attempts to connect to the server. The profile specifies that if a user has not authenticated with the correct password in three attempts, the account will be locked for one day.

The following example changes the name of `acctg_profile` to `payables_profile`:

```
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

9.5.3 Dropping a Profile

Use the `DROP PROFILE` command to drop a profile. The syntax is:

```
DROP PROFILE [IF EXISTS] <profile_name> [CASCADE|RESTRICT];
```

Include the `IF EXISTS` clause to instruct the server to not throw an error if the specified profile does not exist. The server will issue a notice if the profile does not 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 to not drop any profile that is associated with a role. This is the default behavior.

Parameters

`profile_name`

The name of the profile being dropped.

Examples

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example drops a profile named `acctg_profile`:

```
DROP PROFILE acctg_profile CASCADE;
```

The command first re-associates 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`:

```
DROP PROFILE acctg_profile RESTRICT;
```

The `RESTRICT` clause in the command instructs the server to not drop `acctg_profile` if there are any roles associated with the profile.

9.5.4 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 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 Advanced Server, please see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/sql-commands.html

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 pre-defined profile with a role, or to change which pre-defined profile is associated with a user.
- Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to specify that the user account should be placed 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, the role can only be unlocked by a database superuser 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 will expire. If you omit the `AT 'timestamp'` keywords, the password will expire immediately.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- 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 may only have one profile. To discover the profile that is currently associated with a login role, query the `profile` column of the `DBA_USERS` view.

Parameters

`name`

The name of the role with which the specified profile will be associated.

`password`

The password associated with the role.

`profile_name`

The name of the profile that will be associated with the role.

`timestamp`

The date and time at which the clause will be enforced. When specifying a value for `timestamp`, enclose the value in single-quotes.

Examples

The following 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;
```

9.5.5 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 will not 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 instruct the server to lock the account at the time specified by the given timestamp for the length of time

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 is being locked or unlocked.

`timestamp`

The date and time at which the role will be locked. When specifying a value for `timestamp`, enclose the value in single-quotes.

> **Note**
>
> This command (available only in Advanced Server) is implemented to support Oracle-styled profile management.

Examples

The following example uses the `ACCOUNT LOCK` clause to lock the role named `john`. The account will remain locked until the account is unlocked with the `ACCOUNT UNLOCK` clause:

```
ALTER ROLE john ACCOUNT LOCK;
```

The following example uses the `ACCOUNT UNLOCK` clause to unlock the role named `john`:

```
ALTER USER john ACCOUNT UNLOCK;
```

The following 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 will remain locked for the length of time specified by the `PASSWORD_LOCK_TIME` parameter.

The following 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 will remain locked until a database superuser uses the `ACCOUNT UNLOCK` command to unlock the role.

9.5.6 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> […]]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

where `option` can be the following compatible clauses:

```
    PROFILE <profile_name>
|   ACCOUNT {LOCK|UNLOCK}
|   PASSWORD EXPIRE [AT '<timestamp>']
```

or `option` can be the following non-compatible clauses:

```
|   LOCK TIME '<timestamp>'
```

For information about the administrative clauses of the `CREATE USER` or `CREATE ROLE` command that are supported by Advanced Server, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/sql-commands.html

`CREATE ROLE|USER... PROFILE` adds a new role with an associated profile to an 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 pre-defined profile with a role, or to change which pre-defined profile is associated with a user.
- Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to specify that the user account should be placed 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, the role can only be unlocked by a database superuser 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 will expire. If you omit the `AT 'timestamp'` keywords, the password will expire immediately.

Each login role may only have one profile. To discover the profile that is 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 at which the clause will be enforced. When specifying a value for `timestamp` , enclose the value in single-quotes.

Examples

The following example uses `CREATE USER` to create a login role named `john` who is associated with the `acctg_profile` profile:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE USER john PROFILE acctg_profile IDENTIFIED BY “1safepwd”;
```

`john` can log in to the server, using the password `1safepwd`.

The following example uses `CREATE ROLE` to create a login role named `john` who is associated with the `acctg_profile` profile:

```
CREATE ROLE john PROFILE acctg_profile LOGIN PASSWORD “1safepwd”;
```

`john` can log in to the server, using the password `1safepwd`.

9.5.7 Backing up Profile Management Functions

A profile may include a `PASSWORD_VERIFY_FUNCTION` clause that refers to a user-defined function that specifies the behavior enforced by Advanced Server. Profiles are global objects; they are shared by all of the databases within a cluster. While profiles are global objects, user-defined functions are database objects.

Invoking `pg_dumpall` with the `-g` or `-r` option will create a script that recreates the definition of any existing profiles, but that does not recreate the user-defined functions that are referred to by the `PASSWORD_VERIFY_FUNCTION` clause. You should use the `pg_dump` utility to explicitly dump (and later restore) the database in which those functions reside.

The script created by `pg_dump` will contain a command that includes the clause and function name:

```
ALTER PROFILE… LIMIT PASSWORD_VERIFY_FUNCTION <function_name>
```

to associate the restored function with the profile with which it was previously associated.

If the `PASSWORD_VERIFY_FUNCTION` clause is set to `DEFAULT` or `NULL`, the behavior will be replicated by the script generated by the `pg_dumpall -g` or `pg_dumpall -r` command.

9.6 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 will (generally) return the result set in the least amount of time. The server's choice of plan is dependent upon 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 have been gathered by the ANALYZE command.

As a rule, the query planner will select the least expensive plan. You can use an *optimizer hint* to influence the server as it selects a query plan. An optimizer hint is a directive (or multiple 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 employ or avoid a specific plan when producing the result set.

Synopsis

```
{ DELETE | INSERT | SELECT | UPDATE } /*+ { <hint> [ <comment> ] } [...] */
  <statement_body>

{ DELETE | INSERT | SELECT | UPDATE } --+ { <hint> [ <comment> ] } [...]
  <statement_body>
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Optimizer hints may be included in either of the forms shown above. Note that in both forms, a plus sign (+) must immediately follow the `/*` or `--` opening comment symbols, with no intervening space, or the server will not interpret the following tokens as hints.

If you are using the first form, the hint and optional comment may span multiple lines. The second form requires all hints and comments to occupy a single line; the remainder of the statement must start on a new line.

Description

Please Note:

- The database server will always try 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 will not be used even if it is 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.
- Remember that the hint is embedded within 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 non-existent in the SQL command, there will be no indication that any sort of error has occurred. No syntax error will be given and the entire hint is simply ignored.
- If an alias is used for a table name in the SQL command, then the alias name, not the original table name, must be used in the hint. For example, in the command, `SELECT /*+ FULL(acct) */ * FROM accounts acct ..., acct`, the alias for `accounts`, must be specified in the `FULL` hint, not the table name, `accounts`.

Use the `EXPLAIN` command to ensure that the hint is correctly formed and the planner is using the hint. See the Advanced Server documentation set for information on the `EXPLAIN` command.

In general, optimizer hints should not be used in production applications (where table data changes throughout the life of the application). By ensuring that dynamic columns are `ANALYZED` frequently, the column statistics will be updated to reflect value changes, and the planner will use such information to produce the least cost plan for any given command execution. Use of optimizer hints defeats the purpose of this process and will result in the same plan regardless of how the table data changes.

Parameters

`hint`

An optimizer hint directive.

`comment`

A string with additional information. Note that there are restrictions as to what characters may be included in the comment. Generally, `comment` may only consist 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 will be ignored if the comment is not in this form.

`statement_body`

The remainder of the `DELETE, INSERT, SELECT`, or `UPDATE` command.

The following sections describe the optimizer hint directives in more detail.

9.6.1 Default Optimization Modes

There are a number of optimization modes that can be chosen as the default setting for an Advanced Server database cluster. This setting can also be changed on a per session basis by using the `ALTER SESSION` command as well as in individual `DELETE, SELECT`, and `UPDATE` commands within an optimizer hint. The configuration parameter that controls these default modes is named `OPTIMIZER_MODE`.

The following table shows the possible values.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Hint	Description
`ALL_ROWS`	Optimizes for retrieval of all rows of the result set.
`CHOOSE`	Does no default optimization based on assumed number of rows to be retrieved from the result set. This is the default.
`FIRST_ROWS`	Optimizes for retrieval of only the first row of the result set.
`FIRST_ROWS_10`	Optimizes for retrieval of the first 10 rows of the results set.
`FIRST_ROWS_100`	Optimizes for retrieval of the first 100 rows of the result set.
`FIRST_ROWS_1000`	Optimizes for retrieval of the first 1000 rows of the result set.
`FIRST_ROWS(n)`	Optimizes for retrieval of the first *n* rows of the result set. This form may not be used as the object of the `ALTER SESSION SET OPTIMIZER_MODE` command. It may only be used in the form of a hint in a SQL command.

These optimization modes are based upon the assumption that the client submitting the SQL command is interested in viewing only the first “n” rows of the result set and will then abandon the remainder of the result set. Resources allocated to the query are adjusted as such.

Examples

Alter the current session to optimize for retrieval of the first 10 rows of the result set.

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10;
```

The current value of the `OPTIMIZER_MODE` parameter can be shown by using the `SHOW` command. Note that this command is a utility dependent command. In PSQL, the `SHOW` command is used as follows:

```
SHOW OPTIMIZER_MODE;

optimizer_mode
-----------------
 first_rows_10
(1 row)
```

The `SHOW` command, compatible with Oracle databases, has the following syntax:

```
SHOW PARAMETER OPTIMIZER_MODE;

NAME
--------------------------------------------------
VALUE
--------------------------------------------------
optimizer_mode
first_rows_10
```

The following example shows an optimization mode used in a `SELECT` command as a hint:

```
SELECT /*+ FIRST_ROWS(7) */ * FROM emp;

empno| ename |  job     | mgr  |      hiredate      | sal     | comm  | deptno
-----+-------+----------+------+--------------------+---------+-------+-------
7369 | SMITH | CLERK    | 7902 | 17-DEC-80 00:00:00 | 800.00  |       | 20
7499 | ALLEN | SALESMAN | 7698 | 20-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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 |1400.00| 30
7698 | BLAKE | MANAGER  | 7839 | 01-MAY-81 00:00:00 | 2850.00 |       | 30
7782 | CLARK | MANAGER  | 7839 | 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 | 7698 | 08-SEP-81 00:00:00 | 1500.00 | 0.00  | 30
7876 | ADAMS | CLERK    | 7788 | 23-MAY-87 00:00:00 | 1100.00 |       | 20
7900 | JAMES | CLERK    | 7698 | 03-DEC-81 00:00:00 |  950.00 |       | 30
7902 | FORD  | ANALYST  | 7566 | 03-DEC-81 00:00:00 | 3000.00 |       | 20
7934 | MILLER| CLERK    | 7782 | 23-JAN-82 00:00:00 | 1300.00 |       | 10
(14 rows)
```

9.6.2 Access Method Hints

The following hints influence how the optimizer accesses relations to create the result set.

Hint	Description
`FULL(table)`	Perform a full sequential scan on `table`.
`INDEX(table [ index ] [...])`	Use `index` on `table` to access the relation.
`NO_INDEX(table [ index ] [...])`	Do not use `index` on `table` to access the relation.

In addition, the `ALL_ROWS, FIRST_ROWS`, and `FIRST_ROWS(n)` hints can be used.

Examples

The sample application does not have sufficient data to illustrate the effects of optimizer hints so the remainder of the examples in this section will use a banking database created by the `pgbench` application located in the Advanced Server `bin` subdirectory.

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 twenty, which results in the creation of twenty branches, each with 100,000 accounts, resulting in a total of 2,000,000 rows in the `pgbench_accounts` table and twenty 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 0.11 s, remaining 2.10 s)
200000 of 2000000 tuples (10%) done (elapsed 0.22 s, remaining 1.98 s)
300000 of 2000000 tuples (15%) done (elapsed 0.33 s, remaining 1.84 s)
400000 of 2000000 tuples (20%) done (elapsed 0.42 s, remaining 1.67 s)
500000 of 2000000 tuples (25%) done (elapsed 0.52 s, remaining 1.57 s)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
600000 of 2000000 tuples (30%) done (elapsed 0.62 s, remaining 1.45 s)
700000 of 2000000 tuples (35%) done (elapsed 0.73 s, remaining 1.35 s)
800000 of 2000000 tuples (40%) done (elapsed 0.87 s, remaining 1.31 s)
900000 of 2000000 tuples (45%) done (elapsed 0.98 s, remaining 1.19 s)
1000000 of 2000000 tuples (50%) done (elapsed 1.09 s, remaining 1.09 s)
1100000 of 2000000 tuples (55%) done (elapsed 1.22 s, remaining 1.00 s)
1200000 of 2000000 tuples (60%) done (elapsed 1.36 s, remaining 0.91 s)
1300000 of 2000000 tuples (65%) done (elapsed 1.51 s, remaining 0.82 s)
1400000 of 2000000 tuples (70%) done (elapsed 1.65 s, remaining 0.71 s)
1500000 of 2000000 tuples (75%) done (elapsed 1.78 s, remaining 0.59 s)
1600000 of 2000000 tuples (80%) done (elapsed 1.93 s, remaining 0.48 s)
1700000 of 2000000 tuples (85%) done (elapsed 2.10 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 500,00 transactions are then processed. This will populate the `pgbench_history` table with 500,000 rows.

```
 pgbench -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
number of threads: 1
number of transactions per client: 500000
number of transactions actually processed: 500000/500000
latency average: 0.000 ms
tps = 1464.338375 (including connections establishing)
tps = 1464.350357 (excluding connections establishing)
```

The table definitions are shown below:

```
 \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)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
\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             | Modifiers
--------+-----------------------------+-----------
 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 the following example, `aid` is the primary key column, so an indexed search is used on index, `pgbench_accounts_pkey`.

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE aid = 100;

                                    QUERY PLAN
--------------------------------------------------------------------------------
------------------
Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.43..8.45
rows=1 width=97)
   Index Cond: (aid = 100)
(2 rows)
```

The `FULL` hint is used to force a full sequential scan instead of using the index as shown below:

```
EXPLAIN SELECT /*+ FULL(pgbench_accounts) */ * FROM pgbench_accounts WHERE
aid = 100;

                         QUERY PLAN
------------------------------------------------------------------------
Seq Scan on pgbench_accounts  (cost=0.00..58781.69 rows=1 width=97)
  Filter: (aid = 100)
(2 rows)
```

The `NO_INDEX` hint forces a parallel sequential scan instead of use of the index as shown below:

```
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)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..44094.70 rows=1
   width=97)
           Filter: (aid = 100)
(4 rows)
```

In addition to using the `EXPLAIN` command as shown in the prior examples, more detailed information regarding whether or not a hint was used by the planner can be obtained by setting the `trace_hints` configuration parameter as follows:

```
SET trace_hints TO on;
```

The `SELECT` command with the `NO_INDEX` hint is repeated below to illustrate the additional information produced when the `trace_hints` configuration parameters is set.

```
 EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_pkey) */ *
FROM pgbench_accounts WHERE aid = 100;

INFO:  [HINTS] Index Scan of [pgbench_accounts].[pgbench_accounts_pkey]
rejected due to NO_INDEX hint.
                                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)
```

Note that if a hint is ignored, the `INFO: [HINTS]` line will not appear. This may be an indication that there was a syntax error or some other misspelling in the hint as shown in the following example where the index name is misspelled.

```
 EXPLAIN SELECT /*+ NO_INDEX(pgbench_accounts pgbench_accounts_xxx) */ * FROM
pgbench_accounts WHERE aid = 100;

                                    QUERY PLAN
-------------------------------------------------------------------------------
------------------
Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.43..8.45
rows=1 width=97)
   Index Cond: (aid = 100)
(2 rows)
```

9.6.3 Specifying a Join Order

Include the `ORDERED` directive to instruct the query optimizer to join tables in the order in which they are listed in the `FROM` clause. If you do not include the `ORDERED` keyword, the query optimizer will choose 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:

```
SELECT e.ename, d.dname, h.startdate
  FROM emp e, dept d, jobhist h
  WHERE d.deptno = e.deptno
  AND h.empno = e.empno;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following command instructs the optimizer to join the tables in the ordered specified:

```
SELECT /*+ ORDERED */ 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, Advanced Server will first join `emp e` with `dept d` before joining the results with `jobhist h`. Without the `ORDERED` directive, the join order is selected by the query optimizer.

> **Note**
>
> The `ORDERED` directive does not work for Oracle-style outer joins (those joins that contain a '+' sign).

9.6.4 Joining Relations Hints

When two tables are to be joined, there are three possible plans that may be used 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.

The following table lists the optimizer hints that can be used to influence the planner to use one type of join plan over another.

Hint	Description
`USE_HASH(table [...])`	Use a hash join for `table`.
`NO_USE_HASH(table [...])`	Do not use a hash join for `table`.
`USE_MERGE(table [...])`	Use a merge sort join for `table`.
`NO_USE_MERGE(table [...])`	Do not use a merge sort join for `table`.
`USE_NL(table [...])`	Use a nested loop join for `table`.
`NO_USE_NL(table [...])`	Do not use a nested loop join for `table`.

Examples

In the following 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.

```
 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)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(5 rows)
```

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.

```
 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..343575.53 rows=2014215 width=12)
          ->  Sort  (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)
(9 rows)
```

Finally, the `USE_MERGE` hint forces the planner to use a merge join.

```
 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..343575.53 rows=2014215 width=12)
          ->  Sort  (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)
(9 rows)
```

In this three-table join example, the planner first performs a hash join on the `pgbench_branches` and `pgbench_history` tables, then finally performs a hash join of the result with the `pgbench_accounts` table.

```
 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=86814.29..123103.29 rows=500000 width=20)
  Hash Cond: (h.aid = a.aid)
  ->  Hash Join  (cost=21.45..15081.45 rows=500000 width=20)
        Hash Cond: (h.bid = b.bid)
        ->  Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=500000 width=20)
        ->  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)
  ->  Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
        ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=4)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(9 rows)
```

This plan is altered by using hints to force a combination of a merge sort join and a hash join.

```
 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;

                                        QUERY PLAN
------------------------------------------------------------------------------
---------------------
 Hash Join  (cost=152583.39..182562.49 rows=500000 width=20)
   Hash Cond: (h.aid = a.aid)
   ->  Merge Join  (cost=65790.55..74540.65 rows=500000 width=20)
         Merge Cond: (b.bid = h.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=65768.92..68268.92 rows=500000 width=20)
               ->  Sort  (cost=65768.92..67018.92 rows=500000 width=20)
                     Sort Key: h.bid
                     ->  Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=500000 width=20)
   ->  Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
         ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=4)
(13 rows)
```

9.6.5 Global Hints

Thus far, hints have been applied directly to tables that are referenced in the SQL command. It is also possible to apply hints to tables that appear in a view when the view is referenced in the SQL command. The hint does not appear in the view, itself, but rather in the SQL command that references the view.

When specifying a hint that is to apply to a table within a view, the view and table names are given in dot notation within the hint argument list.

Synopsis

```
<hint>(<view>.<table>)
```

Parameters

`hint`

Any of the hints in table Access Method Hints, Joining Relations Hints.

`view`

The name of the view containing `table`.

`table`

The table on which the hint is to be applied.

Examples

A view named, `tx`, is created from the three-table join of `pgbench_history, pgbench_branches`, and `pgbench_accounts` shown in the final

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 shown below:

```
EXPLAIN SELECT * FROM tx;

                                    QUERY PLAN
--------------------------------------------------------------------------------
-----------
 Hash Join  (cost=86814.29..123103.29 rows=500000 width=20)
   Hash Cond: (h.aid = a.aid)
   ->  Hash Join  (cost=21.45..15081.45 rows=500000 width=20)
         Hash Cond: (h.bid = b.bid)
         ->  Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=500000 width=20)
         ->  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)
   ->  Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
         ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=4)
(9 rows)
```

The same hints that were applied to this join at the end of Joining Relations Hints can be applied to the view as follows:

```
EXPLAIN SELECT /*+ USE_MERGE(tx.h tx.b) USE_HASH(tx.a) */ * FROM tx;

                                        QUERY PLAN
--------------------------------------------------------------------------------
---------------------
 Hash Join  (cost=152583.39..182562.49 rows=500000 width=20)
   Hash Cond: (h.aid = a.aid)
   ->  Merge Join  (cost=65790.55..74540.65 rows=500000 width=20)
         Merge Cond: (b.bid = h.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=65768.92..68268.92 rows=500000 width=20)
               ->  Sort  (cost=65768.92..67018.92 rows=500000 width=20)
                     Sort Key: h.bid
                     ->  Seq Scan on pgbench_history h  (cost=0.00..8185.00 rows=500000 width=20)
   ->  Hash  (cost=53746.15..53746.15 rows=2014215 width=4)
         ->  Seq Scan on pgbench_accounts a  (cost=0.00..53746.15 rows=2014215 width=4)
(13 rows)
```

In addition to applying hints to tables within stored views, hints can be applied to tables within subqueries as illustrated by the following example. 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;

 empno | ename  | mgr empno | mgr ename
-------+--------+-----------+-----------
  7369 | SMITH  |      7902 | FORD
  7499 | ALLEN  |      7698 | BLAKE
  7521 | WARD   |      7698 | BLAKE
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  7566 | JONES  |      7839 | KING
  7654 | MARTIN |      7698 | BLAKE
  7698 | BLAKE  |      7839 | KING
  7782 | CLARK  |      7839 | KING
  7788 | SCOTT  |      7566 | JONES
  7844 | TURNER |      7698 | BLAKE
  7876 | ADAMS  |      7788 | SCOTT
  7900 | JAMES  |      7698 | BLAKE
  7902 | FORD   |      7566 | JONES
  7934 | MILLER |      7782 | CLARK
(13 rows)
```

The plan chosen by the query planner is shown below:

```
 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)
```

A hint can be applied to the `emp` table within 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 /*+ 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)
```

9.6.6 Using the APPEND Optimizer Hint

By default, Advanced Server will add 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 instruct the server to bypass mid-table 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:

```
/*+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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

the `sales` table:

```
INSERT /*+APPEND*/ INTO sales VALUES
(10, 10, '01-Mar-2011', 10, 'OR');
```

Note that Advanced Server supports the `APPEND` hint when adding multiple rows in a single `INSERT` statement:

```
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');
```

The `APPEND` hint can also be included in the `SELECT` clause of an `INSERT INTO` statement:

```
INSERT INTO sales_history SELECT /*+APPEND*/ FROM sales;
```

9.6.7 Parallelism Hints

The `PARALLEL` optimizer hint is used to force parallel scanning.

The `NO_PARALLEL` optimizer hint prevents usage of a parallel scan.

Synopsis

```
PARALLEL (<table> [ <parallel_degree> | DEFAULT ])

NO_PARALLEL (<table>)
```

Description

Parallel scanning is the usage 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.

Parameters

`table`

The table to which the parallel hint is to be applied.

`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` located under `Resource Consumption` in the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/runtime-config-resource.html

If `DEFAULT` is specified, then the maximum possible parallel degree is used.

If both `parallel_degree` and `DEFAULT` are omitted, then the query optimizer determines the parallel degree. In this case, if `table` has been set

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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` was specified. For information on the `parallel_workers` storage parameter, see the `Storage Parameters` located under `CREATE TABLE` in the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/sql-createtable.html

Regardless of the circumstance, the parallel degree never exceeds the setting of configuration parameter `max_parallel_workers_per_gather`.

Examples

The following configuration parameter settings are in effect:

```
SHOW max_worker_processes;

 max_worker_processes
----------------------
 8
(1 row)

SHOW max_parallel_workers_per_gather;

 max_parallel_workers_per_gather
---------------------------------
 2
(1 row)
```

The following example shows the default scan on table `pgbench_accounts`. Note that a sequential scan is shown in the query plan.

```
SET trace_hints TO on;

EXPLAIN SELECT * FROM pgbench_accounts;

                            QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on pgbench_accounts  (cost=0.00..53746.15 rows=2014215 width=97)
(1 row)
```

The following example uses the `PARALLEL` hint. In the query plan, the Gather node, which launches the background workers, indicates that two workers are planned to be used.

> **Note**
>
> If `trace_hints` is set to `on`, the `INFO: [HINTS]` lines appear stating that `PARALLEL` has been accepted for `pgbench_accounts` as well as other hint information. For the remaining examples, these lines will not be displayed as they generally show the same output (that is, `trace_hints` has been reset to `off`).

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;

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=1000.00..244418.06 rows=2014215 width=97)
   Workers Planned: 2
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
   ->  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:

```
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. Note that the number of planned workers has increased to `4` as determined by the optimizer.

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM pgbench_accounts;

                                  QUERY PLAN
------------------------------------------------------------------------------
------------
 Gather  (cost=1000.00..241061.04 rows=2014215 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 now returned as this specified value:

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts 6) */ * FROM pgbench_accounts;

                                  QUERY PLAN
------------------------------------------------------------------------------
------------
 Gather  (cost=1000.00..239382.52 rows=2014215 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.

```
EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts DEFAULT) */ * FROM
pgbench_accounts;

                                  QUERY PLAN
------------------------------------------------------------------------------
------------
 Gather  (cost=1000.00..239382.52 rows=2014215 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**

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> This format of the `ALTER TABLE` command to set the `parallel_workers` parameter is not 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 "public.pgbench_accounts"
 Column  |     Type      | Modifiers | Storage  | Stats target | Description
---------+---------------+-----------+----------+--------------+------------
-
 aid     | integer       | not null  | plain    |              |
 bid     | integer       |           | plain    |              |
 abalance| integer       |           | plain    |              |
 filler  | character(84) |           | extended |              |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
Options: fillfactor=100, parallel_workers=3
```

Now, 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=1000.00..242522.97 rows=2014215 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.

The following example shows the `NO_PARALLEL` hint. Note that 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=2014215 width=97)
(1 row)
```

9.6.8 Conflicting Hints

If a command includes two or more conflicting hints, the server will ignore the contradictory hints. The following table lists hints that are contradictory to each other.

Hint	Conflicting Hint
`ALL_ROWS`	`FIRST_ROWS` - all formats

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Hint	Conflicting Hint
`FULL(table)`	`INDEX(table [ index ])` `PARALLEL(table [ degree ])`

`INDEX(table)`	`FULL(table)`
	`NO_INDEX(table)`
	`PARALLEL(table [ degree ])`

`INDEX(table index)`	`FULL(table)`
	`NO_INDEX(table index)`
	`PARALLEL(table [ degree ])`

`PARALLEL(table [ degree ])`	`FULL(table)`
	`INDEX(table)`
	`NO_PARALLEL(table)`

`USE_HASH(table)`	`NO_USE_HASH(table)`
`USE_MERGE(table)`	`NO_USE_MERGE(table)`
`USE_NL(table)`	`NO_USE_NL(table)`

9.7 dblink_ora

dblink_ora enables you to issue arbitrary queries to a remote Oracle server. It provides an OCI-based database link that allows you to `SELECT, INSERT, UPDATE` or `DELETE` data stored on an Oracle system from within Advanced Server.

Connecting to an Oracle Database

To enable Oracle connectivity, download Oracle's freely available OCI drivers from their website, presently at:
http://www.oracle.com/technetwork/database/database-technologies/instant-client/overview/index.html

For Linux, if the Oracle instant client that you've downloaded does not 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
```

Before creating a link to an Oracle server, you must direct 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 will override the `LD_LIBRARY_PATH` environment variable in Linux and `PATH` environment variable in Windows.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> **Note**
>
> The `oracle_home` configuration parameter should provide the correct path to the Oracle client i.e. `OCI` library.

To set the `oracle_home` configuration parameter in the `postgresql.conf` file, edit the file, adding the following line:

```
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`.

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.

9.7.1 dblink_ora Functions and Procedures

dblink_ora supports the following functions and procedures.

9.7.1.1 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:

```
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 wish to request `SYSDBA` privileges on the Oracle server. This parameter is optional; if omitted, the default value is `FALSE`.

The first form of `dblink_ora_connect()` returns a `TEXT` value.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The signature of the second form of the `dblink_ora_connect()` function is:

```
dblink_ora_connect(<foreign_server_name>, <asDBA>)
```

Where:

- `foreign_server_name` specifies the name of a foreign server.
- `asDBA` is `True` if you wish to request `SYSDBA` privileges on the Oracle server. This parameter is optional; if omitted, the default value is `FALSE`.

The second form of the `dblink_ora_connect()` function allows you to use the connection properties of a pre-defined foreign server when establishing a connection 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.

9.7.1.2 dblink_ora_status()

The `dblink_ora_status()` function returns the database connection status. The signature is:

```
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`.

9.7.1.3 dblink_ora_disconnect()

The `dblink_ora_disconnect()` function closes a database connection. The signature is:

```
dblink_ora_disconnect(<conn_name>)
```

Where:

`conn_name` specifies the name of the link.

The function returns a `TEXT` value.

9.7.1.4 dblink_ora_record()

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 that will be invoked on the Oracle server.

The function returns a `SETOF` record.

9.7.1.5 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 that will be invoked on the Oracle server.
- `iterations` specifies the number of times the statement is executed.

The function returns a `SETOF` record.

9.7.1.6 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 that will be invoked on the Oracle server.

The function returns a `VOID`.

9.7.1.7 dblink_ora_copy()

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 that will be invoked on the Oracle server.
- `schema_name` specifies the name of the target schema.
- `table_name` specifies the name of the target table.
- `truncate` specifies if the server should `TRUNCATE` the table prior to copying; specify `TRUE` to indicate that the server should `TRUNCATE` the table. `truncate` is optional; if omitted, the value is `FALSE`.
- `count` instructs the server to report status information every `n` record, where `n` is the number specified. During the execution of the function, Advanced Server raises a notice of severity `INFO` with each iteration of the count. For example, if FeedbackCount is `10, dblink_ora_copy()` raises a notice every `10` records. `count` is optional; if omitted, the value is `0`.

9.7.2 Calling dblink_ora Functions

The following command establishes a connection using the `dblink_ora_connect()` function:

```
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` and a password of `pwd`. You can use the connection name `acctg` to refer to this connection when calling other dblink_ora functions.

The following command uses the `dblink_ora_copy()` function over a connection named `edb_conn` to copy 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 Advanced Server named `as_acctg`. The `TRUNCATE` option is enforced, and a feedback count of `3` is specified:

```
edb=# SELECT dblink_ora_copy('edb_conn','select empid, deptno FROM
ora_acctg', 'public', 'as_acctg', true, 3);

INFO:  Row: 0
INFO:  Row: 3
INFO:  Row: 6
INFO:  Row: 9
INFO:  Row: 12

 dblink_ora_copy
-----------------
 12

(1 row)
```

The following `SELECT` statement uses `dblink_ora_record()` function and the `acctg` connection to retrieve information from the Oracle server:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

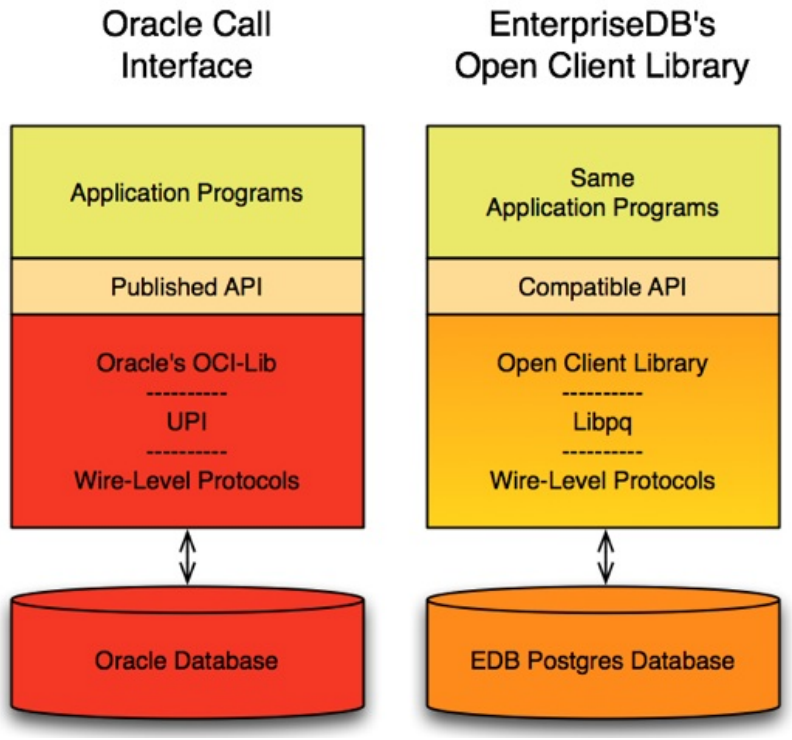
```
SELECT * FROM dblink_ora_record( 'acctg', 'SELECT first_name from employees') AS t1(id VARCHAR);
```

The command retrieves a list that includes all of the entries in the `first_name` column of the `employees` table.

9.8 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 EnterpriseDB 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, seeEDB OCL Connector.

> **Note**
>
> EnterpriseDB does not support use of the Open Client Library with Oracle Real Application Clusters (RAC) and Oracle Exadata; the aforementioned Oracle products have not been evaluated nor certified with this EnterpriseDB product.

9.9 Oracle Catalog Views

The Oracle Catalog Views provide information about database objects in a manner compatible with the Oracle data dictionary views. Information about the supported views is now available in the *Database Compatibility for Oracle Developers Catalog Views Guide*, available at:

https://www.enterprisedb.com/docs

9.10 Tools and Utilities

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Compatible tools and utility programs can allow a developer to work with Advanced Server in a familiar environment. The tools supported by Advanced Server include:

- EDB*Plus
- EDB*Loader
- EDB*Wrap
- The Dynamic Runtime Instrumentation Tools Architecture (DRITA)

For detailed information about the functionality supported by EDB Postgres Advanced Server, seeTools, utilities, and components.

9.11 ECPGPlus

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) 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 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 do not need to include the `BEGIN DECLARE SECTION` and `END DECLARE SECTION` directives.

For more information about using ECPGPlus, see the *EDB Postgres Advanced Server ECPG Connector Guide* available from the EnterpriseDB website at:

https://www.enterprisedb.com/docs

9.12 System Catalog Tables

The system catalog tables contain definitions of database objects that are available to Advanced Server; the layout of the system tables is subject to change. If you are writing an application that depends on information stored in the system tables, it would be prudent to use an existing catalog view, or create a catalog view to isolate the application from changes to the system table.

For detailed information about the system catalog tables, see the *Database Compatibility for Oracle Developers Catalog Views Guide*, available at:

https://www.enterprisedb.com/docs

10 Database Compatibility for Oracle Developers Reference Guide

Database Compatibility for Oracle means that an application runs in an Oracle environment as well as in the EDB Postgres Advanced Server (Advanced Server) environment with minimal or no changes to the application code.

This guide provides reference material about the compatible data types supported by Advanced Server. Reference information about:

- Compatible SQL Language syntax is provided in the *Database Compatibility for Oracle Developers SQL Guide*.
- Compatible Catalog Views is provided in the *Database Compatibility for Oracle Developers Catalog Views Guide*.

Developing an application that is compatible with Oracle databases in the Advanced Server requires special attention to which features are used in the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

construction of the application. For example, developing a compatible application means selecting:

- Data types to define the application's database tables 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 that are compatible with Oracle databases
- Stored Procedure Language (SPL) to create database server-side application logic for stored procedures, functions, triggers, and packages
- System catalog views that are compatible with Oracle's data dictionary

10.1 The SQL Language

The following sections describe the subset of the Advanced Server SQL language compatible with Oracle databases. The following SQL syntax, data types, and functions work in both EDB Postgres Advanced Server and Oracle.

The Advanced Server documentation set includes syntax and commands for extended functionality (functionality that does not provide database compatibility for Oracle or support Oracle-styled applications) that is not included in this guide.

This section is organized into the following sections:

- General discussion of Advanced Server SQL syntax and language elements
- Data types
- Summary of SQL Commands
- Built-in functions

10.1.1 SQL Syntax

This section describes the general syntax of SQL. It forms the foundation for understanding the following chapters that include detail about how the SQL commands are applied to define and modify data.

10.1.1.1 Lexical Structure

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. Which tokens are valid depends on the syntax of the particular 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 need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

Additionally, *comments* can occur in SQL input. They are not tokens -they are effectively 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 this is not required; more than one command can be on a line, and commands can usually be split across lines).

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

the command name, so in the above example we would usually speak of 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 particular variation of `INSERT` also requires a `VALUES` token in order to be complete. The precise syntax rules for each command are described in *Database Compatibility for Oracle Developers SQL Guide*.

10.1.1.2 Identifiers and Key Words

Tokens such as `SELECT`, `UPDATE`, or `VALUES` in the example above 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 they are used in. Therefore they are sometimes simply called, "names". Key words and identifiers have the same *lexical structure*, meaning that one cannot 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 number signs (`#`).

Identifier and key word names are case insensitive. Therefore

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as:

```
uPDaTE my_TabLE SeT a = 5;
```

A convention often used is to write key words in upper case and names in lower case, e.g.,

```
UPDATE my_table SET a = 5;
```

There is a second kind of identifier: the *delimited identifier* or *quoted identifier*. It is formed by enclosing an arbitrary sequence of characters in double-quotes ("). A delimited identifier is always an identifier, never a key word. So `"select"` could be used to refer to a column or table named `"select"`, whereas an unquoted select would be taken as a key word and would therefore provoke 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 would otherwise not be 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 Advanced Server, but `"Foo"` and `"FOO"` are different from these three and each other. The folding of unquoted names to lower case is not compatible with Oracle databases. In Oracle syntax, unquoted names are folded to upper case: for example, `foo` is equivalent to `"FOO"` not `"foo"`. If you want to write portable applications you are advised to always quote a particular name or never quote it.

10.1.1.3 Constants

The kinds of implicitly-typed constants in Advanced Server are *strings* and *numbers*. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 within a string constant, write two adjacent single quotes, e.g. `'Dianne''s horse'`. Note that this is not the same as a double-quote character `(")`.

Numeric Constants

Numeric constants are accepted in these general forms:

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

where `digits` is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker `(e)`, if one is present. There may not be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator 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 contains neither a decimal point nor an exponent is initially presumed to be type `INTEGER` if its value fits in type `INTEGER` (32 bits); otherwise it is presumed to be type `BIGINT` if its value fits in type `BIGINT` (64 bits); otherwise it is taken to be type `NUMBER`. Constants that contain decimal points and/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 will be 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 the following section.

Constants of Other Types

CAST

A constant of an arbitrary type can be entered using the following notation:

```
CAST('string' AS type)
```

The string constant's text is passed to the input conversion routine for the type called `type`. The result is a constant of the indicated type. The explicit type cast may be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

`CAST` can also be used to specify runtime type conversions of arbitrary expressions.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

10.1.1.4 Comments

A comment is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard SQL comment
```

Alternatively, C-style block comments can be used:

```
 /* multiline comment
* block
*/
```

where the comment begins with /* and extends to the matching occurrence of */.

10.1.2 Data Types

The following table shows the built-in general-purpose data types:

Name	Alias	Description
`BLOB`	`LONG RAW, RAW(n), BYTEA`	Binary data
`BOOLEAN`		Logical Boolean (true/false)
`CHAR [ (n) ]`	`CHARACTER [ (n) ]`	Fixed-length character string of `n` characters
`CLOB`	`LONG, LONG VARCHAR`	Long character string
`DATE`	`TIMESTAMP`	Date and time to the second
`DOUBLE PRECISION`	`FLOAT, FLOAT(25) - FLOAT(53)`	Double precision floating-point number
`INTEGER`	`INT, BINARY_INTEGER, PLS_INTEGER`	Signed four-byte integer
`NUMBER`	`DEC, DECIMAL, NUMERIC`	Exact numeric with optional decimal places
`NUMBER(p [, s ])`	`DEC(p [, s ]),DECIMAL(p [, s ]),NUMERIC(p [, s ])`	Exact numeric of maximum precision, `p`, and optional scale, `s`
`REAL`	`FLOAT(1) - FLOAT(24)`	Single precision floating-point number
`TIMESTAMP [ (p) ]`		Date and time with optional, fractional second precision, `p`
`TIMESTAMP [ (p) ] WITH TIME ZONE`		Date and time with optional, fractional second precision, `p`, and with time zone
`VARCHAR2(n)`	`CHAR VARYING(n), CHARACTER VARYING(n), VARCHAR(n)`	Variable-length character string with a maximum length of `n` characters
`XMLTYPE`		XML data

10.1.2.1 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Storage Size	Description	Range
`BINARY_INTEGER`	4 bytes	Signed integer, Alias for `INTEGER`	-2,147,483,648 to +2,147,483,647
`DOUBLE PRECISION`	8 bytes	Variable-precision, inexact	15 decimal digits precision
`INTEGER`	4 bytes	Usual choice for integer	-2,147,483,648 to +2,147,483,647
`NUMBER`	Variable	User-specified precision, exact	Up to 1000 digits of precision
`NUMBER(p [, s ])`	Variable	Exact numeric of maximum precision, `p`, and optional scale, `s`	Up to 1000 digits of precision
`PLS_INTEGER`	4 bytes	Signed integer, Alias for `INTEGER`	-2,147,483,648 to +2,147,483,647
`REAL`	4 bytes	Variable-precision, inexact	6 decimal digits precision
`ROWID`	8 bytes	Signed 8 bit integer.	-9223372036854775808 to 9223372036854775807

The following sections describe the types in detail.

Integer Types

The `BINARY_INTEGER`, `INTEGER`, `PLS_INTEGER`, and `ROWID` types store whole numbers (without fractional components) as specified in table *Numeric Types*. Attempts to store values outside of the allowed range will result in an error.

Arbitrary Precision Numbers

The type, `NUMBER`, can store practically an unlimited number of digits of precision and perform calculations exactly. It is especially recommended 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 the next section.

In what follows we use these terms: 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 can be considered to have a scale of zero.

Both the precision and the scale of the `NUMBER` type can be configured. To declare a column of type `NUMBER` use the syntax

```
NUMBER(precision, scale)
```

The precision must be positive, the scale zero or positive. Alternatively,

```
NUMBER(precision)
```

selects a scale of 0. Specifying `NUMBER` without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas `NUMBER` columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. For maximum portability, it is 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 will attempt to round the value. If the value cannot be rounded so as to satisfy the declared limits, an error is raised.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 cannot be converted exactly to the internal format and are stored as approximations, so that storing and printing back out a value may 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 will not be 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), you should evaluate the implementation carefully.

Comparing two floating-point values for equality may or may not work as expected.

On most platforms, the `REAL` type has a range of at least `1E-37` to `1E+37` with a precision of at least 6 decimal digits. The `DOUBLE PRECISION` type typically has a range of around `1E-307` to `1E+308` with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding may take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

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. Advanced Server accepts `FLOAT(1)` to `FLOAT(24)` as selecting the `REAL` type, while `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`.

10.1.2.2 Character Types

The following table lists the general-purpose character types available in Advanced Server:

Name	Description
`CHAR[(n)]`	Fixed-length character string, blank-padded to the size specified by `n`
`CLOB`	Large variable-length up to 1 GB
`LONG`	Variable unlimited length.
`NVARCHAR(n)`	Variable-length national character string, with limit
`NVARCHAR2(n)`	Variable-length national character string, with limit
`STRING`	Alias for `VARCHAR2`
`VARCHAR(n)`	Variable-length character string, with limit (considered deprecated, but supported for compatibility)
`VARCHAR2(n)`	Variable-length character string, with limit

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` will result in an error, unless the excess characters are all spaces, in which case the string will be 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 may be less. Long values are stored in background tables so they do not interfere with rapid access to the shorter column values.

The database character set determines the character set used to store textual values.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`CHAR`

If you do not specify a value for `n`, `n` will default to `1`. If the string to be assigned is shorter than `n`, values of type `CHAR` will be space-padded to the specified width (`n`), and will be 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 will be 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 will be truncated to `n` characters without raising an error (as specified by the SQL standard).

```
VARCHAR, VARCHAR2, NVARCHAR and NVARCHAR2
```

If the string to be assigned is shorter than `n`, values of type `VARCHAR`, `VARCHAR2`, `NVARCHAR` and `NVARCHAR2` will store the shorter string without padding.

Note: The trailing spaces *are* semantically significant in `VARCHAR` values.

If you explicitly cast a value to a `VARCHAR` type, an over-length value will be 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, you should use a `CLOB` type if the maximum string length is not known.

The longest possible character string that can be stored 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 at https://www.postgresql.org/docs/11/static/sql-createdomain.html

Thus, usage of the `CLOB` type is limited by what can be done for `TEXT` such as a maximum size of approximately 1 GB.

For usage of 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 at:

https://www.postgresql.org/docs/11/static/largeobjects.html

10.1.2.3 Binary Data

The following table shows data types that allow the storage of binary strings:

Name	Storage Size	Description
`BINARY`	The length of the binary string.	Fixed-length binary string, with a length between 1 and 8300.
`BLOB`	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.	Variable-length binary string, with a maximum size of 1 GB.
`VARBINARY`	The length of the binary string	Variable-length binary string, with a length between 1 and 8300.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from characters strings by two characteristics: First, binary strings specifically allow storing octets of value zero and other "non-printable" octets (defined as octets outside the range 32 to 126). Second, operations on binary strings process the actual bytes, whereas the encoding and processing of character strings depends on locale settings.

10.1.2.4 Date/Time Types

The following discussion of the date/time types assumes that the configuration parameter, `edb_redwood_date`, has been set to `TRUE` whenever a table is created or altered.

Advanced Server supports the date/time types shown in the following table:

Name	Storage Size	Description	Low Value	High Value	Resolution
`DATE`	8 bytes	Date and time	4713 BC	5874897 AD	1 second
`INTERVAL DAY TO SECOND [(p)]`	12 bytes	Period of time	-178000000 years	178000000 years	1 microsecond / 14 digits
`INTERVAL YEAR TO MONTH`	12 bytes	Period of time	-178000000 years	178000000 years	1 microsecond / 14 digits
`TIMESTAMP [(p)]`	8 bytes	Date and time	4713 BC	5874897 AD	1 microsecond
`TIMESTAMP [(p)] WITH TIME ZONE`	8 bytes	Date and time with time zone	4713 BC	5874897 AD	1 microsecond

When `DATE` appears as the data type of a column in the data definition language (DDL) commands, `CREATE TABLE` or `ALTER TABLE`, it is translated to `TIMESTAMP` at the time the table definition is stored in the database. Thus, a time component will also be stored in the column along with the date.

When `DATE` appears as a data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or an SPL function, or the return type of an SPL function, it is always translated to `TIMESTAMP` and thus can handle a time component if present.

`TIMESTAMP` accepts an optional precision value `p` which specifies the number of fractional digits retained in the seconds field. The allowed range of `p` is from `0` to `6` with the default being `6`.

When `TIMESTAMP` values are stored as double precision floating-point numbers (currently the default), the effective limit of precision may 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 eight-byte integers (a compile-time option), microsecond precision is available over the full range of values. However eight-byte integer timestamps have a more limited range of dates than shown above: 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:

Field Name	INTERVAL Values Allowed
`YEAR`	Integer value (positive or negative)
`MONTH`	0 through 11

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Field Name	INTERVAL Values Allowed
`DAY`	Integer value (positive or negative)
`HOUR`	0 through 23
`MINUTE`	0 through 59
`SECOND`	0 through 59.9(p) where 9(p) is the precision of fractional seconds

The fields must be presented in descending order – from `YEARS` to `MONTHS`, and from `DAYS` to `HOURS`, `MINUTES` and then `SECONDS`.

Advanced Server supports two `INTERVAL` types compatible with Oracle databases.

The first variation supported by 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.

Advanced Server interprets the value:

```
INTERVAL '1 2:34:5.678' DAY TO SECOND(3)
```

as 1 day, 2 hours, 34 minutes, 5 seconds and 678 thousandths of a second.

Advanced Server interprets the value:

```
INTERVAL '1 23' DAY TO HOUR
```

as 1 day and 23 hours.

Advanced Server interprets the value:

```
INTERVAL '2:34' HOUR TO MINUTE
```

as 2 hours and 34 minutes.

Advanced Server interprets the value:

```
INTERVAL '2:34:56.129' HOUR TO SECOND(2)
```

as 2 hours, 34 minutes, 56 seconds and 13 thousandths of a second. Note that the fractional second is rounded up to 13 because of the specified precision.

The second variation supported by Advanced Server that is compatible with Oracle databases is `INTERVAL YEAR TO MONTH`. This variation stores a time interval in years and months.

Advanced Server interprets the value:

```
INTERVAL '12-3' YEAR TO MONTH
```

as 12 years and 3 months.

Advanced Server interprets the value:

```
INTERVAL '456' YEAR(2)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

as 12 years and 3 months.

Advanced Server interprets the value:

```
INTERVAL '300' MONTH
```

as 25 years.

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 is no ambiguity as to which component is the year, month, and day. However, use of the `TO_DATE` function is strongly recommended to avoid ambiguities.

Any date or time literal input needs to be enclosed 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.

```
 Example
January 8, 1999
1999-01-08
1999-Jan-08
Jan-08-1999
08-Jan-1999
08-Jan-99
Jan-08-99
19990108
990108
```

The date values can be assigned to a `DATE` or `TIMESTAMP` column or variable. The hour, minute, and seconds fields will be set to zero if the date value is not appended with a time value.

Times

Some examples of the time component of a date or time stamp are shown in the following table:

Example	Description
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Example	Description
`040506`	ISO 8601
`04:05 AM`	Same as 04:05; AM does not affect value
`04:05 PM`	Same as 16:05; input hour must be <= 12

Time Stamps

Valid input for time stamps consists of a concatenation of a date and a time. The date portion of the time stamp can be formatted according to any of the examples shown in the block under section `Dates`. The time portion of the time stamp can be formatted according to any of examples shown in table under section `Times`.

The following is an example of a time stamp which follows the Oracle default format.

```
08-JAN-99 04:05:06
```

The following is an example of a time stamp which follows the ISO 8601 standard.

```
1999-01-08 04:05:06
```

Date/Time Output

The default output format of the date/time types will be either `dd-MON-yy`) referred to as the *Redwood date style*, compatible with Oracle databases, or (`yyyy-mm-dd`) referred to as the ISO 8601 format, depending upon 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:

Description	Example
Redwood style	`31-DEC-05 07:37:16`
ISO 8601/SQL standard	`1997-12-17 07:37:16`

Internals

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.

10.1.2.5 Boolean Types

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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Storage Size	Description
BOOLEAN	1 byte	Logical Boolean (true/false)

The valid literal value for representing the true state is `TRUE`. The valid literal for representing the false state is `FALSE`.

10.1.2.6 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 well-formedness, and there are support functions to 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 does not support the storage of content fragments in `XMLTYPE` columns.

The following example shows the creation and insertion of a row into a table with an `XMLTYPE` column.

```
CREATE TABLE books (
    content         XMLTYPE
);

INSERT INTO books VALUES (XMLPARSE (DOCUMENT '<?xml
version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>'));

SELECT * FROM books;

                         content
-------------------------------------------------------------
 <book><title>Manual</title><chapter>...</chapter></book>
(1 row)
```

10.1.3 Functions and Operators

Advanced Server provides a large number of functions and operators for the built-in data types.

10.1.3.1 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`

a b a AND b a OR b

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

a	b	a AND b	a OR b
True	True	True	True
True	False	False	True
True	Null	Null	True
False	False	False	False
False	Null	False	Null
Null	Null	Null	Null

`NOT Truth Table`

a	NOT a
True	False
False	True
Null	Null

The operators `AND` and `OR` are commutative, that is, you can switch the left and right operand without affecting the result.

10.1.3.2 Comparison Operators

The usual comparison operators are shown in the following table:

Operator	Description
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`=`	Equal
`<>`	Not equal
`!=`	Not equal

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type `BOOLEAN;` expressions like `1 < 2 < 3` are not valid (because there is no `<` operator to compare a Boolean value with `3`).

In addition to the comparison operators, the special `BETWEEN` construct is available.

`a BETWEEN x AND y`

is equivalent to

`a >= x AND a <= y`

Similarly,

`a NOT BETWEEN x AND y`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

is equivalent to

```
a < x OR a > y
```

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally.

To check whether a value is or is not null, use the constructs

```
expression IS NULL
```

```
expression IS NOT NULL
```

Do not write `expression = NULL` because `NULL` is not "equal to" `NULL` . (The null value represents an unknown value, and it is not known whether two unknown values are equal.) This behavior conforms to the SQL standard.

Some applications may expect that `expression = NULL` returns true if `expression` evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard.

10.1.3.3 Mathematical Functions and Operators

Mathematical operators are provided for many Advanced Server types. For types without common mathematical conventions for all possible permutations (e.g., date/time types) the actual behavior is described in subsequent sections.

The following table shows the available mathematical operators:

Operator	Description	Example	Result
`+`	Addition	`2 + 3`	`5`
`-`	Subtraction	`2 - 3`	`-1`
`*`	Multiplication	`2 * 3`	`6`
`/`	Division (See the following note)	`4 / 2`	`2`
`**`	Exponentiation Operator	`2 ** 3`	`8`

> **Note**
>
> If the `db_dialect` configuration parameter in the `postgresql.conf` file is set to `redwood` , then division of a pair of `INTEGER` data types does not result in a truncated value. Any fractional result is retained as shown by the following example:

```
 edb=# SET db_dialect TO redwood;
SET
edb=# SHOW db_dialect;
 db_dialect
------------
 redwood
(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS INTEGER) FROM dual;
      ?column?
--------------------
 3.3333333333333333
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

This behavior is compatible with Oracle databases where there is no native `INTEGER` data type, and 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 as shown by the following example:

```
 edb=# SET db_dialect TO postgres;
SET
edb=# SHOW db_dialect;
 db_dialect
------------
 postgres
(1 row)

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 truncation of the result. The same truncated result is returned by Advanced Server when `db_dialect` is set to `postgres` as shown in the previous example.

Note however, that 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, does result in truncation in the PostgreSQL fashion without retaining the fractional portion as shown by the following where `INTEGER` and `SMALLINT` are involved in the division:

```
 edb=# SHOW db_dialect;
 db_dialect
------------
 redwood
(1 row)

edb=# SELECT CAST('10' AS INTEGER) / CAST('3' AS SMALLINT) FROM dual;
 ?column?
----------
        3
(1 row)
```

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 may therefore vary depending on the host system.

Function	Return Type	Description	Example	Result
`ABS(x)`	Same as `x`	Absolute value	`ABS(-17.4)`	`17.4`
`CEIL(DOUBLE PRECISION or NUMBER)`	Same as input	Smallest integer not less than argument	`CEIL(-42.8)`	`-42`
`EXP(DOUBLE PRECISION or NUMBER)`	Same as input	Exponential	`EXP(1.0)`	`2.71828182845 90452`
`FLOOR(DOUBLE PRECISION or NUMBER)`	Same as input	Largest integer not greater than argument	`FLOOR(-42.8)`	`43`
`LN(DOUBLE PRECISION or NUMBER)`	Same as input	Natural logarithm	`LN(2.0)`	`0.6931 4718055 99453`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function	Return Type	Description	Example	Result
`LOG(b NUMBER, x NUMBER)`	`NUMBER`	Logarithm to base `b`	`LOG(2.0, 64.0)`	`6.0000000000000000`
`MOD(y, x)`	Same as argument types	Remainder of `y/x`	`MOD(9, 4)`	`1`
`NVL(x, y)`	Same as argument types; where both arguments are of the same data type	If `x` is null, then NVL returns `y`	`NVL(9, 0)`	`9`
`POWER(a DOUBLE PRECISION, b DOUBLE PRECISION)`	`DOUBLE PRECISION`	`a` raised to the power of `b`	`POWER(9.0, 3.0)`	`729.00000000000000`
`POWER(a NUMBER, b NUMBER)`	`NUMBER`	`a` raised to the power of `b`	`POWER(9.0, 3.0)`	`729.00000000000000`
`ROUND(DOUBLE PRECISION or NUMBER)`	Same as input	Round to nearest integer	`ROUND(42.4)`	`42`
`ROUND(v NUMBER, s INTEGER)`	`NUMBER`	Round to `s` decimal places	`ROUND(42.4382, 2)`	`42.44`
`SIGN(DOUBLE PRECISION or NUMBER)`	Same as input	Sign of the argument (-1, 0, +1)	`SIGN(-8.4)`	`-1`
`SQRT(DOUBLE PRECISION or NUMBER)`	Same as input	Square root	`SQRT(2.0)`	`1.41421356237 3095`
`TRUNC(DOUBLE PRECISION or NUMBER)`	Same as input	Truncate toward zero	`TRUNC(42.8)`	`42`
`TRUNC(v NUMBER, s INTEGER)`	`NUMBER`	Truncate to `s` decimal places	`TRUNC(42.4382, 2)`	`42.43`
`WIDTH_BUCKET(op NUMBER, b1 NUMBER, b2 NUMBER, count INTEGER)`	`INTEGER`	Return the bucket to which `op` would be assigned in an equidepth histogram with `count` buckets, in the range `b1` to `b2`	`WIDTH_BUCKET (5.35, 0.024, 10.06, 5)`	`3`

The following table shows the available trigonometric functions. All trigonometric functions take arguments and return values of type `DOUBLE PRECISION`.

Function	Description
`ACOS(x)`	Inverse cosine
`ASIN(x)`	Inverse sine
`ATAN(x)`	Inverse tangent
`ATAN2(x, y)`	Inverse tangent of `x/y`
`COS(x)`	Cosine
`SIN(x)`	Sine
`TAN(x)`	Tangent

10.1.3.4 String Functions and Operators

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types `CHAR`, `VARCHAR2`, and `CLOB`. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of automatic padding when using the `CHAR` type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first.

Function	Return Type	Description	Example	Result
string || string	`CLOB`	String concatenation	'Enterprise' || 'DB'	`EnterpriseDB`
`CONCAT(string, string)`	`CLOB`	String concatenation	'a' || 'b'	`ab`
`HEXTORAW(varchar2)`	`RAW`	Converts a `VARCHAR2` value to a `RAW value`	`HEXTORAW('303132')`	`'012'`
`RAWTOHEX(raw)`	`VARCHAR2`	Converts a `RAW` value to a `HEXADECIMAL` value	`RAWTOHEX('012')`	`'303132'`
`INSTR(string, set, [ start [, occurrence ] ])`	`INTEGER`	Finds the location of a set of characters in a string, starting at position `start` in the string, `string`, and looking for the first, second, third and so on occurrences of the set. Returns `0` if the set is not found.	`INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',1,3)`	`30`
`INSTRB(string, set)`	`INTEGER`	Returns the position of the `set` within the `string`. Returns `0` if `set` is not found.	`INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PICK')`	`13`
`INSTRB(string, set, start)`	`INTEGER`	Returns the position of the `set` within the `string`, beginning at `start`. Returns `0` if `set` is not found.	`INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 14)`	`30`
`INSTRB(string, set, start, occurrence)`	`INTEGER`	Returns the position of the specified `occurrence` of `set` within the `string`, beginning at `start`. Returns `0` if `set` is not found.	`INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 1, 2)`	`30`
`LOWER(string)`	`CLOB`	Convert `string` to lower case	`LOWER('TOM')`	`tom`
`SUBSTR(string, start [, count ])`	`CLOB`	Extract substring starting from `start` and going for `count` characters. If `count` is not specified, the string is clipped from the start till the end.	`SUBSTR('This is a test',6,2)`	`is`
`SUBSTRB(string, start [, count ])`	`CLOB`	Same as `SUBSTR` except `start` and `count` are in number of bytes.	`SUBSTRB('abc',3)` (assuming a double-byte character set)	`c`
`SUBSTR2(string, start [, count ])`	`CLOB`	Alias for `SUBSTR`.	`SUBSTR2('This is a test',6,2)`	`is`
`SUBSTR2(string, start [, count ])`	`CLOB`	Alias for `SUBSTRB`.	`SUBSTR2('abc',3)` (assuming a double-byte character set)	`c`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function	Return Type	Description	Example	Result
`SUBSTR4(string, start [, count ])`	`CLOB`	Alias for `SUBSTR`.	`SUBSTR4('This is a test',6,2)`	`is`
`SUBSTR4(string, start [, count ])`	`CLOB`	Alias for `SUBSTRB`.	`SUBSTR4('abc',3)` (assuming a double-byte character set)	`c`
`SUBSTRC(string, start [, count ])`	`CLOB`	Alias for `SUBSTR`.	`SUBSTRC('This is a test',6,2)`	`is`
`SUBSTRC(string, start [, count ])`	`CLOB`	Alias for `SUBSTRB`.	`SUBSTRC('abc',3)` (assuming a double-byte character set)	`c`
TRIM([LEADING | TRAILING | BOTH] [characters] FROM string)	`CLOB`	Remove the longest string containing only the characters (a space by default) from the start/end/both ends of the string.	`TRIM(BOTH 'x' FROM 'xTomxx')`	`Tom`
`LTRIM(string [, set])`	`CLOB`	Removes all the characters specified in `set` from the left of a given `string`. If `set` is not specified, a blank space is used as default.	`LTRIM('abcdefghi', 'abc')`	`defghi`
`RTRIM(string [, set])`	`CLOB`	Removes all the characters specified in `set` from the right of a given `string`. If `set` is not specified, a blank space is used as default.	`RTRIM('abcdefghi', 'ghi')`	`abcdef`
`UPPER(string)`	`CLOB`	Convert `string` to upper case	`UPPER('tom')`	`TOM`

Additional 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.

Function	Return Type	Description	Example	Result
`ASCII(string)`	`INTEGER`	ASCII code of the first byte of the argument	`ASCII('x')`	`120`
`CHR(INTEGER)`	`CLOB`	Character with the given ASCII code	`CHR(65)`	`A`
DECODE(expr, expr1a, expr1b [, expr2a, expr2b]... [, default])	Same as argument types of `expr1b, expr2b,..., default`	Finds first match of `expr` with `expr1a, expr2a`, 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.	`DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')`	`Three`
`INITCAP(string)`	`CLOB`	Convert the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.	`INITCAP('hi THOMAS')`	`Hi Thomas`
`LENGTH`	`INTEGER`	Returns the number of characters in a string value.	`LENGTH('Côte d''Azur')`	`11`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function	Return Type	Description	Example	Result
`LENGTHC`	`INTEGER`	This function is identical in functionality to `LENGTH`; the function name is supported for compatibility.	`LENGTHC('Côte d''Azur')`	`11`
`LENGTH2`	`INTEGER`	This function is identical in functionality to `LENGTH`; the function name is supported for compatibility.	`LENGTH2('Côte d''Azur')`	`11`
`LENGTH4`	`INTEGER`	This function is identical in functionality to `LENGTH`; the function name is supported for compatibility.	`LENGTH4('Côte d''Azur')`	`11`
`LENGTHB`	`INTEGER`	Returns the number of bytes required to hold the given value.	`LENGTHB('Côte d''Azur')`	`12`
`LPAD(string, length INTEGER [, fill ])`	`CLOB`	Fill 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).	`LPAD('hi', 5, 'xy')`	`xyxhi`
`REPLACE(string, search_string [, replace_string ]`	`CLOB`	Replaces one value in a string with another. If you do not specify a value for `replace_string`, the `search_string` value when found, is removed.	`REPLACE('GEORGE', 'GE', 'EG')`	`EGORG`
`RPAD(string, length INTEGER [, fill ])`	`CLOB`	Fill up `string` to size, `length` by appending the characters, `fill` (a space by default). If `string` is already longer than `length` then it is truncated.	`RPAD('hi', 5, 'xy')`	`hixyx`
`TRANSLATE(string, from, to)`	`CLOB`	Any character in `string` that matches a character in the `from` set is replaced by the corresponding character in the `to` set.	`TRANSLATE('12345', '14', 'ax')`	`a23x5`

Truncation of String Text Resulting from Concatenation with NULL

> **Note**
>
> This section describes a functionality that is not compatible with Oracle databases, which may lead to some inconsistency when converting data from Oracle to Advanced Server.

For Advanced Server, when a column value is `NULL`, the concatenation of the column with a text string may result in either of the following:

- Return of the text string
- Disappearance of the text string (that is, a null result)

The result is dependent upon the data type of the `NULL` column and the way in which the concatenation is done.

If one uses the string concatenation operator `'||'`, then the types that have implicit coercion to text as listed in Table Data Types with Implicit Coercion to Text will not truncate the string if one of the input parameters is `NULL`, whereas for other types it will truncate the string unless the explicit type cast is used (that is, `::text`). Also, to see the consistent behavior in the presence of nulls, one can use the `CONCAT` function.

The following query lists the data types that have implicit coercion to text:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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'
      AND castcontext='i';
```

The result of the query is listed in the following table:

castsource	casttarget	castfunc	castcontext	castmethod
character	text	pg_catalog.text	implicit in expressions	function
character varying	text	-	implicit in expressions	binary-coercible
"char"	text	pg_catalog.text	implicit in expressions	function
name	text	pg_catalog.text	implicit in expressions	function
pg_node_tree	text	-	implicit in expressions	binary-coercible
pg_ndistinct	text	-	implicit in expressions	input/output function
pg_dependencies	text	-	implicit in expressions	input/output function
integer	text	-	implicit in expressions	input/output function
smallint	text	-	implicit in expressions	input/output function
oid	text	-	implicit in expressions	input/output function
date	text	-	implicit in expressions	input/output function
double precision	text	-	implicit in expressions	input/output function
real	text	-	implicit in expressions	input/output function
time with time zone	text	-	implicit in expressions	input/output function
time without time zone	text	-	implicit in expressions	input/output function
timestamp with time zone	text	-	implicit in expressions	input/output function
interval	text	-	implicit in expressions	input/output function

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

castsource	casttarget	castfunc	castcontext	castmethod
bigint	text	-	implicit in expressions	input/output function
numeric	text	-	implicit in expressions	input/output function
timestamp without time zone	text	-	implicit in expressions	input/output function
record	text	-	implicit in expressions	input/output function
boolean	text	pg_catalog.text	implicit in expressions	function
bytea	text	-	implicit in expressions	input/output function

For information on the column output, see the `pg_cast` system catalog in the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/catalog-pg-cast.html

So for example, data type `UUID` is not in this list and therefore does not 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.

```
CREATE TABLE null_concat_types (
    boolean_type    BOOLEAN,
    uuid_type       UUID,
    char_type       CHARACTER
);

INSERT INTO null_concat_types VALUES (NULL, NULL, NULL);
```

Columns `boolean_type` and `char_type` have the implicit coercion to text while column `uuid_type` does not.

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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 ?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 upon the data type of a `NULL` column, explicit casting or the `CONCAT` function should be used to avoid loss of some text string.

10.1.3.5 Pattern Matching String Functions

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 within the string. The pattern should be a POSIX-style regular expression; for more information about forming a POSIX-style regular expression, please refer to the core documentation at:

https://www.postgresql.org/docs/11/static/functions-matching.html

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 1
  modifier  DEFAULT NULL
)
```

Parameters

`srcstr`

`srcstr` specifies the string to search.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`pattern`

`pattern` specifies the regular expression for which `REGEXP_COUNT` will search.

`position`

`position` is an integer value that indicates the position in the source string at which `REGEXP_COUNT` will begin 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 Advanced Server, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/functions-matching.html

Example

In the following simple 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)
```

In the first example, the command instructs `REGEXP_COUNT` begins counting in the first position; if we modify the command to start the count on the 6th position:

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 6) FROM DUAL;
 regexp_count
--------------
            3
(1 row)
```

`REGEXP_COUNT` returns `3`; the count now excludes any occurrences of the letter `i` that occur before the 6th position.

REGEXP_INSTR

`REGEXP_INSTR` searches a string for a POSIX-style regular expression. This function returns the position within 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,
)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameters:

`srcstr`

`srcstr` specifies the string to search.

`pattern`

`pattern` specifies the regular expression for which `REGEXP_INSTR` will search.

`position`

`position` specifies an integer value that indicates the start position in a source string. The default value is `1`.

`occurrence`

`occurrence` specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is `1`.

`returnparam`

`returnparam` is an integer value that specifies the location within the string that `REGEXP_INSTR` should return. The default value is `0`. Specify:

- `0` to return the location within 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`

`modifier` specifies values that control the pattern matching behavior. The default value is `NULL`. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/functions-matching.html

`subexpression`

`subexpression` is an integer value that identifies the portion of the `pattern` that will be returned by `REGEXP_INSTR`. The default value of `subexpression` is `0`.

If you specify a value for `subexpression`, you must include one (or more) set of parentheses in the `pattern` that isolate a portion of the value being searched for. The value specified by `subexpression` indicates which set of parentheses should be returned; for example, if `subexpression` is `2`, `REGEXP_INSTR` will return the position of the second set of parentheses.

Example

In the following simple example, `REGEXP_INSTR` searches a string that contains the a phone number for the first occurrence of a pattern that contains three consecutive digits:

```
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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

occurrence of three consecutive digits:

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;
 regexp_instr
--------------
            5
(1 row)
```

`REGEXP_INSTR` returns `5`; the second occurrence of three consecutive digits begins in the 5th position.

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:

```
TEXT REGEXP_SUBSTR
(
  srcstr        TEXT,
  pattern       TEXT,
  position      INT  DEFAULT 1,
  occurrence    INT  DEFAULT 1,
  modifier      TEXT DEFAULT NULL,
  subexpression INT  DEFAULT 0
)
```

Parameters:

`srcstr`

`srcstr` specifies the string to search.

`pattern`

`pattern` specifies the regular expression for which `REGEXP_SUBSTR` will search.

`position`

`position` specifies an integer value that indicates the start position in a source string. The default value is `1`.

`occurrence`

`occurrence` specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. 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 Advanced Server, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/functions-matching.html

`subexpression`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`subexpression` is an integer value that identifies the portion of the `pattern` that will be returned by `REGEXP_SUBSTR`. The default value of `subexpression` is `0`.

If you specify a value for `subexpression`, you must include one (or more) set of parentheses in the `pattern` that isolate a portion of the value being searched for. The value specified by `subexpression` indicates which set of parentheses should be returned; for example, if `subexpression` is `2`, `REGEXP_SUBSTR` will return the value contained within the second set of parentheses.

Example

In the following simple 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:

```
edb=# SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;
 regexp_substr
---------------
 555
(1 row)
```

`REGEXP_SUBSTR` returns `555`, the contents of the second substring.

10.1.3.6 Pattern Matching Using the LIKE Operator

Advanced Server provides pattern matching using the traditional SQL `LIKE` operator. The syntax for the `LIKE` operator is as follows.

```
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` does not contain percent signs or underscore, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (`_`) in `pattern` stands for (matches) any single character; a percent sign (`%`) matches any string of zero or more characters.

Some examples:

```
 'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

`LIKE` pattern matches always cover the entire string. To match a pattern anywhere within a string, the pattern must therefore start and end with a percent sign.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

To match a literal underscore or percent sign without matching other characters, the respective character in `pattern` must be preceded by the escape character. The default escape character is the backslash but a different one may be selected by using the `ESCAPE` clause. To match the escape character itself, write two escape characters.

Note that 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 actually 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 is not special to `LIKE` anymore. (But it is 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 effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

10.1.3.7 Data Type Formatting Functions

The Advanced Server formatting functions described in the following table 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 functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a string template that defines the output or input format.

Function	Return Type	Description	Example	Result
`TO_CHAR(DATE [, format ])`	`VARCHAR2`	Convert a date/time to a string with output, `format`. If omitted default format is DD-MON-YY.	`TO_CHAR(SYSDATE, 'MM/DD/YYYY HH12:MI:SS AM')`	`07/25/2007 09:43:02 AM`
`TO_CHAR(TIMESTAMP [, format ])`	`VARCHAR2`	Convert a timestamp to a string with output, `format`. If omitted default format is DD-MON-YY.	`TO_CHAR (CURRENT_TIMESTAMP, 'MM/DD/YYYY HH12:MI:SS AM')`	`08/13/2015 08:55:22 PM`
`TO_CHAR(INTEGER [, format ])`	`VARCHAR2`	Convert an integer to a string with output, `format`	`TO_CHAR(2412, '999,999S')`	`2,412+`
`TO_CHAR(NUMBER [, format ])`	`VARCHAR2`	Convert a decimal number to a string with output, `format`	`TO_CHAR(10125.35, '999,999.99')`	`10,125.35`
`TO_CHAR(DOUBLE PRECISION, format)`	`VARCHAR2`	Convert a floating-point number to a string with output, `format`	`TO_CHAR (CAST(123.5282 AS REAL), '999.99')`	`123.53`
`TO_DATE(string [, format ])`	`TIMESTAMP`	Convert a date or timestamp formatted string to a `TIMESTAMP` data type	`TO_DATE('2007-07-04 13:39:10', 'YYYY-MM-DD HH24:MI:SS')` `TO_DATE('2007-07-04','YYYY-MM-DD')`	`04-JUL-07 13:39:10` `04-JUL-07 00:00:00`
`TO_NUMBER(string [, format ])`	`NUMBER`	Convert a number formatted string to a `NUMBER` data type	`TO_NUMBER('2,412-', '999,999S')`	`-2412`
`TO_TIMESTAMP(string, format)`	`TIMESTAMPTZ`	Convert a timestamp formatted string to a `TIMESTAMPTZ` data type	`TO_TIMESTAMP('05 Dec 2000 08:30:25 pm', 'DD Mon YYYY hh12:mi:ss pm')`	`05-DEC-00 20:30:25 +05:30`

In an output template string (for `TO_CHAR`), there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for anything but `TO_CHAR`), template patterns identify the parts of the input data string to be looked at and the values to be found there.

The following table shows the template patterns available for formatting date values using the `TO_CHAR`, `TO_DATE`, and `TO_TIMESTAMP` functions.

Pattern	Description

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Pattern	Description
`HH`	Hour of day (01-12)
`HH12`	Hour of day (01-12)
`HH24`	Hour of day (00-23)
`MI`	Minute (00-59)
`SS`	Second (00-59)
`SSSSS`	Seconds past midnight (0-86399)
`FFn`	Fractional seconds where `n` is an optional integer from 1 to 9 for the number of digits to return. If omitted, the default is 6.
`AM` or `A.M.` or `PM` or `P.M.`	Meridian indicator (uppercase)
`am` or `a.m.` or `pm` or `p.m.`	Meridian indicator (lowercase)
`Y,YYY`	Year (4 and more digits) with comma
`YEAR`	Year (spelled out)
`SYEAR`	Year (spelled out) (BC dates prefixed by a minus sign)
`YYYY`	Year (4 and more digits)
`SYYYY`	Year (4 and more digits) (BC dates prefixed by a minus sign)
`YYY`	Last 3 digits of year
`YY`	Last 2 digits of year
`Y`	Last digit of year
`IYYY`	ISO year (4 and more digits)
`IYY`	Last 3 digits of ISO year
`IY`	Last 2 digits of ISO year
`I`	Last 1 digit of ISO year
`BC` or `B.C.` or `AD` or `A.D.`	Era indicator (uppercase)
`bc` or `b.c.` or `ad` or `a.d.`	Era indicator (lowercase)
`MONTH`	Full uppercase month name

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Pattern	Description
`Month`	Full mixed-case month name
`month`	Full lowercase month name
`MON`	Abbreviated uppercase month name (3 chars in English, localized lengths vary)
`Mon`	Abbreviated mixed-case month name (3 chars in English, localized lengths vary)
`mon`	Abbreviated lowercase month name (3 chars in English, localized lengths vary)
`MM`	Month number (01-12)
`DAY`	Full uppercase day name
`Day`	Full mixed-case day name
`day`	Full lowercase day name
`DY`	Abbreviated uppercase day name (3 chars in English, localized lengths vary)
`Dy`	Abbreviated mixed-case day name (3 chars in English, localized lengths vary)
`dy`	Abbreviated lowercase day name (3 chars in English, localized lengths vary)
`DDD`	Day of year (001-366)
`DD`	Day of month (01-31)
`D`	Day of week (1-7; Sunday is 1)
`W`	Week of month (1-5) (The first week starts on the first day of the month)
`WW`	Week number of year (1-53) (The first week starts on the first day of the year)
`IW`	ISO week number of year; the first Thursday of the new year is in week 1
`CC`	Century (2 digits); the 21st century starts on 2001-01-01
`SCC`	Same as CC except BC dates are prefixed by a minus sign
`J`	Julian Day (days since January 1, 4712 BC)
`Q`	Quarter
`RM`	Month in Roman numerals (I-XII; I=January) (uppercase)
`rm`	Month in Roman numerals (i-xii; i=January) (lowercase)
`RR`	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 will be the same as the first 2 digits of the current year with the following exceptions: If the given 2-digit year is <50 and the last 2 digits of the current year is >= 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 >= 50 and the last 2 digits of the current year is <50, then the first 2 digits for the given year is 1 less than the first 2 digits of the current year.
`RRRR`	Only affects `TO_DATE` 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.

Certain modifiers may be applied to any template pattern to alter its behavior. For example, `FMMonth` is the `Month` pattern with the `FM` modifier. The following table shows the modifier patterns for date/time formatting.

Modifier	Description	Example
`FM` prefix	Fill mode (suppress padding blanks and zeros)	`FMMonth`
`TH` suffix	Uppercase ordinal number suffix	`DDTH`
`th` suffix	Lowercase ordinal number suffix	`DDth`
`FX` prefix	Fixed format global option (see usage notes)	`FX Month DD Day`
`SP` suffix	Spell mode	`DDSP`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Usage notes for date/time formatting:

- `FM` suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern fixed-width.
- `TO_TIMESTAMP` and `TO_DATE` skip multiple blank spaces in the input string if the `FX` option is not used. `FX` must be specified as the first item in the template. For example:

```
 TO_TIMESTAMP('2000 - JUN', 'YYYY-MON') is correct, but
TO_TIMESTAMP('2000    JUN', 'FXYYYY MON') returns an error
because TO_TIMESTAMP expects one space only.
```

- Ordinary text is allowed in `TO_CHAR` templates and will be output literally.
- In conversions from string to `timestamp` or `date`, the `CC` field is ignored if there is a `YYY`, `YYYY` or `Y,YYY` field. If `CC` is used with `YY` or `Y` then the year is computed as `(CC-1)*100+YY`.

The following table shows the template patterns available for formatting numeric values.

Pattern	Description
`9`	Value with the specified number of digits
`0`	Value with leading zeroes
`. (period)`	Decimal point
`, (comma)`	Group (thousand) separator
`$`	Dollar sign
`PR`	Negative value in angle brackets
`S`	Sign anchored to number (uses locale)
`L`	Currency symbol (uses locale)
`D`	Decimal point (uses locale)
`G`	Group separator (uses locale)
`MI`	Minus sign specified in righ-most position (if number < 0)
`RN` or `rn`	Roman numeral (input between 1 and 3999)
`V`	Shift specified number of digits (see notes)

Usage notes for numeric formatting:

- `9` results in a value with the same number of digits as there are `9s`. If a digit is not available it ouputs a space.
- `TH` does not convert values less than zero and does not convert fractional numbers.

`V` effectively multiplies the input values by 10^n, where `n` is the number of digits following `V`. `TO_CHAR` does not support the use of `V` combined with a decimal point. (E.g., `99.9V99` is not allowed.)

The following table shows some examples of the use of the `TO_CHAR` and `TO_DATE` functions:

Expression	Result
`TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD  HH12:MI:SS')`	`'Tuesday  , 06  05:39:18'`
`TO_CHAR(CURRENT_TIMESTAMP, 'FM-Day, FMDD  HH12:MI:SS')`	`'Tuesday, 6  05:39:18'`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Expression	Result
`TO_CHAR(-0.1, '99.99')`	`'  -.10'`
`TO_CHAR(-0.1, 'FM9.99')`	`'-.1'`
`TO_CHAR(0.1, '0.9')`	`' 0.1'`
`TO_CHAR(12, '9990999.9')`	`'    0012.0'`
`TO_CHAR(12, 'FM9990999.9')`	`'0012.'`
`TO_CHAR(485, '999')`	`' 485'`
`TO_CHAR(-485, '999')`	`'-485'`
`TO_CHAR(1485, '9,999')`	`' 1,485'`
`TO_CHAR(1485, '9G999')`	`' 1,485'`
`TO_CHAR(148.5, '999.999')`	`' 148.500'`
`TO_CHAR(148.5, 'FM999.999')`	`'148.5'`
`TO_CHAR(148.5, 'FM999.990')`	`'148.500'`
`TO_CHAR(148.5, '999D999')`	`' 148.500'`
`TO_CHAR(3148.5, '9G999D999')`	`' 3,148.500'`
`TO_CHAR(-485, '999S')`	`'485-'`
`TO_CHAR(-485, '999MI')`	`'485-'`
`TO_CHAR(485, '999MI')`	`'485 '`
`TO_CHAR(485, 'FM999MI')`	`'485'`
`TO_CHAR(-485, '999PR')`	`'<485>'`
`TO_CHAR(485, 'L999')`	`'$ 485'`
`TO_CHAR(485, 'RN')`	`'        CDLXXXV'`
`TO_CHAR(485, 'FMRN')`	`'CDLXXXV'`
`TO_CHAR(5.2, 'FMRN')`	`'V'`
`TO_CHAR(12, '99V999')`	`' 12000'`
`TO_CHAR(12.4, '99V999')`	`' 12400'`
`TO_CHAR(12.45, '99V9')`	`' 125'`

IMMUTABLE TO_CHAR(TIMESTAMP, format) Function

There are certain cases of the `TO_CHAR` function that can result in usage of an `IMMUTABLE` form of the function. Basically, a function is `IMMUTABLE` if the function does not 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, etc. do not affect the results returned by the function.

For more information about function volatility categories `VOLATILE`, `STABLE`, and `IMMUTABLE`, see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/11/static/xfunc-volatility.html

A particular advantage of an `IMMUTABLE` function is that it can be used in the `CREATE INDEX` command to create an index based on that function.

In order for the `TO_CHAR` function to use the `IMMUTABLE` form the following conditions must be satisfied:

- The first parameter of the `TO_CHAR` function must be of data type `TIMESTAMP`.
- The format specified in the second parameter of the `TO_CHAR` function must not affect the return value of the function based on factors such as language, locale, etc. For example a format of `'YYYY-MM-DD HH24:MI:SS'` can be used for an `IMMUTABLE` form of the function since,

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

regardless of locale settings, the result of the function is the date and time expressed solely in numeric form. However, a format of `'DD-MON-YYYY'` cannot be used for an `IMMUTABLE` form of the function because the 3-character abbreviation of the month may 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 `(MONTH, MON)`, days `(DAY, DY)`, median indicators `(AM, PM)`, or era indicators `(BC, AD)`.

For the following example, a table with a `TIMESTAMP` column is created.

```
CREATE TABLE ts_tbl (ts_col TIMESTAMP);
```

The following shows the successful creation of an index with the `IMMUTABLE` form of the `TO_CHAR` function.

```
 edb=# CREATE INDEX ts_idx ON ts_tbl (TO_CHAR(ts_col,'YYYY-MM-DD HH24:MI:SS'));
CREATE INDEX
edb=# \dS ts_idx
                                  Index "public.ts_idx"
 Column  |       Type        |                        Definition
---------+-------------------+-----------------------------------------------------------
 to_char | character varying | to_char(ts_col, 'YYYY-MM-DD HH24:MI:SS'::character varying)
btree, for table "public.ts_tbl"
```

The following results in an error because the format specified in the `TO_CHAR` function prevents the use of the `IMMUTABLE` form since the 3-character month abbreviation, `MON`, may result in different return values based on the locale setting.

```
 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
```

10.1.3.8 Date/Time Functions and Operators

The Date/Time Functions table shows the available functions for date/time value processing, with details appearing in the following subsections. The following table illustrates the behaviors of the basic arithmetic operators `(+, -)`. For formatting functions, refer to IMMUTABLE TO_CHAR(TIMESTAMP, format) Function. You should be familiar with the background information on date/time data types, see Date/Time Types.

Operator	Example	Result
plus `(+)`	`DATE '2001-09-28' + 7`	`05-OCT-01 00:00:00`
plus `(+)`	`TIMESTAMP '2001-09-28 13:30:00' + 3`	`01-OCT-01 13:30:00`
minus `(-)`	`DATE '2001-10-01' - 7`	`24-SEP-01 00:00:00`
minus `(-)`	`TIMESTAMP '2001-09-28 13:30:00' - 3`	`25-SEP-01 13:30:00`
minus `(-)`	`TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00'`	`@ 1 day 15 hours`

In the date/time functions of the following table the use of the `DATE` and `TIMESTAMP` data types are interchangeable.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function	Return Type	Description	Example	Result
`ADD_MONTHS (DATE, NUMBER)`	`DATE`	Add months to a date	`ADD_MONTHS ('28-FEB-97', ,3.8)`	`31-MAY-97 00:00:00`
`CURRENT_DATE`	`DATE`	Current date	`CURRENT_DATE`	`04-JUL-07`
`CURRENT_TIMESTAMP`	`TIMESTAMP`	Returns the current date and time	`CURRENT_TIMESTAMP`	`04-JUL-07 15:33:23.484`
`EXTRACT(field FROM TIMESTAMP)`	`DOUBLE PRECISION`	Get subfield	`EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40')`	`20`
`LAST_DAY(DATE)`	`DATE`	Returns the last day of the month represented by the given date. If the given date contains a time portion, it is carried forward to the result unchanged	`LAST_DAY('14-APR-98')`	`30-APR-98 00:00:00`
`LOCALTIMESTAMP [ (precision) ]`	`TIMESTAMP`	Current date and time (start of current transaction)	`LOCALTIMESTAMP`	`04-JUL-07 15:33:23.484`
`MONTHS_BETWEEN (DATE, DATE)`	`NUMBER`	Number of months between two dates	`MONTHS_BETWEEN ('28-FEB-07', '30-NOV-06')`	`3`
`NEXT_DAY(DATE, dayofweek)`	`DATE`	Date falling on `dayofweek` following specified date	`NEXT_DAY('16-APR-07',FRI' )`	`20-APR-07 00:00:00`
`NEW_TIME(DATE, VARCHAR, VARCHAR)`	`DATE`	Converts a date and time to an alternate time zone	`NEW_TIME (TO_DATE '2005/05/29 01:45', 'AST', 'PST')`	`2005/05/29 21:45:00`
`NUMTODSINTERVAL(NUMBER, INTERVAL)`	`INTERVAL`	Converts a number to a specified day or second interval	`SELECT numtodsinterval(100, ‘hour’);`	`4 days 04:00:00`
`NUMTOYMINTERVAL(NUMBER, INTERVAL)`	`INTERVAL`	Converts a number to a specified year or month interval	`SELECT numtoyminterval(100, ‘month’);`	`8 years 4 mons`
`ROUND(DATE [, format ])`	`DATE`	Date rounded according to `format`	`ROUND(TO_DATE('29-MAY-05'),'MON')`	`01-JUN-05 00:00:00`
`SYS_EXTRACT_UTC(TIMESTAMP WITH TIME ZONE)`	`TIMESTAMP`	Returns Coordinated Universal Time	`SYS_EXTRACT_UTC(CAST ('24-MAR-11 12:30:00PM -04:00' AS TIMESTAMP WITH TIME ZONE))`	`24-MAR-11 16:30:00`
`SYSDATE`	`DATE`	Returns current date and time	`SYSDATE`	`01-AUG-12 11:12:34`
`SYSTIMESTAMP()`	`TIMESTAMP`	Returns current date and time	`SYSTIMESTAMP`	`01-AUG-12 11:11:23.665229 -07:00`
`TRUNC(DATE [format])`	`DATE`	Truncate according to `format`	`TRUNC(TO_DATE ('29-MAY-05'), 'MON')`	`01-MAY-05 00:00:00`

ADD_MONTHS

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `ADD_MONTHS` functions adds (or subtracts if the second parameter is negative) the specified number of months to the given date. 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 which 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 is carried forward to the result unchanged.

The following are examples of the `ADD_MONTHS` function.

```
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)
```

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

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
     2001
(1 row)
```

`MONTH`

The number of the month within the year (1 - 12)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
-----------
       2
(1 row)
```

`DAY`

The day (of the month) field (1 - 31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
       16
(1 row)
```

`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 which is positive if the first date is greater than the second date or negative if the first date is less than the second date.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

The following are some examples of 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. At least the first three letters of the weekday must be specified - e.g., `SAT`. If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the `NEXT_DAY` function.

```
SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'SUNDAY') FROM DUAL;

      next_day
--------------------
 19-AUG-07 00:00:00
(1 row)

SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'MON') FROM DUAL;

      next_day
--------------------
 20-AUG-07 00:00:00
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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:

Time Zone	Offset from UTC	Description
`AST`	UTC+4	Atlantic Standard Time
`ADT`	UTC+3	Atlantic Daylight Time
`BST`	UTC+11	Bering Standard Time
`BDT`	UTC+10	Bering Daylight Time
`CST`	UTC+6	Central Standard Time
`CDT`	UTC+5	Central Daylight Time
`EST`	UTC+5	Eastern Standard Time
`EDT`	UTC+4	Eastern Daylight Time
`GMT`	UTC	Greenwich Mean Time
`HST`	UTC+10	Alaska-Hawaii Standard Time
`HDT`	UTC+9	Alaska-Hawaii Daylight Time
`MST`	UTC+7	Mountain Standard Time
`MDT`	UTC+6	Mountain Daylight Time
`NST`	UTC+3:30	Newfoundland Standard Time
`PST`	UTC+8	Pacific Standard Time
`PDT`	UTC+7	Pacific Daylight Time
`YST`	UTC+9	Yukon Standard Time
`YDT`	UTC+8	Yukon Daylight Time

Following is an example of the `NEW_TIME` function:

```
 SELECT NEW_TIME(TO_DATE('08-13-07 10:35:15','MM-DD-YY HH24:MI:SS'),'AST',
'PST') "Pacific Standard Time" FROM DUAL;

Pacific Standard Time
---------------------
 13-AUG-07 06:35:15
(1 row)
```

ROUND

The `ROUND` function returns a date rounded according to a specified template pattern. If the template pattern is omitted, the date is rounded to the nearest day. The following table shows the template patterns for the `ROUND` function.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Pattern	Description
`CC, SCC`	Returns January 1, `cc` 01 where `cc` is first 2 digits of the given year if last 2 digits <= 50, or 1 greater than the first 2 digits of the given year if last 2 digits > 50; (for AD years)
`SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y`	Returns January 1, `yyyy` where `yyyy` is rounded to the nearest year; rounds down on June 30, rounds up on July 1
`IYYY, IYY, IY, I`	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
`Q`	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
`MONTH, MON, MM, RM`	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
`WW`	Round to the nearest date that corresponds to the same day of the week as the first day of the year
`IW`	Round to the nearest date that corresponds to the same day of the week as the first day of the ISO year
`W`	Round to the nearest date that corresponds to the same day of the week as the first day of the month
`DDD, DD, J`	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
`DAY, DY, D`	Rounds to the nearest Sunday
`HH, HH12, HH24`	Round to the nearest hour
`MI`	Round to the nearest minute

Following are examples of usage of the `ROUND` function.

The following examples round to the nearest hundred years.

```
SELECT TO_CHAR(ROUND(TO_DATE('1950','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;

   Century
-------------
 01-JAN-1901
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century" FROM DUAL;

   Century
-------------
 01-JAN-2001
(1 row)
```

The following examples round to the nearest year.

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM DUAL;

    Year
-------------
 01-JAN-1999
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY') "Year" FROM DUAL;

    Year
-------------
 01-JAN-2000
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(1 row)
```

The following examples round to the nearest ISO year. The first example rounds to 2004 and the ISO year for 2004 begins on December 29th of 2003. The second example rounds to 2005 and the ISO year for 2005 begins on January 3rd of that same year.

(An ISO year begins on the first Monday from which a 7 day span, Monday thru Sunday, contains at least 4 days of the new year. Thus, it is possible for the beginning of an ISO year to start in December of the prior year.)

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year" FROM DUAL;

  ISO Year
-------------
 29-DEC-2003
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-YYYY') "ISO Year" FROM DUAL;

  ISO Year
-------------
 03-JAN-2005
(1 row)
```

The following example round to the nearest quarter:

```
SELECT ROUND(TO_DATE('15-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-APR-07 00:00:00
(1 row)
```

The following example round to the nearest month:

```
SELECT ROUND(TO_DATE('15-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
 01-DEC-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
 01-JAN-08 00:00:00
(1 row)
```

The following examples round to the nearest week. The first day of 2007 lands on a Monday so in the first example, January 18th is closest to the Monday that lands on January 15th. In the second example, January 19th is closer to the Monday that falls on January 22nd.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SELECT ROUND(TO_DATE('18-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 15-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 22-JAN-07 00:00:00
(1 row)
```

The following 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.

```
SELECT ROUND(TO_DATE('01-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 29-DEC-03 00:00:00
(1 row)

SELECT ROUND(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 05-JAN-04 00:00:00
(1 row)
```

The following examples round to the nearest week where a week is considered to start on the same day as the first day of the month.

```
SELECT ROUND(TO_DATE('05-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
 08-MAR-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
 01-MAR-07 00:00:00
(1 row)
```

The following examples round to the nearest day.

```
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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        Day
--------------------
 05-AUG-07 00:00:00
(1 row)
```

The following examples round to the start of the nearest day of the week (Sunday).

```
 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)
```

The following examples round to the nearest hour.

```
 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 08:00:00
(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 09:00:00
(1 row)
```

The following examples round to the nearest minute.

```
 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 08:30:00
(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 08:31:00
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

TRUNC

The `TRUNC` function returns a date truncated according to a specified template pattern. If the template pattern is omitted, the date is truncated to the nearest day. The following table shows the template patterns for the `TRUNC` function.

Pattern	Description
`CC, SCC`	Returns January 1, `cc` 01 where `cc` is first 2 digits of the given year
`SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y`	Returns January 1, `yyyy` where `yyyy` is the given year
`IYYY, IYY, IY, I`	Returns the start date of the ISO year containing the given date
`Q`	Returns the first day of the quarter containing the given date
`MONTH, MON, MM, RM`	Returns the first day of the specified month
`WW`	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
`IW`	Returns the start of the ISO week containing the given date
`W`	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 month
`DDD, DD, J`	Returns the start of the day for the given date
`DAY, DY, D`	Returns the start of the week (Sunday) containing the given date
`HH, HH12, HH24`	Returns the start of the hour
`MI`	Returns the start of the minute

Following are examples of usage of the `TRUNC` function.

The following 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)
```

The following 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)
```

The following 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
-------------
 29-DEC-2003
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example truncates down to the start date of the quarter.

```
SELECT TRUNC(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-JAN-07 00:00:00
(1 row)
```

The following example truncates to the start of the month.

```
SELECT TRUNC(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
 01-DEC-07 00:00:00
(1 row)
```

The following 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 19th is January 15th.

```
SELECT TRUNC(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 15-JAN-07 00:00:00
(1 row)
```

The following example truncates to the start of an ISO week. An ISO week begins on a Monday. January 2, 2004 falls in the ISO week that starts on Monday, December 29, 2003.

```
SELECT TRUNC(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 29-DEC-03 00:00:00
(1 row)
```

The following 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;

        Week
--------------------
 15-MAR-07 00:00:00
(1 row)
```

The following 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;

        Day
--------------------
 04-AUG-07 00:00:00
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following 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;

    Day of Week
--------------------
 05-AUG-07 00:00:00
(1 row)
```

The following 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'),'DD-MON-YY HH24:MI:SS') "Hour"
FROM DUAL;

        Hour
--------------------
 09-AUG-07 08:00:00
(1 row)
```

The following example truncates to the minute.

```
SELECT TO_CHAR(TRUNC(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 08:30:00
(1 row)
```

CURRENT DATE/TIME

Advanced Server provides a number of functions that return values related to the current date and time. These functions all 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` will not change if called multiple times within 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 will return the same value for each occurrence within the statement. If called from multiple statements within a transaction, may return different values for each occurrence. If called from a function, may return a different value than the value returned by current_timestamp in the caller.

```
SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP FROM DUAL;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
               current_timestamp | current_timestamp
---------------------------------+---------------------------------
 02-SEP-13 17:52:29.261473 +05:00 | 02-SEP-13 17:52:29.261474 +05:00
```

`LOCALTIMESTAMP` can optionally be given a precision parameter which causes the result to be rounded to that many 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
(1 row)

SELECT LOCALTIMESTAMP(2) FROM DUAL;

       timestamp
-----------------------
 06-AUG-07 16:11:44.58
(1 row)
```

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the “current” time, so that multiple modifications within the same transaction bear the same time stamp. Other database systems may advance these values more frequently.

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 be included in the result set. The valid interval types are `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

The following example converts a numeric value to a time interval that includes days and hours:

```
SELECT numtodsinterval(100, ‘hour’);
numtodsinterval
---------------
4 days 04:00:00
(1 row)
```

The following example converts a numeric value to a time interval that includes minutes and seconds:

```
SELECT numtodsinterval(100, ‘second’);
numtodsinterval
---------------
1 min 40 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 be included in the result set. The valid interval types are `YEAR` and `MONTH`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example converts a numeric value to a time interval that includes years and months:

```
 SELECT numtoyminterval(100, 'month');
numtoyminterval
---------------
8 years 4 mons
(1 row)
```

The following example converts a numeric value to a time interval that includes years only:

```
 SELECT numtoyminterval(100, 'year');
numtoyminterval
---------------
100 years
(1 row)
```

10.1.3.9 Sequence Manipulation Functions

This section describes Advanced Server's functions for operating on sequence objects. Sequence objects (also called sequence generators or just sequences) are special single-row tables created with the `CREATE SEQUENCE` command. A sequence object is usually used to generate unique identifiers for rows of a table. The sequence functions, listed below, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

```
 sequence.NEXTVAL
sequence.CURRVAL
```

`sequence` is the identifier assigned to the sequence in the `CREATE SEQUENCE` command. The following describes the usage of these functions.

`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 will safely receive a distinct sequence value.

`CURRVAL`

Return the value most recently obtained by `NEXTVAL` for this sequence in the current session. (An error is reported if `NEXTVAL` has never been called for this sequence in this session.) Notice that because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed `NEXTVAL` since the current session did.

If a sequence object has been created with default parameters, `NEXTVAL` calls on it will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the `CREATE SEQUENCE` command.

Important: To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a `NEXTVAL` operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the `NEXTVAL` later aborts. This means that aborted transactions may leave unused "holes" in the sequence of assigned values.

10.1.3.10 Conditional Expressions

The following section describes the SQL-compliant conditional expressions available in Advanced Server.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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
```

`CASE` clauses can be used 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`.

An example:

```
SELECT * FROM test;

 a
---
 1
 2
 3
(3 rows)

SELECT a,
    CASE WHEN a=1 THEN 'one'
         WHEN a=2 THEN 'two'
         ELSE 'other'
    END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

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 above:

```
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.

The example above can be written using the simple `CASE` syntax:

```
SELECT a,
    CASE a WHEN 1 THEN 'one'
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
          WHEN 2 THEN 'two'
          ELSE 'other'
    END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

A `CASE` expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

```
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 is not null. Null is returned only if all arguments are null.

```
COALESCE(value [, value2 ] ... )
```

It is often used to substitute a default value for null values when data is retrieved for display or further computation. For example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Like a `CASE` expression, `COALESCE` will not evaluate arguments that are not needed to determine the result; that is, arguments to the right of the first non-null argument are not 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`.

```
NULLIF(value1, value2)
```

This can be used to perform the inverse operation of the `COALESCE` example given above:

```
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 is not null. `NVL` evaluates the first expression; if that expression evaluates to `NULL`, `NVL`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

returns the second expression.

```
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 are not coercible to the common data type then `NVL` throws an error.

Examples

1. The following example computes a bonus for non-commissioned employees, If an employee is a commissioned employee, this expression returns the employees commission; if the employee is not a commissioned employee (that is, his commission is `NULL`), this expression returns a bonus that is 10% of his salary.

   ```
   bonus = NVL(emp.commission, emp.salary * .10)
   ```

2. In the following example, the type of 1 is numeric and the type of '' is considered as unknown. Therefore PostgreSQL decides that the common type ought to be numeric and tries to interpret the empty string as a numeric value, which produces the indicated error.

   ```
   edb=# select nvl('',1);
   ERROR:  invalid input syntax for type numeric:
   ""
   ```

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 is not `NULL`, `NVL2` returns the value in `expr2`; if the first expression is `NULL`, `NVL2` returns the value in `expr3`.

```
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).

The following example computes a bonus for commissioned employees - if a given employee is a commissioned employee, this expression returns an amount equal to 110% of his commission; if the employee is not a commissioned employee (that is, his commission is `NULL`), this expression returns `0`.

```
bonus = NVL2(emp.commission, emp.commission * 1.1, 0)
```

GREATEST and LEAST

The `GREATEST` and `LEAST` functions select the largest or smallest value from a list of any number of expressions.

```
GREATEST(value [, value2 ] ... )
LEAST(value [, value2 ] ... )
```

The expressions must all be convertible to a common data type, which will be the type of the result. Null values in the list are ignored. The result will be null only if all the expressions evaluate to null.

> **Note**

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> The `GREATEST` and `LEAST` are not in the SQL standard, but are a common extension.

10.1.3.11 Aggregate Functions

`Aggregate` functions compute a single result value from a set of input values. The built-in aggregate functions are listed in the following tables.

Function	Argument Type	Return Type	Description
`AVG (expression)`	`INTEGER, REAL, DOUBLE PRECISION, NUMBER`	`NUMBER` for any integer type, `DOUBLE PRECISION` for a floating-point argument, otherwise the same as the argument data type	The average (arithmetic mean) of all input values
`COUNT(*)`		`BIGINT`	Number of input rows
`COUNT (expression)`	Any	`BIGINT`	Number of input rows for which the value of expression is not null
`MAX (expression)`	Any numeric, string, date/time, or bytea type	Same as argument type	Maximum value of expression across all input values
`MIN (expression)`	Any numeric, string, date/time, or bytea type	Same as argument type	Minimum value of expression across all input values
`SUM (expression)`	`INTEGER, REAL, DOUBLE PRECISION, NUMBER`	`BIGINT` for `SMALLINT` or `INTEGER` arguments, `NUMBER` for `BIGINT` arguments, `DOUBLE PRECISION` for floating-point arguments, otherwise the same as the argument data type	Sum of expression across all input values

It should be noted that except for `COUNT` , these functions return a null value when no rows are selected. In particular, `SUM` of no rows returns null, not zero as one might expect. The `COALESCE` function may be used to substitute zero for null when necessary.

The following table shows the aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) 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.

Function	Argument Type	Return Type	Description
`CORR(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Correlation coefficient
`COVAR_POP(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Population covariance
`COVAR_SAMP(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Sample covariance
`REGR_AVGX(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Average of the independent variable (sum `(X) / N`)
`REGR_AVGY(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Average of the dependent variable (sum `(Y) / N`)
`REGR_COUNT(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Number of input rows in which both expressions are nonnull
`REGR_INTERCEPT(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	y-intercept of the least-squares-fit linear equation determined by the `(X, Y)` pairs

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function	Argument Type	Return Type	Description
`REGR_R2(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Square of the correlation coefficient
`REGR_SLOPE(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Slope of the least-squares-fit linear equation determined by the `(X, Y)` pairs
`REGR_SXX(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Sum (X^2) – sum (X) ^{2} / N ("sum of squares" of the independent variable)
`REGR_SXY(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Sum (X*Y) – sum (X) * sum (Y) / N ("sum of products" of independent times dependent variable)
`REGR_SYY(Y, X)`	`DOUBLE PRECISION`	`DOUBLE PRECISION`	Sum (Y^2) – sum (Y) ^{2} / N ("sum of squares" of the dependent variable)
`STDDEV(expression)`	`INTEGER, REAL, DOUBLE PRECISION, NUMBER`	`DOUBLE PRECISION` for floating-point arguments, otherwise `NUMBER`	Historic alias for `STDDEV_SAMP`
`STDDEV_POP(expression)`	`INTEGER, REAL, DOUBLE PRECISION, NUMBER`	`DOUBLE PRECISION` for floating-point arguments, otherwise `NUMBER`	Population standard deviation of the input values
`STDDEV_SAMP(expression)`	`INTEGER, REAL, DOUBLE PRECISION, NUMBER`	`DOUBLE PRECISION` for floating-point arguments, otherwise `NUMBER`	Sample standard deviation of the input values
`VARIANCE(expression)`	`INTEGER, REAL, DOUBLE PRECISION, NUMBER`	`DOUBLE PRECISION` for floating-point arguments, otherwise `NUMBER`	Historical alias for `VAR_SAMP`
`VAR_POP(expression)`	`INTEGER, REAL, DOUBLE PRECISION, NUMBER`	`DOUBLE PRECISION` for floating-point arguments, otherwise `NUMBER`	Population variance of the input values (square of the population standard deviation)
`VAR_SAMP(expression)`	`INTEGER, REAL, DOUBLE PRECISION, NUMBER`	`DOUBLE PRECISION` for floating-point arguments, otherwise `NUMBER`	Sample variance of the input values (square of the sample standard deviation)

10.1.3.12 Subquery Expressions

This section describes the SQL-compliant subquery expressions available in Advanced Server. All of the expression forms documented in this section 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`.

```
EXISTS(subquery)
```

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed far enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has any side effects (such as calling sequence functions); whether the side effects occur or not may 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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:

```
 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).

```
expression IN (subquery)
```

Note that 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 will be `NULL`, not `FALSE`. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, it's unwise to assume that the subquery will be evaluated completely.

NOT 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 `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.

```
expression NOT IN (subquery)
```

Note that 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 will be `NULL`, not `TRUE`. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, it's unwise to assume that the subquery will be evaluated 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).

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

`SOME` is a synonym for `ANY`. `IN` is equivalent to `= ANY`.

Note that 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 will be `NULL`, not `FALSE`. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, it's unwise to assume that the subquery will be evaluated 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 does not return `FALSE` for any row, and it returns `NULL` for at least one row.

```
expression operator ALL (subquery)
```

`NOT IN` is equivalent to `<> ALL`. As with `EXISTS`, it's unwise to assume that the subquery will be evaluated completely.

10.2 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 are writing an application that depends on information stored in the system tables, it would be prudent to 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 is provided for compatibility with Oracle databases only.

Column	Type	Modifiers	Description
`dummy`	`VARCHAR2(1)`		Provided for compatibility only.

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 within the file system. For example, the `UTL_FILE` package offers functions that permit a user to read and write files and directories in the host file system, but only allows access to paths that the database administrator has granted access to via a `CREATE DIRECTORY` command.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	Modifiers	Description
dirname	"name"	not null	The name of the alias.
dirowner	oid	not null	The OID of the user that owns the alias.
dirpath	text		The directory name to which access is granted.
diracl	aclitem[]		The access control list that determines which users may access the alias.

edb_password_history

The edb_password_history table contains one row for each password change. The table is shared across all databases within a cluster.

Column	Type	References	Description
passhistroleid	oid	pg_authid.oid	The ID of a role.
passhistpassword	text		Role password in md5 encrypted form.
passhistpasswordsetat	timestamptz		The time the password was set.

edb_policy

The edb_policy table contains one row for each policy.

Column	Type	Modifiers	Description
policyname	name	not null	The policy name.
policygroup	oid	not null	Currently unused.
policyobject	oid	not null	The OID of the table secured by this policy (the object_schema plus the object_name).
policykind	char	not null	The kind of object secured by this policy: 'r' for a table, 'v' for a view, = for a synonym. Currently always 'r'.
policyproc	oid	not null	The OID of the policy function (function_schema plus policy_function).
policyinsert	boolean	not null	True if the policy is enforced by INSERT statements.
policyselect	boolean	not null	True if the policy is enforced by SELECT statements.
policydelete	boolean	not null	True if the policy is enforced by DELETE statements.
policyupdate	boolean	not null	True if the policy is enforced by UPDATE statements.
policyindex	boolean	not null	Currently unused.
policyenabled	boolean	not null	True if the policy is enabled.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	Modifiers	Description
policyupdatecheck	boolean	not null	True if rows updated by an UPDATE statement must satisfy the policy.
policystatic	boolean	not null	Currently unused.
policytype	integer	not null	Currently unused.
policyopts	integer	not null	Currently unused.
policyseccols	int2vector	not null	The column numbers for columns listed in sec_relevant_cols.

edb_profile

The edb_profile table stores information about the available profiles. edb_profiles is shared across all databases within a cluster.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected).
prfname	name		The name of the profile.
prffailedloginattempts	integer		The number of failed login attempts allowed by the profile. -1 indicates that the value from the default profile should be used. -2 indicates no limit on failed login attempts.
prfpasswordlocktime	integer		The password lock time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the account should be locked permanently.
prfpasswordlifetime	integer		The password life time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
prfpasswordgracetime	integer		The password grace time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
prfpasswordreusetime	integer		The number of seconds that a user must wait before reusing a password. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordreusemax	integer		The number of password changes that have to occur before a password can be reused. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordallowhashed	integer		The password allow hashed parameter specifies whether an encrypted password to be allowed for use or not. The possible values can be true/on/yes/1, false/off/no/0, and DEFAULT.
prfpasswordverifyfuncdb	oid	pg_database.oid	The OID of the database in which the password verify function exists.
prfpasswordverifyfunc	oid	pg_proc.oid	The OID of the password verify function associated with the profile.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

edb_variable

The `edb_variable` table contains one row for each package level variable (each variable declared within a package).

Column	Type	Modifiers	Description
`varname`	`"name"`	`not null`	The name of the variable.
`varpackage`	`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 only visible within the package. Note: Public variables are declared within the package header; private variables are declared within the package body.
`varsrc`	`text`		Contains the source of the variable declaration, including any default value expressions for the variable.
`varseq`	`smallint`	`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.
`synobjschema`	`"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 at installation or run time to verify that features used by the application are available with this deployment.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

11 Database Compatibility for Oracle Developers Built-in Package Guide

Database Compatibility for Oracle means that an application runs in an Oracle environment as well as in the EDB Postgres Advanced Server (Advanced Server) environment with minimal or no changes to the application code. This guide focuses solely on the features that are related to the package support provided by Advanced Server.

11.1 Packages

This chapter discusses the concept of packages in Advanced Server. 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:

- Packages provide a convenient means of organizing the functions and procedures that perform a related purpose. Permission to use the package functions and procedures is dependent upon one privilege granted to the entire package. All of the package programs must be referenced with a common name.
- Certain functions, procedures, variables, types, etc. 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 - the program names, parameters if any, and return types of functions. The SPL code of these functions and procedures is not accessible to others, therefore applications that utilize a package are dependent only upon the information available in the signature – not in the procedural logic itself.
- Other functions, procedures, variables, types, etc. in the package can be declared as *private*. Private entities can be referenced and used by function and procedures within the package, but not by other external applications. Private entities are for use only by programs within the package.
- Function and procedure names can be overloaded within a package. One or more functions/procedures can be defined with the same name, but with different signatures. This provides the capability 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*: This is the public interface, (these are the elements which can be referenced outside the package). We declare all database objects that are to be a part of our package within the specification.
- The *package body*: This contains the actual implementation of all the database objects declared within the package specification.

The package body implements the specifications in the package specification. It contains implementation details and private declarations which 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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`

`package_name` is an identifier assigned to the package - each package must have a name unique within 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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`

`declaration` is an identifier of a public variable. A public variable can be accessed from outside of 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 CURSORs`
- 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 within the function or procedure body.

`IN | IN OUT | OUT`

The argument mode. `IN` declares the argument for input only. This is the default. `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 type(s) of an argument. An argument type may 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. A length must not be specified for any base type - for example, specify `VARCHAR2`, not `VARCHAR2(10)`.

The type of a column is referenced by writing `tablename.columnname %TYPE`; using this 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 one is not supplied in the invocation. `DEFAULT` may not be specified for arguments with modes `IN OUT` or `OUT`.

`func_name`

The name of a public function.

`rettype`

The return data type.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`DETERMINISTIC`

`DETERMINISTIC` is a synonym for `IMMUTABLE`. A `DETERMINISTIC` function cannot modify the database and always reaches the same result when given the same argument values; it does not 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`

`WNDS`

`WNPS`

Package Body Syntax

Package implementation details reside in the package body; the package body may contain objects that are not visible to the package user. 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; }
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
   [...]
]
```

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 execution_cost
ROWS result_rows
SET config_param { TO value | = value | FROM CURRENT }
```

Where `package_initializer` :=

```
BEGIN
 statement; [...]
END;
```

Parameters

`package_name`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`package_name` is the name of the package for which this is the package body. There must be an existing package specification with this name.

`private_declaration`

`private_declaration` is an identifier of a private variable that can be accessed by any procedure or function within 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 subprogram declarations are included, they must be declared after all other variable, type, and `REF CURSOR` declarations.

`statement`

An SPL program statement. Note that a `DECLARE - BEGIN - END` block is considered an SPL statement unto itself. Thus, the function body may contain nested blocks.

`exception`

An exception condition name such as `NO_DATA_FOUND, OTHERS`, etc.

`func_name`

The name of the function being created.

`rettype`

The return data type, which may be any of the types listed for `argtype`. As for `argtype`, a length must not be specified for `rettype`.

`DETERMINISTIC`

Include `DETERMINISTIC` to specify that the function will 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, it must also be specified for the function declaration in the package specification. (For private functions, there is no function declaration in the package specification.)

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 within the procedure body.

`IN | IN OUT | OUT`

The argument mode. `IN` declares the argument for input only. This is the default. `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 type(s) of an argument. An argument type may 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. A length must not be specified for any base type - for example, specify `VARCHAR2`, not `VARCHAR2(10)`.

The type of a column is referenced by writing `tablename.columnname%TYPE`; using this 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 one is not supplied in the procedure call. `DEFAULT` may not be specified for arguments with modes `IN OUT` or `OUT`.

Please note: The following options are not compatible with Oracle databases; they are extensions to Oracle package syntax provided by Advanced Server only.

`STRICT`

The `STRICT` keyword specifies that the function will not be executed if called with a `NULL` argument; instead the function will return `NULL`.

`LEAKPROOF`

The `LEAKPROOF` keyword specifies that the function will not 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 cannot be executed in parallel mode. The presence of such a procedure or function forces a serial execution plan. This is the default setting if the `PARALLEL` clause is omitted.

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 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`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`result_rows`

`result_rows` is the estimated number of rows that the query planner should 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.

`package_initializer`

The statements in the `package_initializer` are executed once per user's session when the package is first referenced.

> **Note**
>
> The `STRICT, LEAKPROOF, PARALLEL, COST, ROWS` and `SET` keywords provide extended functionality for Advanced Server and are not supported by Oracle.

11.1.2 Creating Packages

A package is not an executable piece of code; rather it is a repository of code. When you use a package, you actually 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 can be referenced 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:

```
--
-- 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,
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
      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. We 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 above `emp_admin` package specification, we shall now create a package body which will implement the specifications. The body will contain 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
      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.
   --
   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
   )
   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
   BEGIN
      DELETE FROM emp WHERE empno = p_empno;
   END;
END;
```

11.1.3 Referencing a Package

To reference the types, items and subprograms that are declared within a package specification, we use the dot notation. For example:

```
package_name.type_name
```

```
package_name.item_name
```

```
package_name.subprogram_name
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

To invoke a function from the `emp_admin` package specification, we will execute the following SQL command.

```
SELECT emp_admin.get_dept_name(10) FROM DUAL;
```

Here we are invoking the `get_dept_name` function declared within the package `emp_admin`. We are passing the department number as an argument to the function, which will return the name of the department. Here the value returned should be `ACCOUNTING`, which corresponds to department number `10`.

11.1.4 Using Packages With User Defined Types

The following example incorporates the various user-defined types discussed in earlier chapters within the context of a package.

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 along with two functions and two procedures. 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
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10)
    );
    TYPE emp_refcur IS REF CURSOR;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2;
    FUNCTION open_emp_by_dept (
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR;
    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    );
    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    );
END 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`.

```
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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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
            WHERE deptno = p_deptno;
        RETURN emp_by_dept;
    END;

    PROCEDURE fetch_emp (
        p_refcur      IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO ENAME');
        DBMS_OUTPUT.PUT_LINE('----- -------');
        LOOP
            FETCH p_refcur INTO r_emp;
            EXIT WHEN p_refcur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(r_emp.empno || ' ' || r_emp.ename);
        END LOOP;
    END;

    PROCEDURE close_refcur (
        p_refcur      IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        CLOSE p_refcur;
    END;
BEGIN
    OPEN dept_cur;
    LOOP
        FETCH dept_cur INTO t_dept(t_dept_max);
        EXIT WHEN dept_cur%NOTFOUND;
        t_dept_max := t_dept_max + 1;
    END LOOP;
    CLOSE dept_cur;
    t_dept_max := t_dept_max - 1;
END emp_rpt;
```

This package contains an initialization section that loads the private table variable, `t_dept`, using the private static cursor, `dept_cur.t_dept` serves as a department name lookup table in function, `get_dept_name`.

Function, `open_emp_by_dept` returns a `REF CURSOR` variable for a result set of employee numbers and names for a given department. This `REF CURSOR` variable can then be passed to procedure, `fetch_emp`, to retrieve and list the individual rows of the result set. Finally, procedure, `close_refcur`, can be used to close the `REF CURSOR` variable associated with this result set.

The following anonymous block runs the package function and procedures. In the anonymous block's declaration section, note the declaration of cursor variable, `v_emp_cur`, using the package's public `REF CURSOR` type, `EMP_REFCUR. v_emp_cur` contains the pointer to the result set that is

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

passed between the package function and procedures.

```
DECLARE
    v_deptno dept.deptno%TYPE DEFAULT 30;
    v_emp_cur emp_rpt.EMP_REFCUR;
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));
    emp_rpt.fetch_emp(v_emp_cur);
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    emp_rpt.close_refcur(v_emp_cur);
END;
```

The following is the result of this anonymous block.

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

The following anonymous block illustrates another means of achieving 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`.

```
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 is the result of this anonymous block.

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

11.1.5 Dropping a Package

The syntax for deleting an entire package or just the package body is as follows:

```
DROP PACKAGE [ BODY ] package_name;
```

If the keyword, `BODY` , is omitted, both the package specification and the package body are deleted - i.e., the entire package is dropped. If the keyword, `BODY` , is specified, then only the package body is dropped. The package specification remains intact. `package_name` is the identifier of the package to be dropped.

Following statement will destroy only the package body of `emp_admin` :

```
DROP PACKAGE BODY emp_admin;
```

The following statement will drop the entire `emp_admin` package:

```
DROP PACKAGE emp_admin;
```

11.2 Built-In Packages

This chapter describes the built-in packages that are provided with 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 has been granted to `PUBLIC` by default.

For information about using the `GRANT` command to provide access to a package, see the *Database Compatibility for Oracle Developers SQL Guide*, available at:

https://www.enterprisedb.com/docs

All built-in packages are owned by the special `sys` user which must be specified when granting or revoking privileges on built-in packages:

```
GRANT EXECUTE ON PACKAGE SYS.UTL_FILE TO john;
```

11.2.1 DBMS_ALERT

The `DBMS_ALERT` package provides the capability to register for, send, and receive alerts. The following table lists the supported procedures:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function/Procedure	Return Type	Description
`REGISTER(name)`	n/a	Register to be able to receive alerts named, `name`
`REMOVE(name)`	n/a	Remove registration for the alert named, `name`
`REMOVEALL`	n/a	Remove registration for all alerts.
`SIGNAL(name, message)`	n/a	Signals the alert named, `name`, with `message`
`WAITANY(name OUT, message OUT, status OUT, timeout)`	n/a	Wait for any registered alert to occur.
`WAITONE(name, message OUT, status OUT, timeout)`	n/a	Wait for the specified alert, `name`, to occur.

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 above are supported.

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 (located by default in `/opt/PostgresPlus/11AS/data`) with your choice of editor, and edit the `dbms_alert.max_alerts` parameter as shown:

```
dbms_alert.max_alerts = alert_count
```

`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`:

```
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.

```
REGISTER(<name> VARCHAR2)
```

Parameters

`name`

Name of the alert to be registered.

Examples

The following anonymous block registers for an alert named, `alert_test`, then waits for the signal.

```
DECLARE
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    v_name           VARCHAR2(30) := 'alert_test';
    v_msg            VARCHAR2(80);
    v_status         INTEGER;
    v_timeout        NUMBER(3) := 120;
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.

```
REMOVE(<name> VARCHAR2)
```

Parameters

`name`

Name of the alert to be unregistered.

REMOVEALL

The `REMOVEALL` procedure unregisters the session for all alerts.

```
REMOVEALL
```

SIGNAL

The `SIGNAL` procedure signals the occurrence of the named alert.

```
SIGNAL(<name> VARCHAR2, <message> VARCHAR2)
```

Parameters

`name`

Name of the alert.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`message`

Information to pass with this alert.

Examples

The following anonymous block signals an alert for `alert_test`.

```
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;
Issued alert for alert_test
```

WAITANY

The `WAITANY` procedure waits for any of the registered alerts to occur.

```
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

The following anonymous block uses the `WAITANY` procedure to receive an alert named, `alert_test` or `any_alert`:

```
DECLARE
    v_name           VARCHAR2(30);
    v_msg            VARCHAR2(80);
    v_status         INTEGER;
    v_timeout        NUMBER(3) := 120;
BEGIN
    DBMS_ALERT.REGISTER('alert_test');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    DBMS_ALERT.REGISTER('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);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.REMOVEALL;
END;

Registered for alert alert_test and any_alert
Waiting for signal...
```

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;

Issued alert for any_alert
```

Control returns to the first anonymous block and the remainder of the code is executed:

```
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.

```
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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

The following 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_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...
```

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;

Issued alert for alert_test
```

First session is alerted, control returns to the anonymous block, and the remainder of the code is executed:

```
 Registered for alert alert_test
Waiting for signal...
Alert name   : alert_test
Alert msg    : This is the message from alert_test
Alert status : 0
Alert timeout: 120 seconds
```

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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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
    v_action        VARCHAR2(25);
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;
    DBMS_ALERT.SIGNAL('emp_alert',USER || v_action || 'on ' ||
        SYSDATE);
END;
```

The following anonymous block is executed in a session while updates to the `dept` and `emp` tables occur in other sessions:

```
 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_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_ALERT.REMOVEALL;
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
Registered for alerts dept_alert and emp_alert
Waiting for signal...
```

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...
Alert name   : dept_alert
Alert msg    : mary added department(s) on 25-OCT-07 16:41:01
Alert status : 0
------------------------------------------------------------
Alert name   : emp_alert
Alert msg    : mary added employee(s) on 25-OCT-07 16:41:02
Alert status : 0
------------------------------------------------------------
Alert name   : dept_alert
Alert msg    : john added department(s) on 25-OCT-07 16:41:22
Alert status : 0
------------------------------------------------------------
Alert status : 1
```

11.2.2 DBMS_AQ

EDB Postgres Advanced Server Advanced Queueing provides message queueing and message processing for the Advanced Server database. User-defined messages are stored in a queue; 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 a queue or remove messages from a queue, or register or unregister a PL/SQL callback procedure.

Advanced Server also provides extended (non-compatible) functionality for the `DBMS_AQ` package with SQL commands, see the *Database Compatibility for Oracle Developers SQL Guide* 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:

Function/Procedure	Return Type	Description

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function/Procedure	Return Type	Description
ENQUEUE	n/a	Post a message to a queue.
DEQUEUE	n/a	Retrieve a message from a queue if or when a message is available.
REGISTER	n/a	Register a callback procedure.
UNREGISTER	n/a	Unregister a callback procedure.

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.

Advanced Server supports use of the constants listed below:

Constant	Description	For Parameters
DBMS_AQ.BROWSE (0)	Read the message without locking.	dequeue_options_t.dequeue_mode
DBMS_AQ.LOCKED (1)	This constant is defined, but will return an error if used.	dequeue_options_t.dequeue_mode
DBMS_AQ.REMOVE (2)	Delete the message after reading; the default.	dequeue_options_t.dequeue_mode
DBMS_AQ.REMOVE_NODATA (3)	This constant is defined, but will return an error if used.	dequeue_options_t.dequeue_mode
DBMS_AQ.FIRST_MESSAGE (0)	Return the first available message that matches the search terms.	dequeue_options_t.navigation
DBMS_AQ.NEXT_MESSAGE (1)	Return the next available message that matches the search terms.	dequeue_options_t.navigation
DBMS_AQ.NEXT_TRANSACTION (2)	This constant is defined, but will return an error if used.	dequeue_options_t.navigation
DBMS_AQ.FOREVER (0)	Wait forever if a message that matches the search term is not found, the default.	dequeue_options_t.wait
DBMS_AQ.NO_WAIT (1)	Do not wait if a message that matches the search term is not found.	dequeue_options_t.wait
DBMS_AQ.ON_COMMIT (0)	The dequeue is part of the current transaction.	enqueue_options_t.visibility, dequeue_options_t.visibility
DBMS_AQ.IMMEDIATE (1)	This constant is defined, but will return an error if used.	enqueue_options_t.visibility, dequeue_options_t.visibility
DBMS_AQ.PERSISTENT (0)	The message should be stored in a table.	enqueue_options_t.delivery_mode
DBMS_AQ.BUFFERED (1)	This constant is defined, but will return an error if used.	enqueue_options_t.delivery_mode
DBMS_AQ.READY (0)	Specifies that the message is ready to process.	message_properties_t.state
DBMS_AQ.WAITING (1)	Specifies that the message is waiting to be processed.	message_properties_t.state
DBMS_AQ.PROCESSED (2)	Specifies that the message has been processed.	message_properties_t.state
DBMS_AQ.EXPIRED (3)	Specifies that the message is in the exception queue.	message_properties_t.state
DBMS_AQ.NO_DELAY (0)	This constant is defined, but will return an error if used	message_properties_t.delay
DBMS_AQ.NEVER (NULL)	This constant is defined, but will return an error if used	message_properties_t.expiration
DBMS_AQ.NAMESPACE_AQ (0)	Accept notifications from DBMS_AQ queues.	sys.aq$_reg_info.namespace
DBMS_AQ.NAMESPACE_ANONYMOUS (1)	This constant is defined, but will return an error if used	sys.aq$_reg_info.namespace

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

Parameter	Description
dbms_aq.max_workers	The maximum number of workers to run.
dbms_aq.max_idle_time	The idle time a worker must wait before exiting.
dbms_aq.min_work_time	The minimum time a worker can run before exiting.
dbms_aq.launch_delay	The minimum time between creating workers.
dbms_aq.batch_size	The maximum number of messages to process in a single transaction. The default batch size is 10.
dbms_aq.max_databases	The size of DBMS_AQ 's hash table of databases. The default value is 1024.
dbms_aq.max_pending_retries	The size of DBMS_AQ 's hash table of pending retries. The default value is 1024.

11.2.2.1 ENQUEUE

The ENQUEUE procedure adds an entry to a queue. The signature is:

```
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 will use the schema specified in the SEARCH_PATH . Please note that 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

enqueue_options is a value of the type, enqueue_options_t :

```
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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

visibility	ON_COMMIT.
sequence_deviation	NULL
transformation	NULL
relative_msgid	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,
  transaction_group CHARACTER VARYING(30) COLLATE pg_catalog.”C”,
  delivery_mode INTEGER
DBMS_AQ.PERSISTENT);
```

The supported values for message_properties_t are:

priority	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.
delay	Specify the number of seconds that will pass before a message is available for dequeueing or NO_DELAY.
expiration	Use the expiration parameter to specify the number of seconds until a message expires.
correlation	Use correlation to specify a message that will be associated with the entry; the default is NULL.
attempts	This is a system-maintained value that specifies the number of attempts to dequeue the message.
recipient_list	This parameter is not supported.
exception_queue	Use the exception_queue parameter to specify the name of an exception queue to which a message will be moved if it expires or is dequeued by a transaction that rolls back too many times.
enqueue_time	enqueue_time is the time the record was added to the queue; this value is provided by the system.
state	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.
original_msgid	This parameter is accepted for compatibility and ignored.
transaction_group	This parameter is accepted for compatibility and ignored.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`delivery_mode`	This parameter is not supported; specify a value of `DBMS_AQ.PERSISTENT`.

`payload`

Use the `payload` parameter to provide the data that will be associated 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`:

```
DECLARE

  enqueue_options    DBMS_AQ.ENQUEUE_OPTIONS_T;
  message_properties DBMS_AQ.MESSAGE_PROPERTIES_T;
  message_handle     raw(16);
  payload            work_order;

BEGIN

  payload := work_order('Smith', 'system upgrade');

DBMS_AQ.ENQUEUE(
  queue_name         => 'work_order',
  enqueue_options    => enqueue_options,
  message_properties => message_properties,
  payload            => payload,
  msgid              => message_handle
    );
 END;
```

11.2.2.2 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 will use the schema specified in the `SEARCH_PATH`. Please note that unlike Oracle, unquoted identifiers are converted to lower case before storing. To include special characters or use a case-sensitive name,

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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:

`consumer_name`	Must be `NULL`.
`dequeue_mode`	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 do not delete the message.
`navigation`	Identifies the message that will be retrieved. Must be either: `FIRST_MESSAGE` – The first message within the queue that matches the search term. `NEXT_MESSAGE` – The next message that is available that matches the first term.
`visibility`	Must be `ON_COMMIT` – if you roll back the current transaction the dequeued item will remain in the queue.
`wait`	Must be a number larger than 0, or: `DBMS_AQ.FOREVER` – Wait indefinitely. `DBMS_AQ.NO_WAIT` – Do not wait.
`msgid`	The message ID of the message that will be dequeued.
`correlation`	Accepted for compatibility, and ignored.
`deq_condition`	A `VARCHAR2` expression that evaluates to a `BOOLEAN` value indicating if the message should be dequeued.
`transformation`	Accepted for compatibility, and ignored.
`delivery_mode`	Must be `PERSISTENT`; buffered messages are not supported at this time.

`message_properties` is a value of the type, `message_properties_t`:

```
message_properties_t IS RECORD(
  priority INTEGER,
  delay INTEGER,
  expiration INTEGER,
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  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,
  transaction_group CHARACTER VARYING(30) COLLATE pg_catalog.”C”,
  delivery_mode INTEGER
DBMS_AQ.PERSISTENT);
```

The supported values for `message_properties_t` are:

`priority`	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.
`delay`	Specify the number of seconds that will pass before a message is available for dequeueing or `NO_DELAY`.
`expiration`	Use the expiration parameter to specify the number of seconds until a message expires.
`correlation`	Use correlation to specify a message that will be associated with the entry; the default is `NULL`.
`attempts`	This is a system-maintained value that specifies the number of attempts to dequeue the message.
`recipient_list`	This parameter is not supported.
`exception_queue`	Use the `exception_queue` parameter to specify the name of an exception queue to which a message will be moved if it expires or is dequeued by a transaction that rolls back too many times.
`enqueue_time`	`enqueue_time` is the time the record was added to the queue; this value is provided by the system.
`state`	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.
`original_msgid`	This parameter is accepted for compatibility and ignored.
`transaction_group`	This parameter is accepted for compatibility and ignored.
`delivery_mode`	This parameter is not 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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    => dequeue_options,
    message_properties => message_properties,
    payload            => 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`.

11.2.2.3 REGISTER

Use the `REGISTER` procedure to register an email address, procedure or URL that will be notified 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$_REG_INFO_LIST`; that provides information about each subscription that you would like to register. Each entry within the list is of the type `AQ$_REG_INFO`, and may contain:

Attribute	Type	Description
`name`	VARCHAR2 (128)	The (optionally schema-qualified) name of the subscription.
`namespace`	NUMERIC	The only supported value is `DBMS_AQ.NAMESPACE_AQ (0)`
`callback`	VARCHAR2 (4000)	Describes the action that will be performed upon notification. Currently, only calls to PL/SQL procedures are supported. The call should take the form: `plsql://schema.procedure` Where: schema specifies the schema in which the procedure resides. procedure specifies the name of the procedure that will be notified.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Attribute	Type	Description
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 that will be notified 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:

```
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.

11.2.2.4 UNREGISTER

Use the UNREGISTER procedure to turn off notifications related to enqueueing and dequeueing. The signature is:

```
UNREGISTER(
  <reg_list> IN SYS.AQ$_REG_INFO_LIST,
  <count> IN NUMBER)
```

Parameter

reg_list

reg_list is a list of type AQ$_REG_INFO_LIST; that provides information about each subscription that you would like to register. Each entry within the list is of the type AQ$_REG_INFO, and may contain:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Attribute	Type	Description
`name`	VARCHAR2 (128)	The (optionally schema-qualified) name of the subscription.
`namespace`	NUMERIC	The only supported value is `DBMS_AQ.NAMESPACE_AQ (0)`
`callback`	VARCHAR2 (4000)	Describes the action that will be performed upon notification. Currently, only calls to PL/SQL procedures are supported. The call should take the form: `plsql://schema.procedure` Where: schema specifies the schema in which the procedure resides. procedure specifies the name of the procedure that will be notified.
`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.UNREGISTER`, disabling the notifications specified in the example for `DBMS_AQ.REGISTER`:

```
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$_reg_info_list`, and contains the previously described `sys.aq$_reg_info` objects. The list name and an object count are passed to `dbms_aq.unregister`.

11.2.3 DBMS_AQADM

EDB Postgres Advanced Server Advanced Queueing provides message queueing and message processing for the Advanced Server database. User-defined messages are stored in a queue; a collection of queues is stored in a queue table. Procedures in the `DBMS_AQADM` package create and manage message

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

queues and queue tables. Use the `DBMS_AQ` package to add messages to a queue or remove messages from a queue, or register or unregister a PL/SQL callback procedure.

Advanced Server also provides extended (non-compatible) functionality for the `DBMS_AQ` package with SQL commands, see the *Database Compatibility for Oracle Developers SQL Guide* 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.

Function/Procedure	Return Type	Description
`ALTER_QUEUE`	n/a	Modify an existing queue.
`ALTER_QUEUE_TABLE`	n/a	Modify an existing queue table.
`CREATE_QUEUE`	n/a	Create a queue.
`CREATE_QUEUE_TABLE`	n/a	Create a queue table.
`DROP_QUEUE`	n/a	Drop an existing queue.
`DROP_QUEUE_TABLE`	n/a	Drop an existing queue table.
`PURGE_QUEUE_TABLE`	n/a	Remove one or more messages from a queue table.
`START_QUEUE`	n/a	Make a queue available for enqueueing and dequeueing procedures.
`STOP_QUEUE`	n/a	Make a queue unavailable for enqueueing and dequeueing procedures

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 above are supported.

Advanced Server supports use of the arguments listed below:

Constant	Description	For Parameters
`DBMS_AQADM.TRANSACTIONAL(1)`	This constant is defined, but will return an error if used.	`message_grouping`
`DBMS_AQADM.NONE(0)`	Use to specify message grouping for a queue table.	`message_grouping`
`DBMS_AQADM.NORMAL_QUEUE(0)`	Use with `create_queue` to `specify queue_type`.	`queue_type`
`DBMS_AQADM.EXCEPTION_QUEUE (1)`	Use with `create_queue` to specify `queue_type`.	`queue_type`
`DBMS_AQADM.INFINITE(-1)`	Use with `create_queue` to specify `retention_time`.	`retention_time`
`DBMS_AQADM.PERSISTENT (0)`	The message should be stored in a table.	`enqueue_options_t.delivery_mode`
`DBMS_AQADM.BUFFERED (1)`	This constant is defined, but will return an error if used.	`enqueue_options_t.delivery_mode`
`DBMS_AQADM.PERSISTENT_OR_BUFFERED (2)`	This constant is defined, but will return an error if used.	`enqueue_options_t.delivery_mode`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.3.1 ALTER_QUEUE

Use the `ALTER_QUEUE` procedure to modify an existing queue. The signature is:

```
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`

`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`

`retry_delay` specifies the number of seconds until a message is scheduled for re-processing after a `ROLLBACK`. Specify `0` to indicate that the message should be retried immediately (the default).

`retention_time`

`retention_time` specifies the length of time (in seconds) that a message will be stored after being dequeued. You can also specify `0` (the default) to indicate the message should not be retained after dequeueing, or `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:

```
EXEC DBMS_AQADM.ALTER_QUEUE(queue_name => 'work_order', retry_delay => 5);
```

11.2.3.2 ALTER_QUEUE_TABLE

Use the `ALTER_QUEUE_TABLE` procedure to modify an existing queue table. The signature is:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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`:

```
EXEC DBMS_AQADM.ALTER_QUEUE_TABLE
     (queue_table => 'work_order_table', comment => 'This queue table
contains work orders for the shipping department.');
```

The queue table is named `work_order_table`; the command adds a comment to the definition of the queue table.

11.2.3.3 CREATE_QUEUE

Use the `CREATE_QUEUE` procedure to create a queue in an existing queue table. The signature is:

```
CREATE_QUEUE(
  <queue_name> IN VARCHAR2
  <queue_table> IN VARCHAR2,
  <queue_type> IN BINARY_INTEGER DEFAULT NORMAL_QUEUE,
  <max_retries> IN NUMBER DEFAULT 5,
  <retry_delay> IN NUMBER DEFAULT 0
  <retention_time> IN NUMBER DEFAULT 0,
  <dependency_tracking> IN BOOLEAN DEFAULT FALSE,
  <comment> IN VARCHAR2 DEFAULT NULL,
  <auto_commit> IN BOOLEAN DEFAULT TRUE)
```

Parameters

`queue_name`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The name of the new queue.

`queue_table`

The name of the table in which the new queue will reside.

`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 will support 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 re-processing after a `ROLLBACK`. Specify `0` to indicate that the message should be retried immediately (the default).

`retention_time`

`retention_time` specifies the length of time (in seconds) that a message will be stored after being dequeued. You can also specify `0` (the default) to indicate the message should not be retained after dequeueing, 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:

```
 BEGIN
DBMS_AQADM.CREATE_QUEUE ( queue_name => 'work_order', queue_table =>
'work_order_table', comment => 'This queue contains pending work orders.');
END;
```

11.2.3.4 CREATE_QUEUE_TABLE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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,
  <storage_clause> IN VARCHAR2 DEFAULT NULL,
  <sort_list> 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,
  <compatible> IN VARCHAR2 DEFAULT NULL,
  <secure> IN BOOLEAN 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 will be stored in the queue table. Please note that 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. Please note that 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 the table will be created.

`storage_clause` may be one or more of the following:

```
 TABLESPACE tablespace_name, PCTFREE integer, PCTUSED integer,
INITRANS integer, MAXTRANS integer or STORAGE storage_option.
```

`storage_option` may 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 column(s) that will be used 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `enq_time`

`multiple_consumers`

If specified, `message_consumers` must be `FALSE`.

`message_grouping`

If specified, `message_grouping` must be `NONE`.

`comment`

Use the `comment` parameter to provide a comment about the queue table.

`auto_commit`

`auto_commit` is accepted for compatibility, but is ignored.

`primary_instance`

`primary_instance` is accepted for compatibility and stored, but is ignored.

`secondary_instance`

`secondary_instance` is accepted for compatibility, but is ignored.

`compatible`

`compatible` is accepted for compatibility, but is ignored.

`secure`

`secure` is accepted for compatibility, but is ignored.

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:

```
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`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.3.5 DROP_QUEUE

Use the `DROP_QUEUE` procedure to delete a queue. The signature is:

```
DROP_QUEUE(
  <queue_name>  IN VARCHAR2,
  <auto_commit> IN BOOLEAN DEFAULT TRUE)
```

Parameters

`queue_name`

The name of the queue that you wish to drop.

`auto_commit`

`auto_commit` is accepted for compatibility, but is ignored.

Example

The following anonymous block drops the queue named `work_order`:

```
 BEGIN
DBMS_AQADM.DROP_QUEUE(queue_name => 'work_order');
END;
```

11.2.3.6 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 will fail, and the server will issue an error.
- If the target table contains entries and force is `TRUE`, the command will drop the table and any dependent objects.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 ('work_order_table', force => TRUE);
END;
```

11.2.3.7 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 are deleting a message.

`purge_condition`

Use `purge_condition` to specify a condition (a SQL `WHERE` clause) that the server will evaluate 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:

Attribute	Type	Description
`block`	Boolean	Specify `TRUE` if an exclusive lock should be held on all queues within the table; the default is `FALSE`.
`delivery_mode`	INTEGER	`delivery_mode` specifies the type of message that will be purged. The only accepted value is `DBMS_AQ.PERSISTENT`.

Example

The following anonymous block removes any messages from the `work_order_table` with a value in the `completed` column of `YES`:

```
DECLARE
   purge_options dbms_aqadm.aq$_purge_options_t;
BEGIN
   dbms_aqadm.purge_queue_table('work_order_table', 'completed = YES',
purge_options);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  END;
```

11.2.3.8 START_QUEUE

Use the `START_QUEUE` procedure to make a queue available for enqueuing and dequeueing. The signature is:

```
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 are starting.

`enqueue`

Specify `TRUE` to enable enqueueing (the default), or `FALSE` to leave the current setting unchanged.

`dequeue`

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:

```
 BEGIN
DBMS_AQADM.START_QUEUE
(queue_name => 'work_order);
END;
```

11.2.3.9 STOP_QUEUE

Use the `STOP_QUEUE` procedure to disable enqueuing or dequeueing on a specified queue. The signature is:

```
STOP_QUEUE(
  <queue_name> IN VARCHAR2,
  <enqueue> IN BOOLEAN DEFAULT TRUE,
  <dequeue> IN BOOLEAN DEFAULT TRUE,
  <wait> IN BOOLEAN DEFAULT TRUE)
```

Parameters

`queue_name`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`queue_name` specifies the name of the queue that you are stopping.

`enqueue`

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;
```

Enqueueing and dequeueing will stop after any outstanding transactions complete.

11.2.4 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 following table lists the `DBMS_CRYPTO` Functions and Procedures.

Function/Procedure	Return Type	Description
`DECRYPT(src, typ, key, iv)`	`RAW`	Decrypts `RAW` data.
`DECRYPT(dst INOUT, src, typ, key, iv)`	N/A	Decrypts `BLOB` data.
`DECRYPT(dst INOUT, src, typ, key, iv)`	N/A	Decrypts `CLOB` data.
`ENCRYPT(src, typ, key, iv)`	`RAW`	Encrypts `RAW` data.
`ENCRYPT(dst INOUT, src, typ, key, iv)`	N/A	Encrypts `BLOB` data.
`ENCRYPT(dst INOUT, src, typ, key, iv)`	N/A	Encrypts `CLOB` data.
`HASH(src, typ)`	`RAW`	Applies a hash algorithm to `RAW` data.
`HASH(src)`	`RAW`	Applies a hash algorithm to `CLOB` data.
`MAC(src, typ, key)`	`RAW`	Returns the hashed `MAC` value of the given `RAW` data using the specified hash algorithm and key.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function/Procedure	Return Type	Description
`MAC(src, typ, key)`	`RAW`	Returns the hashed `MAC` value of the given `CLOB` data using the specified hash algorithm and key.
`RANDOMBYTES(number_bytes)`	`RAW`	Returns a specified number of cryptographically strong random bytes.
`RANDOMINTEGER()`	`INTEGER`	Returns a random `INTEGER`.
`RANDOMNUMBER()`	`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, Advanced Server will not return error `ORA-28233` if you re-encrypt previously encrypted information.

Please note that `RAW` and `BLOB` are synonyms for the PostgreSQL `BYTEA` data type, and `CLOB` is a synonym for `TEXT`.

11.2.4.1 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:

```
DECRYPT
  (<src> IN RAW, <typ> IN INTEGER, <key> IN RAW, <iv> IN RAW
   DEFAULT NULL) RETURN RAW
```

The signature of the `DECRYPT` procedure is:

```
DECRYPT
  (<dst> INOUT BLOB, <src> IN BLOB, <typ> IN INTEGER, <key> IN RAW,
   <iv> IN RAW DEFAULT NULL)
```

or

```
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`

`dst` specifies the name of a `BLOB` to which the output of the `DECRYPT` procedure will be written. The `DECRYPT` procedure will overwrite any existing

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

data currently in `dst`.

`src`

`src` specifies the source data that will be decrypted. If you are invoking `DECRYPT` as a function, specify `RAW` data; if invoking `DECRYPT` as a procedure, specify `BLOB` or `CLOB` data.

`typ`

`typ` specifies the block cipher type and any modifiers. This should match the type specified when the `src` was encrypted. Advanced Server supports the following block cipher algorithms, modifiers and cipher suites:

Block Cipher Algorithms	
`ENCRYPT_DES`	`CONSTANT INTEGER := 1;`
`ENCRYPT_3DES`	`CONSTANT INTEGER := 3;`
`ENCRYPT_AES`	`CONSTANT INTEGER := 4;`
`ENCRYPT_AES128`	`CONSTANT INTEGER := 6;`
Block Cipher Modifiers	
`CHAIN_CBC`	`CONSTANT INTEGER := 256;`
`CHAIN_ECB`	`CONSTANT INTEGER := 768;`
Block Cipher Padding Modifiers	
`PAD_PKCS5`	`CONSTANT INTEGER := 4096;`
`PAD_NONE`	`CONSTANT INTEGER := 8192;`
Block Cipher Suites	
`DES_CBC_PKCS5`	`CONSTANT INTEGER := ENCRYPT_DES + CHAIN_CBC + PAD_PKCS5;`
`DES3_CBC_PKCS5`	`CONSTANT INTEGER := ENCRYPT_3DES + CHAIN_CBC + PAD_PKCS5;`
`AES_CBC_PKCS5`	`CONSTANT INTEGER := ENCRYPT_AES + CHAIN_CBC + PAD_PKCS5;`

`key`

`key` specifies the user-defined decryption key. This should match the key specified when the `src` was encrypted.

`iv`

`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

The following example uses the `DBMS_CRYPTO.DECRYPT` function to decrypt an encrypted password retrieved from the `passwords` table:

```
 CREATE TABLE passwords
(
  principal VARCHAR2(90) PRIMARY KEY, -- username
  ciphertext RAW(9) -- encrypted password
);

CREATE FUNCTION get_password(username VARCHAR2) RETURN RAW AS
 typ       INTEGER := DBMS_CRYPTO.DES_CBC_PKCS5;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 key       RAW(128) := 'my secret key';
 iv        RAW(100) := 'my initialization vector';
 password  RAW(2048);
BEGIN

  SELECT ciphertext INTO password FROM passwords WHERE principal = username;

  RETURN dbms_crypto.decrypt(password, typ, key, iv);
END;
```

Note that when calling `DECRYPT`, you must pass the same cipher type, key value and initialization vector that was used when `ENCRYPTING` the target.

11.2.4.2 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:

```
ENCRYPT
  (<src> IN RAW, <typ> IN INTEGER, <key> IN RAW,
   <iv> IN RAW DEFAULT NULL) RETURN RAW
```

The signature of the `ENCRYPT` procedure is:

```
ENCRYPT
  (<dst> INOUT BLOB, <src> IN BLOB, <typ> IN INTEGER, <key> IN RAW,
   <iv> IN RAW DEFAULT NULL)
```

or

```
ENCRYPT
  (<dst> INOUT BLOB, <src> IN CLOB, <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 the output of the `ENCRYPT` procedure will be written. The `ENCRYPT` procedure will overwrite any existing data currently in `dst`.

`src`

`src` specifies the source data that will be encrypted. If you are 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 that will be used by `ENCRYPT`, and any modifiers. Advanced Server supports the block cipher algorithms, modifiers and cipher suites listed below:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Block Cipher Algorithms	
`ENCRYPT_DES`	`CONSTANT INTEGER := 1;`
`ENCRYPT_3DES`	`CONSTANT INTEGER := 3;`
`ENCRYPT_AES`	`CONSTANT INTEGER := 4;`
`ENCRYPT_AES128`	`CONSTANT INTEGER := 6;`
Block Cipher Modifiers	
`CHAIN_CBC`	`CONSTANT INTEGER := 256;`
`CHAIN_ECB`	`CONSTANT INTEGER := 768;`
Block Cipher Padding Modifiers	
`PAD_PKCS5`	`CONSTANT INTEGER := 4096;`
`PAD_NONE`	`CONSTANT INTEGER := 8192;`
Block Cipher Suites	
`DES_CBC_PKCS5`	`CONSTANT INTEGER := ENCRYPT_DES + CHAIN_CBC + PAD_PKCS5;`
`DES3_CBC_PKCS5`	`CONSTANT INTEGER := ENCRYPT_3DES + CHAIN_CBC + PAD_PKCS5;`
`AES_CBC_PKCS5`	`CONSTANT INTEGER := ENCRYPT_AES + CHAIN_CBC + PAD_PKCS5;`

`key`

`key` specifies the encryption key.

`iv`

`iv` (optional) specifies an initialization vector. By default, `iv` is `NULL`.

Examples

The following example uses the `DBMS_CRYPTO.DES_CBC_PKCS5` Block Cipher Suite (a pre-defined set of algorithms and modifiers) to encrypt a value retrieved from the `passwords` table:

```
 CREATE TABLE passwords
(
  principal   VARCHAR2(90) PRIMARY KEY, -- username
  ciphertext  RAW(9) -- encrypted password
);
CREATE PROCEDURE set_password(username VARCHAR2, cleartext RAW) AS
 typ         INTEGER := DBMS_CRYPTO.DES_CBC_PKCS5;
 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`.

11.2.4.3 HASH

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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:

```
HASH
  (<src> IN RAW, <typ> IN INTEGER) RETURN RAW

HASH
  (<src> IN CLOB, <typ> IN INTEGER) RETURN RAW
```

Parameters

`src`

`src` specifies the value for which the hash value will be generated. You can specify a `RAW`, a `BLOB`, or a `CLOB` value.

`typ`

`typ` specifies the `HASH` function type. Advanced Server supports the `HASH` function types listed below:

HASH Functions

`HASH_MD4`	`CONSTANT INTEGER := 1;`
`HASH_MD5`	`CONSTANT INTEGER := 2;`
`HASH_SH1`	`CONSTANT INTEGER := 3;`

Examples

The following example uses `DBMS_CRYPTO.HASH` to find the `md5` hash value of the string, `cleartext source`:

```
DECLARE
  typ INTEGER := DBMS_CRYPTO.HASH_MD5;
  hash_value RAW(100);
BEGIN

  hash_value := DBMS_CRYPTO.HASH('cleartext source', typ);

END;
```

11.2.4.4 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:

```
MAC
  (<src> IN RAW, <typ> IN INTEGER, <key> IN RAW) RETURN RAW

MAC
  (<src> IN CLOB, <typ> IN INTEGER, <key> IN RAW) RETURN RAW
```

Parameters

`src`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`src` specifies the value for which the `MAC` value will be generated. Specify a `RAW`, `BLOB`, or `CLOB` value.

`typ`

`typ` specifies the `MAC` function used. Advanced Server supports the `MAC` functions listed below.

MAC Functions

`HMAC_MD5`	`CONSTANT INTEGER := 1;`
`HMAC_SH1`	`CONSTANT INTEGER := 2;`

`key`

`key` specifies the key that will be used to calculate the hashed `MAC` value.

Examples

The following example finds the hashed `MAC` value of the string `cleartext source`:

```
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`.

11.2.4.5 RANDOMBYTES

The `RANDOMBYTES` function returns a `RAW` value of the specified length, containing cryptographically random bytes. The signature is:

```
RANDOMBYTES
  (<number_bytes> IN INTEGER) RETURNS RAW
```

Parameter

`number_bytes`

`number_bytes` specifies the number of random bytes to be returned

Examples

The following example uses `RANDOMBYTES` to return a value that is `1024` bytes long:

```
DECLARE
  result RAW(1024);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
BEGIN
  result := DBMS_CRYPTO.RANDOMBYTES(1024);
END;
```

11.2.4.6 RANDOMINTEGER

The `RANDOMINTEGER()` function returns a random `INTEGER` between `0` and `268,435,455`. The signature is:

```
RANDOMINTEGER() RETURNS INTEGER
```

Examples

The following example uses the `RANDOMINTEGER` function to return a cryptographically strong random `INTEGER` value:

```
DECLARE
  result INTEGER;
BEGIN
  result := DBMS_CRYPTO.RANDOMINTEGER();
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

11.2.4.7 RANDOMNUMBER

The `RANDOMNUMBER()` function returns a random `NUMBER` between `0` and `268,435,455`. The signature is:

```
RANDOMNUMBER() RETURNS NUMBER
```

Examples

The following example uses the `RANDOMNUMBER` function to return a cryptographically strong random number:

```
DECLARE
  result NUMBER;
BEGIN
  result := DBMS_CRYPTO.RANDOMNUMBER();
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

11.2.5 DBMS_JOB

The `DBMS_JOB` package provides for the creation, scheduling, and managing of jobs. A job runs a stored procedure which has been previously stored in the database. The `SUBMIT` procedure is used to create and store a job definition. A job identifier is assigned to a job along with its associated stored procedure and the attributes describing when and how often the job is to be run.

This package relies on the `pgAgent` scheduler. By default, the Advanced Server installer installs `pgAgent`, but you must start the `pgAgent` service manually prior to using `DBMS_JOB`. If you attempt to use this package to schedule a job after un-installing `pgAgent, DBMS_JOB` will throw an error.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`DBMS_JOB` verifies that `pgAgent` is installed, but does not verify that the service is running.

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:

Function/Procedure	Return Type	Description
`BROKEN(job, broken [, next_date ])`	n/a	Specify that a given job is either broken or not broken.
`CHANGE(job, what, next_date, interval, instance, force)`	n/a	Change the job's parameters.
`INTERVAL(job, interval)`	n/a	Set the execution frequency by means of a date function that is recalculated each time the job is run. This value becomes the next date/time for execution.
`NEXT_DATE(job, next_date)`	n/a	Set the next date/time the job is to be run.
`REMOVE(job)`	n/a	Delete the job definition from the database.
`RUN(job)`	n/a	Forces execution of a job even if it is marked broken.
`SUBMIT(job OUT, what [, next_date [, interval [, no_parse ]]])`	n/a	Creates a job and stores its definition in the database.
`WHAT(job, what)`	n/a	Change the stored procedure run by a job.

Before using `DBMS_JOB`, a database superuser must create the `pgAgent` and `DBMS_JOB` extension. Use the `psql` client to connect to a database and invoke the command:

```
CREATE EXTENSION pgagent;
CREATE EXTENSION dbms_job;
```

When and how often a job is run is dependent upon two interacting parameters – `next_date` and `interval`. The `next_date` parameter is a date/time value that specifies the next date/time when the job is to be executed. The `interval` parameter is a string that contains a date function that evaluates to a date/time value.

Just prior to any execution of the job, the expression in the `interval` parameter is evaluated. The resulting value replaces the `next_date` value stored with the job. The job is then executed. In this manner, the expression in `interval` is repeatedly re-evaluated prior to each job execution, supplying the `next_date` date/time for the next execution.

> **Note**
>
> The database user must be the same that created a job and schedule to start the `pgAgent` server and execute the job.

The following examples use the following stored procedure, `job_proc`, which simply inserts a timestamp into table, `jobrun`, containing a single `VARCHAR2` column.

```
CREATE TABLE jobrun (
    runtime VARCHAR2(40)
);

CREATE OR REPLACE PROCEDURE job_proc
IS
BEGIN
    INSERT INTO jobrun VALUES ('job_proc run at ' || TO_CHAR(SYSDATE,
        'yyyy-mm-dd hh24:mi:ss'));
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.5.1 BROKEN

The `BROKEN` procedure sets the state of a job to either broken or not broken. A broken job cannot be executed except by using the `RUN` procedure.

```
BROKEN(<job> BINARY_INTEGER, <broken> BOOLEAN [, <next_date> DATE ])
```

Parameters

`job`

Identifier of the job to be 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. Broken jobs cannot be run except 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:

```
BEGIN
    DBMS_JOB.BROKEN(104,true);
END;
```

Change the state back to not broken:

```
BEGIN
    DBMS_JOB.BROKEN(104,false);
END;
```

11.2.5.2 CHANGE

The `CHANGE` procedure modifies certain job attributes including the stored procedure to be run, the next date/time the job is to be run, and how often it is to be run.

```
CHANGE(<job> BINARY_INTEGER <what> VARCHAR2, <next_date> DATE,
  <interval> VARCHAR2, <instance> BINARY_INTEGER, <force> BOOLEAN)
```

Parameters

`job`

Identifier of the job to modify.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`what`

Stored procedure name. Set this parameter to null if the existing value is to remain unchanged.

`next_date`

Date/time when the job is to be run next. Set this parameter to null if the existing value is to remain unchanged.

`interval`

Date function that when evaluated, provides the next date/time the job is to run. Set this parameter to null if the existing value is 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.

```
BEGIN
    DBMS_JOB.CHANGE(104,NULL,TO_DATE('13-DEC-07','DD-MON-YY'),NULL, NULL,
    NULL);
END;
```

11.2.5.3 INTERVAL

The `INTERVAL` procedure sets the frequency of how often a job is to be run.

```
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 the job is to be run. 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:

```
BEGIN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    DBMS_JOB.INTERVAL(104,'SYSDATE + 7');
END;
```

11.2.5.4 NEXT_DATE

The `NEXT_DATE` procedure sets the date/time of when the job is to be run next.

```
NEXT_DATE(<job> BINARY_INTEGER, <next_date> DATE)
```

Parameters

`job`

Identifier of the job whose next run date is to be set.

`next_date`

Date/time when the job is to be run next.

Examples

Change the job to run next on December 14, 2007:

```
BEGIN
    DBMS_JOB.NEXT_DATE(104, TO_DATE('14-DEC-07','DD-MON-YY'));
END;
```

11.2.5.5 REMOVE

The `REMOVE` procedure deletes the specified job from the database. The job must be resubmitted using the `SUBMIT` procedure in order to have it executed again. Note that the stored procedure that was associated with the job is not deleted.

```
REMOVE(<job> BINARY_INTEGER)
```

Parameter

`job`

Identifier of the job that is to be removed from the database.

Examples

Remove a job from the database:

```
BEGIN
    DBMS_JOB.REMOVE(104);
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.5.6 RUN

The `RUN` procedure forces the job to be run, even if its state is broken.

```
RUN(<job> BINARY_INTEGER)
```

Parameter

`job`

Identifier of the job to be run.

Examples

Force a job to be run.

```
BEGIN
    DBMS_JOB.RUN(104);
END;
```

11.2.5.7 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 be executed, when the job is to be first run, and a date function that calculates the next date/time the job is to be run.

```
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 to be executed by the job.

`next_date`

Date/time when the job is to be run next. The default is `SYSDATE`.

`interval`

Date function that when evaluated, provides the next date/time the job is to run. If `interval` is set to null, then the job is run only once. Null is the default.

`no_parse`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

If set to `TRUE`, do not 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 is not supported in this implementation of `SUBMIT()`. It is included for compatibility only.

Examples

The following example creates a job using stored procedure, `job_proc`. The job will execute immediately and run once a day thereafter as set by the `interval` parameter, `SYSDATE + 1`.

```
DECLARE
    jobid INTEGER;
BEGIN
    DBMS_JOB.SUBMIT(jobid,'job_proc;',SYSDATE,
        'SYSDATE + 1');
    DBMS_OUTPUT.PUT_LINE('jobid: ' || jobid);
END;

jobid: 104
```

The job immediately executes procedure, `job_proc`, populating table, `jobrun`, with a row:

```
SELECT * FROM jobrun;

            runtime
-----------------------------------
 job_proc run at 2007-12-11 11:43:25
(1 row)
```

11.2.5.8 WHAT

The `WHAT` procedure changes the stored procedure that the job will execute.

```
WHAT(<job> BINARY_INTEGER, <what> VARCHAR2)
```

Parameters

`job`

Identifier of the job for which the stored procedure is to be changed.

`what`

Name of the stored procedure to be executed.

Examples

Change the job to run the `list_emp` procedure:

```
BEGIN
    DBMS_JOB.WHAT(104,'list_emp;');
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.6 DBMS_LOB

The `DBMS_LOB` package provides the capability to operate on large objects. The following table lists the supported functions and procedures:

Function/Procedure	Return Type	Description
`APPEND(dest_lob IN OUT, src_lob)`	n/a	Appends one large object to another.
`COMPARE(lob_1, lob_2 [, amount [, offset_1 [, offset_2 ]]])`	`INTEGER`	Compares two large objects.
`CONVERTOBLOB(dest_lob IN OUT, src_clob, amount, dest_offset IN OUT, src_offset IN OUT, blob_csid, lang_context IN OUT, warning OUT)`	n/a	Converts character data to binary.
`CONVERTTOCLOB(dest_lob IN OUT, src_blob, amount, dest_offset IN OUT, src_offset IN OUT, blob_csid, lang_context IN OUT, warning OUT)`	n/a	Converts binary data to character.
`COPY(dest_lob IN OUT, src_lob, amount [, dest_offset [, src_offset ]])`	n/a	Copies one large object to another.
`ERASE(lob_loc IN OUT, amount IN OUT [, offset ])`	n/a	Erase a large object.
`GET_STORAGE_LIMIT(lob_loc)`	`INTEGER`	Get the storage limit for large objects.
`GETLENGTH(lob_loc)`	`INTEGER`	Get the length of the large object.
`INSTR(lob_loc, pattern [, offset [, nth ]])`	`INTEGER`	Get the position of the nth occurrence of a pattern in the large object starting at `offset`
`READ(lob_loc, amount IN OUT, offset, buffer OUT)`	n/a	Read a large object.
`SUBSTR(lob_loc [, amount [, offset ]])`	`RAW, VARCHAR2`	Get part of a large object.
`TRIM(lob_loc IN OUT, newlen)`	n/a	Trim a large object to the specified length.
`WRITE(lob_loc IN OUT, amount, offset, buffer)`	n/a	Write data to a large object.
`WRITEAPPEND(lob_loc IN OUT, amount, buffer)`	n/a	Write data from the buffer to the end of a large object.

Advanced Server's implementation of `DBMS_LOB` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

The following table lists the public variables available in the package.

Public Variables	Data Type	Value
`compress off`	`INTEGER`	`0`
`compress_on`	`INTEGER`	`1`
`deduplicate_off`	`INTEGER`	`0`
`deduplicate_on`	`INTEGER`	`4`
`default_csid`	`INTEGER`	`0`
`default_lang_ctx`	`INTEGER`	`0`
`encrypt_off`	`INTEGER`	`0`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Public Variables	Data Type	Value
encrypt_on	INTEGER	1
file_readonly	INTEGER	0
lobmaxsize	INTEGER	1073741823
lob_readonly	INTEGER	0
lob_readwrite	INTEGER	1
no_warning	INTEGER	0
opt_compress	INTEGER	1
opt_deduplicate	INTEGER	4
opt_encrypt	INTEGER	2
warn_inconvertible_char	INTEGER	1

In the following sections, lengths and offsets are measured in bytes if the large objects are `BLOBs`. Lengths and offsets are measured in characters if the large objects are `CLOBs`.

11.2.6.1 APPEND

The `APPEND` procedure provides the capability to append one large object to another. Both large objects must be of 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`.

11.2.6.2 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 be compared. Must be the same data type as `lob_2`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`lob_2`

Large object locator of the second large object to be compared. 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 within the first large object to begin the comparison. The first byte/character is offset 1. The default is 1.

`offset_2`

Position within 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. Non-zero, if the objects are not the same. `NULL` if `amount`, `offset_1`, or `offset_2` are less than zero.

11.2.6.3 CONVERTTOBLOB

The `CONVERTTOBLOB` procedure provides the capability to convert 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 the character data is to be converted.

`src_clob`

`CLOB` large object locator of the character data to be converted.

`amount`

Number of characters of `src_clob` to be converted.

`dest_offset IN`

Position in bytes in the destination `BLOB` where writing of the source `CLOB` should begin. The first byte is offset 1.

`dest_offset OUT`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 conversion to the destination `BLOB` should begin. 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 an inconvertible character was encountered.

11.2.6.4 CONVERTTOCLOB

The `CONVERTTOCLOB` procedure provides the capability to convert 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 the binary data is to be converted.

`src_blob`

`BLOB` large object locator of the binary data to be converted.

`amount`

Number of bytes of `src_blob` to be converted.

`dest_offset IN`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Position in characters in the destination `CLOB` where writing of the source `BLOB` should begin. 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 conversion to the destination `CLOB` should 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 an inconvertible character was encountered.

11.2.6.5 COPY

The `COPY` procedure provides the capability to copy 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>
{ BLOB | CLOB },
  <amount> INTEGER
  [, <dest_offset> INTEGER [, <src_offset> INTEGER ]])
```

Parameters

`dest_lob`

Large object locator of the large object to which `src_lob` is to be copied. Must be the same data type as `src_lob`.

`src_lob`

Large object locator of the large object to be copied to `dest_lob`. Must be the same data type as `dest_lob`.

`amount`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Number of bytes/characters of `src_lob` to be copied.

`dest_offset`

Position in the destination large object where writing of the source large object should begin. The first position is offset 1. The default is 1.

`src_offset`

Position in the source large object where copying to the destination large object should begin. The first position is offset 1. The default is 1.

11.2.6.6 ERASE

The `ERASE` procedure provides the capability to erase a portion of a large object. To erase a large object means to replace the specified portion with zero-byte fillers for `BLOBs` or with spaces for `CLOBs`. The actual size of the large object is not altered.

```
ERASE(<lob_loc> IN OUT { BLOB | CLOB }, <amount> IN OUT INTEGER
  [, <offset> INTEGER ])
```

Parameters

`lob_loc`

Large object locator of the large object to be erased.

`amount IN`

Number of bytes/characters to be erased.

`amount OUT`

Number of bytes/characters actually erased. This value can be smaller than the input value if the end of the large object is reached before `amount` bytes/characters have been erased.

`offset`

Position in the large object where erasing is to begin. The first byte/character is position 1. The default is 1.

11.2.6.7 GET_STORAGE_LIMIT

The `GET_STORAGE_LIMIT` function returns the limit on the largest allowable large object.

```
 <size> INTEGER GET_STORAGE_LIMIT(<lob_loc> BLOB)

<size> INTEGER GET_STORAGE_LIMIT(<lob_loc> CLOB)
```

Parameters

`size`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Maximum allowable size of a large object in this database.

`lob_loc`

This parameter is ignored, but is included for compatibility.

11.2.6.8 GETLENGTH

The `GETLENGTH` function returns the length of a large object.

```
<amount> INTEGER GETLENGTH(<lob_loc> BLOB)

<amount> INTEGER GETLENGTH(<lob_loc> CLOB)
```

Parameters

`lob_loc`

Large object locator of the large object whose length is to be obtained.

`amount`

Length of the large object in bytes for `BLOBs` or characters for `CLOBs`.

11.2.6.9 INSTR

The `INSTR` function returns the location of the nth occurrence of a given pattern within a large object.

```
<position> INTEGER INSTR(<lob_loc> { BLOB | CLOB },
   <pattern> { RAW | VARCHAR2 } [, <offset> INTEGER [, <nth> INTEGER ]])
```

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 within `lob_loc` to start search for `pattern`. The first byte/character is position 1. The default is 1.

`nth`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Search for `pattern`, `nth` number of times starting at the position given by `offset`. The default is 1.

`position`

Position within the large object where `pattern` appears the nth time specified by `nth` starting from the position given by `offset`.

11.2.6.10 READ

The `READ` procedure provides the capability to read a portion of a large object into a buffer.

```
READ(<lob_loc> { BLOB | CLOB }, <amount> IN OUT BINARY_INTEGER,
  <offset> INTEGER, <buffer> OUT { RAW | VARCHAR2 })
```

Parameters

`lob_loc`

Large object locator of the large object to be read.

`amount IN`

Number of bytes/characters to read.

`amount OUT`

Number of bytes/characters actually read. If there is no more data to be 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`.

11.2.6.11 SUBSTR

The `SUBSTR` function provides the capability to return 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 be read.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`amount`

Number of bytes/characters to be returned. Default is 32,767.

`offset`

Position within 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 be 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`.

11.2.6.12 TRIM

The `TRIM` procedure provides the capability to truncate 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 be trimmed.

`newlen`

Number of bytes/characters to which the large object is to be trimmed.

11.2.6.13 WRITE

The `WRITE` procedure provides the capability to write 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.

```
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 be written.

`amount`

The number of bytes/characters in `buffer` to be written to the large object.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 be written 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`.

11.2.6.14 WRITEAPPEND

The `WRITEAPPEND` procedure provides the capability to add data to the end of a large object.

```
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 data is to be appended.

`amount`

Number of bytes/characters from `buffer` to be appended the large object.

`buffer`

Data to be appended 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`.

11.2.7 DBMS_LOCK

Advanced Server provides support for the `DBMS_LOCK.SLEEP` procedure.

Function/Procedure	Return Type	Description
`SLEEP(seconds)`	n/a	Suspends a session for the specified number of `seconds`.

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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SLEEP(<seconds> NUMBER)
```

Parameters

`seconds`

`seconds` specifies the number of seconds for which you wish to suspend the session. `seconds` can be a fractional value; for example, enter `1.75` to specify one and three-fourths of a second.

11.2.8 DBMS_MVIEW

Use procedures in the `DBMS_MVIEW` package to manage and refresh materialized views and their dependencies. Advanced Server provides support for the following `DBMS_MVIEW` procedures:

Procedure	Return Type	Description
`GET_MV_DEPENDENCIES(list VARCHAR2, deplist VARCHAR2);`	n/a	The `GET_MV_DEPENDENCIES` procedure returns a list of dependencies for a specified view.
`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);`	n/a	This variation of the `REFRESH` procedure refreshes all views named in a comma-separated list of view names.
`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);`	n/a	This variation of the `REFRESH` procedure refreshes all views named in a table of `dbms_utility.uncl_array` values.
`REFRESH_ALL_MVIEWS(number_of_failures BINARY_INTEGER, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN);`	n/a	The `REFRESH_ALL_MVIEWS` procedure refreshes all materialized views.
`REFRESH_DEPENDENT(number_of_failures BINARY_INTEGER, list VARCHAR2, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN, nested BOOLEAN);`	n/a	This variation of the `REFRESH_DEPENDENT` procedure refreshes all views that are dependent on the views listed in a comma-separated list.
`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);`	n/a	This variation of the `REFRESH_DEPENDENT` procedure refreshes all views that are dependent on the views listed in a table of `dbms_utility.uncl_array` values.

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 above are supported.

11.2.8.1 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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

The following example:

```
DECLARE
  deplist VARCHAR2(1000);
BEGIN
  DBMS_MVIEW.GET_MV_DEPENDENCIES('public.emp_view', deplist);
  DBMS_OUTPUT.PUT_LINE('deplist: ' || deplist);
END;
```

Displays a list of the dependencies on a materialized view named `public.emp_view`.

11.2.8.2 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:

```
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:

```
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,
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  <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 may be schema-qualified.

`tab`

`tab` is a table of `DBMS_UTILITY.UNCL_ARRAY` values that specify the name (or names) of a materialized view.

`method`

`method` is a `VARCHAR2` value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is `C`; this 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example uses `DBMS_MVIEW.REFRESH` to perform a `COMPLETE` refresh on the `public.emp_view` materialized view:

```
EXEC DBMS_MVIEW.REFRESH(list => 'public.emp_view', method => 'C');
```

11.2.8.3 REFRESH_ALL_MVIEWS

Use the `REFRESH_ALL_MVIEWS` procedure to refresh any materialized views that have not been refreshed since the table or view on which the view depends has been modified. The signature is:

```
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 operation.

`method`

`method` is a `VARCHAR2` value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is `C;` this 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

The following 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.8.4 REFRESH_DEPENDENT

Use the `REFRESH_DEPENDENT` procedure to refresh all material views that are dependent 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 are dependent 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 are dependent 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`

`number_of_failures` is a `BINARY_INTEGER` that contains the number of failures that occurred during the refresh operation.

`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 may be schema-qualified.

`tab`

`tab` is a table of `DBMS_UTILITY.UNCL_ARRAY` values that specify the name (or names) of a materialized view.

`method`

`method` is a `VARCHAR2` value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is `C`; this performs a complete refresh of the view.

`rollback_seg`

`rollback_seg` is accepted for compatibility and ignored. The default is `NULL`.

`refresh_after_errors`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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

The following example performs a `COMPLETE` refresh on all materialized views dependent on a materialized view named `emp_view` that resides in the `public` schema:

```
 DECLARE
   errors INTEGER;
BEGIN
   DBMS_MVIEW.REFRESH_DEPENDENT(errors, list => 'public.emp_view', method =>
'C');
END;
```

Upon completion, `errors` contains the number of failures.

11.2.9 DBMS_OUTPUT

The `DBMS_OUTPUT` package provides the capability to send messages (lines of text) to a message buffer, or get 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.

Function/Procedure	Return Type	Description
`DISABLE`	n/a	Disable the capability to send and receive messages.
`ENABLE(buffer_size)`	n/a	Enable the capability to send and receive messages.
`GET_LINE(line OUT, status OUT)`	n/a	Get a line from the message buffer.
`GET_LINES(lines OUT, numlines IN OUT)`	n/a	Get multiple lines from the message buffer.
`NEW_LINE`	n/a	Puts an end-of-line character sequence.
`PUT(item)`	n/a	Puts a partial line without an end-of-line character sequence.
`PUT_LINE(item)`	n/a	Puts a complete line with an end-of-line character sequence.
`SERVEROUTPUT(stdout)`	n/a	Direct messages from `PUT, PUT_LINE,` or `NEW_LINE` to either standard output or the message buffer.

The following table lists the public variables available in the `DBMS_OUTPUT` package.

Public Variables	Data Type	Value	Description

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Public Variables	Data Type	Value	Description
chararr	TABLE		For message lines.

CHARARR

The CHARARR is for storing multiple message lines.

```
TYPE chararr IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

DISABLE

The DISABLE procedure clears out the message buffer. Any messages in the buffer at the time the DISABLE procedure is executed will no longer be accessible. Any messages subsequently 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 have been disabled.

Use the ENABLE procedure or SERVEROUTPUT(TRUE) procedure to re-enable the sending and receiving of messages.

```
DISABLE
```

Examples

This anonymous block disables the sending and receiving messages in the current session.

```
BEGIN
    DBMS_OUTPUT.DISABLE;
END;
```

ENABLE

The ENABLE procedure enables the capability to send messages to the message buffer 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 upon 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.

```
ENABLE [ (<buffer_size> INTEGER) ]
```

Parameter

buffer_size

Maximum length of the message buffer in bytes. If a buffer_size of less than 2000 is specified, the buffer size is set to 2000.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Examples

The following anonymous block enables messages. Setting `SERVEROUTPUT(TRUE)` forces them to standard output.

```
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Messages enabled');
END;

Messages enabled
```

The same effect could have been achieved by simply using `SERVEROUTPUT(TRUE)` .

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Messages enabled');
END;

Messages enabled
```

The following anonymous block enables messages, but setting `SERVEROUTPUT(FALSE)` directs messages to the message buffer.

```
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    DBMS_OUTPUT.PUT_LINE('Message sent to buffer');
END;
```

GET_LINE

The `GET_LINE` procedure provides the capability to retrieve a line of text from the message buffer. Only text that has been terminated by an end-of-line character sequence is retrieved – that is complete lines generated using `PUT_LINE` , or by 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

The following anonymous block writes the `emp` table out to the message buffer as a comma-delimited string for each row.

```
EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DECLARE
    v_emprec        VARCHAR2(120);
    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;
```

The following 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.

```
 CREATE TABLE messages (
    status          INTEGER,
    msg             VARCHAR2(100)
);

DECLARE
    v_line          VARCHAR2(100);
    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;

                              msg
-----------------------------------------------------------------
 7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
 7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
 7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
 7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
 7782,CLARK,MANAGER,7839,09-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,7698,08-SEP-81 00:00:00,1500.00,0.00,30
 7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
 7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
 7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
 7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
(14 rows)
```

GET_LINES

The `GET_LINES` procedure provides the capability to retrieve one or more lines of text from the message buffer into a collection. Only text that has been terminated by an end-of-line character sequence is retrieved – that is complete lines generated using `PUT_LINE`, or by a series of `PUT` calls followed by

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

a `NEW_LINE` call.

```
GET_LINES(<lines> OUT CHARARR, <numlines> IN OUT INTEGER)
```

Parameters

`lines`

Table receiving the lines of text from the message buffer. See `CHARARR` for a description of `lines.`

`numlines IN`

Number of lines to be retrieved from the message buffer.

`numlines OUT`

Actual number of lines retrieved from the message buffer. If the output value of `numlines` is less than the input value, then there are no more lines left in the message buffer.

Examples

The following example uses the `GET_LINES` procedure to store all rows from the `emp` table that were placed on the message buffer, into an array.

```
 EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE
    v_emprec        VARCHAR2(120);
    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;
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;

                              msg
----------------------------------------------------------------
 7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
 7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
 7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
 7782,CLARK,MANAGER,7839,09-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,7698,08-SEP-81 00:00:00,1500.00,0.00,30
 7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
 7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
 7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
 7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
(14 rows)
```

NEW_LINE

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.

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(i.empno);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.ename);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.job);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.mgr);
        DBMS_OUTPUT.PUT(',');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        DBMS_OUTPUT.PUT(i.hiredate);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.sal);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.comm);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.deptno);
        DBMS_OUTPUT.NEW_LINE;
    END LOOP;
END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-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,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

PUT_LINE

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 be written to the message buffer.

Examples

The following example uses the `PUT_LINE` procedure to display a comma-delimited list of employees from the `emp` table.

```
DECLARE
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-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,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

SERVEROUTPUT

The `SERVEROUTPUT` procedure provides the capability to direct 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` is implementation dependent. For example, in Oracle SQL*Plus, `SERVEROUTPUT(FALSE)` is the default. In PSQL, `SERVEROUTPUT(TRUE)` is the default. Also note that in Oracle SQL*Plus, this setting is controlled using the SQL*Plus `SET` command, not by a stored procedure as implemented in Advanced Server.

```
SERVEROUTPUT(<stdout> BOOLEAN)
```

Parameter

`stdout`

Set to `TRUE` if subsequent `PUT, PUT_LINE`, or `NEW_LINE` commands are to send text directly to standard output of the command line. Set to `FALSE` if text is to be sent to the message buffer.

Examples

The following anonymous block sends the first message to the command line and the second message to the message buffer.

```
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 within the same session, the following anonymous block is executed, the message stored in the message buffer from the prior example is flushed and displayed on the command line as well as the new message.

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Flush messages from the buffer');
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
This message goes to the message buffer
Flush messages from the buffer
```

11.2.10 DBMS_PIPE

The `DBMS_PIPE` package provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.

The procedures and functions available in the `DBMS_PIPE` package are listed in the following table:

Function/Procedure	Return Type	Description
`CREATE_PIPE(pipename [, maxpipesize ] [, private ])`	`INTEGER`	Explicitly create a private pipe if `private` is "true" (the default) or a public pipe if `private` is "false".
`NEXT_ITEM_TYPE`	`INTEGER`	Determine the data type of the next item in a received message.
`PACK_MESSAGE(item)`	n/a	Place `item` in the session's local message buffer.
`PURGE(pipename)`	n/a	Remove unreceived messages from the specified pipe.
`RECEIVE_MESSAGE(pipename [, timeout ])`	`INTEGER`	Get a message from a specified pipe.
`REMOVE_PIPE(pipename)`	`INTEGER`	Delete an explicitly created pipe.
`RESET_BUFFER`	n/a	Reset the local message buffer.
`SEND_MESSAGE(pipename [, timeout ] [, maxpipesize ])`	`INTEGER`	Send a message on a pipe.
`UNIQUE_SESSION_NAME`	`VARCHAR2`	Obtain a unique session name.
`UNPACK_MESSAGE(item OUT)`	n/a	Retrieve the next data item from a message into a type-compatible variable, `item`.

Pipes are categorized as implicit or explicit. An *implicit pipe* is created if a reference is made to a pipe name that was not previously created by the `CREATE_PIPE` function. For example, if the `SEND_MESSAGE` function is executed using a non-existent pipe name, a new implicit pipe is created with that name. An *explicit pipe* is created using the `CREATE_PIPE` function whereby the first parameter specifies the pipe name for the new pipe.

Pipes are also categorized as private or public. A *private pipe* can only be accessed by the user who created the pipe. Even a superuser cannot access a private pipe that was created by another user. A *public pipe* can be accessed by any user who has access to the `DBMS_PIPE` package.

A public pipe can only be created by using the `CREATE_PIPE` function with the third parameter set to `FALSE`. The `CREATE_PIPE` function can be used 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. The `SEND_MESSAGE` function is then used to send the message through the pipe.

Receipt of a message involves the reverse operation. The `RECEIVE_MESSAGE` function is used to get a message from the specified pipe. The message is written to the session's local message buffer. The `UNPACK_MESSAGE` procedure is then used 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 will be preserved in the local message buffer.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.10.1 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`

Status code returned by the operation. 0 indicates successful creation.

Examples

The following example creates a private pipe named `messages:`

```
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
```

The following example creates a public pipe named `mailbox:`

```
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
```

11.2.10.2 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 has been retrieved into the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 will return the data type code for the next available item. A code of 0 is returned when there are no more items left in the message.

```
<typecode> INTEGER NEXT_ITEM_TYPE
```

Parameters

`typecode`

Code identifying the data type of the next data item as shown in the following table.

Type Code	Data Type
`0`	No more data items
`9`	`NUMBER`
`11`	`VARCHAR2`
`13`	`DATE`
`23`	`RAW`

Note: The type codes list in the table are not compatible with Oracle databases. Oracle assigns a different numbering sequence to the data types.

Examples

The following 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.

```
DECLARE
    v_number        NUMBER := 123;
    v_varchar       VARCHAR2(20) := 'Character data';
    v_date          DATE := SYSDATE;
    v_raw           RAW(4) := '21222324';
    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;

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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('datatypes');
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
    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('---------------------------------');

    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
    DBMS_OUTPUT.PUT_LINE('NEXT_ITEM_TYPE: ' || v_status);
    DBMS_OUTPUT.PUT_LINE('---------------------------------');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

RECEIVE_MESSAGE status: 0
---------------------------------
NEXT_ITEM_TYPE: 9
NUMBER Item   : 123
---------------------------------
NEXT_ITEM_TYPE: 11
VARCHAR2 Item : Character data
---------------------------------
NEXT_ITEM_TYPE: 13
DATE Item     : 02-OCT-07 11:11:43
---------------------------------
NEXT_ITEM_TYPE: 23
RAW Item      : 21222324
---------------------------------
NEXT_ITEM_TYPE: 0
```

11.2.10.3 PACK_MESSAGE

The `PACK_MESSAGE` procedure places an item of data in the session's local message buffer. `PACK_MESSAGE` must be executed at least once before issuing a `SEND_MESSAGE` call.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
PACK_MESSAGE(<item> { DATE | NUMBER | VARCHAR2 | RAW })
```

Use the `UNPACK_MESSAGE` procedure to obtain data items once the message is retrieved using a `RECEIVE_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.

11.2.10.4 PURGE

The `PURGE` procedure removes the unreceived messages from a specified implicit pipe.

```
PURGE(<pipename> VARCHAR2)
```

Use the `REMOVE_PIPE` function to delete an explicit pipe.

Parameters

`pipename`

Name of the pipe.

Examples

Two messages are sent on a pipe:

```
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:

```
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);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
END;

RECEIVE_MESSAGE status: 0
Item: Message #1
```

Purge the pipe:

```
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.

```
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
```

11.2.10.5 RECEIVE_MESSAGE

The `RECEIVE_MESSAGE` function obtains a message from a specified pipe.

```
<status> INTEGER RECEIVE_MESSAGE(<pipename> VARCHAR2
   [, <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:

Status Code	Description
`0`	Success
`1`	Time out
`2`	Message too large for the buffer

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.10.6 REMOVE_PIPE

The `REMOVE_PIPE` function deletes an explicit private or explicit public pipe.

```
<status> INTEGER REMOVE_PIPE(<pipename> VARCHAR2)
```

Use the `REMOVE_PIPE` function to delete explicitly created pipes – i.e., 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 non-existent.

Examples

Two messages are sent on a pipe:

```
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);
END;

CREATE_PIPE status : 0
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

Receive the first message and unpack it:

```
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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Remove the pipe:

```
SELECT DBMS_PIPE.REMOVE_PIPE('pipe') FROM DUAL;

remove_pipe
-------------
        0
(1 row)
```

Try to retrieve the next message. The `RECEIVE_MESSAGE` call returns status code 1 indicating it timed out because the pipe had been 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;

RECEIVE_MESSAGE status: 1
```

11.2.10.7 RESET_BUFFER

The `RESET_BUFFER` procedure resets a "pointer" to the session's local message buffer back to the beginning of the buffer. This has the effect of causing subsequent `PACK_MESSAGE` calls to overwrite any data items that existed in the message buffer prior to 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?');
    DBMS_PIPE.RESET_BUFFER;
    DBMS_PIPE.PACK_MESSAGE('Hi, Bob');
    DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 9:30, tomorrow?');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END;

SEND_MESSAGE status: 0
```

The message to Bob is in the received message.

```
DECLARE
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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);
    DBMS_PIPE.UNPACK_MESSAGE(v_item);
    DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END;

RECEIVE_MESSAGE status: 0
Item: Hi, Bob
Item: Can you attend a meeting at 9:30, tomorrow?
```

11.2.10.8 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:

Status Code	Description
`0`	Success
`1`	Time out
`3`	Function interrupted

11.2.10.9 UNIQUE_SESSION_NAME

The `UNIQUE_SESSION_NAME` function returns a name, unique to the current session.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
<name> VARCHAR2 UNIQUE_SESSION_NAME
```

Parameters

`name`

Unique session name.

Examples

The following anonymous block retrieves and displays a unique session name.

```
DECLARE
    v_session       VARCHAR2(30);
BEGIN
    v_session := DBMS_PIPE.UNIQUE_SESSION_NAME;
    DBMS_OUTPUT.PUT_LINE('Session Name: ' || v_session);
END;

Session Name: PG$PIPE$5$2752
```

11.2.10.10 UNPACK_MESSAGE

The `UNPACK_MESSAGE` procedure copies the data items of a message from the local message buffer to a specified program variable. The message must be placed in the local message buffer with the `RECEIVE_MESSAGE` function before using `UNPACK_MESSAGE`.

```
UNPACK_MESSAGE(<item> OUT { DATE | NUMBER | VARCHAR2 | RAW })
```

Parameters

`item`

Type-compatible variable that receives a data item from the local message buffer.

11.2.10.11 Comprehensive Example

The following example uses a pipe as a “mailbox”. The procedures to create the mailbox, add a multi-item message to the mailbox (up to three items), and display the full contents of the mailbox are enclosed in a package named, `mailbox`.

```
CREATE OR REPLACE PACKAGE mailbox
IS
    PROCEDURE create_mailbox;
    PROCEDURE add_message (
        p_mailbox   VARCHAR2,
        p_item_1    VARCHAR2,
        p_item_2    VARCHAR2 DEFAULT 'END',
        p_item_3    VARCHAR2 DEFAULT 'END'
    );
    PROCEDURE empty_mailbox (
        p_mailbox   VARCHAR2,
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        p_waittime  INTEGER DEFAULT 10
    );
END mailbox;

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 (
        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 (
        p_mailbox   VARCHAR2,
        p_waittime  INTEGER DEFAULT 10
    )
    IS
        v_msgno     INTEGER DEFAULT 0;
        v_itemno    INTEGER DEFAULT 0;
        v_item      VARCHAR2(100);
        v_status    INTEGER;
    BEGIN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        v_status := 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 ||
                    ' *******');
                DBMS_OUTPUT.PUT_LINE('*');
                v_itemno := 0;
                v_status := DBMS_PIPE.RECEIVE_MESSAGE(p_mailbox,1);
            END;
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('Number of messages received: ' || v_msgno);
        v_status := DBMS_PIPE.REMOVE_PIPE(p_mailbox);
        IF v_status = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Deleted mailbox ' || p_mailbox);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Could not delete mailbox - status: '
                || v_status);
        END IF;
    END empty_mailbox;
END mailbox;
```

The following demonstrates the execution of the procedures in `mailbox`. The first procedure creates a public pipe using a name generated by the `UNIQUE_SESSION_NAME` function.

```
 EXEC mailbox.create_mailbox;

Created mailbox: PG$PIPE$13$3940
```

Using the mailbox name, any user in the same database with access to the `mailbox` package and `DBMS_PIPE` package can add messages:

```
 EXEC mailbox.add_message('PG$PIPE$13$3940','Hi, John','Can you attend a
meeting at 3:00, today?','-- Mary');

Added message with 3 item(s) to mailbox PG$PIPE$13$3940

EXEC mailbox.add_message('PG$PIPE$13$3940','Don''t forget to submit your
report','Thanks,','-- Joe');

Added message with 3 item(s) to mailbox PG$PIPE$13$3940
```

Finally, the contents of the mailbox can be emptied:

```
 EXEC mailbox.empty_mailbox('PG$PIPE$13$3940');

****** Start message #1 ******
Item #1: Hi, John
Item #2: Can you attend a meeting at 3:00, today?
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
Item #3: -- Mary
******* End message #1 *******
*
****** Start message #2 ******
Item #1: Don't forget to submit your report
Item #2: Thanks,
Item #3: Joe
******* End message #2 *******
*
Number of messages received: 2
Deleted mailbox PG$PIPE$13$3940
```

11.2.11 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 the functions and procedures listed below to control the profiling tool.

Function/Procedure	Return Type	Description
`FLUSH_DATA`	Status Code or Exception	Flushes performance data collected in the current session without terminating the session (profiling continues).
`GET_VERSION(major OUT, minor OUT)`	n/a	Returns the version number of this package.
`INTERNAL_VERSION_CHECK`	Status Code	Confirms that the current version of the profiler will work with the current database.
`PAUSE_PROFILER`	Status Code or Exception	Pause data collection.
`RESUME_PROFILER`	Status Code or Exception	Resume data collection.
`START_PROFILER(run_comment, run_comment1 [, run_number OUT ])`	Status Code or Exception	Start data collection.
`STOP_PROFILER`	Status Code or Exception	Stop data collection and flush performance data to the `PLSQL_PROFILER_RAWDATA` table.

The functions within 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 status codes and messages returned by the functions, and the exceptions raised by the procedures are listed in the table below.

Status Code	Message	Exception	Description
`-1`	`error version`	`version_mismatch`	The profiler version and the database are incompatible.
`0`	`success`	n/a	The operation completed successfully.
`1`	`error_param`	`profiler_error`	The operation received an incorrect parameter.
`2`	`error_io`	`profiler_error`	The data flush operation has failed.

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

tables described in the Advanced Server Performance Features Guide. The function and procedure signatures are:

```
<status> INTEGER FLUSH_DATA

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:

```
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` will work 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
PAUSE_PROFILER
```

Parameters

`status`

Status code returned by the operation.

RESUME_PROFILER

The `RESUME_PROFILER` function/procedure pauses a profiling session. The function and procedure signatures are:

```
<status> INTEGER RESUME_PROFILER

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:

```
<status> INTEGER START_PROFILER(<run_comment> TEXT := SYSDATE,
   <run_comment1> TEXT := '' [, <run_number> OUT INTEGER ])

START_PROFILER(<run_comment> TEXT := SYSDATE,
  <run_comment1> TEXT := '' [, <run_number> OUT INTEGER ])
```

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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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:

```
 <status> INTEGER STOP_PROFILER

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 within 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` will write a row to the performance statistics table for each line of PL/pgSQL or SPL code that executed within the session. For example, if you execute the following function:

```
 1 - CREATE OR REPLACE FUNCTION getBalance(acctNumber INTEGER)
2 - RETURNS NUMERIC AS $$
3 - DECLARE
4 -   result NUMERIC;
5 - BEGIN
6 -   SELECT INTO result balance FROM acct WHERE id = acctNumber;
7 -
8 -   IF (result IS NULL) THEN
9 -       RAISE INFO 'Balance is null';
10-   END IF;
11-
12-   RETURN result;
13- END;
14- $$ LANGUAGE 'plpgsql';
```

`DBMS_PROFILER` adds one `PLSQL_PROFILER_DATA` entry for each line of code within 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 will always contain zero values.

To start a profiling session, invoke the `DBMS_PROFILER.START_PROFILER` function (or procedure). Once you've invoked `START_PROFILER`, Advanced Server will profile 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`).

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

It is important to note that when you start (or resume) the profiler, the profiler will only gather performance statistics for functions/procedures/triggers that start after the call to `START_PROFILER` (or `RESUME_PROFILER`).

While the profiler is active, 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 within 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 within a session.

In addition, `DBMS_PROFILER` defines a view, `SYS.PLSQL_PROFILER_DATA`, which contains a subset of the `PLSQL_PROFILER_RAWDATA` table.

Please note that a non-superuser may gather profiling information, but may not view that profiling information unless a superuser grants specific privileges on the profiling tables (stored in the `SYS` schema). This permits a non-privileged user to gather performance statistics without exposing information that the administrator may want to keep secret.

Querying the DBMS_PROFILER Tables and View

The following step-by-step example uses `DBMS_PROFILER` to retrieve performance information for procedures, functions, and triggers included in the sample data distributed with Advanced Server.

1. Open the EDB-PSQL command line, and establish a connection to the 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. Then 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
INFO:  7900     JAMES
INFO:  7902     FORD
INFO:  7934     MILLER
 list_emp
----------

(1 row)
```

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 = 7902;
INFO: Updating employee 7902
INFO: ..Old salary: 3000.00
INFO: ..New salary: 500.00
INFO: ..Raise: -2500.00
INFO: User enterprisedb updated employee(s) on 04-FEB-14
UPDATE 1
```

7. Terminate the profiling session and flush the performance information to the profiling tables:

```
acctg=# EXEC dbms_profiler.stop_profiler;

EDB-SPL Procedure successfully completed
```

8. Now, query the `plsql_profiler_runs` table to view a list of the profiling sessions, arranged by `runid:`

```
acctg=# SELECT * FROM plsql_profiler_runs;
 runid | related_run |  run_owner   |         run_date
 |              run_comment               | run_total_time | run_system_info
 | run_comment1 | spare1
-------+-------------+--------------+---------------------------+-------------
---------------------------+----------------+-----------------+--------------
+--------
      1 |             | enterprisedb | 04-FEB-14 09:32:48.874315 | profile
      list_emp                        |           4154 |
      |              |
      2 |             | enterprisedb | 04-FEB-14 09:41:30.546503 | profile
      get_dept_name and emp_sal_trig |           2088 |
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
      |              |
(2 rows)
```

9. Query the `plsql_profiler_units` table to view the amount of time consumed by each unit (each function, procedure, or trigger):

```
acctg=# SELECT * FROM plsql_profiler_units;
 runid | unit_number | unit_type |  unit_owner  |
 unit_name            | unit_timestamp | total_time | spare1 | spare2
-------+-------------+-----------+--------------+----------------------------
-----+----------------+------------+--------+--------
     1 |       16999 | FUNCTION  | enterprisedb |
     list_emp()                     |                |          4 |        |
     2 |       17002 | FUNCTION  | enterprisedb |
     user_audit_trig()              |                |          1 |        |
     2 |       17000 | FUNCTION  | enterprisedb | get_dept_name(p_deptno
     numeric) |                |          1 |        |
     2 |       17004 | FUNCTION  | enterprisedb |
     emp_sal_trig()                 |                |          1 |        |
(4 rows)
```

10. Query the `plsql_profiler_rawdata` table to view a list of the wait event counters and wait event times:

```
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         NUMERIC(4);
    |    16999 |           2 |          0 |               0 |          0
    1 |     v_ename         VARCHAR(10);
    |    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    ENAME';
    |    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 '%     %', v_empno, v_ename;
    |    16999 |          13 |         15 |               0 |    8.3e-05
    1 |     END LOOP;
    |    16999 |          14 |         14 |               0 |   0.000773
    1 |     CLOSE emp_cur;
    |    16999 |          15 |          0 |               0 |          0
    1 |     RETURN;
    |    16999 |          16 |          1 |               0 |      1e-05
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
1 | END;
|    16999 |          17 |          1 |               0 |          0
1 |
|    16999 |          18 |          0 |               0 |          0
2 | DECLARE
|    17002 |           1 |          0 |               0 |          0
2 |     v_action        VARCHAR(24);
|    17002 |           2 |          0 |               0 |          0
2 |     v_text          TEXT;
|    17002 |           3 |          0 |               0 |          0
2 | BEGIN
|    17002 |           4 |          0 |               0 |          0
2 |     IF TG_OP = 'INSERT' THEN
|    17002 |           5 |          0 |               0 |          0
2 |         v_action := ' added employee(s) on ';
|    17002 |           6 |          1 |               0 |   0.000143
2 |     ELSIF TG_OP = 'UPDATE' THEN
|    17002 |           7 |          0 |               0 |          0
2 |         v_action := ' updated employee(s) on ';
|    17002 |           8 |          0 |               0 |          0
2 |     ELSIF TG_OP = 'DELETE' THEN
|    17002 |           9 |          1 |               0 |    3.2e-05
2 |         v_action := ' deleted employee(s) on ';
|    17002 |          10 |          0 |               0 |          0
2 |     END IF;
|    17002 |          11 |          0 |               0 |          0
2 |     v_text := 'User ' || USER || v_action || CURRENT_DATE;
|    17002 |          12 |          0 |               0 |          0
2 |     RAISE INFO ' %', v_text;
|    17002 |          13 |          1 |               0 |   0.000383
2 |     RETURN NULL;
|    17002 |          14 |          1 |               0 |    6.3e-05
2 | END;
|    17002 |          15 |          1 |               0 |    3.6e-05
2 |
|    17002 |          16 |          0 |               0 |          0
2 | DECLARE
|    17000 |           1 |          0 |               0 |          0
2 |     v_dname         VARCHAR(14);
|    17000 |           2 |          0 |               0 |          0
2 | BEGIN
|    17000 |           3 |          0 |               0 |          0
2 |     SELECT INTO v_dname dname FROM dept WHERE deptno = p_deptno;
|    17000 |           4 |          0 |               0 |          0
2 |     RETURN v_dname;
|    17000 |           5 |          1 |               0 |   0.000647
2 |     IF NOT FOUND THEN
|    17000 |           6 |          1 |               0 |    2.6e-05
2 |         RAISE INFO 'Invalid department number %', p_deptno;
|    17000 |           7 |          0 |               0 |          0
2 |         RETURN '';
|    17000 |           8 |          0 |               0 |          0
2 |     END IF;
|    17000 |           9 |          0 |               0 |          0
2 | END;
|    17000 |          10 |          0 |               0 |          0
2 |
|    17000 |          11 |          0 |               0 |          0
2 | DECLARE
|    17004 |           1 |          0 |               0 |          0
2 |     sal_diff       NUMERIC(7,2);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    |    17004 |           2 |          0 |               0 |          0
    2 | BEGIN
    |    17004 |           3 |          0 |               0 |          0
    2 |    IF TG_OP = 'INSERT' THEN
    |    17004 |           4 |          0 |               0 |          0
    2 |        RAISE INFO 'Inserting employee %', NEW.empno;
    |    17004 |           5 |          1 |               0 |    8.4e-05
    2 |        RAISE INFO '..New salary: %', NEW.sal;
    |    17004 |           6 |          0 |               0 |          0
    2 |        RETURN NEW;
    |    17004 |           7 |          0 |               0 |          0
    2 |    END IF;
    |    17004 |           8 |          0 |               0 |          0
    2 |    IF TG_OP = 'UPDATE' THEN
    |    17004 |           9 |          0 |               0 |          0
    2 |        sal_diff := NEW.sal - OLD.sal;
    |    17004 |          10 |          1 |               0 |   0.000355
    2 |        RAISE INFO 'Updating employee %', OLD.empno;
    |    17004 |          11 |          1 |               0 |   0.000177
    2 |        RAISE INFO '..Old salary: %', OLD.sal;
    |    17004 |          12 |          1 |               0 |    5.5e-05
    2 |        RAISE INFO '..New salary: %', NEW.sal;
    |    17004 |          13 |          1 |               0 |    3.1e-05
    2 |        RAISE INFO '..Raise     : %', sal_diff;
    |    17004 |          14 |          1 |               0 |    2.8e-05
    2 |        RETURN NEW;
    |    17004 |          15 |          1 |               0 |    2.7e-05
    2 |    END IF;
    |    17004 |          16 |          1 |               0 |      1e-06
    2 |    IF TG_OP = 'DELETE' THEN
    |    17004 |          17 |          0 |               0 |          0
    2 |        RAISE INFO 'Deleting employee %', OLD.empno;
    |    17004 |          18 |          0 |               0 |          0
    2 |        RAISE INFO '..Old salary: %', OLD.sal;
    |    17004 |          19 |          0 |               0 |          0
    2 |        RETURN OLD;
    |    17004 |          20 |          0 |               0 |          0
    2 |    END IF;
    |    17004 |          21 |          0 |               0 |          0
    2 | END;
    |    17004 |          22 |          0 |               0 |          0
    2 |
    |    17004 |          23 |          0 |               0 |          0
 (68 rows)
```

11. Query the `plsql_profiler_data` view to review a subset of the information found in `plsql_profiler_rawdata` table:

```
 acctg=# SELECT * FROM plsql_profiler_data;
runid | unit_number | line# | total_occur | total_time | min_time | max_time
| spare1 | spare2 | spare3 | spare4
-------+-------------+-------+-------------+------------+----------+---------
-+--------+--------+--------+--------
     1 |       16999 |     1 |           0 |          0 |        0 |
     0 |        |        |        |
     1 |       16999 |     2 |           0 |          0 |        0 |
     0 |        |        |        |
     1 |       16999 |     3 |           0 |          0 |        0 |
     0 |        |        |        |
     1 |       16999 |     4 |           0 |          0 |        0 |
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
0 |        |        |        |
1 |       16999 |     5 |           0 |           0 |        0 |
0 |        |        |        |
1 |       16999 |     6 |           0 |           0 |        0 |
0 |        |        |        |
1 |       16999 |     7 |           0 |           0 |        0 |
0 |        |        |        |
1 |       16999 |     8 |           1 |   0.001621 | 0.001621 |
0.001621 |        |        |        |
1 |       16999 |     9 |           1 |   0.000301 | 0.000301 |
0.000301 |        |        |        |
1 |       16999 |    10 |           1 |    4.6e-05 |  4.6e-05 |  4.6e-
05 |        |        |        |
1 |       16999 |    11 |           1 |   0.001114 | 0.001114 |
0.001114 |        |        |        |
1 |       16999 |    12 |          15 |   0.000206 |    5e-06 |  7.8e-
05 |        |        |        |
1 |       16999 |    13 |          15 |    8.3e-05 |    2e-06 |  4.7e-
05 |        |        |        |
1 |       16999 |    14 |          14 |   0.000773 |  4.7e-05 |
0.000116 |        |        |        |
1 |       16999 |    15 |           0 |           0 |        0 |
0 |        |        |        |
1 |       16999 |    16 |           1 |      1e-05 |    1e-05 |    1e-
05 |        |        |        |
1 |       16999 |    17 |           1 |           0 |        0 |
0 |        |        |        |
1 |       16999 |    18 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |     1 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |     2 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |     3 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |     4 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |     5 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |     6 |           1 |   0.000143 | 0.000143 |
0.000143 |        |        |        |
2 |       17002 |     7 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |     8 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |     9 |           1 |    3.2e-05 |  3.2e-05 |  3.2e-
05 |        |        |        |
2 |       17002 |    10 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |    11 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |    12 |           0 |           0 |        0 |
0 |        |        |        |
2 |       17002 |    13 |           1 |   0.000383 | 0.000383 |
0.000383 |        |        |        |
2 |       17002 |    14 |           1 |    6.3e-05 |  6.3e-05 |  6.3e-
05 |        |        |        |
2 |       17002 |    15 |           1 |    3.6e-05 |  3.6e-05 |  3.6e-
05 |        |        |        |
2 |       17002 |    16 |           0 |           0 |        0 |
0 |        |        |        |
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 2 |       17000 |     1 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17000 |     2 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17000 |     3 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17000 |     4 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17000 |     5 |           1 |   0.000647 | 0.000647 |
0.000647 |        |        |        |
 2 |       17000 |     6 |           1 |    2.6e-05 |  2.6e-05 |  2.6e-
05 |        |        |        |
 2 |       17000 |     7 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17000 |     8 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17000 |     9 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17000 |    10 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17000 |    11 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |     1 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |     2 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |     3 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |     4 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |     5 |           1 |    8.4e-05 |  8.4e-05 |  8.4e-
05 |        |        |        |
 2 |       17004 |     6 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |     7 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |     8 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |     9 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |    10 |           1 |   0.000355 | 0.000355 |
0.000355 |        |        |        |
 2 |       17004 |    11 |           1 |   0.000177 | 0.000177 |
0.000177 |        |        |        |
 2 |       17004 |    12 |           1 |    5.5e-05 |  5.5e-05 |  5.5e-
05 |        |        |        |
 2 |       17004 |    13 |           1 |    3.1e-05 |  3.1e-05 |  3.1e-
05 |        |        |        |
 2 |       17004 |    14 |           1 |    2.8e-05 |  2.8e-05 |  2.8e-
05 |        |        |        |
 2 |       17004 |    15 |           1 |    2.7e-05 |  2.7e-05 |  2.7e-
05 |        |        |        |
 2 |       17004 |    16 |           1 |      1e-06 |    1e-06 |    1e-
06 |        |        |        |
 2 |       17004 |    17 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |    18 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |    19 |           0 |          0 |        0 |
 0 |        |        |        |
 2 |       17004 |    20 |           0 |          0 |        0 |
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
     0 |        |        |        |
     2 |       17004 |    21 |          0 |          0 |        0 |
     0 |        |        |        |
     2 |       17004 |    22 |          0 |          0 |        0 |
     0 |        |        |        |
     2 |       17004 |    23 |          0 |          0 |        0 |
     0 |        |        |        |
(68 rows)
```

DBMS_PROFILER - Reference

The Advanced Server installer creates the following tables and views that you can query to review PL/SQL performance profile information:

Table Name	Description
`PLSQL_PROFILER_RUNS`	Table containing information about all profiler runs, organized by `runid` .
`PLSQL_PROFILER_UNITS`	Table containing information about all profiler runs, organized by unit.
`PLSQL_PROFILER_DATA`	View containing performance statistics.
`PLSQL_PROFILER_RAWDATA`	Table containing the performance statistics `and` the extended performance statistics for DRITA counters and timers.

PLSQL_PROFILER_RUNS

The `PLSQL_PROFILER_RUNS` table contains the following columns:

Column	Data Type	Description
`runid`	`INTEGER (NOT NULL)`	Unique identifier (`plsql_profiler_runnumber`)
`related_run`	`INTEGER`	The `runid` of a related run.
`run_owner`	`TEXT`	The role that recorded the profiling session.
`run_date`	`TIMESTAMP WITHOUT TIME ZONE`	The profiling session start time.
`run_comment`	`TEXT`	User comments relevant to this run
`run_total_time`	`BIGINT`	Run time (in microseconds)
`run_system_info`	`TEXT`	Currently Unused
`run_comment1`	`TEXT`	Additional user comments
`spare1`	`TEXT`	Currently Unused

PLSQL_PROFILER_UNITS

The `PLSQL_PROFILER_UNITS` table contains the following columns:

Column	Data Type	Description
`runid`	`INTEGER`	Unique identifier (`plsql_profiler_runnumber`)
`unit_number`	`OID`	Corresponds to the OID of the row in the pg_proc table that identifies the unit.
`unit_type`	`TEXT`	PL/SQL function, procedure, trigger or anonymous block

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Data Type	Description
unit_owner	TEXT	The identity of the role that owns the unit.
unit_name	TEXT	The complete signature of the unit.
unit_timestamp	TIMESTAMP WITHOUT TIME ZONE	Creation date of the unit (currently NULL).
total_time	BIGINT	Time spent within the unit (in milliseconds)
spare1	BIGINT	Currently Unused
spare2	BIGINT	Currently Unused

PLSQL_PROFILER_DATA

The PLSQL_PROFILER_DATA view contains the following columns:

Column	Data Type	Description
runid	INTEGER	Unique identifier (plsql_profiler_runnumber)
unit_number	OID	Object ID of the unit that contains the current line.
line#	INTEGER	Current line number of the profiled workload.
total_occur	BIGINT	The number of times that the line was executed.
total_time	DOUBLE PRECISION	The amount of time spent executing the line (in seconds)
min_time	DOUBLE PRECISION	The minimum execution time for the line.
max_time	DOUBLE PRECISION	The maximum execution time for the line.
spare1	NUMBER	Currently Unused
spare2	NUMBER	Currently Unused
spare3	NUMBER	Currently Unused
spare4	NUMBER	Currently Unused

PLSQL_PROFILER_RAWDATA

The PLSQL_PROFILER_RAWDATA table contains the statistical and wait events information that is found in the PLSQL_PROFILER_DATA view, as well as the performance statistics returned by the DRITA counters and timers.

Column	Data Type	Description
runid	INTEGER	The run identifier (plsql_profiler_runnumber).
sourcecode	TEXT	The individual line of profiled code.
func_oid	OID	Object ID of the unit that contains the current line.
line_number	INTEGER	Current line number of the profiled workload.
exec_count	BIGINT	The number of times that the line was executed.
tuples_returned	BIGINT	Currently Unused

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Data Type	Description
time_total	DOUBLE PRECISION	The amount of time spent executing the line (in seconds)
time_shortest	DOUBLE PRECISION	The minimum execution time for the line.
time_longest	DOUBLE PRECISION	The maximum execution time for the line.
num_scans	BIGINT	Currently Unused
tuples_fetched	BIGINT	Currently Unused
tuples_inserted	BIGINT	Currently Unused
tuples_updated	BIGINT	Currently Unused
tuples_deleted	BIGINT	Currently Unused
blocks_fetched	BIGINT	Currently Unused
blocks_hit	BIGINT	Currently Unused
wal_write	BIGINT	A server has waited for a write to the write-ahead log buffer (expect this value to be high).
wal_flush	BIGINT	A server has waited for the write-ahead log to flush to disk.
wal_file_sync	BIGINT	A server has waited for the write-ahead log to sync to disk (related to the wal_sync_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open_sync).
db_file_read	BIGINT	A server has waited for the completion of a read (from disk).
db_file_write	BIGINT	A server has waited for the completion of a write (to disk).
db_file_sync	BIGINT	A server has waited for the operating system to flush all changes to disk.
db_file_extend	BIGINT	A server has waited for the operating system while adding a new page to the end of a file.
sql_parse	BIGINT	Currently Unused.
query_plan	BIGINT	A server has generated a query plan.
other_lwlock_acquire	BIGINT	A server has waited for other light-weight lock to protect data.
shared_plan_cache_collision	BIGINT	A server has waited for the completion of the shared_plan_cache_collision event.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Data Type	Description
shared_plan_cache_insert	BIGINT	A server has waited for the completion of the shared_plan_cache_insert event.
shared_plan_cache_hit	BIGINT	A server has waited for the completion of the shared_plan_cache_hit event.
shared_plan_cache_miss	BIGINT	A server has waited for the completion of the shared_plan_cache_miss event.
shared_plan_cache_lock	BIGINT	A server has waited for the completion of the shared_plan_cache_lock event.
shared_plan_cache_busy	BIGINT	A server has waited for the completion of the shared_plan_cache_busy event.
shmemindexlock	BIGINT	A server has waited to find or allocate space in the shared memory.
oidgenlock	BIGINT	A server has waited to allocate or assign an OID.
xidgenlock	BIGINT	A server has waited to allocate or assign a transaction ID.
procarraylock	BIGINT	A server has waited to get a snapshot or clearing a transaction ID at transaction end.
sinvalreadlock	BIGINT	A server has waited to retrieve or remove messages from shared invalidation queue.
sinvalwritelock	BIGINT	A server has waited to add a message to the shared invalidation queue.
walbufmappinglock	BIGINT	A server has waited to replace a page in WAL buffers.
walwritelock	BIGINT	A server has waited for WAL buffers to be written to disk.
controlfilelock	BIGINT	A server has waited to read or update the control file or creation of a new WAL file.
checkpointlock	BIGINT	A server has waited to perform a checkpoint.
clogcontrollock	BIGINT	A server has waited to read or update the transaction status.
subtranscontrollock	BIGINT	A server has waited to read or update the sub-transaction information.
multixactgenlock	BIGINT	A server has waited to read or update the shared multixact state.
multixactoffsetcontrollock	BIGINT	A server has waited to read or update multixact offset mappings.
multixactmembercontrollock	BIGINT	A server has waited to read or update multixact member mappings.
relcacheinitlock	BIGINT	A server has waited to read or write the relation cache initialization file.
checkpointercommlock	BIGINT	A server has waited to manage the fsync requests.
twophasestatelock	BIGINT	A server has waited to read or update the state of prepared transactions.
tablespacecreatelock	BIGINT	A server has waited to create or drop the tablespace.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Data Type	Description
btreevacuumlock	BIGINT	A server has waited to read or update the vacuum related information for a B-tree index.
addinshmeminitlock	BIGINT	A server has waited to manage space allocation in shared memory.
autovacuumlock	BIGINT	The autovacuum launcher waiting to read or update the current state of autovacuum workers.
autovacuumschedulelock	BIGINT	A server has waited to ensure that the table selected for a vacuum still needs vacuuming.
syncscanlock	BIGINT	A server has waited to get the start location of a scan on a table for synchronized scans.
relationmappinglock	BIGINT	A server has waited to update the relation map file used to store catalog to file node mapping.
asyncctllock	BIGINT	A server has waited to read or update shared notification state.
asyncqueuelock	BIGINT	A server has waited to read or update the notification messages.
serializablexacthashlock	BIGINT	A server has waited to retrieve or store information about serializable transactions.
serializablefinishedlistlock	BIGINT	A server has waited to access the list of finished serializable transactions.
serializablepredicatelocklistlock	BIGINT	A server has waited to perform an operation on a list of locks held by serializable transactions.
oldserxidlock	BIGINT	A server has waited to read or record the conflicting serializable transactions.
syncreplock	BIGINT	A server has waited to read or update information about synchronous replicas.
backgroundworkerlock	BIGINT	A server has waited to read or update the background worker state.
dynamicsharedmemorycontrollock	BIGINT	A server has waited to read or update the dynamic shared memory state.
autofilelock	BIGINT	A server has waited to update the `postgresql.auto.conf` file.
replicationslotallocationlock	BIGINT	A server has waited to allocate or free a replication slot.
replicationslotcontrollock	BIGINT	A server has waited to read or update replication slot state.
committscontrollock	BIGINT	A server has waited to read or update transaction commit timestamps.
committslock	BIGINT	A server has waited to read or update the last value set for the transaction timestamp.
replicationoriginlock	BIGINT	A server has waited to set up, drop, or use replication origin.
multixacttruncationlock	BIGINT	A server has waited to read or truncate multixact information.
oldsnapshottimemaplock	BIGINT	A server has waited to read or update old snapshot control information.
backendrandomlock	BIGINT	A server has waited to generate a random number.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Data Type	Description
logicalrepworkerlock	BIGINT	A server has waited for the action on logical replication worker to finish.
clogtruncationlock	BIGINT	A server has waited to truncate the write-ahead log or waiting for write-ahead log truncation to finish.
bulkloadlock	BIGINT	A server has waited for the bulkloadlock to bulk upload the data.
edbresourcemanagerlock	BIGINT	The edbresourcemanagerlock provides detail about edb resource manager lock module.
wal_write_time	BIGINT	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).
wal_flush_time	BIGINT	The amount of time that the server has waited for a wal_flush wait event to write-ahead log to flush to disk.
wal_file_sync_time	BIGINT	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 'fsync' - better performance can be gained by changing this parameter to open_sync).
db_file_read_time	BIGINT	The amount of time that the server has waited for the db_file_read wait event for completion of a read (from disk).
db_file_write_time	BIGINT	The amount of time that the server has waited for the db_file_write wait event for completion of a write (to disk).
db_file_sync_time	BIGINT	The amount of time that the server has waited for the db_file_sync wait event to sync all changes to disk.
db_file_extend_time	BIGINT	The amount of time that the server has waited for the db_file_extend wait event while adding a new page to the end of a file.
sql_parse_time	BIGINT	The amount of time that the server has waited for the sql_parse wait event to parse a SQL statement.
query_plan_time	BIGINT	The amount of time that the server has waited for the query_plan wait event to compute the execution plan for a SQL statement.
other_lwlock_acquire_time	BIGINT	The amount of time that the server has waited for the other_lwlock_acquire wait event to protect data.
shared_plan_cache_collision_time	BIGINT	The amount of time that the server has waited for the shared_plan_cache_collision wait event.
shared_plan_cache_insert_time	BIGINT	The amount of time that the server has waited for the shared_plan_cache_insert wait event.
shared_plan_cache_hit_time	BIGINT	The amount of time that the server has waited for the shared_plan_cache_hit wait event.
shared_plan_cache_miss_time	BIGINT	The amount of time that the server has waited for the shared_plan_cache_miss wait event.
shared_plan_cache_lock_time	BIGINT	The amount of time that the server has waited for the shared_plan_cache_lock wait event.
shared_plan_cache_busy_time	BIGINT	The amount of time that the server has waited for the shared_plan_cache_busy wait event.
shmemindexlock_time	BIGINT	The amount of time that the server has waited for the shmemindexlock wait event to find or allocate space in the shared memory.
oidgenlock_time	BIGINT	The amount of time that the server has waited for the oidgenlock wait event to allocate or assign an OID.
xidgenlock_time	BIGINT	The amount of time that the server has waited for xidgenlock wait event to allocate or assign a transaction ID.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Data Type	Description
procarraylock_time	BIGINT	The amount of time that the server has waited for a procarraylock wait event to clear a transaction ID at transaction end.
sinvalreadlock_time	BIGINT	The amount of time that the server has waited for a sinvalreadlock wait event to retrieve or remove messages from shared invalidation queue.
sinvalwritelock_time	BIGINT	The amount of time that the server has waited for a sinvalwritelock wait event to add a message to the shared invalidation queue.
walbufmappinglock_time	BIGINT	The amount of time that the server has waited for a walbufmappinglock wait event to replace a page in WAL buffers.
walwritelock_time	BIGINT	The amount of time that the server has waited for a walwritelock wait event to write the WAL buffers to disk.
controlfilelock_time	BIGINT	The amount of time that the server has waited for a controlfilelock wait event to read or update the control file or to create a new WAL file.
checkpointlock_time	BIGINT	The amount of time that the server has waited for a checkpointlock wait event to perform a checkpoint.
clogcontrollock_time	BIGINT	The amount of time that the server has waited for a clogcontrollock wait event to read or update the transaction status.
subtranscontrollock_time	BIGINT	The amount of time that the server has waited for the subtranscontrollock wait event to read or update the sub-transaction information.
multixactgenlock_time	BIGINT	The amount of time that the server has waited for the multixactgenlock wait event to read or update the shared multixact state.
multixactoffsetcontrollock_time	BIGINT	The amount of time that the server has waited for the multixactoffsetcontrollock wait event to read or update multixact offset mappings.
multixactmembercontrollock_time	BIGINT	The amount of time that the server has waited for the multixactmembercontrollock wait event to read or update multixact member mappings.
relcacheinitlock_time	BIGINT	The amount of time that the server has waited for the relcacheinitlock wait event to read or write the relation cache initialization file.
checkpointercommlock_time	BIGINT	The amount of time that the server has waited for the checkpointercommlock wait event to manage the fsync requests.
twophasestatelock_time	BIGINT	The amount of time that the server has waited for the twophasestatelock wait event to read or update the state of prepared transactions.
tablespacecreatelock_time	BIGINT	The amount of time that the server has waited for the tablespacecreatelock wait event to create or drop the tablespace.
btreevacuumlock_time	BIGINT	The amount of time that the server has waited for the btreevacuumlock wait event to read or update the vacuum related information for a B-tree index.
addinshmeminitlock_time	BIGINT	The amount of time that the server has waited for the addinshmeminitlock wait event to manage space allocation in shared memory.
autovacuumlock_time	BIGINT	The amount of time that the server has waited for the autovacuumlock wait event to read or update the current state of autovacuum workers.
autovacuumschedulelock_time	BIGINT	The amount of time that the server has waited for the autovacuumschedulelock wait event to ensure that the table selected for a vacuum still needs vacuuming.
syncscanlock_time	BIGINT	The amount of time that the server has waited for the syncscanlock wait event to get the start location of a scan on a table for synchronized scans.
relationmappinglock_time	BIGINT	The amount of time that the server has waited for the relationmappinglock wait event to update the relation map file used to store catalog to file node mapping.
asyncctllock_time	BIGINT	The amount of time that the server has waited for the asyncctllock wait event to read or update shared notification state.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Data Type	Description
asyncqueuelock_time	BIGINT	The amount of time that the server has waited for the asyncqueuelock wait event to read or update the notification messages.
serializablexacthashlock_time	BIGINT	The amount of time that the server has waited for the serializablexacthashlock wait event to retrieve or store information about serializable transactions.
serializablefinishedlistlock_time	BIGINT	The amount of time that the server has waited for the serializablefinishedlistlock wait event to access the list of finished serializable transactions.
serializablepredicatelocklistlock_time	BIGINT	The amount of time that the server has waited for the serializablepredicatelocklistlock wait event to perform an operation on a list of locks held by serializable transactions.
oldserxidlock_time	BIGINT	The amount of time that the server has waited for the oldserxidlock wait event to read or record the conflicting serializable transactions.
syncreplock_time	BIGINT	The amount of time that the server has waited for the syncreplock wait event to read or update information about synchronous replicas.
backgroundworkerlock_time	BIGINT	The amount of time that the server has waited for the backgroundworkerlock wait event to read or update the background worker state.
dynamicsharedmemorycontrollock_time	BIGINT	The amount of time that the server has waited for the dynamicsharedmemorycontrollock wait event to read or update the dynamic shared memory state.
autofilelock_time	BIGINT	The amount of time that the server has waited for the autofilelock wait event to update the postgresql.auto.conf file.
replicationslotallocationlock_time	BIGINT	The amount of time that the server has waited for the replicationslotallocationlock wait event to allocate or free a replication slot.
replicationslotcontrollock_time	BIGINT	The amount of time that the server has waited for the replicationslotcontrollock wait event to read or update replication slot state.
committscontrollock_time	BIGINT	The amount of time that the server has waited for the committscontrollock wait event to read or update transaction commit timestamps.
committslock_time	BIGINT	The amount of time that the server has waited for the committslock wait event to read or update the last value set for the transaction timestamp.
replicationoriginlock_time	BIGINT	The amount of time that the server has waited for the replicationoriginlock wait event to set up, drop, or use replication origin.
multixacttruncationlock_time	BIGINT	The amount of time that the server has waited for the multixacttruncationlock wait event to read or truncate multixact information.
oldsnapshottimemaplock_time	BIGINT	The amount of time that the server has waited for the oldsnapshottimemaplock wait event to read or update old snapshot control information.
backendrandomlock_time	BIGINT	The amount of time that the server has waited for the backendrandomlock wait event to generate a random number.
logicalrepworkerlock_time	BIGINT	The amount of time that the server has waited for the logicalrepworkerlock wait event for an action on logical replication worker to finish.
clogtruncationlock_time	BIGINT	The amount of time that the server has waited for the clogtruncationlock wait event to truncate the write-ahead log or waiting for write-ahead log truncation to finish.
bulkloadlock_time	BIGINT	The amount of time that the server has waited for the bulkloadlock wait event to bulk upload the data.
edbresourcemanagerlock_time	BIGINT	The amount of time that the server has waited for the edbresourcemanagerlock wait event.
totalwaits	BIGINT	The total number of event waits.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Data Type	Description
totalwaittime	BIGINT	The total time spent waiting for an event.

11.2.12 DBMS_RANDOM

The DBMS_RANDOM package provides a number of methods to generate random values. The procedures and functions available in the DBMS_RANDOM package are listed in the following table.

Function/Procedure	Return Type	Description
INITIALIZE(val)	n/a	Initializes the DBMS_RANDOM package with the specified seed value. Deprecated, but supported for backward compatibility.
NORMAL()	NUMBER	Returns a random NUMBER.
RANDOM	INTEGER	Returns a random INTEGER with a value greater than or equal to -2^31 and less than 2^31. Deprecated, but supported for backward compatibility.
SEED(val)	n/a	Resets the seed with the specified value.
SEED(val)	n/a	Resets the seed with the specified value.
STRING(opt, len)	VARCHAR2	Returns a random string.
TERMINATE	n/a	TERMINATE has no effect. Deprecated, but supported for backward compatibility.
VALUE	NUMBER	Returns a random number with a value greater than or equal to 0 and less than 1, with 38 digit precision.
VALUE(low, high)	NUMBER	Returns a random number with a value greater than or equal to low and less than high.

INITIALIZE

The INITIALIZE procedure initializes the DBMS_RANDOM package with a seed value. The signature is:

```
INITIALIZE(<val> IN INTEGER)
```

This procedure should be considered deprecated; it is included for backward compatibility only.

Parameters

val

val is the seed value used by the DBMS_RANDOM package algorithm.

Example

The following code snippet demonstrates a call to the INITIALIZE procedure that initializes the DBMS_RANDOM package with the seed value, 6475.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DBMS_RANDOM.INITIALIZE(6475);
```

NORMAL

The `NORMAL` function returns a random number of type `NUMBER`. The signature is:

```
<result> NUMBER NORMAL()
```

Parameters

`result`

`result` is a random value of type `NUMBER`.

Example

The following code snippet demonstrates a call to the `NORMAL` function:

```
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:

```
<result> INTEGER RANDOM()
```

This function should be considered deprecated; it is included for backward compatibility only.

Parameters

`result`

`result` is a random value of type `INTEGER`.

Example

The following code snippet demonstrates a call to the `RANDOM` function. The call returns a random number:

```
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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SEED(<val> IN INTEGER)
```

Parameters

`val`

`val` is the seed value used by the `DBMS_RANDOM` package algorithm.

Example

The following code snippet demonstrates a call to the `SEED` procedure; the call sets the seed value at `8495`.

```
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:

```
SEED(<val> IN VARCHAR2)
```

Parameters

`val`

`val` is the seed value used by the `DBMS_RANDOM` package algorithm.

Example

The following code snippet demonstrates a call to the `SEED` procedure; the call sets the seed value to `abc123`.

```
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:

```
<result> VARCHAR2 STRING(<opt> IN CHAR, <len> IN NUMBER)
```

Parameters

`opt`

Formatting option for the returned string. `option` may be:

Option	Specifies Formatting Option

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Option	Specifies Formatting Option
`u` or `U`	Uppercase alpha string
`l` or `L`	Lowercase alpha string
`a` or `A`	Mixed case string
`x` or `X`	Uppercase alpha-numeric string
`p` or `P`	Any printable characters

`len`

The length of the returned string.

`result`

`result` is a random value of type `VARCHAR2`.

Example

The following code snippet demonstrates a call to the `STRING` function; the call returns a random alpha-numeric character string that is 10 characters long.

```
x := DBMS_RANDOM.STRING('X', 10);
```

TERMINATE

The `TERMINATE` procedure has no effect. The signature is:

```
TERMINATE
```

The `TERMINATE` procedure should be considered deprecated; the procedure is supported for compatibility only.

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:

```
<result> NUMBER VALUE()
```

Parameters

`result`

`result` is a random value of type `NUMBER`.

Example

The following code snippet demonstrates a call to the `VALUE` function. The call returns a random `NUMBER`:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
x := DBMS_RANDOM.VALUE();
```

VALUE

The `VALUE` function returns a random `NUMBER` with a value that is between user-specified boundaries. The `VALUE` function has two forms; the signature of the second form is:

```
<result> NUMBER VALUE(<low> IN NUMBER, <high> IN NUMBER)
```

Parameters

`low`

`low` specifies the lower boundary for the random value. The random value may be equal to `low`.

`high`

`high` specifies the upper boundary for the random value; the random value will be less than `high`.

`result`

`result` is a random value of type `NUMBER`.

Example

The following code snippet demonstrates 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:

```
x := DBMS_RANDOM.VALUE(1, 100);
```

11.2.13 DBMS_REDACT

The `DBMS_REDACT` package enables the redacting or masking of data returned by a query. The `DBMS_REDACT` package provides a procedure to create policies, alter policies, enable policies, disable policies, and drop policies. The procedures available in the `DBMS_REDACT` package are listed in the following table.

Function/Procedure	Function or Procedure	Return Type	Description
`ADD_POLICY(object_schema, object_name, policy_name, policy_description, column_name, column_description, function_type, function_parameters, expression, enable, regexp_pattern, regexp_replace_string, regexp_position, regexp_occurence, regexp_match_parameter, custom_function_expression)`	Procedure	n/a	Adds a data redaction policy.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function/Procedure	Function or Procedure	Return Type	Description
`ALTER_POLICY(object_schema, object_name, policy_name, action, column_name, function_type, function_parameters, expression, regexp_pattern, regexp_replace_string, regexp_position, regexp_occurence, regexp_match_parameter, policy_description, column_description, custom_function_expression)`	Procedure	n/a	Alters the existing data redaction policy.
`DISABLE_POLICY(object_schema, object_name, policy_name)`	Procedure	n/a	Disables the existing data redaction policy.
`ENABLE_POLICY(object_schema, object_name, policy_name)`	Procedure	n/a	Enables a previously disabled data redaction policy.
`DROP_POLICY(object_schema, object_name, policy_name)`	Procedure	n/a	Drops a data redaction policy.
`UPDATE_FULL_REDACTION_VALUES(number_val, binfloat_val, bindouble_val, char_val, varchar_val, nchar_val, nvarchar_val, datecol_val, ts_val, tswtz_val, blob_val, clob_val, nclob_val)`	Procedure	n/a	Updates the full redaction default values for the specified datatype.

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.

Note that 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 super-user 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 below table highlights the values for `function_type` parameters of `dbms_redact.add_policy` and `dbms_redact.alter_policy`.

Constant	Type	Value	Description
`NONE`	`INTEGER`	`0`	No redaction, zero effect on the result of a query against table.
`FULL`	`INTEGER`	`1`	Full redaction, redacts full values of the column data.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Constant	Type	Value	Description
PARTIAL	INTEGER	2	Partial redaction, redacts a portion of the column data.
RANDOM	INTEGER	4	Random redaction, each query results in a different random value depending on the datatype of the column.
REGEXP	INTEGER	5	Regular Expression based redaction, searches for the pattern of data to redact.
CUSTOM	INTEGER	99	Custom redaction type.

The following table shows the values for the `action` parameter of `dbms_redact.alter_policy`.

Constant	Type	Value	Description
ADD_COLUMN	INTEGER	1	Adds a column to the redaction policy.
DROP_COLUMN	INTEGER	2	Drops a column from the redaction policy.
MODIFY_EXPRESSION	INTEGER	3	Modifies the expression of a redaction policy. The redaction is applied when the expression evaluates to the `BOOLEAN` value to `TRUE`.
MODIFY_COLUMN	INTEGER	4	Modifies a column in the redaction policy to change the redaction function type or function parameter.
SET_POLICY_DESCRIPTION	INTEGER	5	Sets the redaction policy description.
SET_COLUMN_DESCRIPTION	INTEGER	6	Sets a description for the redaction performed on the column.

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 datatype:

- `Character`
- `Number`
- `Datetime`

The following table highlights the format descriptor for partial redaction with respect to datatype. The example described below 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.

Datatype	Format Descriptor	Description	Examples
Character	REDACT_PARTIAL_INPUT_FORMAT	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.	Consider `'VVVFVVFVVVV,VVV-VV-VVVV,X,1,5'` for masking first 5 digits of SSN strings such as `123-45-6789`, adding hyphen to format it and thereby resulting in strings such as `XXX-XX-6789.` The field value `VVVFVVFVVVV` for matching SSN strings such as `123-45-6789`.
	REDACT_PARTIAL_OUTPUT_FORMAT	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.	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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Datatype	Format Descriptor	Description	Examples
	`REDACT_PARTIAL_MASKCHAR`	Specifies the character to be used for redaction.	The value `X` for redacting SSN strings into `XXX-XX-6789`.
	`REDACT_PARTIAL_MASKFROM`	Specifies which `V` within the input format from which to start the redaction.	The value `1` for redacting SSN strings starting at the first `V` of the input format of `VVVFVVFVVVV` into strings such as `XXX-XX-6789`.
	`REDACT_PARTIAL_MASKTO`	Specifies which `V` within the input format at which to end the redaction.	The value `5` for redacting SSN strings up to and including the fifth `V` within the input format of `VVVFVVFVVVV` into strings such as `XXX-XX-6789`.
Number	`REDACT_PARTIAL_MASKCHAR`	Specifies the character to be displayed in the range between 0 and 9.	`'9, 1, 5'` for redacting the first five digits of the Social Security Number `123456789` into `999996789`.
	`REDACT_PARTIAL_MASKFROM`	Specifies the start digit position for redaction.	
	`REDACT_PARTIAL_MASKTO`	Specifies the end digit position for redaction.	
Datetime	`REDACT_PARTIAL_DATE_MONTH`	`'m'` redacts the month. To mask a specific month, specify `'m#'` where # indicates the month specified by its number between `1` and `12`.	`m3` displays as March.
	`REDACT_PARTIAL_DATE_DAY`	`'d'` redacts the day of the month. To mask with a day of the month, append `1-31` to a lowercase `d`.	`d3` displays as `03`.
	`REDACT_PARTIAL_DATE_YEAR`	`'y'` redacts the year. To mask with a year, append `1-9999` to a lowercase `y`.	`y1960` displays as `60`.
	`REDACT_PARTIAL_DATE_HOUR`	`'h'` redacts the hour. To mask with an hour, append `0-23` to a lowercase `h`.	`h18` displays as `18`.
	`REDACT_PARTIAL_DATE_MINUTE`	`'m'` redacts the minute. To mask with a minute, append `0-59` to a lowercase `m`.	`m20` displays as `20`.
	`REDACT_PARTIAL_DATE_SECOND`	`'s'` redacts the second. To mask with a second, append `0-59` to a lowercase `s`.	`s40` displays as `40`.

The following table represents `function_parameters` values that can be used in partial redaction.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function Parameter	Data Type	Value	Description
REDACT_US_SSN_F5	VARCHAR2	'VVVFVVFVVVV,VVV-VV-VVVV,X,1,5'	Redacts the first 5 numbers of SSN. **Example:** The number 123-45-6789 becomes XXX-XX-6789.
REDACT_US_SSN_L4	VARCHAR2	'VVVFVVFVVVV,VVV-VV-VVVV,X,6,9'	Redacts the last 4 numbers of SSN. **Example:** The number 123-45-6789 becomes 123-45-XXXX.
REDACT_US_SSN_ENTIRE	VARCHAR2	'VVVFVVFVVVV,VVV-VV-VVVV,X,1,9'	Redacts the entire SSN. **Example:** The number 123-45-6789 becomes XXX-XX-XXXX.
REDACT_NUM_US_SSN_F5	VARCHAR2	'9,1,5'	Redacts the first 5 numbers of SSN when the column is a number datatype. **Example:** The number 123456789 becomes 999996789.
REDACT_NUM_US_SSN_L4	VARCHAR2	'9,6,9'	Redacts the last 4 numbers of SSN when the column is a number datatype. **Example:** The number 123456789 becomes 123459999.
REDACT_NUM_US_SSN_ENTIRE	VARCHAR2	'9,1,9'	Redacts the entire SSN when the column is a number datatype. **Example:** The number 123456789 becomes 999999999.
REDACT_ZIP_CODE	VARCHAR2	'VVVVV,VVVVV,X,1,5'	Redacts a 5 digit zip code. **Example:** 12345 becomes XXXXX.
REDACT_NUM_ZIP_CODE	VARCHAR2	'9,1,5'	Redacts a 5 digit zip code when the column is a number datatype. **Example:** 12345 becomes 99999.
REDACT_CCN16_F12	VARCHAR2	'VVVVFVVVVFVVVVFVVVV,VVVV-VVVV-VVVV-VVVV,*,1,12'	Redacts a 16 digit credit card number and displays only 4 digits. **Example:** 1234 5678 9000 2358 becomes ****-****-****-2358.
REDACT_DATE_MILLENNIUM	VARCHAR2	'm1d1y2000'	Redacts a date that is in the DD-MM-YY format. **Example:** Redacts all date to 01-JAN-2000.
REDACT_DATE_EPOCH	VARCHAR2	'm1d1y1970'	Redacts all dates to 01-JAN-70.
REDACT_AMEX_CCN_FORMATTED	VARCHAR2	'VVVVFVVVVVVFVVVVV,VVVV-VVVVVV-VVVVV,*,1,10'	Redacts the Amercian Express credit card number and replaces the digit with * except for the last 5 digits. **Example:** The credit card number 1234 567890 34500 becomes **** ****** 34500.
REDACT_AMEX_CCN_NUMBER	VARCHAR2	'0,1,10'	Redacts the Amercian Express credit card number and replaces the digit with 0 except for the last 5 digits. **Example:** The credit card number 1234 567890 34500 becomes 0000 000000 34500.
REDACT_SIN_FORMATTED	VARCHAR2	'VVVFVVVFVVV,VVV-VVV-VVV,*,1,6'	Redacts the Social Insurance Number by replacing the first 6 digits by *. **Example:** 123-456-789 becomes ***-***-789.
REDACT_SIN_NUMBER	VARCHAR2	'9,1,6'	Redacts the Social Insurance Number by replacing the first 6 digits by 9. **Example:** 123456789 becomes 999999789.
REDACT_SIN_UNFORMATTED	VARCHAR2	'VVVVVVVVV,VVVVVVVVV,*,1,6'	Redacts the Social Insurance Number by replacing the first 6 digits by *. **Example:** 123456789 becomes ******789.
REDACT_CCN_FORMATTED	VARCHAR2	'VVVVFVVVVFVVVVFVVVV,VVVV-VVVV-VVVV-VVVV,*,1,12'	Redacts a credit card number by * and displays only 4 digits. **Example:** The credit card number 1234-5678-9000-4671 becomes ****-****-****-4671.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function Parameter	Data Type	Value	Description
REDACT_CCN_NUMBER	VARCHAR2	'9,1,12'	Redacts a credit card number by 0 except the last 4 digits. **Example:** The credit card number 1234567890004671 becomes 0000000000004671.
REDACT_NA_PHONE_FORMATTED	VARCHAR2	'VVVFVVVFVVVV,VVV-VVV-VVVV,X,4,10'	Redacts the North American phone number by X leaving the area code. **Example:** 123-456-7890 becomes 123-XXX-XXXX.
REDACT_NA_PHONE_NUMBER	VARCHAR2	'0,4,10'	Redacts the North American phone number by 0 leaving the area code. **Example:** 1234567890 becomes 1230000000.
REDACT_NA_PHONE_UNFORMATTED	VARCHAR2	'VVVVVVVVVV,VVVVVVVVVV,X,4,10'	Redacts the North American phone number by X leaving the area code. **Example:** 1234567890 becomes 123XXXXXXX.
REDACT_UK_NIN_FORMATTED	VARCHAR2	'VVFVVFVVFVVFV,VV VV VV V,X,3,8'	Redacts the UK National Insurance Number by X but leaving the alphabetic characters. **Example:** NY 22 01 34 D becomes NY XX XX XX D.
REDACT_UK_NIN_UNFORMATTED	VARCHAR2	'VVVVVVVVV,VVVVVVVVV,X,3,8'	Redacts the UK National Insurance Number by X but leaving the alphabetic characters. **Example:** NY220134D becomes NYXXXXXXD.

A regular expression-based redaction searches for patterns of data to redact. The regexp_pattern search the values in order for the regexp_replace_string to change the value. The following table illustrates the regexp_pattern values that you can use during REGEXP based redaction.

Function Parameter and Description	Data Type	Value
RE_PATTERN_CC_L6_T4: Searches for the middle digits of a credit card number that includes 6 leading digits and 4 trailing digits. The regexp_replace_string setting to use with the format is RE_REDACT_CC_MIDDLE_DIGITS that replaces the identified pattern with the characters specified by the RE_REDACT_CC_MIDDLE_DIGITS parameter.	VARCHAR2	'(\d\d\d\d\d\d)(\d\d\d*)(\d\d\d\d)'
RE_PATTERN_ANY_DIGIT: Searches for any digit and replaces the identified pattern with the characters specified by the following values of the regexp_replace_string parameter. regexp_replace_string=> RE_REDACT_WITH_SINGLE_X (replaces any matched digit with the X character). regexp_replace_string=> RE_REDACT_WITH_SINGLE_1 (replaces any matched digit with the 1 character).	VARCHAR2	'\d'
RE_PATTERN_US_PHONE: Searches for the U.S phone number and replaces the identified pattern with the characters specified by the regexp_replace_string parameter. regexp_replace_string=> RE_REDACT_US_PHONE_L7 (searches the phone number and then replaces the last 7 digits).	VARCHAR2	'(\(\d\d\d\)|\d\d\d)-(\d\d\d)-(\d\d\d\d)'
RE_PATTERN_EMAIL_ADDRESS: Searches for the email address and replaces the identified pattern with the characters specified by the following values of the regexp_replace_string parameter. regexp_replace_string=> RE_REDACT_EMAIL_NAME (finds the email address and redacts the email username). regexp_replace_string=> RE_REDACT_EMAIL_DOMAIN (finds the email address and redacts the email domain). regexp_replace_string=> RE_REDACT_EMAIL_ENTIRE (finds the email address and redacts the entire email address).	VARCHAR2	'([A-Za-z0-9._%+-]+)@([A-Za-z0-9.-]+\.[A-Za-z]{2,4})'
RE_PATTERN_IP_ADDRESS: Searches for an IP address and replaces the identified pattern with the characters specified by the regexp_replace_string parameter. The regexp_replace_string parameter to be used is RE_REDACT_IP_L3 that replaces the last section of an IP address with 999 and indicates it is redacted.	VARCHAR2	'(\d{1,3}\.\d{1,3}\.\d{1,3})\.\d{1,3}'

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function Parameter and Description	Data Type	Value
`RE_PATTERN_AMEX_CCN` : Searches for the American Express credit card number. The `regexp_replace_string` parameter to be used is `RE_REDACT_AMEX_CCN` that redacts all of the digits except the last 5.	`VARCHAR2`	`'.*(\d\d\d\d\d)$'`
`RE_PATTERN_CCN` : Searches for the credit card number other than American Express credit cards. The `regexp_replace_string` parameter to be used is `RE_REDACT_CCN` that redacts all of the digits except the last 4.	`VARCHAR2`	`'.*(\d\d\d\d)$'`
`RE_PATTERN_US_SSN` : Searches the SSN number and replaces the identified pattern with the characters specified by the `regexp_replace_string` parameter. `'\1-XXX-XXXX'` or `'XXX-XXX-\3'` will return `123-XXX-XXXX` or `XXX-XXX-6789` for the value `'123-45-6789'` respectively.	`VARCHAR2`	`'(\d\d\d)-(\d\d)-(\d\d\d\d)'`

The below table illustrates the `regexp_replace_string` values that you can use during `REGEXP` based redaction.

Function Parameter	Data Type	Value	Description
`RE_REDACT_CC_MIDDLE_DIGITS`	`VARCHAR2`	`'\1XXXXXX\3'`	Redacts the middle digits of a credit card number according to the `regexp_pattern` parameter with the `RE_PATTERN_CC_L6_T4` format and replaces each redacted character with an `X`. **Example:** The credit card number `1234 5678 9000 2490` becomes `1234 56XX XXXX 2490`.
`RE_REDACT_WITH_SINGLE_X`	`VARCHAR2`	`'X'`	Replaces the data with a single `X` character for each matching pattern as specified by setting the `regexp_pattern` parameter with the `RE_PATTERN_ANY_DIGIT` format. **Example:** The credit card number `1234 5678 9000 2490` becomes `XXXX XXXX XXXX XXXX`.
`RE_REDACT_WITH_SINGLE_1`	`VARCHAR2`	`'1'`	Replaces the data with a single `1` digit for each of the data digits as specified by setting the `regexp_pattern` parameter with the `RE_PATTERN_ANY_DIGIT` format. **Example:** The credit card number `1234 5678 9000 2490` becomes `1111 1111 1111 1111`.
`RE_REDACT_US_PHONE_L7`	`VARCHAR2`	`'\1-XXX-XXXX'`	Redacts the last 7 digits of U.S phone number according to the `regexp_pattern` parameter with the `RE_PATTERN_US_PHONE` format and replaces each redacted character with an `X`. **Example:** The phone number `123-444-5900` becomes `123-XXX-XXXX`.
`RE_REDACT_EMAIL_NAME`	`VARCHAR2`	`'xxxx@\2'`	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`.
`RE_REDACT_EMAIL_DOMAIN`	`VARCHAR2`	`'\1@xxxxx.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`.
`RE_REDACT_EMAIL_ENTIRE`	`VARCHAR2`	`'xxxx@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`.
`RE_REDACT_IP_L3`	`VARCHAR2`	`'\1.999'`	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.
`RE_REDACT_AMEX_CCN`	`VARCHAR2`	`'**********\1'`	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`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function Parameter	Data Type	Value	Description
RE_REDACT_CCN	VARCHAR2	'************\1'	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_occurence values that you can use during REGEXP based redaction.

Function Parameter	Data Type	Value	Description
RE_BEGINNING	INTEGER	1	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 source_char.

Function Parameter	Data Type	Value	Description
RE_ALL	INTEGER	0	Specifies the replacement occurrence of a substring. If the value is 0, then the replacement of each matching substring occurs.
RE_FIRST	INTEGER	1	Specifies the replacement occurrence of a substring. If the value is 1, then the replacement of the first matching substring occurs.

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.

Function Parameter	Data Type	Value	Description
RE_CASE_SENSITIVE	VARCHAR2	'c'	Specifies the case-sensitive matching.
RE_CASE_INSENSITIVE	VARCHAR2	'i'	Specifies the case-insensitive matching.
RE_MULTIPLE_LINES	VARCHAR2	'm'	Treats the source string as multiple lines but if you omit this parameter, then it indicates as a single line.
RE_NEWLINE_WILDCARD	VARCHAR2	'n'	Specifies the period (.), but if you omit this parameter, then the period does not match the newline character.
RE_IGNORE_WHITESPACE	VARCHAR2	'x'	Ignores the whitespace characters.

> **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 accordingly set the replacement string 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 accordingly set the replacement string to avoid runtime errors.

ADD_POLICY

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `add_policy` procedure creates a new data redaction policy for a table.

```
PROCEDURE add_policy (
  <object_schema>                IN VARCHAR2 DEFAULT NULL,
  <object_name>                  IN VARCHAR2,
  <policy_name>                  IN VARCHAR2,
  <policy_description>           IN VARCHAR2 DEFAULT NULL,
  <column_name>                  IN VARCHAR2 DEFAULT NULL,
  <column_description>           IN VARCHAR2 DEFAULT NULL,
  <function_type>                IN INTEGER DEFAULT DBMS_REDACT.FULL,
  <function_parameters>          IN VARCHAR2 DEFAULT NULL,
  <expression>                   IN VARCHAR2,
  <enable>                       IN BOOLEAN DEFAULT TRUE,
  <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,
  <custom_function_expression>   IN VARCHAR2 DEFAULT NULL
  )
```

Parameters

`object_schema`

Specifies the name of the schema in which the object resides and on which the data redaction policy will be applied. If you specify `NULL` then the given object is searched by the order specified by `search_path` setting.

`object_name`

Name of the table on which the data redaction policy is created.

`policy_name`

Name of the policy to be added. 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 additional columns.

`column_description`

Description of the column to be redacted. The `column_description` is not supported, but if you specify the description for a column then, you will get a warning message.

`function_type`

The type of redaction function to be used. The possible values are `NONE, FULL, PARTIAL, RANDOM, REGEXP`, and `CUSTOM`.

`function_parameters`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 the policy is to be applied. The redaction occurs if this policy expression is evaluated to `TRUE`.

`enable`

When set to `TRUE`, the policy is enabled upon creation. The default is set as `TRUE`. When set to `FALSE`, the policy is disabled but the policy can be enabled by calling the `enable_policy` procedure.

`regexp_pattern`

Specifies the regular expression pattern to redact data. If the `regexp_pattern` does not 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', 'RE_IGNORE_WHITESPACE'`.

Note: For more information on `constants`, `function_parameters`, or `regexp` (regular expressions) see, Using `DBMS_REDACT Constants and Function Parameters`.

`custom_function_expression`

The `custom_function_expression` is applicable only for the `CUSTOM` redaction type. The `custom_function_expression` is a function expression that is, 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

The following example illustrates how to create a policy and use full redaction for values in the `payment_details_tab` table `customer id` column.

```
edb=# CREATE TABLE payment_details_tab (
customer_id NUMBER       NOT NULL,
card_string VARCHAR2(19) NOT NULL);
CREATE TABLE

edb=# BEGIN
  INSERT INTO payment_details_tab VALUES (4000, '1234-1234-1234-1234');
  INSERT INTO payment_details_tab VALUES (4001, '2345-2345-2345-2345');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
END;

EDB-SPL Procedure successfully completed

edb=# CREATE USER redact_user;
CREATE ROLE
edb=# GRANT SELECT ON payment_details_tab TO redact_user;
GRANT

\c edb base_user

BEGIN
  DBMS_REDACT.add_policy(
     object_schema              => 'public',
     object_name                => 'payment_details_tab',
     policy_name                => 'redactPolicy_001',
     policy_description         => 'redactPolicy_001 for payment_details_tab table',
     column_name                => 'customer_id',
     function_type              => DBMS_REDACT.full,
     expression                 => '1=1',
     enable                     => TRUE);
END;
```

Redacted Result:

```
 edb=# \c edb redact_user
You are now connected to database "edb" as user "redact_user".

edb=> select customer_id from payment_details_tab order by 1;
 customer_id
-------------
           0
           0
(2 rows)
```

ALTER_POLICY

The `alter_policy` procedure alters or modifies an existing data redaction policy for a table.

```
 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  )
```

Parameters

`object_schema`

Specifies the name of the schema in which the object resides and on which the data redaction policy will be altered. 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 be altered.

`action`

The action to perform. For more information about action parameters see, `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 be used. 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 the policy is to be applied. The redaction occurs if this policy expression is evaluated to `TRUE`.

`regexp_pattern`

Enables the use of regular expressions to redact data. If the `regexp_pattern` does not 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_occurence`

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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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', 'RE_IGNORE_WHITESPACE'`.

Note: For more information on `constants, function_parameters`, or `regexp` (regular expressions) see, `Using DBMS_REDACT Constants and Function Parameters`.

`policy_description`

Specify the description of a redaction policy.

`column_description`

Description of the column to be redacted. The `column_description` is not supported, but if you specify the description for a column then, you will get a warning message.

`custom_function_expression`

The `custom_function_expression` is applicable only for the `CUSTOM` redaction type. The `custom_function_expression` is a function expression that is, 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

The following example illustrates to alter a policy using partial redaction for values in the `payment_details_tab` table `card_string` (usually a credit card number) column.

```
\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:

```
edb=# \c - redact_user
You are now connected to database "edb" as user "redact_user".
edb=> SELECT * FROM payment_details_tab;
 customer_id |     card_string
-------------+---------------------
           0 | ****-****-****-1234
           0 | ****-****-****-2345
(2 rows)
```

DISABLE_POLICY

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `disable_policy` procedure disables an existing data redaction policy.

```
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 the data redaction policy will be applied. 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 be disabled.

Example

The following example illustrates 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.

```
PROCEDURE enable_policy (
  <object_schema>          IN VARCHAR2 DEFAULT NULL,
  <object_name>            IN VARCHAR2,
  <policy_name>            IN VARCHAR2
  )
```

Parameters

`object_schema`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Specifies the name of the schema in which the object resides and on which the data redaction policy will be applied. 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 be enabled.

Example

The following example illustrates how to enable a policy.

```
\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.

```
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 the data redaction policy will be applied. 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 be dropped.

Example

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example illustrates how to drop a policy.

```
\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 and these default values can be viewed using the `redaction_values_for_type_full` view that use the full redaction type.

```
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 is not supported.

`bindouble_val`

The `FLOAT8` datatype is a random value. The binary double datatype is not supported.

`char_val`

Updates the default value for columns of the `CHAR` datatype.

`varchar_val`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

`nclob_val`

The `nclob_val` is mapped to `CLOB` datatype and returns the `CLOB` value.

Example

The following example illustrates how to update the full redaction values but before updating the values, you can:

View the default values using `redaction_values_for_type_full` view as shown below:

```
 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;
-[ RECORD 1 ]------------------+--------------------------
number_value                   | 0
char_value                     |
varchar_value                  |
date_value                     | 01-JAN-01 00:00:00
timestamp_value                | 01-JAN-01 01:00:00
timestamp_with_time_zone_value | 31-DEC-00 20:00:00 -05:00
blob_value                     | \x5b72656461637465645d
clob_value                     | [redacted]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(1 row)
```

Now, update the default values for full redaction type. The `NULL` values will be 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 `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;
-[ RECORD 1 ]------------------+----------------------------------------
number_value                   | 9999999
char_value                     | Z
varchar_value                  | V
date_value                     | 17-OCT-18 00:00:00
timestamp_value                | 17-OCT-18 11:12:13
timestamp_with_time_zone_value | 31-DEC-00 20:00:00 -05:00
blob_value                     | \x4e455720524544414354454420564 14c5545
clob_value                     | NEW REDACTED VALUE
(1 row)
```

Redacted Result:

```
 edb=# \c edb redact_user
You are now connected to database "edb" as user "redact_user".

edb=> select * from payment_details_tab order by 1;
 customer_id | card_string
-------------+-------------
     9999999 | V
     9999999 | V
(2 rows)
```

11.2.14 DBMS_RLS

The `DBMS_RLS` package enables the implementation of Virtual Private Database on certain Advanced Server database objects.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function/Procedure	Function or Procedure	Return Type	Description
`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 ]]]]]]]])`	Procedure	n/a	Add a security policy to a database object.
`DROP_POLICY(object_schema, object_name, policy_name)`	Procedure	n/a	Remove a security policy from a database object.
`ENABLE_POLICY(object_schema, object_name, policy_name, enable)`	Procedure	n/a	Enable or disable a security policy.

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 above are supported.

Virtual Private Database is a type of fine-grained access control using security policies. *Fine-grained access control* in Virtual Private Database means that access to data can be controlled 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 value. The *security policy* is the named association of the policy function to a particular database object, typically a table.

> **Note**
>
> In Advanced Server, the policy function can be written in any language supported by Advanced Server such as SQL, PL/pgSQL and SPL.

> **Note**
>
> The database objects currently supported by Advanced Server Virtual Private Database are tables. Policies cannot be applied to views or synonyms.

The advantages of using Virtual Private Database are the following:

- 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 instance.
- A different security policy can be applied depending upon 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 upon factors such as the session user of the application accessing the database object.
- Invocation of the security policy is transparent to all applications that access the database object and thus, individual applications do not have to be modified to apply the security policy.
- Once a security policy is enabled, it is not possible for any application (including new applications) to circumvent the security policy except by the system privilege noted by the following.
- Even superusers cannot circumvent the security policy except by the system privilege noted by the following.

> **Note**
>
> The only way security policies can be circumvented is if the `EXEMPT ACCESS POLICY` system privilege has been granted to a user. The `EXEMPT ACCESS POLICY` privilege should be granted with extreme care as a user with this privilege is exempted from all policies in the database.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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:

- 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 is to apply and the second input parameter is for the name of that database object. The function must have a `VARCHAR2` return type. The function must 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 upon the database object. Thus, rows that do not satisfy the policy function predicate are filtered out from the SQL command result set.
- 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 is to apply, whether or not to enable the policy at the time of its creation, and if the policy should apply to newly inserted rows or the modified image of updated rows.
- Use the `ENABLE_POLICY` procedure to disable or enable an existing policy.
- Use the `DROP_POLICY` procedure to remove an existing policy. The `DROP_POLICY` procedure does not drop the policy function or the associated database object.

Once policies are created, they can be viewed in the catalog views, compatible with Oracle databases: `ALL_POLICIES, DBA_POLICIES`, or `USER_POLICIES`. The supported compatible views are listed in the *Database Compatibility for Oracle Developers Catalog Views Guide*, available at the EDB website at:

https://www.enterprisedb.com/docs/

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 will return `NULL`.

`attribute` is a `VARCHAR2`. `attribute` may be:

attribute Value	Equivalent Value
`SESSION_USER`	`pg_catalog.session_user`
`CURRENT_USER`	`pg_catalog.current_user`
`CURRENT_SCHEMA`	`pg_catalog.current_schema`
`HOST`	`pg_catalog.inet_host`
`IP_ADDRESS`	`pg_catalog.inet_client_addr`
`SERVER_HOST`	`pg_catalog.inet_server_addr`

> **Note**
>
> The examples used to illustrate the `DBMS_RLS` package are based on a modified copy of the sample `emp` table provided with 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 shown by the following:

```
 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
empno  | ename  |    job    |   sal   |  comm   | deptno |   authid
-------+--------+-----------+---------+---------+--------+-------------
  7782 | CLARK  | MANAGER   | 2450.00 |         |     10 |
  7839 | KING   | PRESIDENT | 5000.00 |         |     10 |
  7934 | MILLER | CLERK     | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK     |  800.00 |         |     20 | researchmgr
  7566 | JONES  | MANAGER   | 2975.00 |         |     20 | researchmgr
  7788 | SCOTT  | ANALYST   | 3000.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK     | 1100.00 |         |     20 | researchmgr
  7902 | FORD   | ANALYST   | 3000.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  | 1600.00 |  300.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN  | 1250.00 |  500.00 |     30 | salesmgr
  7654 | MARTIN | SALESMAN  | 1250.00 | 1400.00 |     30 | salesmgr
  7698 | BLAKE  | MANAGER   | 2850.00 |         |     30 | salesmgr
  7844 | TURNER | SALESMAN  | 1500.00 |    0.00 |     30 | salesmgr
  7900 | JAMES  | CLERK     |  950.00 |         |     30 | salesmgr
(14 rows)

CREATE ROLE salesmgr WITH LOGIN PASSWORD 'password';
GRANT ALL ON vpemp TO salesmgr;
```

ADD_POLICY

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.

```
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 the policy is to be applied.

`object_name`

Name of the database object to which the policy is to be applied. A given database object may 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 within the database.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`function_schema`

Name of the schema containing the policy function.

Note: The policy function may belong to a package in which 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. The same function may be specified in more than one policy.

Note: The policy function may belong to a package in which 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: Advanced Server accepts `INDEX` as a statement type, but it is ignored. Policies are not applied to index operations in 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 do not 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 is not applied to newly inserted rows or the modified image of updated rows. Thus, a newly inserted row may not appear in the result set of a subsequent SQL command that invokes the same policy. Similarly, rows which qualified according to the policy prior to an `UPDATE` command may 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. The policy can be enabled 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 is 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 re-evaluated and the policy function predicate string regenerated for all invocations of the policy.

- The default is `FALSE`.

> **Note**
>
> 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 is not set to its default value of `NULL`, the `policy_type` parameter setting overrides the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> `static_policy` setting.

> **Note**
>
> The setting of `static_policy` is ignored by Advanced Server. Advanced Server implements only the dynamic policy, regardless of the setting of the `static_policy` parameter.

`policy_type`

In Oracle, determines when the policy function is re-evaluated, and hence, if and when the predicate string returned by the policy function changes. The default is `NULL`.

Note: The setting of this parameter is ignored by Advanced Server. Advanced Server always assumes a dynamic policy.

`long_predicate`

In Oracle, allows predicates up to 32K bytes if set to `TRUE`, otherwise predicates are limited to 4000 bytes. The default is `FALSE`.

Note: The setting of this parameter is ignored by Advanced Server. An Advanced Server policy function can return a predicate of unlimited length for all practical purposes.

`sec_relevant_cols`

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 is not 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`.

`sec_relevant_cols_opt`

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` is not set to `DBMS_RLS.ALL_ROWS`, these rows would not be returned at all in the result set.) The default is `NULL`.

Note: Advanced Server does not support the `DBMS_RLS.ALL_ROWS` functionality. Advanced Server throws an error if `sec_relevant_cols_opt` is set to `DBMS_RLS.ALL_ROWS (INTEGER` value of 1).

Examples

This example uses the following policy function:

```
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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 while the second parameter is the name of an attribute set within the application context. `USERENV` is a special built-in namespace that describes the current session. Advanced Server does not support application contexts, but 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` to be applied to the `vpemp` table using function `verify_session_user` whenever an `INSERT, UPDATE`, or `DELETE` SQL command is given referencing the `vpemp` table.

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_update';
    v_function_schema       VARCHAR2(30) := 'enterprisedb';
    v_policy_function       VARCHAR2(30) := 'verify_session_user';
    v_statement_types       VARCHAR2(30) := 'INSERT,UPDATE,DELETE';
    v_update_check          BOOLEAN      := TRUE;
    v_enable                BOOLEAN      := TRUE;
BEGIN
    DBMS_RLS.ADD_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name,
        v_function_schema,
        v_policy_function,
        v_statement_types,
        v_update_check,
        v_enable
    );
END;
```

After successful creation of the policy, a terminal session is started by user `salesmgr`. The following query shows the content of the `vpemp` table:

```
edb=# \c edb salesmgr
Password for user salesmgr:
You are now connected to database "edb" as user "salesmgr".
edb=> SELECT * FROM vpemp;
 empno | ename  |    job    |   sal   |  comm   | deptno |   authid
-------+--------+-----------+---------+---------+--------+-------------
  7782 | CLARK  | MANAGER   | 2450.00 |         |     10 |
  7839 | KING   | PRESIDENT | 5000.00 |         |     10 |
  7934 | MILLER | CLERK     | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK     |  800.00 |         |     20 | researchmgr
  7566 | JONES  | MANAGER   | 2975.00 |         |     20 | researchmgr
  7788 | SCOTT  | ANALYST   | 3000.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK     | 1100.00 |         |     20 | researchmgr
  7902 | FORD   | ANALYST   | 3000.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  | 1600.00 |  300.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN  | 1250.00 |  500.00 |     30 | salesmgr
  7654 | MARTIN | SALESMAN  | 1250.00 | 1400.00 |     30 | salesmgr
  7698 | BLAKE  | MANAGER   | 2850.00 |         |     30 | salesmgr
  7844 | TURNER | SALESMAN  | 1500.00 |    0.00 |     30 | salesmgr
  7900 | JAMES  | CLERK     |  950.00 |         |     30 | salesmgr
(14 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

An unqualified `UPDATE` command (no `WHERE` clause) is issued by the `salesmgr` user:

```
edb=> UPDATE vpemp SET comm = sal * .75;
UPDATE 6
```

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 has been changed only for those rows where `authid` contains `salesmgr`. All other rows are unchanged.

```
edb=> SELECT * FROM vpemp;
 empno | ename  |    job    |   sal   |  comm   | deptno |   authid
-------+--------+-----------+---------+---------+--------+-------------
  7782 | CLARK  | MANAGER   | 2450.00 |         |     10 |
  7839 | KING   | PRESIDENT | 5000.00 |         |     10 |
  7934 | MILLER | CLERK     | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK     |  800.00 |         |     20 | researchmgr
  7566 | JONES  | MANAGER   | 2975.00 |         |     20 | researchmgr
  7788 | SCOTT  | ANALYST   | 3000.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK     | 1100.00 |         |     20 | researchmgr
  7902 | FORD   | ANALYST   | 3000.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  | 1600.00 | 1200.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN  | 1250.00 |  937.50 |     30 | salesmgr
  7654 | MARTIN | SALESMAN  | 1250.00 |  937.50 |     30 | salesmgr
  7698 | BLAKE  | MANAGER   | 2850.00 | 2137.50 |     30 | salesmgr
  7844 | TURNER | SALESMAN  | 1500.00 | 1125.00 |     30 | salesmgr
  7900 | JAMES  | CLERK     |  950.00 |  712.50 |     30 | salesmgr
(14 rows)
```

Furthermore, since the `update_check` parameter was set to `TRUE` in the `ADD_POLICY` procedure, the following `INSERT` command throws an exception since the value given for the `authid` column, `researchmgr`, does not match the session user, which is `salesmgr`, and hence, fails the policy.

```
edb=> INSERT INTO vpemp VALUES (9001,'SMITH','ANALYST',3200.00,NULL,20,
'researchmgr');
ERROR:  policy with check option violation
DETAIL:  Policy predicate was evaluated to FALSE with the updated values
```

If `update_check` was set to `FALSE`, the preceding `INSERT` command would have succeeded.

The following example illustrates the use of the `sec_relevant_cols` parameter to apply a policy only when certain columns are referenced in the SQL command. The following policy function is used for this example, which selects rows where the employee salary is less than `2000`.

```
CREATE OR REPLACE FUNCTION sal_lt_2000 (
    p_schema        VARCHAR2,
    p_object        VARCHAR2
)
RETURN VARCHAR2
IS
BEGIN
    RETURN 'sal < 2000';
END
```

The policy is created so that it is enforced only if a `SELECT` command includes columns `sal` or `comm`:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_salary';
    v_function_schema       VARCHAR2(30) := 'enterprisedb';
    v_policy_function       VARCHAR2(30) := 'sal_lt_2000';
    v_statement_types       VARCHAR2(30) := 'SELECT';
    v_sec_relevant_cols     VARCHAR2(30) := 'sal,comm';
BEGIN
    DBMS_RLS.ADD_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name,
        v_function_schema,
        v_policy_function,
        v_statement_types,
        sec_relevant_cols => v_sec_relevant_cols
    );
END;
```

If a query does not reference columns `sal` or `comm`, then the policy is not applied. The following query returns all 14 rows of table `vpemp`:

```
edb=# SELECT empno, ename, job, deptno, authid FROM vpemp;
 empno | ename  |    job    | deptno |   authid
-------+--------+-----------+--------+-------------
  7782 | CLARK  | MANAGER   |     10 |
  7839 | KING   | PRESIDENT |     10 |
  7934 | MILLER | CLERK     |     10 |
  7369 | SMITH  | CLERK     |     20 | researchmgr
  7566 | JONES  | MANAGER   |     20 | researchmgr
  7788 | SCOTT  | ANALYST   |     20 | researchmgr
  7876 | ADAMS  | CLERK     |     20 | researchmgr
  7902 | FORD   | ANALYST   |     20 | researchmgr
  7499 | ALLEN  | SALESMAN  |     30 | salesmgr
  7521 | WARD   | SALESMAN  |     30 | salesmgr
  7654 | MARTIN | SALESMAN  |     30 | salesmgr
  7698 | BLAKE  | MANAGER   |     30 | salesmgr
  7844 | TURNER | SALESMAN  |     30 | salesmgr
  7900 | JAMES  | CLERK     |     30 | salesmgr
(14 rows)
```

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` as shown by the following:

```
edb=# SELECT empno, ename, job, sal, comm, deptno, authid FROM vpemp;
 empno | ename  |   job    |   sal   |  comm   | deptno |   authid
-------+--------+----------+---------+---------+--------+-------------
  7934 | MILLER | CLERK    | 1300.00 |         |     10 |
  7369 | SMITH  | CLERK    |  800.00 |         |     20 | researchmgr
  7876 | ADAMS  | CLERK    | 1100.00 |         |     20 | researchmgr
  7499 | ALLEN  | SALESMAN | 1600.00 | 1200.00 |     30 | salesmgr
  7521 | WARD   | SALESMAN | 1250.00 |  937.50 |     30 | salesmgr
  7654 | MARTIN | SALESMAN | 1250.00 |  937.50 |     30 | salesmgr
  7844 | TURNER | SALESMAN | 1500.00 | 1125.00 |     30 | salesmgr
  7900 | JAMES  | CLERK    |  950.00 |  712.50 |     30 | salesmgr
(8 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

DROP_POLICY

The `DROP_POLICY` procedure deletes an existing policy. The policy function and database object associated with the policy are not deleted by the `DROP_POLICY` procedure.

You must be a superuser to execute this procedure.

```
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 be deleted.

Examples

The following example deletes policy `secure_update` on table `public.vpemp`:

```
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.

```
ENABLE_POLICY(<object_schema> VARCHAR2, <object_name> VARCHAR2,
  <policy_name> VARCHAR2, <enable> BOOLEAN)
```

Parameters

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 be enabled or disabled.

`enable`

When set to `TRUE`, the policy is enabled. When set to `FALSE`, the policy is disabled.

Examples

The following example disables policy `secure_update` on table `public.vpemp`:

```
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;
```

11.2.15 DBMS_SCHEDULER

The `DBMS_SCHEDULER` package provides a way to create and manage Oracle-styled jobs, programs and job schedules. The `DBMS_SCHEDULER` package implements the following functions and procedures:

Function/Procedure	Return Type	Description
`CREATE_JOB(job_name, job_type, job_action, number_of_arguments, start_date, repeat_interval, end_date, job_class, enabled, auto_drop, comments)`	n/a	Use the first form of the `CREATE_JOB` procedure to create a job, specifying program and schedule details by means of parameters.
`CREATE_JOB(job_name, program_name, schedule_name, job_class, enabled, auto_drop, comments)`	n/a	Use the second form of `CREATE_JOB` to create a job that uses a named program and named schedule.
`CREATE_PROGRAM(program_name, program_type, program_action, number_of_arguments, enabled, comments)`	n/a	Use `CREATE_PROGRAM` to create a program.
`CREATE_SCHEDULE(schedule_name, start_date, repeat_interval, end_date, comments)`	n/a	Use the `CREATE_SCHEDULE` procedure to create a schedule.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function/Procedure	Return Type	Description
`DEFINE_PROGRAM_ARGUMENT(program_name, argument_position, argument_name, argument_type, default_value, out_argument)`	n/a	Use the first form of the `DEFINE_PROGRAM_ARGUMENT` procedure to define a program argument that has a default value.
`DEFINE_PROGRAM_ARGUMENT(program_name, argument_position, argument_name, argument_type, out_argument)`	n/a	Use the first form of the `DEFINE_PROGRAM_ARGUMENT` procedure to define a program argument that does not have a default value.
`DISABLE(name, force, commit_semantics)`	n/a	Use the `DISABLE` procedure to disable a job or program.
`DROP_JOB(job_name, force, defer, commit_semantics)`	n/a	Use the `DROP_JOB` procedure to drop a job.
`DROP_PROGRAM(program_name, force)`	n/a	Use the `DROP_PROGRAM` procedure to drop a program.
`DROP_PROGRAM_ARGUMENT(program_name, argument_position)`	n/a	Use the first form of `DROP_PROGRAM_ARGUMENT` to drop a program argument by specifying the argument position.
`DROP_PROGRAM_ARGUMENT(program_name, argument_name)`	n/a	Use the second form of `DROP_PROGRAM_ARGUMENT` to drop a program argument by specifying the argument name.
`DROP_SCHEDULE(schedule_name, force)`	n/a	Use the `DROP SCHEDULE` procedure to drop a schedule.
`ENABLE(name, commit_semantics)`	n/a	Use the `ENABLE` command to enable a program or job.
`EVALUATE_CALENDAR_STRING(calendar_string, start_date, return_date_after, next_run_date)`	n/a	Use `EVALUATE_CALENDAR_STRING` to review the execution date described by a user-defined calendar schedule.
`RUN_JOB(job_name, use_current_session, manually)`	n/a	Use the `RUN_JOB` procedure to execute a job immediately.
`SET_JOB_ARGUMENT_VALUE(job_name, argument_position, argument_value)`	n/a	Use the first form of `SET_JOB_ARGUMENT` value to set the value of a job argument described by the argument's position.
`SET_JOB_ARGUMENT_VALUE(job_name, argument_name, argument_value)`	n/a	Use the second form of `SET_JOB_ARGUMENT` value to set the value of a job argument described by the argument's name.

Advanced Server's implementation of `DBMS_SCHEDULER` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

The `DBMS_SCHEDULER` package is dependent on the pgAgent service; you must have a pgAgent service installed and running on your server before using `DBMS_SCHEDULER`.

Before using `DBMS_SCHEDULER`, a database superuser must create the catalog tables in which the `DBMS_SCHEDULER` programs, schedules and jobs are stored. Use the `psql` client to connect to the database, and invoke the command:

```
CREATE EXTENSION dbms_scheduler;
```

By default, the `dbms_scheduler` extension resides in the `contrib/dbms_scheduler_ext` subdirectory (under the Advanced Server installation).

Note that after creating the `DBMS_SCHEDULER` tables, only a superuser will be able to perform a dump or reload of the database.

11.2.15.1 Using Calendar Syntax to Specify a Repeating Interval

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `CREATE_JOB` and `CREATE_SCHEDULE` procedures use Oracle-styled calendar syntax to define the interval with which a job or schedule is repeated. You should provide the scheduling information in the `repeat_interval` parameter of each procedure.

`repeat_interval` is a value (or series of values) that define the interval between the executions of the scheduled job. Each value is composed of a token, followed by an equal sign, followed by the unit (or units) on which the schedule will execute. Multiple token values must be separated by a semi-colon (;).

For example, the following value:

```
FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;BYMINUTE=45
```

Defines a schedule that is executed each weeknight at 5:45.

The token types and syntax described in the table below are supported by Advanced Server:

Token type	Syntax	Valid Values
`FREQ`	`FREQ=predefined_interval`	Where `predefined_interval` is one of the following: `YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY`. The `SECONDLY` keyword is not 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`
`BYMONTHDAY`	`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.
		`Monday` | `MON` | `1` |
		`Tuesday` | `TUE` | `2` |
		`Wednesday` | `WED` | `3` |
		`Thursday` | `THU` | `4` |
		`Friday` | `FRI` | `5` |
		`Saturday` | `SAT` | `6` |
		`Sunday` | `SUN` | `7` |
`BYDATE`	`BYDATE=date(,date)...`	Where date is `YYYYMMDD`. `YYYY` is a four-digit year representation of the year, `MM` is a two-digit representation of the month, and `DD` is a two-digit day representation of the day.
`BYDATE`	`BYDATE=date(,date)...`	Where date is `MMDD`. `MM` is a two-digit representation of the month, and `DD` is a two-digit day representation of the day
`BYHOUR`	`BYHOUR=hour`	Where `hour` is a value from `0` through `23`.
`BYMINUTE`	`BYMINUTE=minute`	Where `minute` is a value from `0` through `59`.

11.2.15.2 CREATE_JOB

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Use the `CREATE_JOB` procedure to create a job. The procedure comes in two forms; the first form of the procedure specifies a schedule within the job definition, as well as a job action that will be invoked when the job executes:

```
CREATE_JOB(
  <job_name> IN VARCHAR2,
  <job_type> IN VARCHAR2,
  <job_action> IN VARCHAR2,
  <number_of_arguments> IN PLS_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 will execute, and specifies the name of a program that will 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 `PLSQL_BLOCK` or `STORED_PROCEDURE`.

`job_action`

- If `job_type` is `PLSQL_BLOCK`, `job_action` specifies the content of the PL/SQL block that will be invoked 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 that the job should be scheduled to execute when the job is enabled.

`repeat_interval`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`repeat_interval` is a `VARCHAR2` value that specifies how often the job will repeat. If a `repeat_interval` is not specified, the job will execute 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 will no longer execute. If a date is specified, the `end_date` must be after `start_date`. The default value is `NULL`.

Please note that if an `end_date` is not specified and a `repeat_interval` is specified, the job will repeat indefinitely until it is disabled.

`program_name`

`program_name` is the name of a program that will be executed by the job.

`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 will be changed to `DISABLED` after the time specified in `end_date`.

`comments`

Use the `comments` parameter to specify a comment about the job.

Example

The following example demonstrates a call to the `CREATE_JOB` procedure:

```
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 will execute until it is disabled by the `DBMS_SCHEDULER.DISABLE` procedure.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.15.3 CREATE_PROGRAM

Use the `CREATE_PROGRAM` procedure to create a `DBMS_SCHEDULER` program. The signature is:

```
CREATE_PROGRAM(
  <program_name> IN VARCHAR2,
  <program_type> IN VARCHAR2,
  <program_action> IN VARCHAR2,
  <number_of_arguments> IN PLS_INTEGER DEFAULT 0,
  <enabled> IN BOOLEAN DEFAULT FALSE,
  <comments> IN VARCHAR2 DEFAULT NULL)
```

Parameters

`program_name`

`program_name` specifies the name of the program that is being created.

`program_type`

`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 will execute 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`

`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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following call to the `CREATE_PROGRAM` procedure creates a program named `update_log`:

```
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,
    comments          => '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 will be enabled when the `CREATE_PROGRAM` procedure executes.

11.2.15.4 CREATE_SCHEDULE

Use the `CREATE_SCHEDULE` procedure to create a job schedule. The signature of the `CREATE_SCHEDULE` 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 a `start_date` is not specified, 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 will repeat. If a `repeat_interval` is not specified, the job will execute 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 will return an error.

`end_date`

`end_date` is a `TIMESTAMP WITH TIME ZONE` value that specifies a time after which the schedule will no longer execute. If a date is specified, the `end_date` must be after the `start_date`. The default value is `NULL`.

Note: If a `repeat_interval` is specified and an `end_date` is not specified, the schedule will repeat indefinitely until it is disabled.

`comments`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Use the `comments` parameter to specify a comment about the schedule; by default, this parameter is `NULL`.

Example

The following 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 pm, effective after June 1, 2013. Since no `end_date` is specified, the schedule will execute indefinitely until it is disabled with `DBMS_SCHEDULER.DISABLE`.

11.2.15.5 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:

```
DEFINE_PROGRAM_ARGUMENT(
  <program_name> IN VARCHAR2,
  <argument_position> IN PLS_INTEGER,
  <argument_name> IN VARCHAR2 DEFAULT NULL,
  <argument_type> IN VARCHAR2,
  <out_argument> IN BOOLEAN DEFAULT FALSE)
```

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 is passed to the program.

`argument_name`

`argument_name` specifies the optional name of the argument. By default, `argument_name` is `NULL`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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` will be overridden by a value specified by the job when the job executes.

`out_argument IN BOOLEAN DEFAULT FALSE`

`out_argument` is not currently used; if specified, the value must be `FALSE`.

Example

The following code fragment uses the `DEFINE_PROGRAM_ARGUMENT` procedure to define the first and second arguments in a program named `add_emp`:

```
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`; the second argument does not have a default value.

11.2.15.6 DISABLE

Use the `DISABLE` procedure to disable a program or a job. The signature of the `DISABLE` procedure is:

```
DISABLE(
  <name> IN VARCHAR2,
  <force> IN BOOLEAN DEFAULT FALSE,
  <commit_semantics> IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```

Parameters

`name`

`name` specifies the name of the program or job that is being disabled.

`force`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`force` is accepted for compatibility, and ignored.

`commit_semantics`

`commit_semantics` instructs 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`, instructing the server to stop when it encounters an error. Any programs or jobs that were successfully disabled prior to the error will be 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`:

```
DBMS_SCHEDULER.DISABLE('update_emp');
```

11.2.15.7 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 is being dropped.

`force`

`force` is accepted for compatibility, and ignored.

`defer`

`defer` is accepted for compatibility, and ignored.

`commit_semantics`

`commit_semantics` instructs 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`, instructing 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 `DROP_JOB` drops a job named `update_log`:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DBMS_SCHEDULER.DROP_JOB('update_log');
```

11.2.15.8 DROP_PROGRAM

The `DROP_PROGRAM` procedure to drop a program. The signature of the `DROP_PROGRAM` procedure is:

```
DROP_PROGRAM(
  <program_name> IN VARCHAR2,
  <force> IN BOOLEAN DEFAULT FALSE)
```

Parameters

`program_name`

`program_name` specifies the name of the program that is being dropped.

`force`

`force` is a `BOOLEAN` value that instructs the server how to handle programs with dependent jobs.

- Specify `FALSE` to instruct the server to return an error if the program is referenced by a job.
- Specify `TRUE` to instruct the server 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');
```

11.2.15.9 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 which argument to drop:

```
DROP_PROGRAM_ARGUMENT(
  <program_name> IN VARCHAR2,
  <argument_position> IN PLS_INTEGER)
```

The second form takes the argument name:

```
DROP_PROGRAM_ARGUMENT(
  <program_name> IN VARCHAR2,
  <argument_name> IN VARCHAR2)
```

Parameters

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`program_name`

`program_name` specifies the name of the program that is being modified.

`argument_position`

`argument_position` specifies the position of the argument that is being dropped.

`argument_name`

`argument_name` specifies the name of the argument that is being dropped.

Examples

The following call to `DROP_PROGRAM_ARGUMENT` drops the first argument in the `update_emp` program:

```
DBMS_SCHEDULER.DROP_PROGRAM_ARGUMENT('update_emp', 1);
```

The following call to `DROP_PROGRAM_ARGUMENT` drops an argument named `emp_name`:

```
DBMS_SCHEDULER.DROP_PROGRAM_ARGUMENT(update_emp', 'emp_name');
```

11.2.15.10 DROP_SCHEDULE

Use the `DROP_SCHEDULE` procedure to drop a schedule. The signature is:

```
DROP_SCHEDULE(
  <schedule_name> IN VARCHAR2,
  <force> IN BOOLEAN DEFAULT FALSE)
```

Parameters

`schedule_name`

`schedule_name` specifies the name of the schedule that is being dropped.

`force`

`force` specifies the behavior of the server if the specified schedule is referenced by any job:

- Specify `FALSE` to instruct the server to return an error if the specified schedule is referenced by a job. This is the default behavior.
- Specify `TRUE` to instruct the server to disable to any jobs that use the specified schedule before dropping the schedule. Any running jobs will be allowed to complete before the schedule is dropped.

Example

The following call to `DROP_SCHEDULE` drops a schedule named `weeknights_at_5`:

```
DBMS_SCHEDULER.DROP_SCHEDULE('weeknights_at_5', TRUE);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The server will disable any jobs that use the schedule before dropping the schedule.

11.2.15.11 ENABLE

Use the `ENABLE` procedure to enable a disabled program or job.

The signature of the `ENABLE` procedure is:

```
ENABLE(
  <name> IN VARCHAR2,
  <commit_semantics> IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```

Parameters

`name`

`name` specifies the name of the program or job that is being enabled.

`commit_semantics`

`commit_semantics` instructs 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`, instructing 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');
```

11.2.15.12 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 will return the date and time that a specified schedule will execute 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`calendar_string` is the calendar string that describes a `repeat_interval` that is being evaluated.

`start_date IN TIMESTAMP WITH TIME ZONE`

`start_date` is the date and time after which the `repeat_interval` will become valid.

`return_date_after`

Use the `return_date_after` parameter to specify the date and time that `EVALUATE_CALENDAR_STRING` should 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` will return 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 will contain the first occurrence of the schedule after the date specified by the `return_date_after` parameter.

Example

The following example evaluates a calendar string and returns the first date and time that the schedule will be executed after June 15, 2013:

```
DECLARE
  result     TIMESTAMP;
BEGIN

  DBMS_SCHEDULER.EVALUATE_CALENDAR_STRING
  (
    'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',
    '15-JUN-2013', NULL, result
  );

    DBMS_OUTPUT.PUT_LINE('next_run_date: ' || result);
END;
/

next_run_date: 17-JUN-13 05.00.00.000000 PM
```

June 15, 2013 is a Saturday; the schedule will not execute until Monday, June 17, 2013 at 5:00 pm.

11.2.15.13 RUN_JOB

Use the `RUN_JOB` procedure to execute a job immediately. The signature of the `RUN_JOB` procedure is:

```
RUN_JOB(
  <job_name> IN VARCHAR2,
  <use_current_session> IN BOOLEAN DEFAULT TRUE
```

Parameters

`job_name`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`job_name` specifies the name of the job that will execute.

`use_current_session`

By default, the job will execute in the current session. If specified, `use_current_session` must be set to `TRUE` ; if `use_current_session` is set to `FALSE` , Advanced Server will return an error.

Example

The following call to `RUN_JOB` executes a job named `update_log` :

```
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.

11.2.15.14 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 which argument should be modified by position:

```
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 which argument to modify:

```
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`

`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 will be set.

`argument_name`

Use `argument_name` to specify the argument by name for which the value will be set.

`argument_value`

`argument_value` specifies the new value of the argument.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Examples

The following 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');
```

The following example sets the `emp_name` argument to `SMITH`:

```
DBMS_SCHEDULER.SET_JOB_ARGUMENT_VALUE('update_emp', 'emp_name', 'SMITH');
```

11.2.16 DBMS_SESSION

Advanced Server provides support for the following `DBMS_SESSION.SET_ROLE` procedure:

Function/Procedure	Return Type	Description
`SET_ROLE(role_cmd)`	n/a	Executes a `SET ROLE` statement followed by the string value specified in `role_cmd`.

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 will use 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

The following 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');
```

11.2.17 DBMS_SQL

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The DBMS_SQL package provides an application interface compatible with Oracle databases to the EnterpriseDB dynamic SQL functionality. With DBMS_SQL you can construct queries and other commands at run time (rather than when you write the application). EnterpriseDB Advanced Server offers native support for dynamic SQL; DBMS_SQL provides a way to use dynamic SQL in a fashion compatible with Oracle databases without modifying your application.

DBMS_SQL assumes the privileges of the current user when executing dynamic SQL statements.

Function/Procedure	Function or Procedure	Return Type	Description
BIND_VARIABLE(c, name, value [, out_value_size])	Procedure	n/a	Bind a value to a variable.
BIND_VARIABLE_CHAR(c, name, value [, out_value_size])	Procedure	n/a	Bind a CHAR value to a variable.
BIND_VARIABLE_RAW(c, name, value [, out_value_size])	Procedure	n/a	Bind a RAW value to a variable.
CLOSE_CURSOR(c IN OUT)	Procedure	n/a	Close a cursor.
COLUMN_VALUE(c, position, value OUT [, column_error OUT [, actual_length OUT]])	Procedure	n/a	Return a column value into a variable.
COLUMN_VALUE_CHAR(c, position, value OUT [, column_error OUT [, actual_length OUT]])	Procedure	n/a	Return a CHAR column value into a variable.
COLUMN_VALUE_RAW(c, position, value OUT [, column_error OUT [, actual_length OUT]])	Procedure	n/a	Return a RAW column value into a variable.
DEFINE_COLUMN(c, position, column [, column_size])	Procedure	n/a	Define a column in the SELECT list.
DEFINE_COLUMN_CHAR(c, position, column, column_size)	Procedure	n/a	Define a CHAR column in the SELECT list.
DEFINE_COLUMN_RAW(c, position, column, column_size)	Procedure	n/a	Define a RAW column in the SELECT list.
DESCRIBE_COLUMNS	Procedure	n/a	Defines columns to hold a cursor result set.
EXECUTE(c)	Function	INTEGER	Execute a cursor.
EXECUTE_AND_FETCH(c [, exact])	Function	INTEGER	Execute a cursor and fetch a single row.
FETCH_ROWS(c)	Function	INTEGER	Fetch rows from the cursor.
IS_OPEN(c)	Function	BOOLEAN	Check if a cursor is open.
LAST_ROW_COUNT	Function	INTEGER	Return cumulative number of rows fetched.
OPEN_CURSOR	Function	INTEGER	Open a cursor.
PARSE(c, statement, language_flag)	Procedure	n/a	Parse a statement.

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 above are supported.

The following table lists the public variable available in the DBMS_SQL package.

Public Variables	Data Type	Value	Description
native	INTEGER	1	Provided for compatibility with Oracle syntax. See DBMS_SQL.PARSE for more information.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Public Variables	Data Type	Value	Description
V6	INTEGER	2	Provided for compatibility with Oracle syntax. See DBMS_SQL.PARSE for more information.
V7	INTEGER	3	Provided for compatibility with Oracle syntax. See DBMS_SQL.PARSE for more information

11.2.17.1 BIND_VARIABLE

The BIND_VARIABLE procedure provides the capability to associate a value with an IN or IN OUT bind variable in a SQL command.

```
BIND_VARIABLE(<c> INTEGER, <name> VARCHAR2,
  <value> { BLOB | CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }
  [, <out_value_size> INTEGER ])
```

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 assigned.

out_value_size

If name is an IN OUT variable, defines the maximum length of the output value. If not specified, the length of value is assumed.

Examples

The following anonymous block uses bind variables to insert a row into the emp table.

```
DECLARE
    curid           INTEGER;
    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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;

Number of rows processed: 1
```

11.2.17.2 BIND_VARIABLE_CHAR

The `BIND_VARIABLE_CHAR` procedure provides the capability to associate a `CHAR` value with an `IN` or `IN OUT` bind variable in a SQL command.

```
BIND_VARIABLE_CHAR(<c> INTEGER, <name> VARCHAR2, <value> CHAR
  [, <out_value_size> INTEGER ])
```

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 of type `CHAR` to be assigned.

`out_value_size`

If `name` is an `IN OUT` variable, defines the maximum length of the output value. If not specified, the length of `value` is assumed.

11.2.17.3 BIND_VARIABLE_RAW

The `BIND_VARIABLE_RAW` procedure provides the capability to associate a `RAW` value with an `IN` or `IN OUT` bind variable in a SQL command.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
BIND_VARIABLE_RAW(<c> INTEGER, <name> VARCHAR2, <value> RAW
  [, <out_value_size> INTEGER ])
```

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 of type `RAW` to be assigned.

`out_value_size`

If `name` is an `IN OUT` variable, defines the maximum length of the output value. If not specified, the length of `value` is assumed.

11.2.17.4 CLOSE_CURSOR

The `CLOSE_CURSOR` procedure closes an open cursor. The resources allocated to the cursor are released and it can no longer be used.

```
CLOSE_CURSOR(<c> IN OUT INTEGER)
```

Parameters

`c`

Cursor ID of the cursor to be closed.

Examples

The following example closes a previously opened cursor:

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
             .
             .
             .
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

11.2.17.5 COLUMN_VALUE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `COLUMN_VALUE` procedure defines a variable to receive a value from a cursor.

```
COLUMN_VALUE(<c> INTEGER, <position> INTEGER, <value> OUT { BLOB |
  CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }
  [, <column_error> OUT NUMBER [, <actual_length> OUT INTEGER ]])
```

Parameters

`c`

Cursor id of the cursor returning data to the variable being defined.

`position`

Position within the cursor of the returned data. The first value in the cursor is position 1.

`value`

Variable 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.

Examples

The following example shows the portion of an anonymous block that receives the values from a cursor using the `COLUMN_VALUE` procedure.

```
DECLARE
    curid           INTEGER;
    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
            .
            .
            .
    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,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,10) || '  ' ||
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
            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;
```

11.2.17.6 COLUMN_VALUE_CHAR

The `COLUMN_VALUE_CHAR` procedure defines a variable to receive a `CHAR` value from a cursor.

```
COLUMN_VALUE_CHAR(<c> INTEGER, <position> INTEGER, <value> OUT CHAR
  [, <column_error> OUT NUMBER [, <actual_length> OUT INTEGER ]])
```

Parameters

`c`

Cursor id of the cursor returning data to the variable being defined.

`position`

Position within the cursor of the returned data. 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.

11.2.17.7 COLUMN_VALUE_RAW

The `COLUMN_VALUE_RAW` procedure defines a variable to receive a `RAW` value from a cursor.

```
COLUMN_VALUE_RAW(<c> INTEGER, <position> INTEGER, <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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`position`

Position within the cursor of the returned data. 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.

11.2.17.8 DEFINE_COLUMN

The `DEFINE_COLUMN` procedure defines a column or expression in the `SELECT` list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN(<c> INTEGER, <position> INTEGER, <column> { BLOB |
  CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }
  [, <column_size> INTEGER ])
```

Parameters

`c`

Cursor id of the cursor associated with the `SELECT` command.

`position`

Position of the column or expression in the `SELECT` list that is being defined.

`column`

A variable that is of the same data type as the column or expression in position `position` of the `SELECT` list.

`column_size`

The maximum length of the returned data. `column_size` must be specified only if `column` is `VARCHAR2`. Returned data exceeding `column_size` is truncated to `column_size` characters.

Examples

The following shows how the `empno, ename, hiredate, sal`, and `comm` columns of the `emp` table are defined with the `DEFINE_COLUMN` procedure.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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);
            .
            .
            .
END;
```

The following shows an alternative to the prior example that produces the exact same results. Note that the lengths of the data types are irrelevant – the `empno, sal`, and `comm` columns will 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 will return data up to ten 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.

```
DECLARE
    curid           INTEGER;
    v_num           NUMBER(1);
    v_varchar       VARCHAR2(1);
    v_date          DATE;
    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_num);
    DBMS_SQL.DEFINE_COLUMN(curid,2,v_varchar,10);
    DBMS_SQL.DEFINE_COLUMN(curid,3,v_date);
    DBMS_SQL.DEFINE_COLUMN(curid,4,v_num);
    DBMS_SQL.DEFINE_COLUMN(curid,5,v_num);
            .
            .
            .
END;
```

11.2.17.9 DEFINE_COLUMN_CHAR

The `DEFINE_COLUMN_CHAR` procedure defines a `CHAR` column or expression in the `SELECT` list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN_CHAR(<c> INTEGER, <position> INTEGER, <column>
CHAR, <column_size> INTEGER)
```

Parameters

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`c`

Cursor id of the cursor associated with the `SELECT` command.

`position`

Position of the column or expression in the `SELECT` list that is 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.

11.2.17.10 DEFINE_COLUMN_RAW

The `DEFINE_COLUMN_RAW` procedure defines a `RAW` column or expression in the `SELECT` list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN_RAW(<c> INTEGER, <position> INTEGER, <column> RAW,
  <column_size> INTEGER)
```

Parameters

`c`

Cursor id of the cursor associated with the `SELECT` command.

`position`

Position of the column or expression in the `SELECT` list that is 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.

11.2.17.11 DESCRIBE_COLUMNS

The `DESCRIBE_COLUMNS` procedure describes the columns returned by a cursor.

```
DESCRIBE_COLUMNS(<c> INTEGER, <col_cnt> OUT INTEGER, <desc_t> OUT
  DESC_TAB);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameters

`c`

The cursor ID of the cursor.

`col_cnt`

The number of columns in cursor result set.

`desc_tab`

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:

Column Name	Type
`col_type`	`INTEGER`
`col_max_len`	`INTEGER`
`col_name`	`VARCHAR2(128)`
`col_name_len`	`INTEGER`
`col_schema_name`	`VARCHAR2(128)`
`col_schema_name_len`	`INTEGER`
`col_precision`	`INTEGER`
`col_scale`	`INTEGER`
`col_charsetid`	`INTEGER`
`col_charsetform`	`INTEGER`
`col_null_ok`	`BOOLEAN`

11.2.17.12 EXECUTE

The `EXECUTE` function executes a parsed SQL command or SPL block.

```
<status> INTEGER EXECUTE(<c> INTEGER)
```

Parameters

`c`

Cursor ID of the parsed SQL command or SPL block to be executed.

`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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DECLARE
    curid           INTEGER;
    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;
```

11.2.17.13 EXECUTE_AND_FETCH

Function `EXECUTE_AND_FETCH` executes a parsed `SELECT` command and fetches one row.

```
<status> INTEGER EXECUTE_AND_FETCH(<c> INTEGER [, <exact> BOOLEAN ])
```

Parameters

`c`

Cursor id of the cursor for the `SELECT` command to be executed.

`exact`

If set to `TRUE`, an exception is thrown if the number of rows in the result set is not 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 there is more than one row 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

The following stored procedure uses the `EXECUTE_AND_FETCH` function to retrieve one employee using the employee's name. An exception will be thrown if the employee is not found, or there is more than one employee with the same name.

```
CREATE OR REPLACE PROCEDURE select_by_name(
    p_ename         emp.ename%TYPE
)
IS
    curid           INTEGER;
    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 ' ||
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
                                    '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 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);
        DBMS_SQL.CLOSE_CURSOR(curid);
END;

EXEC select_by_name('MARTIN')

Number    : 7654
Name      : MARTIN
Hire Date : 09/28/1981
Salary    : 1250
Commission: 1400
Department: SALES
```

11.2.17.14 FETCH_ROWS

The `FETCH_ROWS` function retrieves a row from a cursor.

```
<status> INTEGER FETCH_ROWS(<c> INTEGER)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

The following examples fetches the rows from the `emp` table and displays the results.

```
DECLARE
    curid          INTEGER;
    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);
        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,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;

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00       .00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
7566   JONES       1981-04-02  2,975.00       .00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
7698   BLAKE       1981-05-01  2,850.00       .00
7782   CLARK       1981-06-09  2,450.00       .00
7788   SCOTT       1987-04-19  3,000.00       .00
7839   KING        1981-11-17  5,000.00       .00
7844   TURNER      1981-09-08  1,500.00       .00
7876   ADAMS       1987-05-23  1,100.00       .00
7900   JAMES       1981-12-03    950.00       .00
7902   FORD        1981-12-03  3,000.00       .00
7934   MILLER      1982-01-23  1,300.00       .00
```

11.2.17.15 IS_OPEN

The `IS_OPEN` function provides the capability to test if the given cursor is open.

```
<status> BOOLEAN IS_OPEN(<c> INTEGER)
```

Parameters

`c`

Cursor ID of the cursor to be tested.

`status`

Set to `TRUE` if the cursor is open, set to `FALSE` if the cursor is not open.

11.2.17.16 LAST_ROW_COUNT

The `LAST_ROW_COUNT` function returns the number of rows that have been currently fetched.

```
<rowcnt> INTEGER LAST_ROW_COUNT
```

Parameters

`rowcnt`

Number of row fetched thus far.

Examples

The following example uses the `LAST_ROW_COUNT` function to display the total number of rows fetched in the query.

```
DECLARE
    curid          INTEGER;
    v_empno        NUMBER(4);
    v_ename        VARCHAR2(10);
    v_hiredate     DATE;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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);
        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,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_OUTPUT.PUT_LINE('Number of rows: ' || DBMS_SQL.LAST_ROW_COUNT);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00       .00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00       .00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
7698   BLAKE       1981-05-01  2,850.00       .00
7782   CLARK       1981-06-09  2,450.00       .00
7788   SCOTT       1987-04-19  3,000.00       .00
7839   KING        1981-11-17  5,000.00       .00
7844   TURNER      1981-09-08  1,500.00       .00
7876   ADAMS       1987-05-23  1,100.00       .00
7900   JAMES       1981-12-03    950.00       .00
7902   FORD        1981-12-03  3,000.00       .00
7934   MILLER      1982-01-23  1,300.00       .00
Number of rows: 14
```

11.2.17.17 OPEN_CURSOR

The `OPEN_CURSOR` function creates a new cursor. A cursor must be used to parse and execute any dynamic SQL statement. Once a cursor has been opened, it can be re-used with the same or different SQL statements. The cursor does not have to be closed and re-opened in order to be re-used.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
<c> INTEGER OPEN_CURSOR
```

Parameters

`c`

Cursor ID number associated with the newly created cursor.

Examples

The following example creates a new cursor:

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
             .
             .
             .
END;
```

11.2.17.18 PARSE

The `PARSE` procedure parses a SQL command or SPL block. If the SQL command is a DDL command, it is immediately executed and does not require running the `EXECUTE` function.

```
PARSE(<c> INTEGER, <statement> VARCHAR2, <language_flag> INTEGER)
```

Parameters

`c`

Cursor ID of an open cursor.

`statement`

SQL command or SPL block to be parsed. 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 EnterpriseDB Advanced Server form.

Examples

The following anonymous block creates a table named, `job`. Note that DDL statements are executed immediately by the `PARSE` procedure and do not require a separate `EXECUTE` step.

```
DECLARE
    curid           INTEGER;
BEGIN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno NUMBER(3), ' ||
        'jname VARCHAR2(9))',DBMS_SQL.native);
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

The following inserts two rows into the `job` table.

```
DECLARE
    curid           INTEGER;
    v_sql           VARCHAR2(50);
    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;

Number of rows processed: 1
Number of rows processed: 1
```

The following anonymous block uses the `DBMS_SQL` package to execute a block containing two `INSERT` statements. Note that the end of the block contains a terminating semi-colon, while in the prior example, each individual `INSERT` statement does not have a terminating semi-colon.

```
DECLARE
    curid           INTEGER;
    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;
```

11.2.18 DBMS_UTILITY

The `DBMS_UTILITY` package provides support for the following various utility programs:

Function/Procedure	Function or Procedure	Return Type	Description

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function/Procedure	Function or Procedure	Return Type	Description
`ANALYZE_DATABASE(method [, estimate_rows [, estimate_percent [, method_opt ]]])`	Procedure	n/a	Analyze database tables.
`ANALYZE_PART_OBJECT(schema, object_name [, object_type [, command_type [, command_opt [, sample_clause ]]]])`	Procedure	n/a	Analyze a partitioned table.
`ANALYZE_SCHEMA(schema, method [, estimate_rows [, estimate_percent [, method_opt ]]])`	Procedure	n/a	Analyze schema tables.
`CANONICALIZE(name, canon_name OUT, canon_len)`	Procedure	n/a	Canonicalizes a string – e.g., strips off white space.
`COMMA_TO_TABLE(list, tablen OUT, tab OUT)`	Procedure	n/a	Convert a comma-delimited list of names to a table of names.
`DB_VERSION(version OUT, compatibility OUT)`	Procedure	n/a	Get the database version.
`EXEC_DDL_STATEMENT (parse_string)`	Procedure	n/a	Execute a DDL statement.
`FORMAT_CALL_STACK`	Function	`TEXT`	Formats the current call stack.
`GET_CPU_TIME`	Function	`NUMBER`	Get the current CPU time.
`GET_DEPENDENCY(type, schema, name)`	Procedure	n/a	Get objects that are dependent upon the given object..
`GET_HASH_VALUE(name, base, hash_size)`	Function	`NUMBER`	Compute a hash value.
`GET_PARAMETER_VALUE(parnam, intval OUT, strval OUT)`	Procedure	`BINARY_INTEGER`	Get database initialization parameter settings.
`GET_TIME`	Function	`NUMBER`	Get the current time.
`NAME_TOKENIZE(name, a OUT, b OUT, c OUT, dblink OUT, nextpos OUT)`	Procedure	n/a	Parse the given name into its component parts.
`TABLE_TO_COMMA(tab, tablen OUT, list OUT)`	Procedure	n/a	Convert a table of names to a comma-delimited list.

Advanced Server's implementation of `DBMS_UTILITY` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

The following table lists the public variables available in the `DBMS_UTILITY` package.

Public Variables	Data Type	Value	Description
`inv_error_on_restrictions`	`PLS_INTEGER`	`1`	Used by the `INVALIDATE` procedure.
`lname_array`	`TABLE`		For lists of long names.
`uncl_array`	`TABLE`		For lists of users and names.

LNAME_ARRAY

The `LNAME_ARRAY` is for storing lists of long names including fully-qualified names.

```
TYPE lname_array IS TABLE OF VARCHAR2(4000) INDEX BY BINARY_INTEGER;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 provide the capability to 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_statistics system` table.

`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 within the current database.
- `ANALYZE_SCHEMA` analyzes all tables in a given schema (within 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 ]]])

ANALYZE_SCHEMA(<schema> VARCHAR2, <method> VARCHAR2
  [, <estimate_rows> NUMBER [, <estimate_percent> NUMBER
  [, <method_opt> VARCHAR2 ]]])

ANALYZE_PART_OBJECT(<schema> VARCHAR2, <object_name> VARCHAR2
  [, <object_type> CHAR [, <command_type> CHAR
  [, <command_opt> VARCHAR2 [, <sample_clause> ]]]])
```

Parameters - `ANALYZE_DATABASE` and `ANALYZE_SCHEMA`

`method`

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 is 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 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 method is `ESTIMATE`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

This argument is ignored, but is included for compatibility.

`method_opt`

Object types to be analyzed. 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 - `ANALYZE_PART_OBJECT`

`schema`

Name of the schema whose objects are to be analyzed.

`object_name`

Name of the partitioned object to be analyzed.

`object_type`

Type of object to be analyzed. 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 upon on a specified number of rows or a percentage of rows in the `sample_clause` clause; `C` - compute exact statistics; or `V` – validate the structure and integrity of the partitions.

This argument is ignored, but is included for compatibility.

`command_opt`

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_clause`

If `command_type` is `E`, contains the following clause to specify the number of rows or percentage or rows on which to base the estimate.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 is not 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 is not double-quoted and does not contain periods, uppercases all alphabetic characters and eliminates leading and trailing spaces.
- If the string is double-quoted and does not 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 and returns the non-double-quoted portions uppercased and enclosed in double quotes.

```
CANONICALIZE(<name> VARCHAR2, <canon_name> OUT VARCHAR2,
  <canon_len> BINARY_INTEGER)
```

Parameters

`name`

String to be canonicalized.

`canon_name`

The canonicalized string.

`canon_len`

Number of bytes in `name` to canonicalize starting from the first character.

Examples

The following procedure applies the `CANONICALIZE` procedure on its input parameter and displays the results.

```
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 || '<==');
    DBMS_OUTPUT.PUT_LINE('Length: ' || LENGTH(v_canon));
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
EXEC canonicalize('Identifier')
Canonicalized name ==>IDENTIFIER<==
Length: 10

EXEC canonicalize('"Identifier"')
Canonicalized name ==>Identifier<==
Length: 10

EXEC canonicalize('"_+142%"')
Canonicalized name ==>_+142%<==
Length: 6

EXEC canonicalize('abc.def.ghi')
Canonicalized name ==>"ABC"."DEF"."GHI"<==
Length: 17

EXEC canonicalize('"abc.def.ghi"')
Canonicalized name ==>abc.def.ghi<==
Length: 11

EXEC canonicalize('"abc".def."ghi"')
Canonicalized name ==>"abc"."DEF"."ghi"<==
Length: 17

EXEC canonicalize('"abc.def".ghi')
Canonicalized name ==>"abc.def"."GHI"<==
Length: 15
```

COMMA_TO_TABLE

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. The names must be formatted 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`.

`LNAME_ARRAY`

A `DBMS_UTILITY LNAME_ARRAY` (as described in the LNAME_ARRAY section).

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`UNCL_ARRAY`

A `DBMS_UTILITY UNCL_ARRAY` (as described in the UNCL_ARRAY section).

Examples

The following procedure uses the `COMMA_TO_TABLE` procedure to convert a list of names to a table. The table entries are then displayed.

```
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.

```
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.

```
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);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END;

Version: EnterpriseDB 11.0.0 on i686-pc-linux-gnu, compiled by GCC gcc
(GCC) 4.1.2 20080704 (Red Hat 4.1.2-48), 32-bit
Compatibility: EnterpriseDB 11.0.0 on i686-pc-linux-gnu, compiled by GCC
gcc (GCC) 4.1.220080704 (Red Hat 4.1.2-48), 32-bit
```

EXEC_DDL_STATEMENT

The `EXEC_DDL_STATEMENT` provides the capability to execute a `DDL` command.

```
EXEC_DDL_STATEMENT(<parse_string> VARCHAR2)
```

Parameters

`parse_string`

The DDL command to be executed.

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` does not include a valid DDL statement, Advanced Server returns the following 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, Advanced Server's behavior differs from Oracle's; Oracle accepts the invalid `parse_string` without complaint.

FORMAT_CALL_STACK

The `FORMAT_CALL_STACK` function returns the formatted contents of the current call stack.

```
DBMS_UTILITY.FORMAT_CALL_STACK
return VARCHAR2
```

This function can be used in a stored procedure, function or package to return the current call stack in a readable format. This function is useful for debugging purposes.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

GET_CPU_TIME

The `GET_CPU_TIME` function returns the CPU time in hundredths of a second from some arbitrary point in time.

```
<cputime> NUMBER GET_CPU_TIME
```

Parameters

`cputime`

Number of hundredths of a second of CPU time.

Examples

The following `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 provides the capability to list the objects that are dependent upon the specified object. `GET_DEPENDENCY` does not 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 dependencies are to be obtained.

Examples

The following anonymous block finds dependencies on the `EMP` table.

```
BEGIN
    DBMS_UTILITY.GET_DEPENDENCY('TABLE','public','EMP');
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 provides the capability to compute a hash value for a given string.

```
<hash> NUMBER GET_HASH_VALUE(<name> VARCHAR2, <base> NUMBER,
   <hash_size> NUMBER)
```

Parameters

`name`

The string for which a hash value is to be computed.

`base`

Starting value at which hash values are to be generated.

`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;
    FOR r_emp IN emp_cur LOOP
        DBMS_OUTPUT.PUT_LINE(RPAD(r_emp.ename,10) || ' ' ||
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
           r_hash(r_emp.ename));
    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 provides the capability to retrieve 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 is to be returned. The parameters are listed in the `pg_settings` system view.

`intval`

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.

```
DECLARE
    v_intval        INTEGER;
    v_strval        VARCHAR2(80);
BEGIN
    DBMS_UTILITY.GET_PARAMETER_VALUE('max_fsm_pages', v_intval, v_strval);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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 provides the capability to return the current time in hundredths of a second.

```
<time> NUMBER GET_TIME
```

Parameters

`time`

Number of hundredths of a second from the time in which the program is started.

Examples

The following example shows calls to the `GET_TIME` function.

```
SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
----------
  1555860

SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
----------
  1556037
```

NAME_TOKENIZE

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,
  <b> OUT VARCHAR2, <c> OUT VARCHAR2, <dblink> OUT VARCHAR2,
  <nextpos> OUT BINARY_INTEGER)
```

Parameters

`name`

String containing a name in the following format:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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

The following stored procedure is used to display the returned parameter values of the `NAME_TOKENIZE` procedure for various names.

```
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`:

```
BEGIN
    name_tokenize('emp');
END;

name   : emp
a      : EMP
b      :
c      :
dblink :
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
nextpos: 3
```

Tokenize the name, `edb.list_emp`:

```
BEGIN
    name_tokenize('edb.list_emp');
END;

name    : edb.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       : edb
b       : Emp_Admin
c       : UPDATE_EMP_SAL
dblink  :
nextpos: 32
```

Tokenize the name `edb.emp@edb_dblink`:

```
BEGIN
    name_tokenize('edb.emp@edb_dblink');
END;

name    : edb.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. The names must be formatted as valid identifiers.

```
TABLE_TO_COMMA(<tab> { LNAME_ARRAY | UNCL_ARRAY },
  <tablen> OUT BINARY_INTEGER, <list> OUT VARCHAR2)
```

Parameters

`tab`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Table containing names.

`LNAME_ARRAY`

A `DBMS_UTILITY LNAME_ARRAY` (as described in the LNAME ARRAY section).

`UNCL_ARRAY`

A `DBMS_UTILITY UNCL_ARRAY` (as described the UNCL_ARRAY section).

`tablen`

Number of entries in `list`.

`list`

Comma-delimited list of names from `tab`.

Examples

The following 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 is displayed.

```
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');
    DBMS_OUTPUT.PUT_LINE('-------------');
    FOR i IN 1..v_length LOOP
        DBMS_OUTPUT.PUT_LINE(r_lname(i));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('-------------');
    DBMS_UTILITY.TABLE_TO_COMMA(r_lname,v_listlen,v_list);
    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
```

11.2.19 UTL_ENCODE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The UTL_ENCODE package provides a way to encode and decode data. Advanced Server supports the following functions and procedures:

Function/Procedure	Return Type	Description
BASE64_DECODE(r)	RAW	Use the BASE64_DECODE function to translate a Base64 encoded string to the original RAW value.
BASE64_ENCODE(r)	RAW	Use the BASE64_ENCODE function to translate a RAW string to an encoded Base64 value.
BASE64_ENCODE(loid)	TEXT	Use the BASE64_ENCODE function to translate a TEXT string to an encoded Base64 value.
MIMEHEADER_DECODE(buf)	VARCHAR2	Use the MIMEHEADER_DECODE function to translate an encoded MIMEHEADER formatted string to it's original value.
MIMEHEADER_ENCODE(buf, encode_charset, encoding)	VARCHAR2	Use the MIMEHEADER_ENCODE function to convert and encode a string in MIMEHEADER format.
QUOTED_PRINTABLE_DECODE(r)	RAW	Use the QUOTED_PRINTABLE_DECODE function to translate an encoded string to a RAW value.
QUOTED_PRINTABLE_ENCODE(r)	RAW	Use the QUOTED_PRINTABLE_ENCODE function to translate an input string to a quoted-printable formatted RAW value.
TEXT_DECODE(buf, encode_charset, encoding)	VARCHAR2	Use the TEXT_DECODE function to decode a string encoded by TEXT_ENCODE.
TEXT_ENCODE(buf, encode_charset, encoding)	VARCHAR2	Use the TEXT_ENCODE function to translate a string to a user-specified character set, and then encode the string.
UUDECODE(r)	RAW	Use the UUDECODE function to translate a uuencode encoded string to a RAW value.
UUENCODE(r, type, filename, permission)	RAW	Use the UUENCODE function to translate a RAW string to an encoded uuencode value.

11.2.19.1 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 that will be translated 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, refer to the Postgres Core Documentation available at:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

https://www.postgresql.org/docs/11/static/datatype-binary.html

The following example first encodes (using `BASE64_ENCODE`), and then decodes (using `BASE64_DECODE`) a string that contains the text `abc:`

```
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)
```

11.2.19.2 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

`r`

`r` specifies the `RAW` string that will be translated to Base64.

`loid`

`loid` specifies the object ID of a large object that will be translated to Base64.

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, refer to the Postgres Core Documentation available at:

https://www.postgresql.org/docs/11/static/datatype-binary.html

The following example first encodes (using `BASE64_ENCODE`), and then decodes (using `BASE64_DECODE`) a string that contains the text `abc:`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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)
```

11.2.19.3 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

`buf`

`buf` contains the value (encoded by `MIMEHEADER_ENCODE)` that will be decoded.

Examples

The following examples use the `MIMEHEADER_ENCODE` and `MIMEHEADER_DECODE` functions to first encode, and then decode a string:

```
edb=# SELECT UTL_ENCODE.MIMEHEADER_ENCODE('What is the date?') FROM DUAL;
      mimeheader_encode
------------------------------
 =?UTF8?Q?What is the date??=
(1 row)

edb=# SELECT UTL_ENCODE.MIMEHEADER_DECODE('=?UTF8?Q?What is the date??=')
FROM DUAL;
 mimeheader_decode
-------------------
 What is the date?
(1 row)
```

11.2.19.4 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)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

This function returns a `VARCHAR2` value.

Parameters

`buf`

`buf` contains the string that will be formatted and encoded. The string is a `VARCHAR2` value.

`encode_charset`

`encode_charset` specifies the character set to which the string will be converted before being formatted and encoded. 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 do not specify a value, `MIMEHEADER_ENCODE` will use quoted-printable encoding.
- `B` to enable base-64 encoding.

Examples

The following examples use the `MIMEHEADER_ENCODE` and `MIMEHEADER_DECODE` functions to first encode, and then decode a string:

```
edb=# SELECT UTL_ENCODE.MIMEHEADER_ENCODE('What is the date?') FROM DUAL;
      mimeheader_encode
------------------------------
 =?UTF8?Q?What is the date??=
(1 row)

edb=# SELECT UTL_ENCODE.MIMEHEADER_DECODE('=?UTF8?Q?What is the date??=')
FROM DUAL;
 mimeheader_decode
-------------------
 What is the date?
(1 row)
```

11.2.19.5 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 that will be decoded. The string is a `RAW` value, encoded by `QUOTED_PRINTABLE_ENCODE`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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, refer to the Postgres Core Documentation available at:

https://www.postgresql.org/docs/11/static/datatype-binary.html

The following example first encodes and then decodes a string:

```
edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_ENCODE('E=mc2') FROM DUAL;
quoted_printable_encode
-------------------------
 E=3Dmc2
(1 row)

edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_DECODE('E=3Dmc2') FROM DUAL;
 quoted_printable_decode
-------------------------
 E=mc2
(1 row)
```

11.2.19.6 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 string (a `RAW` value) that will be encoded in a quoted-printable format.

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, refer to the Postgres Core Documentation available at:

https://www.postgresql.org/docs/11/static/datatype-binary.html

The following example first encodes and then decodes a string:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_ENCODE('E=mc2') FROM DUAL;
quoted_printable_encode
-------------------------
 E=3Dmc2
(1 row)

edb=# SELECT UTL_ENCODE.QUOTED_PRINTABLE_DECODE('E=3Dmc2') FROM DUAL;
 quoted_printable_decode
-------------------------
 E=mc2
(1 row)
```

11.2.19.7 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`

`buf` contains the encoded string that will be translated to the original value encoded by `TEXT_ENCODE`.

`encode_charset`

`encode_charset` specifies the character set to which the string will be translated before encoding. The default value is `NULL`.

`encoding`

`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

The following 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;
       text_encode
--------------------------
V2hhdCBpcyB0aGUgZGF0ZT8=
(1 row)

edb=# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
    text_decode
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
-------------------
What is the date?
(1 row)
```

11.2.19.8 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_DECODE(<buf> IN VARCHAR2, <encode_charset> IN VARCHAR2 DEFAULT
NULL, <encoding> IN PLS_INTEGER DEFAULT NULL)
```

This function returns a `VARCHAR2` value.

Parameters

`buf`

`buf` contains the encoded string that will be translated to the specified character set and encoded by `TEXT_ENCODE`.

`encode_charset`

`encode_charset` specifies the character set to which the value will be translated before encoding. The default value is `NULL`.

`encoding`

`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

The following 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;
       text_encode
--------------------------
V2hhdCBpcyB0aGUgZGF0ZT8=
(1 row)

edb=# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
    text_decode
-------------------
What is the date?
(1 row)
```

11.2.19.9 UUDECODE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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:

```
UUDECODE(<r> IN RAW)
```

This function returns a `RAW` value.

If you are using the 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 Advanced Server configuration parameter `utl_encode.uudecode_redwood` to `TRUE` before invoking the Advanced Server `UUDECODE` function on the Oracle-created data. (For example, this situation may occur if you migrated Oracle tables containing uuencoded data to an 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 Advanced Server `UUENCODE` function. As a result, attempting to use the 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 are using the Advanced Server `UUDECODE` function on uuencoded data created by the 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 that will be translated to `RAW`.

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, refer to the Postgres Core Documentation available at:

https://www.postgresql.org/docs/11/static/datatype-binary.html

The following example uses `UUENCODE` and `UUDECODE` to first encode and then decode a string:

```
 edb=# SET bytea_output = escape;
SET
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
edb-# ('begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012')
edb-# FROM DUAL;
     uudecode
-------------------
What is the date?
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

11.2.19.10 UUENCODE

Use the `UUENCODE` function to translate `RAW` data into a uuencode formatted encoded string. The signature is:

```
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

`r`

`r` contains the `RAW` string that will be translated to uuencode format.

`type`

`type` is an `INTEGER` value or constant that specifies the type of uuencoded string that will be returned; the default value is `1`. The possible values are:

Value	Constant
`1`	`complete`
`2`	`header_piece`
`3`	`middle_piece`
`4`	`end_piece`

`filename`

`filename` is a `VARCHAR2` value that specifies the file name that you want to embed in the encoded form; if you do not specify a file name, `UUENCODE` will include a filename 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 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, refer to the Postgres Core Documentation available at:

https://www.postgresql.org/docs/11/static/datatype-binary.html

The following example uses `UUENCODE` and `UUDECODE` to first encode and then decode a string:

```
edb=# SET bytea_output = escape;
SET
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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
edb-# ('begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012')
edb-# FROM DUAL;
     uudecode
------------------
 What is the date?
(1 row)
```

11.2.20 UTL_FILE

The `UTL_FILE` package provides the capability to read from, and write to files on the operating system's file system. Non-superusers must be granted `EXECUTE` privilege on the `UTL_FILE` package by a superuser before using any of the functions or procedures in the package. For example the following command grants the privilege to user `mary`:

```
GRANT EXECUTE ON PACKAGE SYS.UTL_FILE TO mary;
```

Also, the operating system username, `enterprisedb`, must have the appropriate read and/or write permissions on the directories and files to be accessed using the `UTL_FILE` functions and procedures. If the required file permissions are not in place, an exception is thrown in the `UTL_FILE` function or procedure.

A handle to the file to be written to, or read from is used to reference the file. The *file handle* is defined by a public variable in the `UTL_FILE` package named, `UTL_FILE.FILE_TYPE`. 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 is 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:

Function/Procedure	Return Type	Description
`FCLOSE(file IN OUT)`	n/a	Closes the specified file identified by `file`.
`FCLOSE_ALL`	n/a	Closes all open files.
`FCOPY(location, filename, dest_dir, dest_file [, start_line [, end_line ] ])`	n/a	Copies `filename` in the directory identified by `location` to file, `dest_file`, in directory, `dest_dir`, starting from line, `start_line`, to line, `end_line`.
`FFLUSH(file)`	n/a	Forces data in the buffer to be written to disk in the file identified by `file`.
`FOPEN(location, filename, open_mode [, max_linesize ])`	`FILE_TYPE`	Opens file, `filename`, in the directory identified by `location`.
`FREMOVE(location, filename)`	n/a	Removes the specified file from the file system.
`FRENAME(location, filename, dest_dir, dest_file [, overwrite ])`	n/a	Renames the specified file.
`GET_LINE(file, buffer OUT)`	n/a	Reads a line of text into variable, `buffer`, from the file identified by `file`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function/Procedure	Return Type	Description
`IS_OPEN(file)`	`BOOLEAN`	Determines whether or not the given file is open.
`NEW_LINE(file [, lines ])`	n/a	Writes an end-of-line character sequence into the file.
`PUT(file, buffer)`	n/a	Writes `buffer` to the given file. `PUT` does not write an end-of-line character sequence.
`PUT_LINE(file, buffer)`	n/a	Writes `buffer` to the given file. An end-of-line character sequence is added by the `PUT_LINE` procedure.
`PUTF(file, format [, arg1 ] [, ...])`	n/a	Writes a formatted string to the given file. Up to five substitution parameters, `arg1,...arg5` may be specified for replacement in `format`.

Advanced Server's implementation of `UTL_FILE` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above 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:

Exception Code	Condition name
`-29283`	`invalid_operation`
`-29285`	`write_error`
`-29284`	`read_error`
`-29282`	`invalid_filehandle`
`-29287`	`invalid_maxlinesize`
`-29281`	`invalid_mode`
`-29280`	`invalid_path`

Setting File Permissions with utl_file.umask

When a `UTL_FILE` function or procedure creates a file, there are default file permissions as shown by the following.

```
-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile
```

Note that 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 wish to have a different set of file permissions on files created by the `UTL_FILE` functions and procedures, you can accomplish this by setting 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 is for usage only within the Advanced Server `UTL_FILE` package.

> **Note**
>
> The `utl_file.umask` parameter is not supported on Windows systems.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The value specified for `utl_file.umask` is a 3 or 4-character octal string that would be 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 usage of the `umask` command.)

The following is an example of setting the file permissions with `utl_file.umask`.

First, set up the directory in the file system to be used by the `UTL_FILE` package. Be sure the operating system account, `enterprisedb` or `postgres`, whichever is applicable, 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 created in the preceding step.

```
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 group users and other users have read and write permissions on the file as well as the file owner.

```
$ pwd
/tmp/utldir
$ ls -l
total 4
-rw-rw-rw- 1 enterprisedb enterprisedb 21 Jul 24 16:04 utlfile
```

This parameter can also be set on a per role basis with the `ALTER ROLE` command, on a per database basis with the `ALTER DATABASE` command, or for the entire database server instance by setting it in the `postgresql.conf` file.

FCLOSE

The `FCLOSE` procedure closes an open file.

```
FCLOSE(<file> IN OUT FILE_TYPE)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameters

`file`

Variable of type `FILE_TYPE` containing a file handle of the file to be closed.

FCLOSE_ALL

The `FLCLOSE_ALL` procedures closes all open files. The procedure executes successfully even if there are no open files to close.

```
FCLOSE_ALL
```

FCOPY

The `FCOPY` procedure copies text from one file to another.

```
FCOPY(<location> VARCHAR2, <filename> VARCHAR2,
  <dest_dir> VARCHAR2, <dest_file> VARCHAR2
  [, <start_line> PLS_INTEGER [, <end_line> PLS_INTEGER ] ])
```

Parameters

`location`

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory containing the file to be copied.

`filename`

Name of the source file to be copied.

`dest_dir`

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory to which the file is to be copied.

`dest_file`

Name of the destination file.

`start_line`

Line number in the source file from which copying will begin. The default is 1.

`end_line`

Line number of the last line in the source file to be copied. If omitted or null, copying will go to the last line of the file.

Examples

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following makes a copy of a file, `C:\TEMP\EMPDIR\empfile.csv`, containing a comma-delimited list of employees from the `emp` table. The copy, `empcopy.csv`, is then listed.

```
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';
    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, ''' ||
        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;
    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;

The following is the destination file, 'empcopy.csv'
7369,SMITH,CLERK,7902,17-DEC-80,800,,20
7499,ALLEN,SALESMAN,7698,20-FEB-81,1600,300,30
7521,WARD,SALESMAN,7698,22-FEB-81,1250,500,30
7566,JONES,MANAGER,7839,02-APR-81,2975,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81,1250,1400,30
7698,BLAKE,MANAGER,7839,01-MAY-81,2850,,30
7782,CLARK,MANAGER,7839,09-JUN-81,2450,,10
7788,SCOTT,ANALYST,7566,19-APR-87,3000,,20
7839,KING,PRESIDENT,,17-NOV-81,5000,,10
7844,TURNER,SALESMAN,7698,08-SEP-81,1500,0,30
7876,ADAMS,CLERK,7788,23-MAY-87,1100,,20
7900,JAMES,CLERK,7698,03-DEC-81,950,,30
7902,FORD,ANALYST,7566,03-DEC-81,3000,,20
7934,MILLER,CLERK,7782,23-JAN-82,1300,,10
14 records retrieved
```

FFLUSH

The `FFLUSH` procedure flushes unwritten data from the write buffer to the file.

```
FFLUSH(<file> FILE_TYPE)
```

Parameters

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`file`

Variable of type `FILE_TYPE` containing a file handle.

Examples

Each line is flushed after the `NEW_LINE` procedure is called.

```
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,',');
        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.NEW_LINE(v_empfile);
        UTL_FILE.FFLUSH(v_empfile);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
END;
```

FOPEN

The `FOPEN` function opens a file for I/O.

```
<filetype> FILE_TYPE FOPEN(<location> VARCHAR2,
  <filename> VARCHAR2,<open_mode> VARCHAR2
  [, <max_linesize> BINARY_INTEGER ])
```

Parameters

`location`

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory containing the file to be opened.

`filename`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name of the file to be opened.

`open_mode`

Mode in which the file will be opened. Modes are: `a` - append to file; `r` - read from file; `w` - write to file.

`max_linesize`

Maximum size of a line in characters. In read mode, an exception is thrown if an attempt is made to read a line exceeding `max_linesize`. In write and append modes, an exception is thrown if an attempt is made to write a line exceeding `max_linesize`. The end-of-line character(s) are not included in determining if the maximum line size is exceeded. This behavior is not compatible with Oracle databases; Oracle does count the end-of-line character(s).

`filetype`

Variable of type `FILE_TYPE` containing the file handle of the opened file.

FREMOVE

The `FREMOVE` procedure removes a file from the system.

```
FREMOVE(<location> VARCHAR2, <filename> VARCHAR2)
```

An exception is thrown if the file to be removed does not exist.

Parameters

`location`

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory containing the file to be removed.

`filename`

Name of the file to be removed.

Examples

The following removes file `empfile.csv`.

```
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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

FRENAME

The `FRENAME` procedure renames a given file. This effectively moves a file from one location to another.

```
FRENAME(<location> VARCHAR2, <filename> VARCHAR2,
  <dest_dir> VARCHAR2, <dest_file> VARCHAR2,
  [ <overwrite> BOOLEAN ])
```

Parameters

`location`

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory containing the file to be renamed.

`filename`

Name of the source file to be renamed.

`dest_dir`

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory to which the renamed file is to exist.

`dest_file`

New name of the original file.

`overwrite`

Replaces any existing file named `dest_file` in `dest_dir` if set to `TRUE`, otherwise an exception is thrown if set to `FALSE`. This is the default.

Examples

The following 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.

```
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);
    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, ''' ||
        v_dest_file || '''');
    LOOP
        UTL_FILE.GET_LINE(v_empfile,v_emprec);
        DBMS_OUTPUT.PUT_LINE(v_emprec);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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 THEN
            DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
            DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

The following is the renamed file, 'newemp.csv'
7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-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,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
14 records retrieved
```

GET_LINE

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.

```
GET_LINE(<file> FILE_TYPE, <buffer> OUT VARCHAR2)
```

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`.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
    v_emprec        VARCHAR2(120);
    v_count         INTEGER := 0;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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
            DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
            DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-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,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
End of file empfile.csv - 14 records retrieved
```

IS_OPEN

The `IS_OPEN` function determines whether or not the given file is open.

```
<status> BOOLEAN IS_OPEN(<file> FILE_TYPE)
```

Parameters

`file`

Variable of type `FILE_TYPE` containing the file handle of the file to be tested.

`status`

`TRUE` if the given file is open, `FALSE` otherwise.

NEW_LINE

The `NEW_LINE` procedure writes an end-of-line character sequence in the file.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
NEW_LINE(<file> FILE_TYPE [, <lines> INTEGER ])
```

Parameters

`file`

Variable of type `FILE_TYPE` containing the file handle of the file to which end-of-line character sequences are to be written.

`lines`

Number of end-of-line character sequences to be written. The default is one.

Examples

A file containing a double-spaced list of employee records is written.

```
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,',');
        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.NEW_LINE(v_empfile,2);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
END;

Created file: empfile.csv
```

This file is then displayed:

```
C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-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,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

PUT

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.

```
PUT(<file> FILE_TYPE, <buffer> { DATE | NUMBER | TIMESTAMP | VARCHAR2 })
```

Parameters

`file`

Variable of type `FILE_TYPE` containing the file handle of the file to which the given string is to be written.

`buffer`

Text to be written to the specified file.

Examples

The following example uses the `PUT` procedure to create a comma-delimited file of employees from the `emp` table.

```
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,',');
        UTL_FILE.PUT(v_empfile,i.ename);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.job);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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.NEW_LINE(v_empfile);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
END;

Created file: empfile.csv
```

The following is the contents of `empfile.csv` created above:

```
 C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-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,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

PUT_LINE

The `PUT_LINE` procedure writes a single line to the given file including an end-of-line character sequence.

```
PUT_LINE(<file> FILE_TYPE, <buffer> { DATE | NUMBER | TIMESTAMP | VARCHAR2 })
```

Parameters

`file`

Variable of type `FILE_TYPE` containing the file handle of the file to which the given line is to be written.

`buffer`

Text to be written to the specified file.

Examples

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example uses the `PUT_LINE` procedure to create a comma-delimited file of employees from the `emp` table.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
    v_emprec        VARCHAR2(120);
    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
        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;
        UTL_FILE.PUT_LINE(v_empfile,v_emprec);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    UTL_FILE.FCLOSE(v_empfile);
END;
```

The following is the contents of `empfile.csv` created above:

```
C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-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,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

PUTF

The `PUTF` procedure writes a formatted string to the given file.

```
PUTF(<file> FILE_TYPE, <format> VARCHAR2 [, <arg1> VARCHAR2] [, ...])
```

Parameters

`file`

Variable of type `FILE_TYPE` containing the file handle of the file to which the formatted line is to be written.

`format`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

String to format the text written to the file. The special character sequence, `%s`, is substituted by the value of arg. The special character sequence, `\n`, indicates a new line. Note, however, in Advanced Server, a new line character must be specified with two consecutive backslashes instead of one - `\\n`. This characteristic is not compatible with Oracle databases.

`arg1`

Up to five arguments, `arg1`,... `arg5`, to be substituted in the format string for each occurrence of `%s`. The first arg is substituted for the first occurrence of `%s`, the second arg is substituted for the second occurrence of `%s`, etc.

Examples

The following anonymous block produces formatted output containing data from the `emp` table. Note the use of the E literal syntax and double backslashes for the new line character sequence in the format string which are not compatible with Oracle databases.

```
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\\nSalary: $%s Commission: $%s\\n\\n';
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
    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;

Created file: empfile.csv
```

The following is the contents of `empfile.csv` created above:

```
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
7788 SCOTT, ANALYST
Salary: $3000.00 Commission: $0
7839 KING, PRESIDENT
Salary: $5000.00 Commission: $0
7844 TURNER, SALESMAN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
Salary: $1500.00 Commission: $0.00
7876 ADAMS, CLERK
Salary: $1100.00 Commission: $0
7900 JAMES, CLERK
Salary: $950.00 Commission: $0
7902 FORD, ANALYST
Salary: $3000.00 Commission: $0
7934 MILLER, CLERK
Salary: $1300.00 Commission: $0
```

11.2.21 UTL_HTTP

The `UTL_HTTP` package provides a way to use the HTTP or HTTPS protocol to retrieve information found at an URL. Advanced Server supports the following functions and procedures:

Function/Procedure	Return Type	Description
`BEGIN_REQUEST(url, method, http_version)`	`UTL_HTTP.REQ`	Initiates a new HTTP request.
`END_REQUEST(r IN OUT)`	n/a	Ends an HTTP request before allowing it to complete.
`END_RESPONSE(r IN OUT)`	n/a	Ends the HTTP response.
`GET_BODY_CHARSET`	`VARCHAR2`	Returns the default character set of the body of future HTTP requests.
`GET_BODY_CHARSET(charset OUT)`	n/a	Returns the default character set of the body of future HTTP requests.
`GET_FOLLOW_REDIRECT(max_redirects OUT)`	n/a	Current setting for the maximum number of redirections allowed.
`GET_HEADER(r IN OUT, n, name OUT, value OUT)`	n/a	Returns the nth header of the HTTP response.
`GET_HEADER_BY_NAME(r IN OUT, name, value OUT, n)`	n/a	Returns the HTTP response header for the specified name.
`GET_HEADER_COUNT(r IN OUT)`	`INTEGER`	Returns the number of HTTP response headers.
`GET_RESPONSE(r IN OUT)`	`UTL_HTTP.RESP`	Returns the HTTP response.
`GET_RESPONSE_ERROR_CHECK(enable OUT)`	n/a	Returns whether or not response error check is set.
`GET_TRANSFER_TIMEOUT(timeout OUT)`	n/a	Returns the transfer timeout setting for HTTP requests.
`READ_LINE(r IN OUT, data OUT, remove_crlf)`	n/a	Returns the HTTP response body in text form until the end of line.
`READ_RAW(r IN OUT, data OUT, len)`	n/a	Returns the HTTP response body in binary form for a specified number of bytes.
`READ_TEXT(r IN OUT, data OUT, len)`	n/a	Returns the HTTP response body in text form for a specified number of characters.
`REQUEST(url)`	`VARCHAR2`	Returns the content of a web page.
`REQUEST_PIECES(url, max_pieces)`	`UTL_HTTP.HTML_PIECES`	Returns a table of 2000-byte segments retrieved from an URL.
`SET_BODY_CHARSET(charset)`	n/a	Sets the default character set of the body of future HTTP requests.
`SET_FOLLOW_REDIRECT(max_redirects)`	n/a	Sets the maximum number of times to follow the redirect instruction.
`SET_FOLLOW_REDIRECT(r IN OUT, max_redirects)`	n/a	Sets the maximum number of times to follow the redirect instruction for an individual request.
`SET_HEADER(r IN OUT, name, value)`	n/a	Sets the HTTP request header.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Function/Procedure	Return Type	Description
`SET_RESPONSE_ERROR_CHECK(enable)`	n/a	Determines whether or not HTTP 4xx and 5xx status codes are to be treated as errors.
`SET_TRANSFER_TIMEOUT(timeout)`	n/a	Sets the default, transfer timeout value for HTTP requests.
`SET_TRANSFER_TIMEOUT(r IN OUT, timeout)`	n/a	Sets the transfer timeout value for an individual HTTP request.
`WRITE_LINE(r IN OUT, data)`	n/a	Writes CRLF terminated data to the HTTP request body in TEXT form.
`WRITE_RAW(r IN OUT, data)`	n/a	Writes data to the HTTP request body in BINARY form.
`WRITE_TEXT(r IN OUT, data)`	n/a	Writes data to the HTTP request body in TEXT form.

Advanced Server's implementation of `UTL_HTTP` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

> **Note**
>
> In Advanced Server, an `HTTP 4xx` or `HTTP 5xx` response produces a database error; in Oracle, this is configurable but `FALSE` by default.

In Advanced Server, the `UTL_HTTP` text interfaces expect the downloaded data to be in the database encoding. All currently-available interfaces are text interfaces. In Oracle, the encoding is detected from HTTP headers; in the absence of the header, the default is configurable and defaults to `ISO-8859-1`.

Advanced Server ignores all cookies it receives.

The `UTL_HTTP` exceptions that can be raised in Oracle are not recognized by Advanced Server. In addition, the error codes returned by Advanced Server are not the same as those returned by Oracle.

There are various public constants available with `UTL_HTTP`. These are listed in the following tables.

The following table contains `UTL_HTTP` public constants defining HTTP versions and port assignments.

HTTP VERSIONS	
`HTTP_VERSION_1_0`	`CONSTANT VARCHAR2(64) := 'HTTP/1.0';`
`HTTP_VERSION_1_1`	`CONSTANT VARCHAR2(64) := 'HTTP/1.1';`
STANDARD PORT ASSIGNMENTS	
`DEFAULT_HTTP_PORT`	`CONSTANT INTEGER := 80;`
`DEFAULT_HTTPS_PORT`	`CONSTANT INTEGER := 443;`

The following table contains `UTL_HTTP` public status code constants.

1XX INFORMATIONAL	
`HTTP_CONTINUE`	`CONSTANT INTEGER := 100;`
`HTTP_SWITCHING_PROTOCOLS`	`CONSTANT INTEGER := 101;`
`HTTP_PROCESSING`	`CONSTANT INTEGER := 102;`
2XX SUCCESS	
`HTTP_OK`	`CONSTANT INTEGER := 200;`
`HTTP_CREATED`	`CONSTANT INTEGER := 201;`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

1XX INFORMATIONAL

```
HTTP_ACCEPTED                   CONSTANT INTEGER := 202;
HTTP_NON_AUTHORITATIVE_INFO     CONSTANT INTEGER := 203;
HTTP_NO_CONTENT                 CONSTANT INTEGER := 204;
HTTP_RESET_CONTENT              CONSTANT INTEGER := 205;
HTTP_PARTIAL_CONTENT            CONSTANT INTEGER := 206;
HTTP_MULTI_STATUS               CONSTANT INTEGER := 207;
HTTP_ALREADY_REPORTED           CONSTANT INTEGER := 208;
HTTP_IM_USED                    CONSTANT INTEGER := 226;
```

3XX REDIRECTION

```
HTTP_MULTIPLE_CHOICES           CONSTANT INTEGER := 300;
HTTP_MOVED_PERMANENTLY          CONSTANT INTEGER := 301;
HTTP_FOUND                      CONSTANT INTEGER := 302;
HTTP_SEE_OTHER                  CONSTANT INTEGER := 303;
HTTP_NOT_MODIFIED               CONSTANT INTEGER := 304;
HTTP_USE_PROXY                  CONSTANT INTEGER := 305;
HTTP_SWITCH_PROXY               CONSTANT INTEGER := 306;
HTTP_TEMPORARY_REDIRECT         CONSTANT INTEGER := 307;
HTTP_PERMANENT_REDIRECT         CONSTANT INTEGER := 308;
```

4XX CLIENT ERROR

```
HTTP_BAD_REQUEST                CONSTANT INTEGER := 400;
HTTP_UNAUTHORIZED               CONSTANT INTEGER := 401;
HTTP_PAYMENT_REQUIRED           CONSTANT INTEGER := 402;
HTTP_FORBIDDEN                  CONSTANT INTEGER := 403;
HTTP_NOT_FOUND                  CONSTANT INTEGER := 404;
HTTP_METHOD_NOT_ALLOWED         CONSTANT INTEGER := 405;
HTTP_NOT_ACCEPTABLE             CONSTANT INTEGER := 406;
HTTP_PROXY_AUTH_REQUIRED        CONSTANT INTEGER := 407;
HTTP_REQUEST_TIME_OUT           CONSTANT INTEGER := 408;
HTTP_CONFLICT                   CONSTANT INTEGER := 409;
HTTP_GONE                       CONSTANT INTEGER := 410;
HTTP_LENGTH_REQUIRED            CONSTANT INTEGER := 411;
HTTP_PRECONDITION_FAILED        CONSTANT INTEGER := 412;
HTTP_REQUEST_ENTITY_TOO_LARGE   CONSTANT INTEGER := 413;
HTTP_REQUEST_URI_TOO_LARGE      CONSTANT INTEGER := 414;
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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

4XX CLIENT ERROR

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_PRECONDITION_REQUIRED	CONSTANT INTEGER := 428;
HTTP_TOO_MANY_REQUESTS	CONSTANT INTEGER := 429;
HTTP_REQUEST_HEADER_FIELDS_TOO_LARGE	CONSTANT INTEGER := 431;
HTTP_NO_RESPONSE	CONSTANT INTEGER := 444;
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 := 494;
HTTP_CERT_ERROR	CONSTANT INTEGER := 495;
HTTP_NO_CERT	CONSTANT INTEGER := 496;
HTTP_HTTP_TO_HTTPS	CONSTANT INTEGER := 497;
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.

```
TYPE html_pieces IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

REQ

The `REQ` record type holds information about each HTTP request.

```
TYPE req IS RECORD (
    url             VARCHAR2(32767),    -- URL to be accessed
    method          VARCHAR2(64),       -- HTTP method
    http_version    VARCHAR2(64),       -- HTTP version
    private_hndl    INTEGER             -- Holds handle for this request
);
```

RESP

The `RESP` record type holds information about the response from each HTTP request.

```
TYPE resp IS RECORD (
    status_code     INTEGER,            -- HTTP status code
    reason_phrase   VARCHAR2(256),      -- HTTP response reason phrase
    http_version    VARCHAR2(64),       -- HTTP version
    private_hndl    INTEGER             -- Holds handle for this response
);
```

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
```

The `BEGIN_REQUEST` function returns a record of type `UTL_HTTP.REQ`.

Parameters

`url`

`url` is the Uniform Resource Locator from which `UTL_HTTP` will return content.

`method`

`method` is the HTTP method to be used. The default is `GET`.

`http_version`

`http_version` is the HTTP protocol version sending the request. The specified values should be either `HTTP/1.0` or `HTTP/1.1`. The default is null in which case the latest HTTP protocol version supported by the `UTL_HTTP` package is used which is 1.1.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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, then close the response. The signature is:

```
END_REQUEST(<r> IN OUT UTL_HTTP.REQ)
```

Parameters

`r`

`r` is the HTTP request record.

END_RESPONSE

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:

```
END_RESPONSE(<r> IN OUT UTL_HTTP.RESP)
```

Parameters

`r`

`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:

```
GET_BODY_CHARSET(<charset> OUT VARCHAR2)
```

The function signature is:

```
GET_BODY_CHARSET() RETURN VARCHAR2
```

This function returns a `VARCHAR2` value.

Parameters

`charset`

`charset` is the character set of the body.

Examples

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following is an example of the `GET_BODY_CHARSET` function.

```
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:

```
GET_FOLLOW_REDIRECT(<max_redirects> OUT INTEGER)
```

Parameters

`max_redirects`

`max_redirects` is maximum number of redirections allowed.

GET_HEADER

The `GET_HEADER` procedure returns the `nth` header of the HTTP response. The signature is:

```
GET_HEADER(<r> IN OUT UTL_HTTP.RESP, <n> INTEGER, <name> OUT
VARCHAR2, <value> OUT VARCHAR2)
```

Parameters

`r`

`r` is the HTTP response record.

`n`

`n` is the `nth` header of the HTTP response record to retrieve.

`name`

`name` is the name of the response header.

`value`

`value` is the value of the response header.

Examples

The following example retrieves the header count, then the headers.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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;
```

The following is the output from the example.

```
 Header Count: 23
Age: 570
Cache-Control: must-revalidate
Content-Type: text/html; charset=utf-8
Date: Wed, 30 Apr 2015 14:57:52 GMT
ETag: "aab02f2bd2d696eed817ca89ef411dda"
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Last-Modified: Wed, 30 Apr 2015 14:15:49 GMT
RTSS: 1-1307-3
Server: Apache/2.2.3 (Red Hat)
Set-Cookie:
SESS2771d0952de2a1a84d322a262e0c173c=jn1u1j1etmdi5gg4lh8hakvs01;
expires=Fri, 23-May-2015 18:21:43 GMT; path=/; domain=.enterprisedb.com
Vary: Accept-Encoding
Via: 1.1 varnish
X-EDB-Backend: ec
X-EDB-Cache: HIT
X-EDB-Cache-Address: 10.31.162.212
X-EDB-Cache-Server: ip-10-31-162-212
X-EDB-Cache-TTL: 600.000
X-EDB-Cacheable: MAYBE: The user has a cookie of some sort. Maybe it's
double choc-chip!
X-EDB-Do-GZIP: false
X-Powered-By: PHP/5.2.17
X-Varnish: 484508634 484506789
transfer-encoding: chunked
Connection: keep-alive
```

GET_HEADER_BY_NAME

Copyright © 2009 - 2024 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`

`r` is the HTTP response record.

`name`

`name` is the name of the response header to retrieve.

`value`

`value` is the value of the response header.

`n`

`n` is the `nth` 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.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_name          VARCHAR2(30) := 'Content-Type';
    v_value         VARCHAR2(200);
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;

Content-Type: text/html; charset=utf-8
```

GET_HEADER_COUNT

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`r`

`r` is the HTTP response record.

GET_RESPONSE

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.

GET_RESPONSE_ERROR_CHECK

The `GET_RESPONSE_ERROR_CHECK` procedure returns whether or not 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`.

GET_TRANSFER_TIMEOUT

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.

READ_LINE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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:

```
READ_LINE(<r> IN OUT UTL_HTTP.RESP, <data> OUT VARCHAR2,
<remove_crlf> BOOLEAN DEFAULT FALSE)
```

Parameters

`r`

`r` is the HTTP response record.

`data`

`data` is the response body in text form.

`remove_crlf`

Set `remove_crlf` to `TRUE` to remove new line characters, otherwise set to `FALSE`. The default is `FALSE`.

Examples

The following example retrieves and displays the body of the specified website.

```
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.

```
<!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">

  <!-- ____________________________ HEAD _____________________________ -->

  <head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />


    <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,
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
mysql migration, open source database, training, replication" />
<meta name="description" content="The leader in open source database
products, services, support, training and expertise based on PostgreSQL.
Free downloads, documentation, and tutorials." />
<meta name="abstract" content="The Enterprise PostgreSQL Company" />
<link rel="EditURI" type="application/rsd+xml" title="RSD" href="http://
www.enterprisedb.com/blogapi/rsd" />
<link rel="alternate" type="application/rss+xml" title="EnterpriseDB RSS"
href="http://www.enterprisedb.com/rss.xml" />
<link rel="shortcut icon" href="/sites/all/themes/edb_pixelcrayons/
favicon.ico" type="image/x-icon" />
    <link type="text/css" rel="stylesheet" media="all" href="/sites/default/
    files/css/css_db11adabae0aed6b79a2c3c52def4754.css" />
<!--[if IE 6]>
<link type="text/css" rel="stylesheet" media="all" href="/sites/all/themes/
oho_basic/css/ie6.css?g" />
<![endif]-->
<!--[if IE 7]>
<link type="text/css" rel="stylesheet" media="all" href="/sites/all/themes/
oho_basic/css/ie7.css?g" />
<![endif]-->
    <script type="text/javascript" src="/sites/default/files/js/
    js_74d97b1176812e2fd6e43d62503a5204.js"></script>
<script type="text/javascript">
<!--//--><![CDATA[//><!--
```

READ_RAW

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:

```
READ_RAW(<r> IN OUT UTL_HTTP.RESP, <data> OUT RAW, <len> INTEGER)
```

Parameters

`r`

`r` is the HTTP response record.

`data`

`data` is the response body in binary form.

`len`

Set `len` to the number of bytes of data to be returned.

Examples

The following example retrieves and displays the first 150 bytes in binary form.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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);
    DBMS_OUTPUT.PUT_LINE(v_data);
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The following is the output from the example.

```
 \x3c21444f43545950452068746d6c205055424c494320222d2f2f5733432f2f4454442058485
44d4c20312e30205374726963742f2f454e220d0a202022687474703a2f2f7777772e77332e6f
72672f54522f7868746d6c312f4454442f7868746d6c312d7374726963742e647464223e0d0a3
c68746d6c20786d6c6e733d22687474703a2f2f7777772e77332e6f72672f313939392f
```

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 be returned.

Examples

The following example retrieves the first 150 characters.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_data          VARCHAR2(150);
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    UTL_HTTP.READ_TEXT(v_resp, v_data, 150);
    DBMS_OUTPUT.PUT_LINE(v_data);
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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/
```

REQUEST

The `REQUEST` function returns the first 2000 bytes retrieved from a user-specified URL. The signature is:

```
REQUEST(<url> IN VARCHAR2) RETURN VARCHAR2
```

If the data found at the given URL is longer than 2000 bytes, the remainder will be discarded. If the data found at the given URL is shorter than 2000 bytes, the result will be shorter than 2000 bytes.

Parameters

`url`

`url` is the Uniform Resource Locator from which `UTL_HTTP` will return content.

Example

The following command returns the first 2000 bytes retrieved from the EnterpriseDB website:

```
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 an URL. The signature is:

```
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` will return content.

`max_pieces`

`max_pieces` specifies the maximum number of 2000-byte segments that the `REQUEST_PIECES` function will return. If `max_pieces` specifies more units than are available at the specified `url`, the final unit will contain fewer bytes.

Example

The following example returns the first four 2000 byte segments retrieved from the EnterpriseDB website:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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:

```
SET_BODY_CHARSET(<charset> VARCHAR2 DEFAULT NULL)
```

Parameters

`charset`

`charset` is the character set of the body of future requests. The default is null in which case the database character set is assumed.

SET_FOLLOW_REDIRECT

The `SET_FOLLOW_REDIRECT` procedure sets the maximum number of times the HTTP redirect instruction is to be followed in the response to this request or future requests. This procedures has two signatures:

```
SET_FOLLOW_REDIRECT(<max_redirects> IN INTEGER DEFAULT 3)
```

and

```
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`

`r` is the HTTP request record.

`max_redirects`

`max_redirects` is 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SET_HEADER(<r> IN OUT UTL_HTTP.REQ, <name> IN VARCHAR2, <value>
IN VARCHAR2 DEFAULT NULL)
```

Parameters

`r`

`r` is the HTTP request record.

`name`

`name` is the name of the request header.

`value`

`value` is the value of the request header. The default is null.

SET_RESPONSE_ERROR_CHECK

The `SET_RESPONSE_ERROR_CHECK` procedure determines whether or not HTTP 4xx and 5xx status codes returned by the `GET_RESPONSE` function should be interpreted as errors. The signature is:

```
SET_RESPONSE_ERROR_CHECK(<enable> IN BOOLEAN DEFAULT FALSE)
```

Parameters

`enable`

Set `enable` to `TRUE` if HTTP 4xx and 5xx status codes are to be treated as errors, otherwise set to `FALSE`. The default is `FALSE`.

SET_TRANSFER_TIMEOUT

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

`r`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`r` is the HTTP request record.

`timeout`

`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

The following example writes data (`Account balance $500.00`) in text form to the request body to be sent using the HTTP `POST` method. The data is sent to a hypothetical web application (`post.php`) that accepts and processes data.

```
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');
    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;
```

Assuming the web application successfully processed the `POST` method, the following output would be displayed:

```
 Status Code: 200
Reason Phrase: OK
```

WRITE_RAW

The `WRITE_RAW` procedure writes data to the HTTP request body in binary form. The signature is:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
WRITE_RAW(<r> IN OUT UTL_HTTP.REQ, <data> IN RAW)
```

Parameters

`r`

`r` is the HTTP request record.

`data`

`data` is the request body in binary form.

Example

The following example writes data in binary form to the request body to be sent using the HTTP `POST` method to a hypothetical web application that accepts and processes such data.

```
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
('54657374696e6720504f5354206d6574686f6420696e20485454502072657175657374'));
    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`.

Assuming the web application successfully processed the `POST` method, the following output would be 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`

`r` is the HTTP request record.

`data`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`data` is the request body in text form.

Example

The following example writes data (`Account balance $500.00`) in text form to the request body to be sent using the HTTP `POST` method. The data is sent to a hypothetical web application (`post.php`) that accepts and processes data.

```
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;
```

Assuming the web application successfully processed the `POST` method, the following output would be displayed:

```
Status Code: 200
Reason Phrase: OK
```

11.2.22 UTL_MAIL

The `UTL_MAIL` package provides the capability to manage e-mail. Advanced Server supports the following procedures:

Function/Procedure	Return Type	Description
`SEND(sender, recipients, cc, bcc, subject, message [, mime_type [, priority ]])`	n/a	Packages and sends an e-mail to an SMTP server.
`SEND_ATTACH_RAW(sender, recipients, cc, bcc, subject, message, mime_type, priority, attachment [, att_inline [, att_mime_type [, att_filename ]]])`	n/a	Same as the `SEND` procedure, but with `BYTEA` or large object attachments.
`SEND_ATTACH_VARCHAR2(sender, recipients, cc, bcc, subject, message, mime_type, priority, attachment [, att_inline [, att_mime_type [, att_filename ]]])`	n/a	Same as the `SEND` procedure, but with `VARCHAR2` attachments.

> **Note**
>
> An administrator must grant execute privileges to each user or group before they can use this package.

SEND

The `SEND` procedure provides the capability to send an e-mail to an SMTP server.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SEND(<sender> VARCHAR2, <recipients> VARCHAR2, <cc> VARCHAR2,
  <bcc> VARCHAR2, <subject> VARCHAR2, <message> VARCHAR2
  [, <mime_type> VARCHAR2 [, <priority> PLS_INTEGER ]])
```

Parameters

`sender`

E-mail address of the sender.

`recipients`

Comma-separated e-mail addresses of the recipients.

`cc`

Comma-separated e-mail addresses of copy recipients.

`bcc`

Comma-separated e-mail addresses of blind copy recipients.

`subject`

Subject line of the e-mail.

`message`

Body of the e-mail.

`mime_type`

Mime type of the message. The default is `text/plain; charset=us-ascii`.

`priority`

Priority of the e-mail The default is 3.

Examples

The following anonymous block sends a simple e-mail message.

```
DECLARE
    v_sender        VARCHAR2(30);
    v_recipients    VARCHAR2(60);
    v_subj          VARCHAR2(20);
    v_msg           VARCHAR2(200);
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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

SEND_ATTACH_RAW

The `SEND_ATTACH_RAW` procedure provides the capability to send an e-mail to an SMTP server with an attachment containing either `BYTEA` data or a large object (identified by the large object's `OID`). The call to `SEND_ATTACH_RAW` can be written in two ways:

```
SEND_ATTACH_RAW(<sender> VARCHAR2, <recipients> VARCHAR2,
  <cc> VARCHAR2, <bcc> VARCHAR2, <subject> VARCHAR2, <message> VARCHAR2,
  <mime_type> VARCHAR2, <priority> PLS_INTEGER,
  <attachment> BYTEA[, <att_inline> BOOLEAN
  [, <att_mime_type> VARCHAR2[, <att_filename> VARCHAR2 ]]])
```

or

```
SEND_ATTACH_RAW(<sender> VARCHAR2, <recipients> VARCHAR2,
  <cc> VARCHAR2, <bcc> VARCHAR2, <subject> VARCHAR2, <message> VARCHAR2,
  <mime_type> VARCHAR2, <priority> PLS_INTEGER, <attachment> OID
  [, <att_inline> BOOLEAN [, <att_mime_type> VARCHAR2
  [, <att_filename> VARCHAR2 ]]])
```

Parameters

`sender`

E-mail address of the sender.

`recipients`

Comma-separated e-mail addresses of the recipients.

`cc`

Comma-separated e-mail addresses of copy recipients.

`bcc`

Comma-separated e-mail addresses of blind copy recipients.

`subject`

Subject line of the e-mail.

`message`

Body of the e-mail.

`mime_type`

Mime type of the message. The default is `text/plain; charset=us-ascii`.

`priority`

Priority of the e-mail. The default is `3`.

`attachment`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The attachment.

`att_inline`

If set to `TRUE`, then the attachment is viewable inline, `FALSE` otherwise. The default is `TRUE`.

`att_mime_type`

Mime type of the attachment. The default is `application/octet`.

`att_filename`

The file name containing the attachment. The default is `NULL`.

SEND_ATTACH_VARCHAR2

The `SEND_ATTACH_VARCHAR2` procedure provides the capability to send an e-mail 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`

E-mail address of the sender.

`recipients`

Comma-separated e-mail addresses of the recipients.

`cc`

Comma-separated e-mail addresses of copy recipients.

`bcc`

Comma-separated e-mail addresses of blind copy recipients.

`subject`

Subject line of the e-mail.

`message`

Body of the e-mail.

`mime_type`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Mime type of the message. The default is `text/plain; charset=us-ascii`.

`priority`

Priority of the e-mail The default is `3.`

`attachment`

The `VARCHAR2` attachment.

`att_inline`

If set to `TRUE`, then the attachment is viewable inline, `FALSE` otherwise. The default is `TRUE`.

`att_mime_type`

Mime type of the attachment. The default is `text/plain; charset=us-ascii`.

`att_filename`

The file name containing the attachment. The default is `NULL`.

11.2.23 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.

Function/Procedure	Function or Procedure	Return Type	Description
`CAST_TO_RAW(c IN VARCHAR2)`	Function	`RAW`	Converts a `VARCHAR2` string to a `RAW` value.
`CAST_TO_VARCHAR2(r IN RAW)`	Function	`VARCHAR2`	Converts a `RAW` value to a `VARCHAR2` string.
`CONCAT(r1 IN RAW, r2 IN RAW, r3 IN RAW,…)`	Function	`RAW`	Concatenate multiple `RAW` values into a single `RAW` value.
`CONVERT(r IN RAW, to_charset IN VARCHAR2, from_charset IN VARCHAR2)`	Function	`RAW`	Converts encoded data from one encoding to another, and returns the result as a `RAW` value.
`LENGTH(r IN RAW)`	Function	`NUMBER`	Returns the length of a `RAW` value.
`SUBSTR(r IN RAW, pos IN INTEGER, len IN INTEGER)`	Function	`RAW`	Returns a portion of a `RAW` value.

Advanced Server's implementation of `UTL_RAW` is a partial implementation when compared to Oracle's version. Only those functions and procedures listed in the table above are supported.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

CAST_TO_RAW

The `CAST_TO_RAW` function converts a `VARCHAR2` string to a `RAW` value. The signature is:

```
CAST_TO_RAW(<c> VARCHAR2)
```

The function returns a `RAW` value if you pass a non-`NULL` value; if you pass a `NULL` value, the function will return `NULL`.

Parameters

`c`

The `VARCHAR2` value that will be converted to `RAW`.

Example

The following 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);
  r := UTL_RAW.CAST_TO_RAW(v);
  dbms_output.put_line(r);
END;
```

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(<r> RAW)
```

The function returns a `VARCHAR2` value if you pass a non-`NULL` value; if you pass a `NULL` value, the function will return `NULL`.

Parameters

`r`

The `RAW` value that will be converted to a `VARCHAR2` value.

Example

The following example uses the `CAST_TO_VARCHAR2` function to convert a `RAW` value to a `VARCHAR2` string:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DECLARE
  r RAW;
  v VARCHAR2;
BEGIN
  r := '\x4163636f756e7473'
  dbms_output.put_line(v);
  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
Accounts
```

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 Advanced Server implementation is a variadic function, and does not place a restriction on the number of values that can be concatenated.

Parameters

`r1, r2, r3,…`

The `RAW` values that `CONCAT` will concatenate.

Example

The following example uses the `CONCAT` function to concatenate multiple `RAW` values into a single `RAW` value:

```
SELECT UTL_RAW.CAST_TO_VARCHAR2(UTL_RAW.CONCAT('\x61', '\x62', '\x63')) FROM
DUAL;
 concat
--------
 abc
(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:

```
CONVERT(<r> RAW, <to_charset> VARCHAR2, <from_charset> VARCHAR2)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The function returns a `RAW` value.

Parameters

`r`

The `RAW` value that will be converted.

`to_charset`

The name of the encoding to which `r` will be converted.

`from_charset`

The name of the encoding from which `r` will be converted.

Example

The following example uses the `UTL_RAW.CAST_TO_RAW` function to convert a `VARCHAR2` string (`Accounts`) to a raw value, and then convert the value from `UTF8` to `LATIN7` , and then from `LATIN7` to `UTF8` :

```
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);
```

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:

```
LENGTH(<r> RAW)
```

The function returns a `RAW` value.

Parameters

`r`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `RAW` value that `LENGTH` will evaluate.

Example

The following 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;
 length
--------
 8
(1 row)
```

The following 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('□□□□'));
 length
--------
     12
(1 row)
```

SUBSTR

The `SUBSTR` function returns a substring of a `RAW` value. The signature is:

```
SUBSTR (<r> RAW, <pos> INTEGER, <len> INTEGER)
```

This function returns a `RAW` value.

Parameters

`r`

The `RAW` value from which the substring will be returned.

`pos`

The position within the `RAW` value of the first byte of the returned substring.

- If `pos` is `0` or `1`, the substring begins at the first byte of the `RAW` value.
- If `pos` is greater than one, the substring begins at the first byte specified by `pos`. For example, if `pos` is `3`, the substring begins at the third byte of the value.
- If `pos` is negative, the substring begins at `pos` bytes from the end of the source value. For example, if `pos` is `-3`, the substring begins at the third byte from the end of the value.

`len`

The maximum number of bytes that will be returned.

Example

The following example uses the `SUBSTR` function to select a substring that begins `3` bytes from the start of a `RAW` value:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 SELECT UTL_RAW.SUBSTR(UTL_RAW.CAST_TO_RAW('Accounts'), 3, 5) FROM DUAL;
  substr
--------
  count
(1 row)
```

The following example uses the `SUBSTR` function to select a substring that starts `5` bytes from the end of a `RAW` value:

```
 SELECT UTL_RAW.SUBSTR(UTL_RAW.CAST_TO_RAW('Accounts'), -5 , 3) FROM DUAL;
  substr
--------
  oun
(1 row)
```

11.2.24 UTL_SMTP

The `UTL_SMTP` package provides the capability to send e-mails 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.

Function/Procedure	Function or Procedure	Return Type	Description
`CLOSE_DATA(c IN OUT)`	Procedure	n/a	Ends an e-mail message.
`COMMAND(c IN OUT, cmd [, arg ])`	Both	`REPLY`	Execute an SMTP command.
`COMMAND_REPLIES(c IN OUT, cmd [, arg ])`	Function	`REPLIES`	Execute an SMTP command where multiple reply lines are expected.
`DATA(c IN OUT, body VARCHAR2)`	Procedure	n/a	Specify the body of an e-mail message.
`EHLO(c IN OUT, domain)`	Procedure	n/a	Perform initial handshaking with an SMTP server and return extended information.
`HELO(c IN OUT, domain)`	Procedure	n/a	Perform initial handshaking with an SMTP server
`HELP(c IN OUT [, command ])`	Function	`REPLIES`	Send the `HELP` command.
`MAIL(c IN OUT, sender [, parameters ])`	Procedure	n/a	Start a mail transaction.
`NOOP(c IN OUT)`	Both	`REPLY`	Send the null command.
`OPEN_CONNECTION(host [, port [, tx_timeout ]])`	Function	`CONNECTION`	Open a connection.
`OPEN_DATA(c IN OUT)`	Both	`REPLY`	Send the `DATA` command.
`QUIT(c IN OUT)`	Procedure	n/a	Terminate the SMTP session and disconnect.
`RCPT(c IN OUT, recipient [, parameters ])`	Procedure	n/a	Specify the recipient of an e-mail message.
`RSET(c IN OUT)`	Procedure	n/a	Terminate the current mail transaction.
`VRFY(c IN OUT, recipient)`	Function	`REPLY`	Validate an e-mail address.
`WRITE_DATA(c IN OUT, data)`	Procedure	n/a	Write a portion of the e-mail message.

Advanced Server's implementation of `UTL_SMTP` is a partial implementation when compared to Oracle's version. Only those functions and procedures

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

listed in the table above are supported.

The following table lists the public variables available in the `UTL_SMTP` package.

Public Variables	Data Type	Value	Description
`connection`	`RECORD`		Description of an SMTP connection.
`reply`	`RECORD`		SMTP reply line.

CONNECTION

The `CONNECTION` record type provides a description of an SMTP connection.

```
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.

```
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 e-mail message by sending the following sequence:

`<CR><LF>.<CR><LF>`

This is a single period at the beginning of a line.

```
CLOSE_DATA(<c> IN OUT CONNECTION)
```

Parameters

`c`

The SMTP connection to be closed.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

COMMAND

The `COMMAND` procedure provides the capability to execute an SMTP command. If you are expecting multiple reply lines, use `COMMAND_REPLIES`.

```
<reply> REPLY COMMAND(<c> IN OUT CONNECTION, <cmd> VARCHAR2
   [, <arg> VARCHAR2 ])

COMMAND(<c> IN OUT CONNECTION, <cmd> VARCHAR2 [, <arg> VARCHAR2 ])
```

Parameters

`c`

The SMTP connection to which the command is to be sent.

`cmd`

The SMTP command to be processed.

`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 only a single reply line is expected.

```
<replies> REPLIES COMMAND(<c> IN OUT CONNECTION, <cmd> VARCHAR2
   [, <arg> VARCHAR2 ])
```

Parameters

`c`

The SMTP connection to which the command is to be sent.

`cmd`

The SMTP command to be processed.

`arg`

An argument to the SMTP command. The default is null.

`replies`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

SMTP reply lines to the command. See Reply/Replies for a description of `REPLY` and `REPLIES`.

DATA

The `DATA` procedure provides the capability to specify the body of the e-mail message. The message is terminated with a `<CR><LF>.<CR><LF>` sequence.

```
DATA(<c> IN OUT CONNECTION, <body> VARCHAR2)
```

Parameters

`c`

The SMTP connection to which the command is to be sent.

`body`

Body of the e-mail message to be sent.

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

`c`

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`c`

The connection to the SMTP server over which to perform handshaking.

`domain`

Domain name of the sending host.

HELP

The `HELP` function provides the capability to send the `HELP` command to the SMTP server.

```
<replies> REPLIES HELP(<c> IN OUT CONNECTION [, <command> VARCHAR2 ])
```

Parameters

`c`

The SMTP connection to which the command is to be sent.

`command`

Command on which help is requested.

`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 e-mail address.

`parameters`

Mail command parameters in the format, `key=value` as defined in RFC 1869.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

NOOP

The `NOOP` function/procedure sends the null command to the SMTP server. The `NOOP` has no effect upon the server except to obtain a successful response.

```
<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.

```
<c> 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`

Time out value in seconds. Do not wait is indicated by specifying 0. Wait indefinitely is indicated by setting timeout to null. 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
OPEN_DATA(<c> IN OUT CONNECTION)
```

Parameters

`c`

SMTP connection on which to send the command.

QUIT

The `QUIT` procedure closes the session with an SMTP server.

```
QUIT(<c> IN OUT CONNECTION)
```

Parameters

`c`

SMTP connection to be terminated.

RCPT

The `RCPT` procedure provides the e-mail address of the recipient. To schedule multiple recipients, invoke `RCPT` multiple times.

```
RCPT(<c> IN OUT CONNECTION, <recipient> VARCHAR2 [, <parameters> VARCHAR2 ])
```

Parameters

`c`

Connection to SMTP server on which to add a recipient.

`recipient`

The recipient's e-mail address.

`parameters`

Mail command parameters in the format, key=value as defined in RFC 1869.

RSET

The `RSET` procedure provides the capability to terminate the current mail transaction.

```
RSET(<c> IN OUT CONNECTION)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameters

`c`

SMTP connection on which to cancel the mail transaction.

VRFY

The `VRFY` function provides the capability to validate and verify the recipient's e-mail address. If valid, the recipient's full name and fully qualified mailbox is returned.

```
<reply> REPLY VRFY(<c> IN OUT CONNECTION, <recipient> VARCHAR2)
```

Parameters

`c`

The SMTP connection on which to verify the e-mail address.

`recipient`

The recipient's e-mail address to be verified.

`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 provides the capability to add `VARCHAR2` data to an e-mail message. The `WRITE_DATA` procedure may be repetitively called to add data.

```
WRITE_DATA(<c> IN OUT CONNECTION, <data> VARCHAR2)
```

Parameters

`c`

The SMTP connection on which to add data.

`data`

Data to be added to the e-mail message. The data must conform to the RFC 822 specification.

Comprehensive Example

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following procedure constructs and sends a text e-mail message using the `UTL_SMTP` package.

```
 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,
        'Dy, DD Mon YYYY HH24:MI:SS') || v_crlf
        || 'From: ' || p_sender || v_crlf
        || 'To: ' || p_recipient || v_crlf
        || 'Subject: ' || p_subj || v_crlf
        || p_msg
        , 1, 32767));
    UTL_SMTP.QUIT(v_conn);
END;

EXEC send_mail('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday
Party','Are you planning to attend?','smtp.enterprisedb.com');
```

The following example uses the `OPEN_DATA, WRITE_DATA`, and `CLOSE_DATA` procedures instead of the `DATA` procedure.

```
 CREATE OR REPLACE PROCEDURE send_mail_2 (
    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.OPEN_DATA(v_conn);
    UTL_SMTP.WRITE_DATA(v_conn,'From: ' || p_sender || v_crlf);
    UTL_SMTP.WRITE_DATA(v_conn,'To: ' || p_recipient || v_crlf);
    UTL_SMTP.WRITE_DATA(v_conn,'Subject: ' || p_subj || v_crlf);
    UTL_SMTP.WRITE_DATA(v_conn,v_crlf || p_msg);
    UTL_SMTP.CLOSE_DATA(v_conn);
    UTL_SMTP.QUIT(v_conn);
END;

EXEC send_mail_2('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
Party','Are you planning to attend?','smtp.enterprisedb.com');
```

11.2.25 UTL_URL

The `UTL_URL` package provides a way to escape illegal and reserved characters within an URL.

Function/Procedure	Return Type	Description
`ESCAPE(url, escape_reserved_chars, url_charset)`	`VARCHAR2`	Use the `ESCAPE` function to escape any illegal and reserved characters in a URL.
`UNESCAPE(url, url_charset)`	`VARCHAR2`	The `UNESCAPE` function to convert an URL to it's original form.

The `UTL_URL` package will return the `BAD_URL` exception if the call to a function includes an incorrectly-formed URL.

ESCAPE

Use the `ESCAPE` function to escape illegal and reserved characters within an URL. The signature is:

```
 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`

`url` specifies the Uniform Resource Locator that `UTL_URL` will escape.

`escape_reserved_chars`

`escape_reserved_chars` is a `BOOLEAN` value that instructs the `ESCAPE` function to escape reserved characters as well as illegal characters:

- If `escaped_reserved_chars` is `FALSE`, `ESCAPE` will escape only the illegal characters in the specified URL.
- If `escape_reserved_chars` is `TRUE`, `ESCAPE` will escape both the illegal characters and the reserved characters in the specified URL.

  By default, `escape_reserved_chars` is `FALSE`.

  Within an URL, legal characters are:

Uppercase A through Z	Lowercase a through z	0 through 9
asterisk (*)	exclamation point (!)	hyphen (-)
left parenthesis (()	period (.)	right parenthesis ())

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Uppercase A through Z	Lowercase a through z	0 through 9
single-quote (')	tilde (~)	underscore (_)

Some characters are legal in some parts of an 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 an URL are:

Illegal Character	Escape Sequence
a blank space ( )	`%20`
curly braces ({ or })	`%7b` and `%7d`
hash mark (#)	`%23`

The `ESCAPE` function considers the following characters to be reserved, and will escape them if `escape_reserved_chars` is set to `TRUE`:

Reserved Character	Escape Sequence
ampersand (&)	`%5C`
at sign (@)	`%25`
colon (:)	`%3a`
comma (,)	`%2c`
dollar sign ($)	`%24`
equal sign (=)	`%3d`
plus sign (+)	`%2b`
question mark (?)	`%3f`
semi-colon (;)	`%3b`
slash (/)	`%2f`

`url_charset`

`url_charset` specifies a character set to which a given character will be converted before it is escaped. If `url_charset` is `NULL`, the character will not be converted. The default value of `url_charset` is `ISO-8859-1`.

Examples

The following anonymous block uses the `ESCAPE` function to escape the blank spaces in the URL:

```
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
```

If you include a value of `TRUE` for the `escape_reserved_chars` parameter when invoking the function:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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%2F%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 it's original form.

The signature is:

```
UNESCAPE(<url> VARCHAR2, <url_charset> VARCHAR2)
```

Parameters

`url`

`url` specifies the Uniform Resource Locator that `UTL_URL` will unescape.

`url_charset`

After unescaping a character, the character is assumed to be in `url_charset` encoding, and will be converted from that encoding to database encoding before being returned. If `url_charset` is `NULL`, the character will not be converted. The default value of `url_charset` is `ISO-8859-1`.

Examples

The following anonymous block uses the `ESCAPE` function to escape the blank spaces in the URL:

```
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 (unescaped) URL is:

```
http://www.example.com/Using the UNESCAPE function.html
```

12 Database Compatibility for Oracle Developers Tools and Utilities Guide

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The tools and utilities documented in this guide allow a developer that is accustomed to working with Oracle utilities to work with Advanced Server in a familiar environment.

The sections in this guide describe compatible tools and utilities that are supported by Advanced Server. These include:

- EDB*Loader
- EDB*Wrap
- Dynamic Runtime Instrumentation

The EDB*Plus command line client provides a user interface to 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

For detailed installation and usage information about EDB*Plus, see the EDB*Plus.

12.1 EDB*Loader

EDB*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for 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
- Syntax for control file directives compatible with Oracle SQL*Loader
- 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 do not meet the selection criteria of any target table
- Log file for recording the EDB*Loader session and any error messages
- Data loading from standard input and remote loading, particularly useful for large data sources on remote hosts

These features are explained in detail in the following sections.

> **Note**
>
> When you invoke the EDB*Loader program (called `edbldr`), you pass in parameters and directive information to the database server. **We strongly recommend that the version 11 EDB*Loader client (the edbldr program supplied with Advanced Server 11) be used to load data only into version 11 of the database server. In general, the EDB*Loader client and database server should be the same version.**

Data Loading Methods

As with Oracle SQL*Loader, EDB*Loader supports three data loading methods:

- Conventional path load
- Direct path load
- Parallel direct path load

Conventional path load is the default method used by EDB*Loader. Basic insert processing is used to add rows to the table.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The advantage of a conventional path load over the other methods is that table constraints and database objects defined on the table such as primary keys, not null constraints, check constraints, unique indexes, foreign key constraints, and triggers are enforced during a conventional path load.

One exception is that the Advanced Server *rules* defined on the table are not enforced. EDB*Loader can load tables on which rules are defined, but the rules are not executed. As a consequence, partitioned tables implemented using rules cannot be loaded using EDB*Loader.

> **Note**
>
> Advanced Server rules are created by the `CREATE RULE` command. Advanced Server rules are not the same database objects as rules and rule sets used in Oracle.

EDB*Loader also supports direct path loads. A direct path load is faster than a conventional path load, but requires the removal of most types of constraints and triggers from the table. For more information, see Direct Path Load.

Finally, EDB*Loader supports parallel direct path loads. A parallel direct path load provides even greater performance improvement by permitting multiple EDB*Loader sessions to run simultaneously to load a single table. For more information, see Parallel Direct Path Load.

General Usage

EDB*Loader can load data files with either delimiter-separated or fixed-width fields, in single-byte or multi-byte character sets. The delimiter can be a string consisting of one or more single-byte or multi-byte characters. Data file encoding and the database encoding may be different. 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 should occur 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 do not 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 do 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 do not abort the transaction:

- Attempt to load non-numeric value into a numeric column
- Numeric value is too large for a numeric column
- Character value is 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:

- Violation of a unique constraint such as a primary key or unique index
- Violation of a referential integrity constraint
- Violation of a check constraint
- Error thrown by a trigger fired as a result of inserting rows

Building the EDB*Loader Control File

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 table (or tables) from the input data file. The control file includes information such as:

- The name of the input data file containing the data to be loaded.
- The name of the table or tables to be loaded from the data file.
- Names of the columns within the table or 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 which records from the data file to load into a given table.
- The name of the file that will collect illegally formatted records.
- The name of the discard file that will collect records that do not meet the selection criteria of any table.

The syntax for the EDB*Loader control file is as follows:

```
 [ 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` that describes the location, data format, or value of the data to be inserted into `column_name` of the `target_table`. The syntax of `field_def` is the following:

```
 <column_name> {
  CONSTANT <val> |
  FILLER [ POSITION (<start:end>) ] [ <fieldtype> ] |
  BOUNDFILLER [ POSITION (<start:end>) ] [ <fieldtype> ] |
  [ POSITION (<start:end>) ] [ <fieldtype> ]
  [ NULLIF <field_condition> [ AND <field_condition> ] ...]
  [ PRESERVE 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>])]
```

Description

The specification of `data_file`, `bad_file`, and `discard_file` may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is then assumed to exist (in the case of `data_file`), or is created (in the case of `bad_file` or `discard_file`), relative to the current working directory from which `edbldr` is invoked.

You can include references to environment variables within the EDB*Loader control file when referring to a directory path and/or file name. Environment variable references are formatted differently on Windows systems than on Linux systems:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- 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 is set to the directory path and/or file name.

The `EDBLDR_ENV_STYLE` environment variable instructs Advanced Server to interpret environment variable references as Windows-styled references or Linux-styled references irregardless 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 `linux` or `unix` to instruct Advanced Server to recognize Linux-style references within the control file.
- On a Linux system, set `EDBLDR_ENV_STYLE` to `windows` to instruct Advanced Server to recognize Windows-style references within the control file.

The operating system account `enterprisedb` must have read permission on the directory and file specified by `data_file`.

The operating system account enterprisedb must have write permission on the directories where `bad_file` and `discard_file` are to be written.

> **Note**
>
> The file names for `data_file`, `bad_file`, and `discard_file` should include extensions of `.dat`, `.bad`, and `.dsc`, respectively. If the provided file name does not contain an extension, EDB*Loader assumes the actual file name includes the appropriate aforementioned extension.

If an EDB*Loader session results in data format errors and the `BADFILE` clause is not specified, nor is the BAD parameter given on the command line when `edbldr` is invoked, a bad file is created with the name `control_file_base.bad` in the current working directory from which `edbldr` is invoked. `control_file_base` is the base name of the control file (that is, the file name without any extension) used in the `edbldr` session.

If all of the following conditions are true, the discard file is not created even if the EDB*Loader session results in discarded records:

- The `DISCARDFILE` clause for specifying the discard file is not included in the control file.
- The `DISCARD` parameter for specifying the discard file is not included on the command line.
- The `DISCARDMAX` clause for specifying the maximum number of discarded records is not included in the control file.
- The `DISCARDS` clause for specifying the maximum number of discarded records is not included in the control file.
- The `DISCARDMAX` parameter for specifying the maximum number of discarded records is not included on the command line.

If neither the `DISCARDFILE` clause nor the `DISCARD` parameter for explicitly specifying the discard file name are specified, but `DISCARDMAX` or `DISCARDS` is specified, then the EDB*Loader session creates a discard file using the data file name with an extension of `.dsc`.

> **Note**
>
> There is a distinction between keywords `DISCARD` and `DISCARDS`. `DISCARD` is an EDB*Loader command line parameter used to specify the discard file name (see General Usage). `DISCARDS` is a clause of the `LOAD DATA` directive that may only appear in the control file. Keywords `DISCARDS` and `DISCARDMAX` 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 termination of the EDB*Loader session due to exceeding the `DISCARDS` or `DISCARDMAX` settings are kept in the database and are not rolled back.

If one of `INSERT`, `APPEND`, `REPLACE`, or `TRUNCATE` is specified, it establishes the default action of how rows are to be added to target tables. If omitted, the default action is as if `INSERT` had been specified.

If the `FIELDS TERMINATED BY` clause is specified, then the `POSITION (start:end)` clause may not be specified for any `field_def`. Alternatively if the `FIELDS TERMINATED BY` clause is not specified, then every `field_def` must contain either the `POSITION (start:end)` clause, the `fieldtype(length)` clause, or the `CONSTANT` clause.

Parameters

`OPTIONS param=value`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Use the `OPTIONS` clause to specify `param=value` pairs that represent an EDB*Loader directive. If a parameter is specified in both the `OPTIONS` clause and on the command line when `edbldr` is invoked, the command line setting is used.

Specify one or more of the following parameter/value pairs:

- `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`.

  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 indicate that the data should be copied with the rows `frozen`. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, see the PostgreSQL core documentation at:

  https://www.postgresql.org/docs/11/static/routine-vacuuming.html

  You must specify a data-loading type of `TRUNCATE` in the control file when using the `FREEZE` option. `FREEZE` is not 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 `PARALLEL` is `FALSE`.

  When `PARALLEL` is `TRUE`, the `DIRECT` parameter must also be set to `TRUE`. For more information about parallel direct path loads, seeParallel Direct Path Load.

- `ROWS=n`

  `n` specifies the number of rows that EDB*Loader will commit 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 will remain in the destination table.

- `SKIP=skip_count`

  `skip_count` specifies the number of records at the beginning of the input data file that should be skipped before loading begins. The default is `0`.

- `SKIP_INDEX_MAINTENANCE={ FALSE | TRUE }`

  If `SKIP_INDEX_MAINTENANCE` is `TRUE`, index maintenance is not 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**

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> During a parallel direct path load, target table indexes are not updated, and 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 availabe at:

https://www.postgresql.org/docs/11/static/sql-reindex.html

`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 available at:

https://www.postgresql.org/docs/11/static/multibyte.html

`data_file`

File containing the data to be loaded into `target_table`. Each record in the data file corresponds to a row to be inserted into `target_table`.

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of `.dat`, for example, `mydatafile.dat`.

Note: If the `DATA` parameter is specified on the command line when `edbldr` is invoked, the file given by the command line `DATA` parameter is used instead.

If the `INFILE` clause is omitted as well as the command line `DATA` parameter, then the data file name is assumed to be identical to the control file name, but with an extension of `.dat`.

`stdin`

Specify `stdin` (all lowercase letters) if you want to use standard input to pipe the data to be loaded directly to EDB*Loader. This is useful for data sources generating a large number of records to be loaded.

`bad_file`

A file that receives `data_file` records that cannot be loaded due to errors. The bad file is generated for collecting rejected or bad records.

From Advanced Server version 11 and onwards, a bad file will be generated only if there are any bad or rejected records. However, if there is an existing bad file with identical name and location, and no bad records are generated after invoking a new version of `ebdldr`, the existing bad file remains untouched.

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of `.bad`, for example, `mybadfile.bad`.

Note: If the `BAD` parameter is specified on the command line when `edbldr` is invoked, the file given by the command line `BAD` parameter is used instead.

`discard_file`

File that receives input data records that are not loaded into any table 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 an extension is not provided in the file name, EDB*Loader assumes the file has an extension of `.dsc`, for example, `mydiscardfile.dsc`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Note: If the `DISCARD` parameter is specified on the command line when `edbldr` is invoked, the file given by the command line `DISCARD` parameter is used instead.

`{ DISCARDMAX | DISCARDS } max_discard_recs`

Maximum number of discarded records that may be encountered from the input data records before terminating the EDB*Loader session. (A discarded record is described in the preceding description of the `discard_file` parameter.) Either keyword `DISCARDMAX` or `DISCARDS` may be used preceding the integer value specified by `max_discard_recs`.

For example, if `max_discard_recs` is `0`, then the EDB*Loader session is terminated if and when a first discarded record is encountered. If `max_discard_recs` is `1`, then the EDB*Loader session is terminated if and 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 have been loaded into the database are retained. They are not rolled back.

`INSERT | APPEND | REPLACE | TRUNCATE`

Specifies how data is to be loaded into the target tables. If one of `INSERT`, `APPEND`, `REPLACE`, or `TRUNCATE` is specified, it establishes the default action for all tables, overriding the default of `INSERT`.

- `INSERT`

  Data is to be loaded into an empty table. EDB*Loader throws an exception and does not load any data if the table is not initially empty.

  > **Note**
  >
  > If the table contains rows, the `TRUNCATE` command must be used to empty the table prior to invoking EDB*Loader. EDB*Loader throws an exception if the `DELETE` command is used to empty the table instead of the `TRUNCATE` command. Oracle SQL*Loader allows the table to be emptied by using either the `DELETE` or `TRUNCATE` command.

- `APPEND`

  Data is to be added to any existing rows in the table. The table may be initially empty as well.

- `REPLACE`

  The `REPLACE` keyword and `TRUNCATE` keywords are functionally identical. The table is truncated by EDB*Loader prior to loading the new data.

  > **Note**
  >
  > Delete triggers on the table are not fired as a result of the `REPLACE` operation.

- `TRUNCATE`

  The table is truncated by EDB*Loader prior to loading the new data. Delete triggers on the table are not fired as a result of the `TRUNCATE` operation.

`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, and if any trailing whitespaces are present, they are left untouched. In the case of predetermined-sized fields with `NO PRESERVE BLANKS`, the trailing whitespaces are omitted, and the leading whitespaces if any, are left untouched.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Note: If neither `PRESERVE BLANKS` nor `NO PRESERVE BLANKS` is explicitly provided, then the behavior defaults to `NO PRESERVE BLANKS`. This option does not work for ideographic whitespaces.

`target_table`

Name of the table into which data is to be loaded. The table name may be schema-qualified (for example, `enterprisedb.emp`). The specified target must not 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, and 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 that is to be compared 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.

Use of either (`start` : `end`) or `column_name` defines the portion of the record in `data_file` that is to be compared 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`.

If for a given record, none of the `WHEN` 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 may be single-byte or multi-byte as long as they are valid in the database encoding. Two consecutive appearances of `termstring` with no intervening character results in the corresponding column set to null.

`enclstring`

String of one or more characters used to enclose a field value in `data_file`. The characters may be single-byte or multi-byte as long as they are 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 may be single-byte or multi-byte as long as they are valid in the database encoding. Two consecutive appearances of `delimstring` with no intervening character results in no corresponding row loaded into the table. The last record (in other words, the end of the data file) must also be terminated by the `delimstring` characters, otherwise the final record is not loaded into the table.

Note: The `RECORDS DELIMITED BY 'delimstring'` clause is not compatible with Oracle databases.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`TRAILING NULLCOLS`

If `TRAILING NULLCOLS` is specified, then the columns in the column list for which there is no data in `data_file` for a given record, are set to null when the row is inserted. This 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 `TRAILING NULLCOLS` is not specified, 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 a field value defined by `field_def` is to be inserted. If the field definition includes the `FILLER` or `BOUNDFILLER` clause, then `column_name` is not required to be the name of a column in the table. It can be any identifier name since the `FILLER` and `BOUNDFILLER` clauses prevent the loading of the field data into a table column.

`CONSTANT val`

Specifies a constant that is type-compatible with the column data type to which it is assigned in a field definition. Single or double quotes may enclose `val`. If `val` contains white space, then enclosing quotation marks must be used.

The use of the `CONSTANT` clause completely determines the value to be assigned to a column in each inserted row. No other clause may appear in the same field definition.

If the `TERMINATED BY` clause is used 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 is no field in `data_file` for any field definition with a `CONSTANT` clause.

`FILLER`

Specifies that the data in the field defined by the field definition is not to be loaded into the associated column if the identifier of the field definition is an actual column name in the table. In such case, the column is set to null. Use of the `FILLER` or `BOUNDFILLER` clause is the only circumstance in which the field definition does not have to be identified by an actual column name.

Unlike the `BOUNDFILLER` clause, an identifier defined with the `FILLER` clause must not be referenced in a `SQL` expression. See the discussion of the `expr` parameter.

`BOUNDFILLER`

Specifies that the data in the field defined by the field definition is not to be loaded into the associated column if the identifier of the field definition is an actual column name in the table. In such case, the column is set to null. Use of the `FILLER` or `BOUNDFILLER` clause is the only circumstance in which the field definition does not have to be identified by an actual column name.

Unlike the `FILLER` clause, an identifier defined with the `BOUNDFILLER` clause may be referenced in a SQL expression. 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`.

```
 CHAR [(<length>)] | DATE [(<length>)] | TIMESTAMP [(<length>)] [ "<datemask>" ] |
INTEGER EXTERNAL [(<length>)] |
FLOAT EXTERNAL [(<length>)] | DECIMAL EXTERNAL [(<length>)] |
ZONED EXTERNAL [(<length>)] | ZONED [(<precision>[,<scale>])]
```

Field type that describes the format of the data field in `data_file`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Note: Specification of a field type is optional (for descriptive purposes only) and has no effect on whether or not EDB*Loader successfully inserts the data in the field into the table column. Successful loading depends upon 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 `2600`, but if the field contains a value such as `26XX`, the insertion fails and the record is written to `bad_file`.

Please note that `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 be loaded into the associated column.

If the `POSITION (start:end)` clause is specified along 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 be loaded 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 be loaded into the column may be shorter than the field defined by `POSITION (start:end)`, or it may go beyond the *end* position depending upon the specified `length` size.

If the `FIELDS TERMINATED BY 'termstring'` clause is specified as part of the `INTO TABLE` clause, and a field definition contains the `fieldtype(length)` clause, then a record is accepted as long as the specified `length` values are 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 the `FIELDS TERMINATED BY 'termstring'` clause is not specified, and the `POSITION (start:end)` clause is not included 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, or by 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 will use the value specified for `precision`.

`scale`

`scale` 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 the `DATE` or `TIMESTAMP` field type is specified along with a SQL expression for the column, then `datemask` must be specified after `DATE` or `TIMESTAMP` and before the SQL expression. See the following 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 case of `time_stamp timestamp "select to_timestamp(:time_stamp, 'yyyymmddhh24miss')"`, the EDB*Loader cannot 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 will not be 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 the simple workaround is to specify the datemask and SQL expression using the `TO_CHAR` function as:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
time_stamp timestamp "yyyymmddhh24miss" "to_char(to_timestamp(:time_stamp, 'yyyymmddhh24miss'),
'yyyymmddhh24miss')"
```

`NULLIF field_condition [ AND field_condition ] ...`

See the description of `field_condition` previously listed in this Parameters section 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 would normally occur 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, and if any trailing whitespaces are present, they are left untouched. In the case of predetermined-sized fields with `NO PRESERVE BLANKS`, the trailing whitespaces are omitted, and the leading whitespaces if any, are left untouched.

Note: If neither `PRESERVE BLANKS` nor `NO PRESERVE BLANKS` is explicitly provided, then the behavior defaults to `NO PRESERVE BLANKS`. This option does not work for ideographic whitespaces.

`expr`

A SQL expression returning a scalar value that is type-compatible with the column data type to which it is assigned in a field definition. Double quotes must enclose `expr`. `expr` may contain a reference to any column in the field list (except for fields with the `FILLER` clause) by prefixing the column name by a colon character `(:)`.

`expr` may also consist of a SQL `SELECT` statement. If a `SELECT` statement is used then the following rules must apply:

- The `SELECT` statement must be enclosed within parentheses `(SELECT ...)`.
- The select list must consist of exactly 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 use of the `SELECT` statement:

  ```
  "(SELECT <expr> [ FROM <table_list> [ WHERE <condition> ] ])"
  ```

  > **Note**
  >
  > Omitting the `FROM table_list` clause is not compatible with Oracle databases. If no tables need to be specified, use of the `FROM DUAL` clause is compatible with Oracle databases.

EDB Loader Control File Examples

The following are some examples of control files and their corresponding data files.

Delimiter-Separated Field Data File

The following control file uses a delimiter-separated data file that appends rows to the `emp` table:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
LOAD DATA
  INFILE '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
  )
```

In the preceding control file, the `APPEND` clause is used to allow the insertion of additional rows into the `emp` table.

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 the last field supplying the `comm` column to be omitted from the first and last records. The `comm` column is set to null for the rows inserted from these records.

The double quotation mark enclosure character surrounds the value `JONES, JR.` in the last record since the comma delimiter character is part of the field value.

The following query displays the rows added to the table after the EDB*Loader session:

```
SELECT * FROM emp WHERE empno > 9100;

 empno| ename     | job     | mgr|   hiredate        | sal   | comm   |deptno
------+-----------+---------+----+-------------------+-------+--------+------
  9101| ROGERS    | CLERK   |7902| 17-DEC-10 00:00:00|1980.00|        |  20
  9102| PETERSON  | SALESMAN|7698| 20-DEC-10 00:00:00|2600.00| 2300.00|  30
  9103| WARREN    | SALESMAN|7698| 22-DEC-10 00:00:00|5250.00| 2500.00|  30
  9104| JONES, JR.| MANAGER |7839| 02-APR-09 00:00:00|7975.00|        |  20
(4 rows)
```

Fixed-Width Field Data File

The following example is a control file that loads the same rows into the `emp` table, but uses a 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),
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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)
  )
```

In the preceding control file, the `FIELDS TERMINATED BY` and `OPTIONALLY ENCLOSED BY` clauses are absent. Instead, each field now includes the `POSITION` clause.

The following is the corresponding data file containing fixed-width fields:

```
 9101ROGERS     CLERK     790217-DEC-10     1980.0020
9102PETERSON   SALESMAN 769820-DEC-10    2600.0030 2300.00
9103WARREN     SALESMAN 769822-DEC-10    5250.0030 2500.00
9104JONES, JR.MANAGER   783902-APR-09     7975.0020
```

Single Physical Record Data File – RECORDS DELIMITED BY Clause

The following example is a control file that loads the same rows into the `emp` table, but uses a data file with one physical record. Each individual record that is to be loaded as a row in the table is terminated by the semicolon character `(;)` specified by the `RECORDS DELIMITED BY` clause.

```
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,;
```

FILLER Clause

The following control file illustrates the use of 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.

```
LOAD DATA
  INFILE       'emp_fixed.dat'
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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         FILLER POSITION (39:46),
    deptno      POSITION (47:48),
    comm        FILLER POSITION (49:56)
  )
```

Using the same fixed-width data file as in the prior fixed-width field example, the resulting rows in the table appear as follows:

```
SELECT * FROM emp WHERE empno > 9100;

 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)
```

BOUNDFILLER Clause

The following control file illustrates the use of 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, columns with the `BOUNDFILLER` clause are permitted to be used in an expression as shown for column `jobdesc`.

```
LOAD DATA
  INFILE    'emp.dat'
    BADFILE 'emp.bad'
  APPEND
  INTO TABLE empjob
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job          BOUNDFILLER,
    mgr          BOUNDFILLER,
    hiredate     FILLER,
    sal          FILLER,
    deptno       FILLER,
    comm         FILLER,
    jobdesc      ":job || ' for manager ' || :mgr"
  )
```

The following is the delimiter-separated data file used in this example.

```
 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 - 2024 EnterpriseDB Corporation. All rights reserved.

The following table is loaded using the preceding control file and data file.

```
CREATE TABLE empjob (
    empno               NUMBER(4) NOT NULL CONSTRAINT empjob_pk PRIMARY KEY,
    ename               VARCHAR2(10),
    job                 VARCHAR2(9),
    mgr                 NUMBER(4),
    jobdesc             VARCHAR2(25)
);
```

The resulting rows in the table appear as follows:

```
SELECT * FROM empjob;

 empno |    ename   | job | mgr |         jobdesc
-------+------------+-----+-----+--------------------------
 9101  | ROGERS     |     |     | CLERK for manager 7902
 9102  | PETERSON   |     |     | SALESMAN for manager 7698
 9103  | WARREN     |     |     | SALESMAN for manager 7698
 9104  | JONES, JR. |     |     | MANAGER for manager 7839
(4 rows)
```

Field Types with Length Specification

The following example is a control file that 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
  (
    empno       CHAR(4),
    ename       CHAR(10),
    job         POSITION (15:23) CHAR(9),
    mgr         INTEGER EXTERNAL(4),
    hiredate    DATE(11) "DD-MON-YY",
    sal         DECIMAL EXTERNAL(8),
    deptno      POSITION (47:48),
    comm        POSITION (49:56) DECIMAL EXTERNAL(8)
  )
```

> **Note**
>
> The `POSITION` clause and the `fieldtype(length)` clause can be used individually or in combination as long as each field definition contains at least one of the two clauses.

The following is the corresponding data file containing fixed-width fields:

```
 9101ROGERS     CLERK      790217-DEC-10  1980.0020
9102PETERSON   SALESMAN   769820-DEC-10  2600.0030  2300.00
9103WARREN     SALESMAN   769822-DEC-10  5250.0030  2500.00
9104JONES, JR. MANAGER    783902-APR-09  7975.0020
```

The resulting rows in the table appear as follows:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SELECT * FROM emp WHERE empno > 9100;

 empno| ename     | job     | mgr|   hiredate        | sal   | comm   |deptno
------+-----------+---------+----+-------------------+-------+--------+------
  9101| ROGERS    | CLERK   |7902| 17-DEC-10 00:00:00|1980.00|        |  20
  9102| PETERSON  | SALESMAN|7698| 20-DEC-10 00:00:00|2600.00| 2300.00|  30
  9103| WARREN    | SALESMAN|7698| 22-DEC-10 00:00:00|5250.00| 2500.00|  30
  9104| JONES, JR.| MANAGER |7839| 02-APR-09 00:00:00|7975.00|        |  20
(4 rows)
```

NULLIF Clause

The following example uses the `NULLIF` clause on the `sal` column to set it to null for employees of job `MANAGER` as well as on the `comm` column to set it to null if the employee is not a `SALESMAN` and is not in department `30`. In other words, a `comm` value is accepted if the employee is a `SALESMAN` or is a member of department `30`.

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) NULLIF job = 'MANAGER',
    deptno      POSITION (47:48),
    comm        POSITION (49:56) NULLIF job <> 'SALESMAN' AND deptno <> '30'
  )
```

The following is the corresponding data file:

```
 9101ROGERS        CLERK         790217-DEC-10     1980.0020
9102PETERSON      SALESMAN      769820-DEC-10     2600.0030     2300.00
9103WARREN        SALESMAN      769822-DEC-10     5250.0030     2500.00
9104JONES, JR.    MANAGER       783902-APR-09     7975.0020
9105ARNOLDS       CLERK         778213-SEP-10     3750.0030     800.00
9106JACKSON       ANALYST       756603-JAN-11     4500.0040     2000.00
9107MAXWELL       SALESMAN      769820-DEC-10     2600.0010     1600.00
```

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;

 empno |   ename    |    job    |   sal    |   comm    | deptno
-------+------------+-----------+----------+-----------+-------
  9101 | ROGERS     | CLERK     | 1980.00  | --null--  |   20
  9102 | PETERSON   | SALESMAN  | 2600.00  | 2300.00   |   30
  9103 | WARREN     | SALESMAN  | 5250.00  | 2500.00   |   30
  9104 | JONES, JR. | MANAGER   | --null-- | --null--  |   20
  9105 | ARNOLDS    | CLERK     | 3750.00  |  800.00   |   30
  9106 | JACKSON    | ANALYST   | 4500.00  | --null--  |   40
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  9107 | MAXWELL    | SALESMAN | 2600.00  | 1600.00  |   10
(7 rows)
```

> **Note**
>
> 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 set to null since this employee is neither a `SALESMAN` nor is a member of department `30`.

SELECT Statement in a Field Expression

The following example uses a `SELECT` statement in the expression of the field definition to return the value to be loaded into the column.

```
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 = :deptno)",
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      POSITION (47:48),
    comm        POSITION (49:56)
  )
```

The content of the `dept` table used in the `SELECT` statement is the following:

```
SELECT * FROM dept;

 deptno |    dname   |   loc
--------+------------+---------
   10   | ACCOUNTING | NEW YORK
   20   | RESEARCH   | DALLAS
   30   | SALES      | CHICAGO
   40   | OPERATIONS | BOSTON
(4 rows)
```

The following is the corresponding data file:

```
 9101ROGERS     CLERK     790217-DEC-10    1980.0020
9102PETERSON   SALESMAN  769820-DEC-10    2600.0030   2300.00
9103WARREN     SALESMAN  769822-DEC-10    5250.0030   2500.00
9104JONES, JR. MANAGER   783902-APR-09    7975.0020
```

The resulting rows in the table appear as follows:

```
SELECT * FROM emp WHERE empno > 9100;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 empno| ename      | job      | mgr|   hiredate        | sal   | comm   |deptno
------+-----------+----------+----+-------------------+-------+--------+------
  9101| ROGERS     | RESEARCH|7902| 17-DEC-10 00:00:00|1980.00|        |  20
  9102| PETERSON   | SALES    |7698| 20-DEC-10 00:00:00|2600.00| 2300.00|  30
  9103| WARREN     | SALES    |7698| 22-DEC-10 00:00:00|5250.00| 2500.00|  30
  9104| JONES, JR.| RESEARCH|7839| 02-APR-09 00:00:00|7975.00|        |  20
(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

The following example illustrates the use of 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, while inserting no rows from the original `emp` table:

```
CREATE TABLE emp_research AS SELECT * FROM emp WHERE deptno = 99;
CREATE TABLE emp_sales AS SELECT * FROM emp WHERE deptno = 99;
```

The following control file contains two `INTO TABLE` clauses. Also note that there is no `APPEND` clause so the default operation of `INSERT` is used, which requires that tables `emp_research` and `emp_sales` be empty.

```
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),
    deptno     CONSTANT '30',
    comm       POSITION (49:56) "ROUND(:comm + (:sal * .25), 0)"
  )
```

The `WHEN` clauses specify that when the field designated by columns 47 thru 48 contains `20` , the record is inserted into the `emp_research` table and 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

named `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.

Finally, column `comm` of the `emp_sales` table is assigned a SQL expression. Column names may be referenced in the expression by prefixing the column name with a colon character `(:)`.

The following is the corresponding data file:

```
 9101ROGERS      CLERK     790217-DEC-10   1980.0020
9102PETERSON    SALESMAN  769820-DEC-10   2600.0030  2300.00
9103WARREN      SALESMAN  769822-DEC-10   5250.0030  2500.00
9104JONES, JR. MANAGER   783902-APR-09   7975.0020
9105ARNOLDS     CLERK     778213-SEP-10   3750.0010
9106JACKSON     ANALYST   756603-JAN-11   4500.0040
```

Since the records for employees `ARNOLDS` and `JACKSON` contain `10` and `40` in columns 47 thru 48, which do not satisfy any of the `WHEN` clauses, EDB*Loader writes these two records to the discard file, `emp_multitbl.dsc`, whose content is shown by the following:

```
 9105ARNOLDS    CLERK     778213-SEP-10   3750.0010
9106JACKSON    ANALYST   756603-JAN-11   4500.0040
```

The following are the rows loaded into the `emp_research` and `emp_sales` tables:

```
 SELECT * FROM emp_research;

empno  |   ename    |   job   |  mgr |      hiredate      |   sal   | comm | deptno
-------+------------+---------+------+--------------------+---------+------+-------
 9101  | ROGERS     | CLERK   | 7902 | 17-DEC-10 00:00:00 | 1980.00 |      | 20.00
 9104  | JONES, JR. | MANAGER | 7839 | 02-APR-09 00:00:00 | 7975.00 |      | 20.00
(2 rows)

SELECT * FROM emp_sales;

empno  |   ename  |    job   |  mgr |       hiredate     |    sal  |   comm  | deptno
-------+----------+----------+------+--------------------+---------+---------+--------
 9102  | PETERSON | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2950.00 | 30.00
 9103  | WARREN   | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 3813.00 | 30.00
(2 rows)
```

Invoking EDB*Loader

You must have superuser privileges to run EDB*Loader. Use the following command to invoke EDB*Loader from the command line:

```
edbldr [ -d <dbname> ] [ -p <port> ] [ -h <host> ]
[ USERID={ <username/password> | <username>/ | <username> | / } ]
  CONTROL=<control_file>
[ DATA=<data_file> ]
[ BAD=<bad_file>]
[ DISCARD=<discard_file> ]
[ DISCARDMAX=<max_discard_recs> ]
[ LOG=<log_file> ]
[ PARFILE=<param_file> ]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
[ DIRECT={ FALSE | TRUE } ]
[ FREEZE={ FALSE | TRUE } ]
[ ERRORS=<error_count> ]
[ PARALLEL={ FALSE | TRUE } ]
[ ROWS=<n> ]
[ SKIP=<skip_count> ]
[ SKIP_INDEX_MAINTENANCE={ FALSE | TRUE } ]
[ edb_resource_group=<group_name> ]
```

Description

If the `-d` option, the `-p` option, or the `-h` option are omitted, the defaults for the database, port, and host are determined according to the same rules as other Advanced Server utility programs such as `edb-psql`, for example.

Any parameter listed in the preceding syntax diagram except for the `-d` option, `-p` option, `-h` option, and the `PARFILE` parameter may be specified in a *parameter file*. The parameter file is specified on the command line when `edbldr` is invoked using `PARFILE=param_file`. Some parameters may be specified in the `OPTIONS` clause in the control file. For more information on the control file, see Building the EDB*Loader Control File.

The specification of `control_file`, `data_file`, `bad_file`, `discard_file`, `log_file`, and `param_file` may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is assumed to exist (in the case of `control_file`, `data_file`, or `param_file`), or to be created (in the case of `bad_file`, `discard_file`, or `log_file`) relative to the current working directory from which `edbldr` is invoked.

> **Note**
>
> The control file must exist in the character set encoding of the client where `edbldr` is invoked. If the client is in a different encoding than the database encoding, then the `PGCLIENTENCODING` environment variable must be set on the client to the client's encoding prior to invoking `edbldr`. This must be done to ensure character set conversion is properly done between the client and the database server.

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 to be written.

> **Note**
>
> The file names for `control_file`, `data_file`, `bad_file`, `discard_file`, and `log_file` should include extensions of `.ctl`, `.dat`, `.bad`, `.dsc`, and `.log`, respectively. If the provided file name does not contain an extension, EDB*Loader assumes the actual file name includes the appropriate aforementioned extension.

Parameters

`dbname`

Name of the database containing the tables to be loaded.

`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 | / }`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

EDB*Loader connects to the database with `username`. `username` must be a superuser. `password` is the password for `username`.

If the `USERID` parameter is omitted, EDB*Loader prompts for `username` and `password`. If `USERID=username/` is specified, then EDB*Loader 1) uses the password file specified by environment variable `PGPASSFILE` if `PGPASSFILE` is set, or 2) uses the `.pgpass` password file `(pgpass.conf` on Windows systems) if `PGPASSFILE` is not set. If `USERID=username` is specified, then EDB*Loader prompts for `password`. If `USERID=/` is specified, the connection is attempted using the operating system account as the user name.

Note: The Advanced Server connection environment variables `PGUSER` and `PGPASSWORD` are ignored by EDB*Loader. See the PostgreSQL core documentation for information on the `PGPASSFILE` environment variable and the password file.

`CONTROL=control_file`

`control_file` specifies the name of the control file containing EDB*Loader directives. If a file extension is not specified, 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 be loaded into the target table. If a file extension is not specified, 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 cannot be loaded due to errors. Specifying a `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 do not meet any table's selection criteria. Specifying a `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 may 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.

`LOG=log_file`

`log_file` specifies the name of the file in which EDB*Loader records the results of the EDB*Loader session.

If the `LOG` parameter is omitted, 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 to be written.

`PARFILE=param_file`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`param_file` specifies the name of the file that contains command line parameters for the EDB*Loader session. Any command line parameter listed in this section except for the `-d`, `-p`, and `-h` options, and the `PARFILE` parameter itself, can be specified in `param_file` instead of on the command line.

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 is no default file name or extension assumed for `param_file`, though by Oracle SQL*Loader convention, `.par` is typically used, but not required, as an extension.

`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`.

For information about direct path loads, see Direct Path Load.

`FREEZE= { FALSE | TRUE }`

Set `FREEZE` to `TRUE` to indicate that the data should be copied with the rows `frozen`. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/11/static/routine-vacuuming.html

You must specify a data-loading type of `TRUNCATE` in the control file when using the `FREEZE` option. `FREEZE` is not supported for direct loading.

By default, `FREEZE` is `FALSE`.

`ERRORS=error_count`

`error_count` specifies the number of errors permitted before aborting the EDB*Loader session. The default is `50`.

`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.

`ROWS=n`

`n` specifies the number of rows that EDB*Loader will commit before loading the next set of `n` rows.

`SKIP=skip_count`

Number of records at the beginning of the input data file that should be skipped before loading begins. The default is `0`.

`SKIP_INDEX_MAINTENANCE= { FALSE | TRUE }`

If set to `TRUE`, index maintenance is not 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`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

During a parallel direct path load, target table indexes are not updated, and 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 available at:

https://www.postgresql.org/docs/11/static/sql-reindex.html

`edb_resource_group=group_name`

`group_name` specifies the name of an EDB Resource Manager resource group to which the EDB*Loader session is to be assigned.

Any default resource group that may have been assigned to the session (for example, a database user running the EDB*Loader session who had been assigned a default resource group with the `ALTER ROLE ... SET edb_resource_group` command) is overridden by the resource group given by the `edb_resource_group` parameter specified on the `edbldr` command line.

Examples

In the following example EDB*Loader is invoked using a control file named `emp.ctl` located in the current working directory to load a table in database `edb`:

```
$ /usr/edb/as11/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl
EDB*Loader: Copyright (c) 2007-2021, EnterpriseDB Corporation.

Successfully loaded (4) records
```

In the following example, EDB*Loader prompts for the user name and password since they are 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/as11/bin/edbldr -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
```

The following example runs EDB*Loader with the same parameters as shown in the preceding example, but using a parameter file located in the current working directory. The `SKIP` and `ERRORS` parameters are altered from their defaults in the parameter file as well. The parameter file, `emp.par`, contains the following:

```
CONTROL=emp.ctl
BAD=/tmp/emp.bad
LOG=/tmp/emp.log
SKIP=1
ERRORS=10
```

EDB*Loader is invoked with the parameter file as shown by the following:

```
$ /usr/edb/as11/bin/edbldr -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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Exit Codes

When EDB*Loader exits, it will return one of the following codes:

Exit Code	Description
`0`	Indicates that all rows loaded successfully.
`1`	Indicates that EDB*Loader encountered command line or syntax errors, or aborted the load operation due to an unrecoverable error.
`2`	Indicates that the load completed, but some (or all) rows were rejected or discarded.
`3`	Indicates that EDB*Loader encountered fatal errors (such as OS errors). This class of errors is equivalent to the `FATAL` or `PANIC` severity levels of PostgreSQL errors.

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, thereby resulting in a performance improvement.

Bypassing insert processing reduces the types of constraints that may exist on the target table. The following types of constraints are permitted on the target table of a direct path load:

- Primary key
- Not null constraints
- Indexes (unique or non-unique)

The restrictions on the target table of a direct path load are the following:

- Triggers are not permitted
- Check constraints are not permitted
- Foreign key constraints on the target table referencing another table are not permitted
- Foreign key constraints on other tables referencing the target table are not permitted
- The table must not be partitioned
- Rules may exist on the target table, but they are not executed

> **Note**
>
> Currently, a direct path load in EDB*Loader is more restrictive than in Oracle SQL*Loader. The preceding restrictions do not apply to Oracle SQL*Loader in most cases. The following restrictions apply to a control file used in a direct path load:
>
> - Multiple table loads are not supported. That is, only one `INTO TABLE` clause may be specified in the control file.
> - SQL expressions may not be used in the data field definitions of the `INTO TABLE` clause.
> - The `FREEZE` option is not supported for direct path loading.

To run a direct path load, add the `DIRECT=TRUE` option as shown by the following example:

```
$ /usr/edb/as11/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl DIRECT=TRUE
EDB*Loader: Copyright (c) 2007-2021, EnterpriseDB Corporation.

Successfully loaded (4) records
```

Parallel Direct Path Load

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The performance of a direct path load can be further improved 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 be loaded into the table must be divided amongst several data files so that each EDB*Loader session uses its own data file and the same record is not loaded more than once into the table.

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 the following:

- Triggers are not permitted
- Check constraints are not permitted
- Foreign key constraints on the target table referencing another table are not permitted
- Foreign key constraints on other tables referencing the target table are not permitted
- The table must not be partitioned
- Rules may exist on the target table, but they are not executed

In addition, the `APPEND` clause must be specified in the control file used by each EDB*Loader session.

To run a parallel direct path load, run EDB*Loader in a separate session for each participant of the parallel direct path load. Invocation of each such EDB*Loader session must include the `DIRECT=TRUE` and `PARALLEL=TRUE` parameters.

Each EDB*Loader session runs as an independent transaction so if one of the parallel sessions aborts and rolls back its changes, the loading done by the other parallel sessions are not affected.

> **Note**
>
> In a parallel direct path load, each EDB*Loader session reserves a fixed number of blocks in the target table in a round-robin fashion. Some of the blocks in the last allocated chunk may not be used,and those blocks remain uninitialized. A subsequent use of the `VACUUM` command on the target table may show warnings regarding 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 is an expected behavior and does not indicate data corruption.

Indexes on the target table are not updated during a parallel direct path load and are therefore marked as invalid after the load is complete. You must use the `REINDEX` command to rebuild the indexes.

The following 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 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 the insertion of a larger number of rows.

The following is the control file used in the first session:

```
LOAD DATA
  INFILE    '/home/user/loader/emp_parallel_1.dat'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

The `APPEND` clause must be specified in the control file for a parallel direct path load.

The following shows the invocation of EDB*Loader in the first session. The `DIRECT=TRUE` and `PARALLEL=TRUE` parameters must be specified.

```
 $ /usr/edb/as11/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. Note that it is the same as the one used in the first session, but uses a different data file.

```
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,
    deptno,
    comm
  )
```

The preceding control file is used in a second session as shown by the following:

```
 $ /usr/edb/as11/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 the following message in each session when its respective load operation completes:

```
Successfully loaded (10000) records
```

The following query shows that the index on the emp table has been marked as `INVALID`:

```
SELECT index_name, status FROM user_indexes WHERE table_name = 'EMP';

  index_name | status
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
-------------+---------
   EMP_PK    | INVALID
 (1 row)
```

> **Note**
>
> `user_indexes` is the view of indexes compatible with Oracle databases owned by the current user.

Queries on the `emp` table will not utilize the index unless it is rebuilt using the `REINDEX` command as shown by the following:

```
REINDEX INDEX emp_pk;
```

A subsequent query on `user_indexes` shows that the index is now marked as `VALID`:

```
SELECT index_name, status FROM user_indexes WHERE table_name = 'EMP';

  index_name | status
-------------+--------
   EMP_PK    | VALID
 (1 row)
```

Remote Loading

EDB*Loader supports a feature called *remote loading*. In remote loading, the database containing the table to be loaded is running on a database server on a different host than from where EDB*Loader is invoked with the input data source.

This feature is useful if you have a large amount of data to be loaded, and you do not 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, which then loads the table in the remote database. This bypasses the process of having to create a data file on disk for EDB*Loader.

Performing remote loading along with using standard input requires the following:

- The `edbldr` program must be installed on the client host on which it is to be invoked with the data source for the EDB*Loader session.
- The control file must contain the clause `INFILE 'stdin'` so you can pipe the data directly into EDB*Loader's standard input. For more information, see Building the EDB*Loader Control File for information on the `INFILE` clause and 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 are created on, the client host on which `edbldr` is invoked.
- When invoking EDB*Loader, use the `-h` option to specify the IP address of the remote database server. For more information, see Invoking EDB*Loader for information on invoking EDB*Loader.
- Use the operating system pipe operator `(|)` or input redirection operator `(<)` to supply the input data to EDB*Loader.

The following example loads a database running on a database server at `192.168.1.14` using data piped from a source named `datasource`.

```
 datasource | ./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password
CONTROL=remote.ctl
```

The following is another example of how standard input can be used:

```
 ./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password
CONTROL=remote.ctl < datasource
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Updating a Table with a Conventional Path Load

You can use EDB*Loader with a conventional path load to update the rows within 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 does not 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. Please note that you cannot use EDB*Loader to `UPDATE` a partitioned table.

To perform an `UPDATE`, use the same steps as when performing a conventional path load:

1. Create a data file that contains the rows you wish 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.

The following example uses the `emp` table that is distributed with the Advanced Server sample data. By default, the table contains:

```
edb=# select * from emp;
empno|ename |   job    |  mgr |       hiredate      |   sal    | comm   | deptno
-----+------+---------+------+---------------------+--------- +-------+--------
7369 |SMITH |CLERK    | 7902 | 17-DEC-80 00:00:00 | 800.00   |        |   20
7499 |ALLEN |SALESMAN | 7698 | 20-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 | 02-APR-81 00:00:00 | 2975.00  |        |   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 | 2850.00  |        |   30
7782 |CLARK |MANAGER  | 7839 | 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 | 7698 | 08-SEP-81 00:00:00 | 1500.00  | 0.00   |   30
7876 |ADAMS |CLERK    | 7788 | 23-MAY-87 00:00:00 | 1100.00  |        |   20
7900 |JAMES |CLERK    | 7698 | 03-DEC-81 00:00:00 | 950.00   |        |   30
7902 |FORD  |ANALYST  | 7566 | 03-DEC-81 00:00:00 | 3000.00  |        |   20
7934 |MILLER|CLERK    | 7782 | 23-JAN-82 00:00:00 | 1300.00  |        |   10
(14 rows)
```

The following 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:

```
LOAD DATA
  INFILE 'emp_update.dat'
  BADFILE 'emp_update.bad'
  DISCARDFILE 'emp_update.dsc'
UPDATE INTO TABLE emp
FIELDS TERMINATED BY ","
(empno, ename, job, mgr, hiredate, sal, comm, deptno)
```

The data that is being updated or inserted is saved in the `emp_update.dat` file. `emp_update.dat` contains:

```
7521,WARD,MANAGER,7839,22-FEB-81 00:00:00,3000.00,0.00,30
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 00:00:00,1800.00,500.00,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1500.00,400.00,30
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Invoke EDB*Loader, specifying the name of the database (`edb`), the name of a database superuser (and their associated password) and the name of the control file (`emp_update.ctl`):

```
edbldr -d edb userid=user_name/password control=emp_update.ctl
```

After performing the update, the `emp` table contains:

```
 edb=# select * from emp;
empno|ename |   job    |  mgr |      hiredate       |   sal   |  comm  | deptno
-----+------+---------+------+---------------------+---------+--------+--------
7369 |SMITH |CLERK     | 7902 | 17-DEC-80 00:00:00 | 800.00  |        |  20
7499 |ALLEN |SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 | 300.00 |  30
7521 |WARD  |MANAGER   | 7839 | 22-FEB-81 00:00:00 | 3000.00 | 0.00   |  30
7566 |JONES |MANAGER   | 7839 | 02-APR-81 00:00:00 | 3500.00 | 0.00   |  20
7654 |MARTIN|SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1500.00 | 400.00 |  30
7698 |BLAKE |MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850.00 |        |  30
7782 |CLARK |MANAGER   | 7839 | 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  | 7698 | 08-SEP-81 00:00:00 | 1500.00 | 0.00   |  30
7876 |ADAMS |CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |        |  20
7900 |JAMES |CLERK     | 7698 | 03-DEC-81 00:00:00 | 950.00  |        |  30
7902 |FORD  |ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |        |  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 | 1800.00 | 500.00 |  20
7934 |MILLER|CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |        |  10
(16 rows)
```

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`)

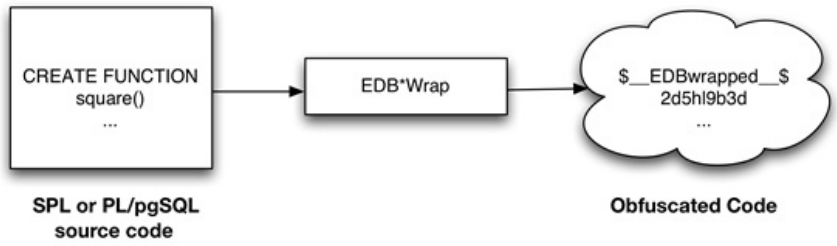
12.2 EDB*Wrap

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 is 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 will store 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 may 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 what 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 will be stored plaintext, while the package body source and package functions and procedures will be stored obfuscated.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.



Once wrapped, source code and programs cannot be unwrapped or debugged. Reverse engineering is possible, but would be very difficult.

The entire source file is wrapped into one unit. Any `psql` meta-commands included in the wrapped file will not be recognized when the file is executed; executing an obfuscated file that contains a psql meta-command will cause a syntax error. `edbwrap` does not validate SQL source code -if the plaintext form contains a syntax error, `edbwrap` will not complain. Instead, the server will report an error and abort the entire file when you try to execute the obfuscated form.

Using EDB*Wrap to Obfuscate Source Code

EDB*Wrap is a command line utility; it 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 may also specify the name of the file where `edbwrap` will write the obfuscated form of the code. `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` does not contain an extension, `edbwrap` will search for a file named `input_file.sql`

The `oname=output_file` argument (which is optional) specifies the name of the output file; if `output_file` does not contain an extension, `edbwrap` will append `.plb` to the name.

If you do not 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 that may feel more familiar:

```
edbwrap --iname <input_file> [--oname <output_file>]
edbwrap -i <input_file> [-o <output_file>]
```

You may mix command-line styles; the rules for deriving input and output file names are identical regardless of which style you use.

Once `edbwrap` has produced 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.

The following sequence demonstrates `edbwrap` functionality.

First, create the source code for the `list_emp` procedure (in plaintext form):

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
[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;
/
```

You can import the `list_emp` procedure with a client application such as `edb-psql`:

```
[bash] edb-psql edb
Welcome 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 commands
       \g or terminate with semicolon to execute query
       \q to quit

edb=# \i listemp.sql
CREATE PROCEDURE
```

You can view the plaintext source code (stored in the server) by examining the `pg_proc` system table:

```
edb=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';
                           prosrc
-----------------------------------------------------------------


     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
(1 row)

edb=# quit
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Next, obfuscate 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+6DL30RVaGjYMIzkuoSzAQgtBw7MhYFuAFkBsfYfhdJ0rjwBv+bHr1FCyH6j9SgH
movU+bYI+jR+hR2jbzq3sovHKEyZIp9y3/GckbQgualRhIlGpyWfE0dltDUpkYRLN
/OUXmk0/P4H6EI98sAHevGDhOWI+58DjJ44qhZ+l5NNEVxbWDztpb/s5sdx4660qQ
Ozx3/gh8VkqS2JbcxYMpjmrwVr6fAXfb68Ml9mW2Hl7fNtxcb5kjSzXvfWR2XYzJf
KFNrEhbL1DTVlSEC5wE6lGlwhYvXOf22m1R2IFns0MtF9fwcnBWAs1YqjR00j6+fc
er/f/efAFh4=
$__EDBwrapped__$
```

You may notice that 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 are running edbwrap while your locale is set to en_US.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 does not match the encoding of the input file, you should change your locale and rewrap the input file.

You can import the obfuscated code into 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
       \h for help with SQL commands
       \? for help with edb-psql commands
       \g or terminate with semicolon to execute query
       \q to quit

edb=# \i listemp.plb
CREATE PROCEDURE

Now, the pg_proc system table contains the obfuscated code:

edb=# SELECT prosrc FROM pg_proc WHERE proname = 'list_emp';
                                    prosrc
-------------------------------------------------------------------
  $__EDBwrapped__$
  UTF8
  dw4B9Tz69J3WOsy0GgYJQa+G2sLZ3IOyxS8pDyuOTFuiYe/EXiEatwwG3h3tdJk
  ea+AIp35dS/4idbN8wpegM3s994dQ3R97NgNHfvTQnO2vtd4wQtsQ/Zc4v4Lhfj
  nlV+A4UpHI5oQEnXeAch2LcRD87hkU0uo1ESeQV8IrXaj9BsZr+ueROnwhGs/Ec
  pva/tRV4m9RusFn0wyr38u4Z8w4dfnPW184Y3o6It4b3aH07WxTkWrMLmOZW1jJ
  Nu6u4o+ezO64G9QKPazgehslv4JB9NQnuocActfDSPMY7R7anmgw
  $__EDBwrapped__$
(1 row)
```

Invoke the obfuscated code in the same way that you would invoke the plaintext form:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# exec list_emp;

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

EDB-SPL Procedure successfully completed
edb=# quit
```

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 will show wrapped programs in plaintext form. Source code is also displayed in plaintext in SQL error messages generated during the execution of a program.

> **Note**
>
> At this time, the bodies of the objects created by the following statements will not be stored in obfuscated form:
>
> ```
> CREATE [OR REPLACE] TYPE type_name AS OBJECT
> CREATE [OR REPLACE] TYPE type_name UNDER type_name
> CREATE [OR REPLACE] TYPE BODY type_name
> ```

12.3 Dynamic Runtime Instrumentation Tools Architecture (DRITA)

The Dynamic Runtime Instrumentation Tools Architecture (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. Each snapshot is identified by a unique ID number; you can use snapshot ID numbers with DRITA reporting functions to return system performance statistics.

Configuring and Using DRITA

Advanced Server's `postgresql.conf` file includes a configuration parameter named `timed_statistics` that controls the collection of timing data. The valid parameter values are `TRUE` or `FALSE`; the default value is `FALSE`.

This is a dynamic parameter which can be modified in the `postgresql.conf` file, or while a session is in progress. To enable DRITA, you must either:

Modify the `postgresql.conf` file, setting the `timed_statistics` parameter to `TRUE`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

or

Connect to the server with the EDB-PSQL client, and invoke the command:

```
SET timed_statistics = TRUE
```

After modifying the `timed_statistics` parameter, take a starting snapshot. A snapshot captures the current state of each timer and event counter. The server will compare 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)
```

Then, run the workload that you would like to evaluate; when the workload has completed (or at a strategic point during the workload), take another snapshot:

```
edb=# SELECT * FROM edbsnap();
    edbsnap
----------------------
 Statement processed.
(1 row)
```

You can capture multiple snapshots during a session. Then, use the DRITA functions and reports to manage and compare the snapshots to evaluate performance information.

DRITA Functions

You can use DRITA functions to gather wait information and manage snapshots. DRITA functions are fully supported by Advanced Server 10 whether your installation is made compatible with Oracle databases or is made in PostgreSQL-compatible mode.

get_snaps()

The `get_snaps()` function returns a list of the current snapshots. The signature is:

```
get_snaps()
```

The following example demonstrates using the `get_snaps()` function to display a list of snapshots:

```
SELECT * FROM get_snaps();
       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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 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
(12 rows)
```

The first column in the result list displays the snapshot identifier; the second column displays the date and time that the snapshot was captured.

sys_rpt()

The `sys_rpt()` function returns system wait information. The signature is:

```
sys_rpt(<beginning_id>, <ending_id>, <top_n>)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

This example demonstrates a call to the `sys_rpt()` function:

```
SELECT * FROM sys_rpt(9, 10, 10);
                                sys_rpt
------------------------------------------------------------------------------
WAIT NAME                    COUNT     WAIT TIME                     % WAIT
------------------------------------------------------------------------------
wal flush                    8359      1.357593                      30.62
wal write                    8358      1.349153                      30.43
wal file sync                8358      1.286437                      29.02
query plan                   33439     0.439324                       9.91
db file extend               54        0.000585                       0.01
db file read                 31        0.000307                       0.01
other lwlock acquire         0         0.000000                       0.00
ProcArrayLock                0         0.000000                       0.00
CLogControlLock              0         0.000000                       0.00
(11 rows)
```

The information displayed in the result set includes:

Column Name	Description
`WAIT NAME`	The name of the wait.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column Name	Description
COUNT	The number of times that the wait event occurred.
WAIT TIME	The time of the wait event in seconds.
% WAIT	The percentage of the total wait time used by this wait for this session.

sess_rpt()

The `sess_rpt()` function returns session wait information. The signature is:

```
sess_rpt(<beginning_id>, <ending_id>, <top_n>)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

The following example demonstrates a call to the `sess_rpt()` function:

```
SELECT * FROM sess_rpt(8, 9, 10);

                                  sess_rpt
-------------------------------------------------------------------------------------
ID    USER       WAIT NAME                 COUNT TIME           % WAIT SES %  WAIT ALL
-------------------------------------------------------------------------------------
3501 enterprise  wal flush                 8354  1.354958      30.61          30.61
3501 enterprise  wal write                 8354  1.348192      30.46          30.46
3501 enterprise  wal file sync             8354  1.285607      29.04          29.04
3501 enterprise  query plan                33413 0.436901      9.87           9.87
3501 enterprise  db file extend            54    0.000578      0.01           0.01
3501 enterprise  db file read              56    0.000541      0.01           0.01
3501 enterprise  ProcArrayLock            0     0.000000      0.00           0.00
3501 enterprise  CLogControlLock          0     0.000000      0.00           0.00
(10 rows)
```

The information displayed in the result set includes:

Column Name	Description
ID	The processID of the session.
USER	The name of the user incurring the wait.
WAIT NAME	The name of the wait event.
COUNT	The number of times that the wait event occurred.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column Name	Description
`TIME`	The length of the wait event in seconds.
`% WAIT SES`	The percentage of the total wait time used by this wait for this session.
`% WAIT ALL`	The percentage of the total wait time used by this wait (for all sessions).

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`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`backend_id`

`backend_id` is an integer value that represents the backend identifier.

The following code sample demonstrates a call to `sessid_rpt()`:

```
SELECT * FROM sessid_rpt(8, 9, 3501);

                                  sessid_rpt
-------------------------------------------------------------------------------
ID    USER       WAIT NAME                 COUNT  TIME          % WAIT SES % WAIT ALL
-------------------------------------------------------------------------------
3501 enterprise CLogControlLock           0      0.000000      0.00         0.00
3501 enterprise ProcArrayLock             0      0.000000      0.00         0.00
3501 enterprise db file read              56     0.000541      0.01         0.01
3501 enterprise db file extend            54     0.000578      0.01         0.01
3501 enterprise query plan                33413  0.436901      9.87         9.87
3501 enterprise wal file sync             8354   1.285607      29.04        29.04
3501 enterprise wal write                 8354   1.348192      30.46        30.46
3501 enterprise wal flush                 8354   1.354958      30.61        30.61
(10 rows)
```

The information displayed in the result set includes:

Column Name	Description
`ID`	The process ID of the wait.
`USER`	The name of the user that owns the session.
`WAIT NAME`	The name of the wait event.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column Name	Description
`COUNT`	The number of times that the wait event occurred.
`TIME`	The length of the wait in seconds.
`% WAIT SES`	The percentage of the total wait time used by this wait for this session.
`% WAIT ALL`	The percentage of the total wait time used by this wait (for all sessions).

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`

`snapshot_id` is an integer value that identifies the snapshot.

`session_id`

`session_id` is an integer value that represents the session.

The following example demonstrates a call to the `sesshist_rpt()` function:

> **Note**
>
> The following example has been shortened; over 1300 rows were actually generated.

```
SELECT * FROM sesshist_rpt (9, 3501);
                                  sesshist_rpt
--------------------------------------------------------------------------------
----------
 ID    USER      SEQ   WAIT NAME      ELAPSED   File   Name                #
 of Blk   Sum of Blks
--------------------------------------------------------------------------------
---------
 3501 enterprise 1     query plan     13        0      N/A
0          0
 3501 enterprise 1     query plan     13        0      edb_password_history
0          0
 3501 enterprise 1     query plan     13        0      edb_password_history
0          0
 3501 enterprise 1     query plan     13        0      edb_password_history
0          0
 3501 enterprise 1     query plan     13        0      edb_profile
0          0
 3501 enterprise 1     query plan     13        0      edb_profile_name_ind
0          0
 3501 enterprise 1     query plan     13        0      edb_profile_oid_inde
0          0
 3501 enterprise 1     query plan     13        0      edb_profile_password
0          0
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 3501 enterprise 1     query plan     13        0      edb_resource_group
0          0
 3501 enterprise 1     query plan     13        0      edb_resource_group_n
0          0
 3501 enterprise 1     query plan     13        0      edb_resource_group_o
0          0
 3501 enterprise 1     query plan     13        0      pg_attribute
0          0
 3501 enterprise 1     query plan     13        0      pg_attribute_relid_a
0          0
 3501 enterprise 1     query plan     13        0      pg_attribute_relid_a
0          0
 3501 enterprise 1     query plan     13        0      pg_auth_members
0          0
 3501 enterprise 1     query plan     13        0      pg_auth_members_memb
0          0
 3501 enterprise 1     query plan     13        0      pg_auth_members_role
0          0
                                      .
                                      .
                                      .
 3501 enterprise 2     wal flush      149       0      N/A
0          0
 3501 enterprise 2     wal flush      149       0      edb_password_history
0          0
 3501 enterprise 2     wal flush      149       0      edb_password_history
0          0
 3501 enterprise 2     wal flush      149       0      edb_password_history
0          0
 3501 enterprise 2     wal flush      149       0      edb_profile
0          0
 3501 enterprise 2     wal flush      149       0      edb_profile_name_ind
0          0
 3501 enterprise 2     wal flush      149       0      edb_profile_oid_inde
0          0
 3501 enterprise 2     wal flush      149       0      edb_profile_password
0          0
 3501 enterprise 2     wal flush      149       0      edb_resource_group
0          0
 3501 enterprise 2     wal flush      149       0      edb_resource_group_n
0          0
 3501 enterprise 2     wal flush      149       0      edb_resource_group_o
0          0
 3501 enterprise 2     wal flush      149       0      pg_attribute
0          0
 3501 enterprise 2     wal flush      149       0      pg_attribute_relid_a
0          0
 3501 enterprise 2     wal flush      149       0      pg_attribute_relid_a
0          0
 3501 enterprise 2     wal flush      149       0      pg_auth_members
0          0
 3501 enterprise 2     wal flush      149       0      pg_auth_members_memb
0          0
 3501 enterprise 2     wal flush      149       0      pg_auth_members_role
0          0
                                      .
                                      .
                                      .
 3501 enterprise 3     wal write      148       0      N/A
0          0
 3501 enterprise 3     wal write      148       0      edb_password_history
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
0          0
 3501 enterprise 3     wal write      148       0      edb_password_history
0          0
 3501 enterprise 3     wal write      148       0      edb_password_history
0          0
 3501 enterprise 3     wal write      148       0      edb_profile
0          0
 3501 enterprise 3     wal write      148       0      edb_profile_name_ind
0          0
 3501 enterprise 3     wal write      148       0      edb_profile_oid_inde
0          0
 3501 enterprise 3     wal write      148       0      edb_profile_password
0          0
 3501 enterprise 3     wal write      148       0      edb_resource_group
0          0
 3501 enterprise 3     wal write      148       0      edb_resource_group_n
0          0
 3501 enterprise 3     wal write      148       0      edb_resource_group_o
0          0
 3501 enterprise 3     wal write      148       0      pg_attribute
0          0
 3501 enterprise 3     wal write      148       0      pg_attribute_relid_a
0          0
 3501 enterprise 3     wal write      148       0      pg_attribute_relid_a
0          0
 3501 enterprise 3     wal write      148       0      pg_auth_members
0          0
 3501 enterprise 3     wal write      148       0      pg_auth_members_memb
0          0
 3501 enterprise 3     wal write      148       0      pg_auth_members_role
0          0
                                       .
                                       .
                                       .
 3501 enterprise 24    wal write      130       0      pg_toast_1255
0          0
 3501 enterprise 24    wal write      130       0      pg_toast_1255_index
0          0
 3501 enterprise 24    wal write      130       0      pg_toast_2396
0          0
 3501 enterprise 24    wal write      130       0      pg_toast_2396_index
0          0
 3501 enterprise 24    wal write      130       0      pg_toast_2964
0          0
 3501 enterprise 24    wal write      130       0      pg_toast_2964_index
0          0
 3501 enterprise 24    wal write      130       0      pg_toast_3592
0          0
 3501 enterprise 24    wal write      130       0      pg_toast_3592_index
0          0
 3501 enterprise 24    wal write      130       0      pg_type
0          0
 3501 enterprise 24    wal write      130       0      pg_type_oid_index
0          0
 3501 enterprise 24    wal write      130       0      pg_type_typname_nsp_
0          0
(1304 rows)
```

The information displayed in the result set includes:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column Name	Description
`ID`	The system-assigned identifier of the wait.
`USER`	The name of the user that incurred the wait.
`SEQ`	The sequence number of the wait event.
`WAIT NAME`	The name of the wait event.
`ELAPSED`	The length of the wait event in microseconds.
`File`	The relfilenode number of the file.
`Name`	If available, the name of the file name related to the wait event.
`# of Blk`	The block number read or written for a specific instance of the event .
`Sum of Blks`	The number of blocks read.

purgesnap()

The `purgesnap()` function purges a range of snapshots from the snapshot tables. The signature is:

```
purgesnap(<beginning_id>, <ending_id>)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`purgesnap()` removes all snapshots between `beginning_id` and `ending_id` (inclusive):

```
SELECT * FROM purgesnap(6, 9);

           purgesnap
-----------------------------------
 Snapshots in range 6 to 9 deleted.
(1 row)
```

A call to the `get_snaps()` function after executing the example shows that snapshots `6` through `9` have been purged from the snapshot tables:

```
SELECT * FROM get_snaps();
       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
 10 25-JUL-18 09:49:49.855821
 11 25-JUL-18 09:49:54.919954
 12 25-JUL-18 09:49:59.987707
(8 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

truncsnap()

Use the truncsnap() function to delete all records from the snapshot table. The signature is:

```
truncsnap()
```

For example:

```
SELECT * FROM truncsnap();

    truncsnap
----------------------
 Snapshots truncated.
(1 row)
```

A call to the `get_snaps()` function after calling the `truncsnap()` function shows that all records have been removed from the snapshot tables:

```
SELECT * FROM get_snaps();
 get_snaps
-----------
(0 rows)
```

Simulating Statspack AWR Reports

The functions described in this section return information comparable to the information contained in an Oracle Statspack/AWR (Automatic Workload Repository) report. When taking a snapshot, performance data from system catalog tables is saved into history tables. The reporting functions listed below report on the differences between two given snapshots.

- `stat_db_rpt()`
- `stat_tables_rpt()`
- `statio_tables_rpt()`
- `stat_indexes_rpt()`
- `statio_indexes_rpt()`

The reporting functions can be executed 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 additional system information. The signature is:

```
edbreport(<beginning_id>, <ending_id>)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`ending_id` is an integer value that represents the ending session identifier.

The call to the `edbreport()` function returns a composite report that contains system information and the reports returned by the other statspack functions.

```
SELECT * FROM edbreport(9, 10);
                                  edbreport
-------------------------------------------------------------------------------
    EnterpriseDB Report for database acctg            25-JUL-18
 Version: PostgreSQL 11.0 (EnterpriseDB Advanced Server 11.0.2)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
```

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                                          RELPAGES
-----------------------------------------------------
pgbench_accounts                               16394
pgbench_history                                391
pg_proc                                        145
pg_attribute                                   92
pg_depend                                      81
pg_collation                                   60
edb$stat_all_indexes                           46
edb$statio_all_indexes                         46
pg_description                                 44
edb$stat_all_tables                            29
```

The information displayed in the `Top 10 Relations by pages` section includes:

Column Name	Description
`TABLE`	The name of the table.
`RELPAGES`	The number of pages in the table.

```
      Top 10 Indexes by pages

INDEX                                          RELPAGES
-----------------------------------------------------
pgbench_accounts_pkey                          2745
pg_depend_reference_index                      68
pg_depend_depender_index                       63
pg_proc_proname_args_nsp_index                 53
pg_attribute_relid_attnam_index                25
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
pg_description_o_c_o_index                  24
pg_attribute_relid_attnum_index             17
pg_proc_oid_index                           14
pg_collation_name_enc_nsp_index             12
edb$stat_idx_pk                             10
```

The information displayed in the `Top 10 Indexes by pages` section includes:

Column Name	Description
`INDEX`	The name of the index.
`RELPAGES`	The number of pages in the index.

```
        Top 10 Relations by DML

SCHEMA        RELATION                        UPDATES     DELETES    INSERTS
-------------------------------------------------------------------------------
public        pgbench_accounts                117209      0          1000000
public        pgbench_tellers                 117209      0          100
public        pgbench_branches                117209      0          10
public        pgbench_history                 0           0          117209
```

The information displayed in the `Top 10 Relations by DML` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`UPDATES`	The number of UPDATES performed on the table.
`DELETES`	The number of DELETES performed on the table.
`INSERTS`	The number of INSERTS performed on the table.

```
       DATA from pg_stat_database

DATABASE    NUMBACKENDS  XACT COMMIT   XACT ROLLBACK   BLKS READ  BLKS HIT  HIT RATIO
-------------------------------------------------------------------------------------
acctg       0            8261          0               117        127985    99.91
```

The information displayed in the DATA from `pg_stat_database` section of the report includes:

Column Name	Description
`DATABASE`	The name of the database.
`NUMBACKENDS`	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.
`XACT COMMIT`	Number of transactions in this database that have been committed.
`XACT ROLLBACK`	Number of transactions in this database that have been rolled back.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column Name	Description
BLKS READ	Number of disk blocks read in this database.
BLKS HIT	Number of times disk blocks were found already in the buffer cache (when a read was not necessary).
HIT RATIO	The percentage of times that a block was found in the shared buffer cache.

```
     DATA from pg_buffercache

RELATION                            BUFFERS
-----------------------------------------
pgbench_accounts                    16665
pgbench_accounts_pkey               2745
pgbench_history                     751
edb$statio_all_indexes              94
edb$stat_all_indexes                94
edb$stat_all_tables                 60
edb$statio_all_tables               56
edb$session_wait_history            34
edb$statio_idx_pk                   17
pg_depend                           17
```

The information displayed in the `DATA from pg_buffercache` section of the report includes:

Column Name	Description
RELATION	The name of the table.
BUFFERS	The number of shared buffers used by the relation.

> **Note**
>
> In order to obtain the report for `DATA from pg_buffercache`, the `pg_buffercache` module must have been installed in the database.Perform the installation with the `CREATE EXTENSION` command.

For more information on the `CREATE EXTENSION` command, see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/11/static/sql-createextension.html

```
     DATA from pg_stat_all_tables ordered by seq scan

 SCHEMA                RELATION                      SEQ SCAN    REL TUP READ IDX SCAN
IDX TUP READ INS   UPD     DEL
-------------------------------------------------------------------------------------
----------------------------------
 public                pgbench_branches              8258        82580       0
0             0     8258    0
 public                pgbench_tellers               8258        825800      0
0             0     8258    0
 pg_catalog            pg_class                      7           3969        92
80            0     0       0
 pg_catalog            pg_index                      5           950         31
38            0     0       0
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 pg_catalog            pg_namespace                4           144         5
4             0     0       0
 pg_catalog            pg_database                 2           12          7
7             0     0       0
 pg_catalog            pg_am                       1           1           0
0             0     0       0
 pg_catalog            pg_authid                   1           10          2
2             0     0       0
 sys                   callback_queue_table        0           0           0
0             0     0       0
 sys                   edb$session_wait_history    0           0           0
0             125   0       0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by seq scan` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`SEQ SCAN`	The number of sequential scans initiated on this table.
`REL TUP READ`	The number of tuples read in the table.
`IDX SCAN`	The number of index scans initiated on the table.
`IDX TUP READ`	The number of index tuples read.
`INS`	The number of rows inserted.
`UPD`	The number of rows updated.
`DEL`	The number of rows deleted.

```
   DATA from pg_stat_all_tables ordered by rel tup read

 SCHEMA              RELATION                  SEQ SCAN  REL TUP READ IDX SCAN
IDX TUP READ INS  UPD    DEL
-------------------------------------------------------------------------------
-------------------------------------------
 public              pgbench_tellers           8258      825800       0
0             0    8258   0
 public              pgbench_branches          8258      82580        0
0             0    8258   0
 pg_catalog          pg_class                  7         3969         92
80            0    0      0
 pg_catalog          pg_index                  5         950          31
38            0    0      0
 pg_catalog          pg_namespace              4         144          5
4             0    0      0
 pg_catalog          pg_database               2         12           7
7             0    0      0
 pg_catalog          pg_authid                 1         10           2
2             0    0      0
 pg_catalog          pg_am                     1         1            0
0             0    0      0
 sys                 callback_queue_table      0         0            0
0             0    0      0
 sys                 edb$session_wait_history  0         0            0
0             125  0      0
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The information displayed in the `DATA from pg_stat_all_tables ordered by rel tup read` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`SEQ SCAN`	The number of sequential scans performed on the table.
`REL TUP READ`	The number of tuples read from the table.
`IDX SCAN`	The number of index scans performed on the table.
`IDX TUP READ`	The number of index tuples read.
`INS`	The number of rows inserted.
`UPD`	The number of rows updated.
`DEL`	The number of rows deleted.

```
   DATA from pg_statio_all_tables

 SCHEMA               RELATION             HEAP     HEAP     IDX      IDX
TOAST     TOAST     TIDX    TIDX
                                          READ     HIT      READ     HIT
READ      HIT       READ    HIT
-------------------------------------------------------------------------------
--------------------------------
 public               pgbench_accounts     32       25016    0        49913
0         0         0       0
 public               pgbench_tellers      0        24774    0        0
0         0         0       0
 public               pgbench_branches     0        16516    0        0
0         0         0       0
 public               pgbench_history      53       8364     0        0
0         0         0       0
 pg_catalog           pg_class             0        199      0        187
0         0         0       0
 pg_catalog           pg_attribute         0        198      0        395
0         0         0       0
 pg_catalog           pg_proc              0        75       0        153
0         0         0       0
 pg_catalog           pg_index             0        56       0        33
0         0         0       0
 pg_catalog           pg_amop              0        48       0        56
0         0         0       0
 pg_catalog           pg_namespace         0        28       0        7
0         0         0       0
```

The information displayed in the `DATA from pg_statio_all_tables` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`HEAP READ`	The number of heap blocks read.
`HEAP HIT`	The number of heap blocks hit.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column Name	Description
IDX READ	The number of index blocks read.
IDX HIT	The number of index blocks hit.
TOAST READ	The number of toast blocks read.
TOAST HIT	The number of toast blocks hit.
TIDX READ	The number of toast index blocks read.
TIDX HIT	The number of toast index blocks hit.

```
    DATA from pg_stat_all_indexes

 SCHEMA                  RELATION                    INDEX
IDX SCAN     IDX TUP READ IDX TUP FETCH
-------------------------------------------------------------------------------
-----------------------------------------
 public                  pgbench_accounts            pgbench_accounts_pkey
16516        16679         16516
 pg_catalog              pg_attribute
pg_attribute_relid_attnum_index   196               402          402
 pg_catalog              pg_proc                     pg_proc_oid_index
70           70            70
 pg_catalog              pg_class                    pg_class_oid_index
61           61            61
 pg_catalog              pg_class                    pg_class_relname_nsp_index
31 19 19
 pg_catalog              pg_type                     pg_type_oid_index
22           22            22
 pg_catalog             edb_policy                   edb_policy_object_name_index
21           0             0
 pg_catalog             pg_amop                     pg_amop_fam_strat_index
16           16            16
 pg_catalog             pg_index                    pg_index_indexrelid_index
16           16            16
 pg_catalog             pg_index                    pg_index_indrelid_index
15           22            22
```

The information displayed in the `DATA from pg_stat_all_indexes` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the index resides.
RELATION	The name of the table on which the index is defined.
INDEX	The name of the index.
IDX SCAN	The number of indexes scans initiated on this index.
IDX TUP READ	Number of index entries returned by scans on this index
IDX TUP FETCH	Number of live table rows fetched by simple index scans using this index.

```
   DATA from pg_statio_all_indexes

 SCHEMA            RELATION                  INDEX
IDX BLKS READ IDX BLKS HIT
-------------------------------------------------------------------------------
-----------------------------------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 public            pgbench_accounts          pgbench_accounts_pkey
0             49913
 pg_catalog        pg_attribute
pg_attribute_relid_attnum_index    0         395
 sys               edb$stat_all_indexes      edb$stat_idx_pk
1             382
 sys               edb$statio_all_indexes    edb$statio_idx_pk
1             382
 sys               edb$statio_all_tables     edb$statio_tab_pk
2             262
 sys               edb$stat_all_tables       edb$stat_tab_pk
0             259
 sys               edb$session_wait_history  session_waits_hist_pk
0             251
 pg_catalog        pg_proc                   pg_proc_oid_index
0             142
 pg_catalog        pg_class                  pg_class_oid_index
0             123
 pg_catalog        pg_class                  pg_class_relname_nsp_index
0             63
```

The information displayed in the `DATA from pg_statio_all_indexes` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the index resides.
`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

WAIT NAME                                COUNT        WAIT TIME        % WAIT
-------------------------------------------------------------------------------
wal flush                                8359         1.357593          30.62
wal write                                8358         1.349153          30.43
wal file sync                            8358         1.286437          29.02
query plan                               33439        0.439324          9.91
db file extend                           54           0.000585          0.01
db file read                             31           0.000307          0.01
other lwlock acquire                     0            0.000000          0.00
ProcArrayLock                            0            0.000000          0.00
CLogControlLock                          0            0.000000          0.00
```

The information displayed in the `System Wait Information` section includes:

Column Name	Description
`WAIT NAME`	The name of the wait.
`COUNT`	The number of times that the wait event occurred.
`WAIT TIME`	The length of the wait time in seconds.
`% WAIT`	The percentage of the total wait time used by this wait for this session.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    Database Parameters from postgresql.conf

 PARAMETER                              SETTING
CONTEXT       MINVAL         MAXVAL
------------------------------------------------------------------------
-------------------------------------------------
 allow_system_table_mods                off
postmaster
 application_name                       psql.bin
user
 archive_command                        (disabled)
sighup
 archive_mode                           off
postmaster
 archive_timeout                        0
sighup      0               1073741823
 array_nulls                            on
user
 authentication_timeout                 60
sighup       1              600
 autovacuum                             on
sighup
 autovacuum_analyze_scale_factor        0.1
sighup       0              100
 autovacuum_analyze_threshold           50
sighup       0              2147483647
 autovacuum_freeze_max_age              200000000
postmaster   100000         2000000000
 autovacuum_max_workers                 3
postmaster   1              262143
 autovacuum_multixact_freeze_max_age    400000000
postmaster   10000          2000000000
 autovacuum_naptime                     60
sighup       1            2147483
 autovacuum_vacuum_cost_delay           20
sighup      -1              100
                        .
                        .
                        .
```

The information displayed in the `Database Parameters from postgresql.conf` section includes:

Column Name	Description
`PARAMETER`	The name of the parameter.
`SETTING`	The current value assigned to the parameter.
`CONTEXT`	The context required to set the parameter value.
`MINVAL`	The minimum value allowed for the parameter.
`MAXVAL`	The maximum value allowed for the parameter.

stat_db_rpt()

The signature is:

```
stat_db_rpt(<beginning_id>, <ending_id>)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

The following example demonstrates the `stat_db_rpt()` function:

```
SELECT * FROM stat_db_rpt(9, 10);

                                  stat_db_rpt
--------------------------------------------------------------------------------
   DATA from pg_stat_database

DATABASE  NUMBACKENDS   XACT COMMIT  XACT ROLLBACK  BLKS READ  BLKS HIT HIT RATIO
--------------------------------------------------------------------------------
acctg     0             8261         0              117        127985   99.91
(5 rows)
```

The information displayed in the DATA from `pg_stat_database` section of the report includes:

Column Name	Description
`DATABASE`	The name of the database.
`NUMBACKENDS`	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.
`XACT COMMIT`	The number of transactions in this database that have been committed.
`XACT ROLLBACK`	The number of transactions in this database that have been 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, or `sys`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should 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');

                        stat_tables_rpt
------------------------------------------------------------------------------
   DATA from pg_stat_all_tables ordered by seq scan

 SCHEMA                RELATION                       SEQ SCAN   REL TUP READ IDX SCAN
IDX TUP READ INS   UPD    DEL
-------------------------------------------------------------------------------------------
-----------------------------------
 public                pgbench_branches               8249       82490       0
0            0    8249    0
 public                pgbench_tellers                8249       824900      0
0            0    8249    0
 pg_catalog            pg_class                       7          3969        92
80           0    0       0
 pg_catalog            pg_index                       5          950         31
38           0    0       0
 pg_catalog            pg_namespace                   4          144         5
4            0    0       0
 pg_catalog            pg_am                          1          1           0
0            0    0       0
 pg_catalog            pg_authid                      1          10          2
2            0    0       0
 pg_catalog            pg_database                    1          6           3
3            0    0       0
 sys                   callback_queue_table           0          0           0
0            0    0       0
 sys                  edb$session_wait_history        0          0           0
0            125 0        0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by seq scan` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column Name	Description
RELATION	The name of the table.
SEQ SCAN	The number of sequential scans on the table.
REL TUP READ	The number of tuples read from the table.
IDX SCAN	The number of index scans performed on the table.
IDX TUP READ	The number of index tuples read from the table.
INS	The number of rows inserted.
UPD	The number of rows updated.
DEL	The number of rows deleted.

The second portion of the report contains:

```
   DATA from pg_stat_all_tables ordered by rel tup read

 SCHEMA                RELATION                 SEQ SCAN REL   TUP READ   IDX SCAN
IDX TUP READ INS    UPD     DEL
-------------------------------------------------------------------------------------
--------------------------------
 public                pgbench_tellers          8249          824900      0
0            0     8249     0
 public                pgbench_branches         8249          82490       0
0            0     8249     0
 pg_catalog            pg_class                 7             3969        92
80           0     0        0
 pg_catalog            pg_index                 5             950         31
38           0     0        0
 pg_catalog            pg_namespace             4             144         5
4            0     0        0
 pg_catalog            pg_authid                1             10          2
2            0     0        0
pg_catalog            pg_database              1             6           3
3            0     0        0
 pg_catalog            pg_am                    1             1           0
0            0     0        0
 sys                   callback_queue_table     0             0           0
0            0     0        0
 sys                   edb$session_wait_history 0             0           0
0            125   0        0
(29 rows)
```

The information displayed in the `DATA from pg_stat_all_tables ordered by rel tup read` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
SEQ SCAN	The number of sequential scans performed on the table.
REL TUP READ	The number of tuples read from the table.
IDX SCAN	The number of index scans performed on the table.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column Name	Description
`IDX TUP READ`	The number of live rows fetched by index scans.
`INS`	The number of rows inserted.
`UPD`	The number of rows updated.
`DEL`	The number of rows deleted.

statio_tables_rpt()

The signature is:

```
statio_tables_rpt(<beginning_id>, <ending_id>, <top_n>, <scope>)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, or `sys`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should return information about all tables.

The `statio_tables_rpt()` function returns a report that contains:

```
SELECT * FROM statio_tables_rpt(9, 10, 10, 'SYS');

                                              statio_tables_rpt
--------------------------------------------------------------------------------
------------------------------
    DATA from pg_statio_all_tables

 SCHEMA                  RELATION             HEAP     HEAP    IDX     IDX   TOAST
TOAST      TIDX      TIDX
                                             READ     HIT     READ    HIT   READ

HIT        READ      HIT
--------------------------------------------------------------------------------
----------------------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 sys                    edb$stat_all_indexes 8        18      1       382    0
0          0         0
 sys                    edb$statio_all_index 8        18      1       382    0
0          0         0
 sys                    edb$statio_all_table 5        12      2       262    0
0          0         0
 sys                    edb$stat_all_tables  4        10      0       259    0
0          0         0
 sys                    edb$session_wait_his 2        6       0       251    0
0          0         0
 sys                    edb$session_waits    1        4       0       12     0
0          0         0
 sys                    callback_queue_table 0        0       0       0      0
0          0         0
 sys                    dual                 0        0       0       0      0
0          0         0
 sys                   edb$snap              0        1       0       2      0
0          0         0
 sys                   edb$stat_database     0        2       0       7      0
0          0         0
(15 rows)
```

The information displayed in the `DATA from pg_statio_all_tables` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the relation.
`HEAP READ`	The number of heap blocks read.
`HEAP HIT`	The number of heap blocks hit.
`IDX READ`	The number of index blocks read.
`IDX HIT`	The number of index blocks hit.
`TOAST READ`	The number of toast blocks read.
`TOAST HIT`	The number of toast blocks hit.
`TIDX READ`	The number of toast index blocks read.
`TIDX HIT`	The number of toast index blocks hit.

stat_indexes_rpt()

The signature is:

```
stat_indexes_rpt(<beginning_id>, <ending_id>, <top_n>, <scope>)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog, information_schema`, or `sys`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should 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

 SCHEMA                    RELATION                  INDEX
IDX SCAN       IDX TUP READ IDX TUP FETCH
--------------------------------------------------------------------------------
-------------------------------------------
 public                    pgbench_accounts          pgbench_accounts_pkey
16516          16679          16516
 pg_catalog                pg_attribute
pg_attribute_relid_attnum_index      196      402      402
 pg_catalog                pg_proc                   pg_proc_oid_index
70             70             70
 pg_catalog                pg_class                  pg_class_oid_index
61             61             61
 pg_catalog                pg_class                  pg_class_relname_nsp_index
31             19             19
 pg_catalog                pg_type                   pg_type_oid_index
22             22             22
 pg_catalog                edb_policy                edb_policy_object_name_index
21             0              0
 pg_catalog                pg_amop                   pg_amop_fam_strat_index
16             16             16
 pg_catalog                pg_index                  pg_index_indexrelid_index
16             16             16
 pg_catalog                pg_index                  pg_index_indrelid_index
15             22             22
(14 rows)
```

The information displayed in the `DATA from pg_stat_all_indexes` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the relation.
`INDEX`	The name of the index.
`IDX SCAN`	The number of indexes scanned.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column Name	Description
IDX TUP READ	The number of index tuples read.
IDX TUP FETCH	The number of index tuples fetched.

statio_indexes_rpt()

The signature is:

```
statio_indexes_rpt(<beginning_id>, <ending_id>, <top_n>, <scope>)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, or `sys`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should return information about all tables.

The `statio_indexes_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM statio_indexes_rpt(9, 10, 10, 'SYS');

                          statio_indexes_rpt
------------------------------------------------------------------------------
-----------------------------------
  DATA from pg_statio_all_indexes

 SCHEMA                       RELATION                       INDEX
IDX BLKS READ         IDX BLKS HIT
------------------------------------------------------------------------------
-----------------------------------
 pg_catalog                 pg_attribute
pg_attribute_relid_attnum_index          0                     395
 sys                        edb$stat_all_indexes           edb$stat_idx_pk
1                    382
 sys                        edb$statio_all_indexes         edb$statio_idx_pk
1                    382
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 sys                       edb$statio_all_tables          edb$statio_tab_pk
2                    262
 sys                       edb$stat_all_tables            edb$stat_tab_pk
0                    259
 sys                       edb$session_wait_history       session_waits_hist_pk
0                    251
 pg_catalog                pg_proc                        pg_proc_oid_index
0                    142
 pg_catalog                pg_class                       pg_class_oid_index
0                    123
 pg_catalog                pg_class                       pg_class_relname_nsp_index
0                    63
 pg_catalog                pg_type                        pg_type_oid_index
0                    45
(14 rows)
```

The information displayed in the `DATA from pg_statio_all_indexes` report includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`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.

Performance Tuning Recommendations

To use DRITA reports for performance tuning, review the top five events in a given report, looking for any event that takes a disproportionately large percentage of resources. In a streamlined system, user I/O will probably make up the largest number of waits. Waits should be evaluated in the context of CPU usage and total time; an event may not be significant if it takes 2 minutes out of a total measurement interval of 2 hours, if the rest of the time is consumed by CPU time. The component of response time (CPU "work" time or other "wait" time) that consumes the highest percentage of overall time should be evaluated.

When evaluating events, watch for:

Event type	Description
Checkpoint waits	Checkpoint waits may indicate that checkpoint parameters need to be adjusted, `checkpoint_segments` and `checkpoint_timeout`).
WAL-related waits	WAL-related waits may indicate `wal_buffers` are under-sized.
SQL Parse waits	If the number of waits is high, try to use prepared statements.
db file random reads	If high, check that appropriate indexes and statistics exist.
db file random writes	If high, may need to decrease `bgwriter_delay`.
btree random lock acquires	May indicate indexes are being rebuilt. Schedule index builds during less active time.

Performance reviews should also include careful scrutiny of the hardware, the operating system, the network and the application SQL statements.

Event Descriptions

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following table lists the basic wait events that are displayed by DRITA.

Event Name	Description
`add in shmem lock acquire`	Obsolete/unused
`bgwriter communication lock acquire`	The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.
`btree vacuum lock acquire`	The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.
`buffer free list lock acquire`	The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory).
`checkpoint lock acquire`	A server process has waited for the short-term lock that prevents simultaneous checkpoints.
`checkpoint start lock acquire`	The server has waited for the short-term lock that synchronizes access to the bgwriter checkpoint schedule.
`clog control lock acquire`	The server has waited for the short-term lock that synchronizes access to the commit log.
`control file lock acquire`	The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number).
`db file extend`	A server process has waited for the operating system while adding a new page to the end of a file.
`db file read`	A server process has waited for the completion of a read (from disk).
`db file write`	A server process has waited for the completion of a write (to disk).
`db file sync`	A server process has waited for the operating system to flush all changes to disk.
`first buf mapping lock acquire`	The server has waited for a short-term lock that synchronizes access to the shared-buffer mapping table.
`freespace lock acquire`	The server has waited for the short-term lock that synchronizes access to the freespace map.
`lwlock acquire`	The server has waited for a short-term lock that has not been described elsewhere in this section.
`multi xact gen lock acquire`	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).
`multi xact member lock acquire`	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).
`multi xact offset lock acquire`	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).
`oid gen lock acquire`	The server has waited for the short-term lock that synchronizes access to the next available OID (object ID).
`query plan`	The server has computed the execution plan for a SQL statement.
`rel cache init lock acquire`	The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.
`shmem index lock acquire`	The server has waited for the short-term lock that synchronizes access to the shared-memory map.
`sinval lock acquire`	The server has waited for the short-term lock that synchronizes access to the cache invalidation state.
`sql parse`	The server has parsed a SQL statement.
`subtrans control lock acquire`	The server has waited for the short-term lock that synchronizes access to the subtransaction log.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Event Name	Description
`tablespace create lock acquire`	The server has waited for the short-term lock that prevents simultaneous `CREATE TABLESPACE` or `DROP TABLESPACE` commands.
`two phase state lock acquire`	The server has waited for the short-term lock that synchronizes access to the list of prepared transactions.
`wal insert lock acquire`	The server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small.
`wal write lock acquire`	The server has waited for the short-term lock that synchronizes write-ahead log flushes.
`wal file sync`	The server has waited for the write-ahead log to sync to disk (related to the wal_sync_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open_sync).
`wal flush`	The server has waited for the write-ahead log to flush to disk.
`wal write`	The server has waited for a write to the write-ahead log buffer (expect this value to be high).
`xid gen lock acquire`	The server has waited for the short-term lock that synchronizes access to the next available transaction ID.

When wait events occur for lightweight locks, they are displayed by DRITA as well. A *lightweight lock* is used 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*. Individual lightweight lock tranches are not displayed by DRITA, but their summation is displayed by a single event named `other lwlock acquire`.

For a list and description of lightweight locks displayed by DRITA, see Section 28.2, `The Statistics Collector` in the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/monitoring-stats.html

Under Section 28.2.2. `Viewing Statistics`, the lightweight locks are listed in Table 28-4 `wait_event` Description where the `Wait Event Type` column designates `LWLock`.

The following example displays lightweight locks `ProcArrayLock`, `CLogControlLock`, `WALBufMappingLock`, and `XidGenLock`.

```
postgres=# select * from sys_rpt(40,70,20);
                                  sys_rpt
-------------------------------------------------------------------------
 WAIT NAME                          COUNT       WAIT TIME        % WAIT
-------------------------------------------------------------------------
 wal flush                          56107       44.456494       47.65
 db file read                       66123       19.543968       20.95
 wal write                          32886       12.780866       13.70
 wal file sync                      32933       11.792972       12.64
 query plan                         223576      4.539186        4.87
 db file extend                     2339        0.087038        0.09
 other lwlock acquire               402         0.066591        0.07
 ProcArrayLock                      135         0.012942        0.01
 CLogControlLock                    212         0.010333        0.01
 WALBufMappingLock                  47          0.006068        0.01
 XidGenLock                         53          0.005296        0.01
(13 rows)
```

DRITA also displays wait events that occur that are related to certain Advanced Server product features.

These Advanced Server feature specific wait events and the `other lwlock acquire` event are listed in the following table.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Event Name	Description
`BulkLoadLock`	The server has waited for access related to EDB*Loader.
`EDBResoureManagerLock`	The server has waited for access related to EDB Resource Manager.
`other lwlock acquire`	Summation of waits for lightweight lock tranches.

13 Database Compatibility for Oracle Developers Catalog Views Guide

The Oracle Catalog Views provide information about database objects in a manner compatible with the Oracle data dictionary views.

For detailed information about Advanced Server's compatibility features and extended functionality, see the complete library of Advanced Server documentation, available at:

https://www.enterprisedb.com/docs

13.1 ALL_ALL_TABLES

The `ALL_ALL_TABLES` view provides information about the tables accessible by the current user.

Name	Type	Description
`owner`	`TEXT`	User name of the table's owner.
`schema_name`	`TEXT`	Name of the schema in which the table belongs.
`table_name`	`TEXT`	The name of the table.
`tablespace_name`	`TEXT`	Name of the tablespace in which the table resides if other than the default tablespace.
`status`	`CHARACTER VARYING (5)`	Included for compatibility only; always set to `VALID`.
`temporary`	`TEXT`	`Y` if the table is temporary; `N` if the table is permanent.

13.2 ALL_CONS_COLUMNS

The `ALL_CONS_COLUMNS` view provides information about the columns specified in constraints placed on tables accessible by the current user.

Name	Type	Description
`owner`	`TEXT`	User name of the constraint's owner.
`schema_name`	`TEXT`	Name of the schema in which the constraint belongs.
`constraint_name`	`TEXT`	The name of the constraint.
`table_name`	`TEXT`	The name of the table to which the constraint belongs.
`column_name`	`TEXT`	The name of the column referenced in the constraint.
`position`	`SMALLINT`	The position of the column within the object definition.
`constraint_def`	`TEXT`	The definition of the constraint.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.3 ALL_CONSTRAINTS

The `ALL_CONSTRAINTS` view provides information about the constraints placed on tables accessible by the current user.

Name	Type	Description
`owner`	`TEXT`	User name of the constraint's owner.
`schema_name`	`TEXT`	Name of the schema in which the constraint belongs.
`constraint_name`	`TEXT`	The name of the constraint.
`constraint_type`	`TEXT`	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
`table_name`	`TEXT`	Name of the table to which the constraint belongs.
`search_condition`	`TEXT`	Search condition that applies to a check constraint.
`r_owner`	`TEXT`	Owner of a table referenced by a referential constraint.
`r_constraint_name`	`TEXT`	Name of the constraint definition for a referenced table.
`delete_rule`	`TEXT`	The delete rule for a referential constraint. Possible values are: `C` – cascade; `R` – restrict; `N` – no action
`deferrable`	`BOOLEAN`	Specified if the constraint is deferrable (`T` or `F`).
`deferred`	`BOOLEAN`	Specifies if the constraint has been deferred (`T` or `F`).
`index_owner`	`TEXT`	User name of the index owner.
`index_name`	`TEXT`	The name of the index.
`constraint_def`	`TEXT`	The definition of the constraint.

13.4 ALL_DB_LINKS

The `ALL_DB_LINKS` view provides information about the database links accessible by the current user.

Name	Type	Description
`owner`	`TEXT`	User name of the database link's owner.
`db_link`	`TEXT`	The name of the database link.
`type`	`CHARACTER VARYING`	Type of remote server. Value will be either `REDWOOD` or `EDB` .

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
username	TEXT	User name of the user logging in.
host	TEXT	Name or IP address of the remote server.

13.5 ALL_DIRECTORIES

The ALL_DIRECTORIES view provides information about all directories created with the CREATE DIRECTORY command.

Name	Type	Description
owner	CHARACTER VARYING(30)	User name of the directory's owner.
directory_name	CHARACTER VARYING(30)	The alias name assigned to the directory.
directory_path	CHARACTER VARYING(4000)	The path to the directory.

13.6 ALL_IND_COLUMNS

The ALL_IND_COLUMNS view provides information about columns included in indexes on the tables accessible by the current user.

Name	Type	Description
index_owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
table_owner	TEXT	User name of the table owner.
table_name	TEXT	The name of the table to which the index belongs.
column_name	TEXT	The name of the column.
column_position	SMALLINT	The position of the column within the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

13.7 ALL_INDEXES

The ALL_INDEXES view provides information about the indexes on tables that may be accessed by the current user.

Name	Type	Description
owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
index_type	TEXT	The index type is always BTREE. Included for compatibility only.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Always set to N (not compressed). Included for compatibility only.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Always set to LOGGING. Included for compatibility only.
status	TEXT	Included for compatibility only; always set to VALID.
partitioned	CHARACTER(3)	Indicates that the index is partitioned. Currently, always set to NO.
temporary	CHARACTER(1)	Indicates that an index is on a temporary table. Always set to N; included for compatibility only.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

13.8 ALL_JOBS

The ALL_JOBS view provides information about all jobs that reside in the database.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date.
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

13.9 ALL_OBJECTS

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `ALL_OBJECTS` view provides information about all objects that reside in the database.

Name	Type	Description
`owner`	`TEXT`	User name of the object's owner.
`schema_name`	`TEXT`	Name of the schema in which the object belongs.
`object_name`	`TEXT`	Name of the object.
`object_type`	`TEXT`	Type of the object – possible values are: `INDEX`, `FUNCTION`, `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, `SEQUENCE`, `SYNONYM`, `TABLE`, `TRIGGER`, and `VIEW`
`status`	`CHARACTER VARYING`	Whether or not the state of the object is valid. Currently, Included for compatibility only; always set to `VALID`.
`temporary`	`TEXT`	`Y` if a temporary object; `N` if this is a permanent object.

13.10 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.

Name	Type	Description
`owner`	`TEXT`	The owner of the table.
`schema_name`	`TEXT`	The name of the schema in which the table resides.
`name`	`TEXT`	The name of the table in which the column resides.
`object_type`	`CHARACTER(5)`	For compatibility only; always `TABLE`.
`column_name`	`TEXT`	The name of the column on which the key is defined.
`column_position`	`INTEGER`	`1` for the first column; `2` for the second column, etc.

13.11 ALL_PART_TABLES

The `ALL_PART_TABLES` view provides information about all of the partitioned tables that reside in the database.

Name	Type	Description
`owner`	`TEXT`	The owner of the partitioned table.
`schema_name`	`TEXT`	The name of the schema in which the table resides.
`table_name`	`TEXT`	The name of the table.
`partitioning_type`	`TEXT`	The partitioning type used to define table partitions.
`subpartitioning_type`	`TEXT`	The subpartitioning type used to define table subpartitions.
`partition_count`	`BIGINT`	The number of partitions in the table.
`def_subpartition_count`	`INTEGER`	The number of subpartitions in the table.
`partitioning_key_count`	`INTEGER`	The number of partitioning keys specified.
`subpartitioning_key_count`	`INTEGER`	The number of subpartitioning keys specified.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_next_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE.
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT.
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL.

13.12 ALL_POLICIES

The ALL_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

Name	Type	Description
object_owner	TEXT	Name of the owner of the object.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object on which the policy applies.
policy_group	TEXT	Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Included for compatibility only; always set to NO.
policy_type	TEXT	Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

13.13 ALL_QUEUES

The ALL_QUEUES view provides information about any currently defined queues.

Name	Type	Description
owner	TEXT	User name of the queue owner.
name	TEXT	The name of the queue.
queue_table	TEXT	The name of the queue table in which the queue resides.
qid	OID	The system-assigned object ID of the queue.
queue_type	CHARACTER VARYING	The queue type; may be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.
max_retries	NUMERIC	The maximum number of dequeue attempts.
retrydelay	NUMERIC	The maximum time allowed between retries.
enqueue_enabled	CHARACTER VARYING	YES if the queue allows enqueuing; NO if the queue does not.
dequeue_enabled	CHARACTER VARYING	YES if the queue allows dequeuing; NO if the queue does not.
retention	CHARACTER VARYING	The number of seconds that a processed message is retained in the queue.
user_comment	CHARACTER VARYING	A user-specified comment.
network_name	CHARACTER VARYING	The name of the network on which the queue resides.
sharded	CHARACTER VARYING	YES if the queue resides on a sharded network; NO if the queue does not.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.14 ALL_QUEUE_TABLES

The `ALL_QUEUE_TABLES` view provides information about all of the queue tables in the database.

Name	Type	Description
`owner`	`TEXT`	Role name of the owner of the queue table.
`queue_table`	`TEXT`	The user-specified name of the queue table.
`type`	`CHARACTER VARYING`	The type of data stored in the queue table.
`object_type`	`TEXT`	The user-defined payload type.
`sort_order`	`CHARACTER VARYING`	The order in which the queue table is sorted.
`recipients`	`CHARACTER VARYING`	Always `SINGLE`.
`message_grouping`	`CHARACTER VARYING`	Always `NONE`.
`compatible`	`CHARACTER VARYING`	The release number of the Advanced Server release with which this queue table is compatible.
`primary_instance`	`NUMERIC`	Always `0`.
`secondary_instance`	`NUMERIC`	Always `0`.
`owner_instance`	`NUMERIC`	The instance number of the instance that owns the queue table.
`user_comment`	`CHARACTER VARYING`	The user comment provided when the table was created.
`secure`	`CHARACTER VARYING`	`YES` indicates that the queue table is secure; `NO` indicates that it is not.

13.15 ALL_SEQUENCES

The `ALL_SEQUENCES` view provides information about all user-defined sequences on which the user has `SELECT`, or `UPDATE` privileges.

Name	Type	Description
`sequence_owner`	`TEXT`	User name of the sequence's owner.
`schema_name`	`TEXT`	Name of the schema in which the sequence resides.
`sequence_name`	`TEXT`	Name of the sequence.
`min_value`	`NUMERIC`	The lowest value that the server will assign to the sequence.
`max_value`	`NUMERIC`	The highest value that the server will assign to the sequence.
`increment_by`	`NUMERIC`	The value added to the current sequence number to create the next sequent number.
`cycle_flag`	`CHARACTER VARYING`	Specifies if the sequence should wrap when it reaches `min_value` or `max_value`.
`order_flag`	`CHARACTER VARYING`	Will always return `Y`.
`cache_size`	`NUMERIC`	The number of pre-allocated sequence numbers stored in memory.
`last_number`	`NUMERIC`	The value of the last sequence number saved to disk.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.16 ALL_SOURCE

The `ALL_SOURCE` view provides a source code listing of the following program types: functions, procedures, triggers, package specifications, and package bodies.

Name	Type	Description
`owner`	`TEXT`	User name of the program's owner.
`schema_name`	`TEXT`	Name of the schema in which the program belongs.
`name`	`TEXT`	Name of the program.
`type`	`TEXT`	Type of program – possible values are: `FUNCTION`, `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, and `TRIGGER`
`line`	`INTEGER`	Source code line number relative to a given program.
`text`	`TEXT`	Line of source code text.

13.17 ALL_SUBPART_KEY_COLUMNS

The `ALL_SUBPART_KEY_COLUMNS` view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

Name	Type	Description
`owner`	`TEXT`	The owner of the table.
`schema_name`	`TEXT`	The name of the schema in which the table resides.
`name`	`TEXT`	The name of the table in which the column resides.
`object_type`	`CHARACTER(5)`	For compatibility only; always `TABLE`.
`column_name`	`TEXT`	The name of the column on which the key is defined.
`column_position`	`INTEGER`	`1` for the first column; `2` for the second column, etc.

13.18 ALL_SYNONYMS

The `ALL_SYNONYMS` view provides information on all synonyms that may be referenced by the current user.

Name	Type	Description
`owner`	`TEXT`	User name of the synonym's owner.
`schema_name`	`TEXT`	The name of the schema in which the synonym resides.
`synonym_name`	`TEXT`	Name of the synonym.
`table_owner`	`TEXT`	User name of the object's owner.
`table_schema_name`	`TEXT`	The name of the schema in which the table resides.
`table_name`	`TEXT`	The name of the object that the synonym refers to.
`db_link`	`TEXT`	The name of any associated database link.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.19 ALL_TAB_COLUMNS

The `ALL_TAB_COLUMNS` view provides information on all columns in all user-defined tables and views.

Name	Type	Description
`owner`	`CHARACTER VARYING`	User name of the owner of the table or view in which the column resides.
`schema_name`	`CHARACTER VARYING`	Name of the schema in which the table or view resides.
`table_name`	`CHARACTER VARYING`	Name of the table or view.
`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 within the table or view.
`data_default`	`CHARACTER VARYING`	Default value assigned to the column.

13.20 ALL_TAB_PARTITIONS

The `ALL_TAB_PARTITIONS` view provides information about all of the partitions that reside in the database.

Name	Type	Description
`table_owner`	`TEXT`	The owner of the table in which the partition resides.
`schema_name`	`TEXT`	The name of the schema in which the table resides.
`table_name`	`TEXT`	The name of the table.
`composite`	`TEXT`	`YES` if the table is subpartitioned; `NO` if the table is not subpartitioned.
`partition_name`	`TEXT`	The name of the partition.
`subpartition_count`	`BIGINT`	The number of subpartitions in the partition.
`high_value`	`TEXT`	The high partitioning value specified in the `CREATE TABLE` statement.
`high_value_length`	`INTEGER`	The length of high partitioning value.
`partition_position`	`INTEGER`	Included for compatibility only; always `NULL`.
`tablespace_name`	`TEXT`	The name of the tablespace in which the partition resides.
`pct_free`	`NUMERIC`	Included for compatibility only; always `0`.
`pct_used`	`NUMERIC`	Included for compatibility only; always `0`.
`ini_trans`	`NUMERIC`	Included for compatibility only; always `0`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
max_trans	NUMERIC	Included for compatibility only; always 0.
initial_extent	NUMERIC	Included for compatibility only; always NULL.
next_extent	NUMERIC	Included for compatibility only; always NULL.
min_extent	NUMERIC	Included for compatibility only; always 0.
max_extent	NUMERIC	Included for compatibility only; always 0.
pct_increase	NUMERIC	Included for compatibility only; always 0.
freelists	NUMERIC	Included for compatibility only; always NULL.
freelist_groups	NUMERIC	Included for compatibility only; always NULL.
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES.
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE.
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL.
avg_space	NUMERIC	Included for compatibility only; always NULL.
chain_cnt	NUMERIC	Included for compatibility only; always NULL.
avg_row_len	NUMERIC	Included for compatibility only; always NULL.
sample_size	NUMERIC	Included for compatibility only; always NULL.
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL.
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL.
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES.
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO.
backing_table	REGCLASS	Name of the partition backing table.

13.21 ALL_TAB_SUBPARTITIONS

The ALL_TAB_SUBPARTITIONS view provides information about all of the subpartitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the subpartition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	Included for compatibility only; always NULL.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0.
pct_used	NUMERIC	Included for compatibility only; always 0.
ini_trans	NUMERIC	Included for compatibility only; always 0.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
max_trans	NUMERIC	Included for compatibility only; always 0.
initial_extent	NUMERIC	Included for compatibility only; always NULL.
next_extent	NUMERIC	Included for compatibility only; always NULL.
min_extent	NUMERIC	Included for compatibility only; always 0.
max_extent	NUMERIC	Included for compatibility only; always 0.
pct_increase	NUMERIC	Included for compatibility only; always 0.
freelists	NUMERIC	Included for compatibility only; always NULL.
freelist_groups	NUMERIC	Included for compatibility only; always NULL.
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES.
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE.
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL.
avg_space	NUMERIC	Included for compatibility only; always NULL.
chain_cnt	NUMERIC	Included for compatibility only; always NULL.
avg_row_len	NUMERIC	Included for compatibility only; always NULL.
sample_size	NUMERIC	Included for compatibility only; always NULL.
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL.
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL.
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES.
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO.
backing_table	REGCLASS	Name of the subpartition backing table.

13.22 ALL_TABLES

The ALL_TABLES view provides information on all user-defined tables.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Whether or not the state of the table is valid. Currently, Included for compatibility only; always set to VALID.
temporary	CHARACTER(1)	Y if this is a temporary table; N if this is not a temporary table.

13.23 ALL_TRIGGERS

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `ALL_TRIGGERS` view provides information about the triggers on tables that may be accessed by the current user.

Name	Type	Description
`owner`	`TEXT`	User name of the trigger's owner.
`schema_name`	`TEXT`	The name of the schema in which the trigger resides.
`trigger_name`	`TEXT`	The name of the trigger.
`trigger_type`	`TEXT`	The type of the trigger. Possible values are: `BEFORE ROW`, `BEFORE STATEMENT`, `AFTER ROW`, `AFTER STATEMENT`
`triggering_event`	`TEXT`	The event that fires the trigger.
`table_owner`	`TEXT`	The user name of the owner of the table on which the trigger is defined.
`base_object_type`	`TEXT`	Included for compatibility only. Value will always be `TABLE`.
`table_name`	`TEXT`	The name of the table on which the trigger is defined.
`referencing_name`	`TEXT`	Included for compatibility only. Value will always be `REFERENCING NEW AS NEW OLD AS OLD`.
`status`	`TEXT`	Status indicates if the trigger is enabled `(VALID)` or disabled `(NOTVALID)`.
`description`	`TEXT`	Included for compatibility only.
`trigger_body`	`TEXT`	The body of the trigger.
`action_statement`	`TEXT`	The SQL command that executes when the trigger fires.

13.24 ALL_TYPES

The `ALL_TYPES` view provides information about the object types available to the current user.

Name	Type	Description
`owner`	`TEXT`	The owner of the object type.
`schema_name`	`TEXT`	The name of the schema in which the type is defined.
`type_name`	`TEXT`	The name of the type.
`type_oid`	`OID`	The object identifier (OID) of the type.
`typecode`	`TEXT`	The typecode of the type. Possible values are: `OBJECT`, `COLLECTION`, `OTHER`
`attributes`	`INTEGER`	The number of attributes in the type.

13.25 ALL_USERS

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `ALL_USERS` view provides information on all user names.

Name	Type	Description
`username`	`TEXT`	Name of the user.
`user_id`	`OID`	Numeric user id assigned to the user.
`created`	`TIMESTAMP WITHOUT TIME ZONE`	Included for compatibility only; always `NULL`.

13.26 ALL_VIEW_COLUMNS

The `ALL_VIEW_COLUMNS` view provides information on all columns in all user-defined views.

Name	Type	Description
`owner`	`CHARACTER VARYING`	User name of the view's owner.
`schema_name`	`CHARACTER VARYING`	Name of the schema in which the view belongs.
`view_name`	`CHARACTER VARYING`	Name of the view.
`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 within the view.
`data_default`	`CHARACTER VARYING`	Default value assigned to the column.

13.27 ALL_VIEWS

The `ALL_VIEWS` view provides information about all user-defined views.

Name	Type	Description
`owner`	`TEXT`	User name of the view's owner.
`schema_name`	`TEXT`	Name of the schema in which the view belongs.
`view_name`	`TEXT`	Name of the view.
`text`	`TEXT`	The `SELECT` statement that defines the view.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.28 DBA_ALL_TABLES

The `DBA_ALL_TABLES` view provides information about all tables in the database.

Name	Type	Description
`owner`	`TEXT`	User name of the table's owner.
`schema_name`	`TEXT`	Name of the schema in which the table belongs.
`table_name`	`TEXT`	Name of the table.
`tablespace_name`	`TEXT`	Name of the tablespace in which the table resides if other than the default tablespace.
`status`	`CHARACTER VARYING(5)`	Included for compatibility only; always set to `VALID`.
`temporary`	`TEXT`	`Y` if the table is temporary; `N` if the table is permanent.

13.29 DBA_CONS_COLUMNS

The `DBA_CONS_COLUMNS` view provides information about all columns that are included in constraints that are specified in on all tables in the database.

Name	Type	Description
`owner`	`TEXT`	User name of the constraint's owner.
`schema_name`	`TEXT`	Name of the schema in which the constraint belongs.
`constraint_name`	`TEXT`	The name of the constraint.
`table_name`	`TEXT`	The name of the table to which the constraint belongs.
`column_name`	`TEXT`	The name of the column referenced in the constraint.
`position`	`SMALLINT`	The position of the column within the object definition.
`constraint_def`	`TEXT`	The definition of the constraint.

13.30 DBA_CONSTRAINTS

The `DBA_CONSTRAINTS` view provides information about all constraints on tables in the database.

Name	Type	Description
`owner`	`TEXT`	User name of the constraint's owner.
`schema_name`	`TEXT`	Name of the schema in which the constraint belongs.
`constraint_name`	`TEXT`	The name of the constraint.
`constraint_type`	`TEXT`	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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade; R - restrict; N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

13.31 DBA_DB_LINKS

The DBA_DB_LINKS view provides information about all database links in the database.

Name	Type	Description
owner	TEXT	User name of the database link's owner.
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB .
username	TEXT	User name of the user logging in.
host	TEXT	Name or IP address of the remote server.

13.32 DBA_DIRECTORIES

The DBA_DIRECTORIES view provides information about all directories created with the CREATE DIRECTORY command.

Name	Type	Description
owner	CHARACTER VARYING(30)	User name of the directory's owner.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
directory_name	CHARACTER VARYING(30)	The alias name assigned to the directory.
directory_path	CHARACTER VARYING(4000)	The path to the directory.

13.33 DBA_IND_COLUMNS

The DBA_IND_COLUMNS view provides information about all columns included in indexes, on all tables in the database.

Name	Type	Description
index_owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	Name of the index.
table_owner	TEXT	User name of the table's owner.
table_name	TEXT	Name of the table in which the index belongs.
column_name	TEXT	Name of column or attribute of object column.
column_position	SMALLINT	The position of the column in the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

13.34 DBA_INDEXES

The DBA_INDEXES view provides information about all indexes in the database.

Name	Type	Description
owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index resides.
index_name	TEXT	The name of the index.
index_type	TEXT	The index type is always BTREE. Included for compatibility only.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Always set to N (not compressed). Included for compatibility only.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Included for compatibility only. Always set to LOGGING.
status	TEXT	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHARACTER(3)	Indicates that the index is partitioned. Always set to NO.
temporary	CHARACTER(1)	Indicates that an index is on a temporary table. Always set to N.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

13.35 DBA_JOBS

The DBA_JOBS view provides information about all jobs in the database.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date.
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

13.36 DBA_OBJECTS

The DBA_OBJECTS view provides information about all objects in the database.

Name	Type	Description
owner	TEXT	User name of the object's owner.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
status	CHARACTER VARYING	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

13.37 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.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

13.38 DBA_PART_TABLES

The DBA_PART_TABLES view provides information about all of the partitioned tables in the database.

Name	Type	Description
owner	TEXT	The owner of the partitioned table.
schema_name	TEXT	The schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_next_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE.
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT.
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL.

13.39 DBA_POLICIES

The DBA_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

Name	Type	Description
object_owner	TEXT	Name of the owner of the object.
schema_name	TEXT	The name of the schema in which the object resides.
object_name	TEXT	Name of the object to which the policy applies.
policy_group	TEXT	Name of the policy group. Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Included for compatibility only; always set to NO.
policy_type	TEXT	Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

13.40 DBA_PROFILES

The DBA_PROFILES view provides information about existing profiles. The table includes a row for each profile/resource combination.

Name	Type	Description
profile	CHARACTER VARYING(128)	The name of the profile.
resource_name	CHARACTER VARYING(32)	The name of the resource associated with the profile.
resource_type	CHARACTER VARYING(8)	The type of resource governed by the profile; currently PASSWORD for all supported resources.
limit	CHARACTER VARYING(128)	The limit values of the resource.
common	CHARACTER VARYING(3)	YES for a user-created profile; NO for a system-defined profile.

13.41 DBA_QUEUES

The DBA_QUEUES view provides information about any currently defined queues.

Name	Type	Description
owner	TEXT	User name of the queue owner.
name	TEXT	The name of the queue.
queue_table	TEXT	The name of the queue table in which the queue resides.
qid	OID	The system-assigned object ID of the queue.
queue_type	CHARACTER VARYING	The queue type; may be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.
max_retries	NUMERIC	The maximum number of dequeue attempts.
retrydelay	NUMERIC	The maximum time allowed between retries.
enqueue_enabled	CHARACTER VARYING	YES if the queue allows enqueuing; NO if the queue does not.
dequeue_enabled	CHARACTER VARYING	YES if the queue allows dequeuing; NO if the queue does not.
retention	CHARACTER VARYING	The number of seconds that a processed message is retained in the queue.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
user_comment	CHARACTER VARYING	A user-specified comment.
network_name	CHARACTER VARYING	The name of the network on which the queue resides.
sharded	CHARACTER VARYING	YES if the queue resides on a sharded network; NO if the queue does not.

13.42 DBA_QUEUE_TABLES

The DBA_QUEUE_TABLES view provides information about all of the queue tables in the database.

Name	Type	Description
owner	TEXT	Role name of the owner of the queue table.
queue_table	TEXT	The user-specified name of the queue table.
type	CHARACTER VARYING	The type of data stored in the queue table.
object_type	TEXT	The user-defined payload type.
sort_order	CHARACTER VARYING	The order in which the queue table is sorted.
recipients	CHARACTER VARYING	Always SINGLE.
message_grouping	CHARACTER VARYING	Always NONE.
compatible	CHARACTER VARYING	The release number of the Advanced Server release with which this queue table is compatible.
primary_instance	NUMERIC	Always 0.
secondary_instance	NUMERIC	Always 0.
owner_instance	NUMERIC	The instance number of the instance that owns the queue table.
user_comment	CHARACTER VARYING	The user comment provided when the table was created.
secure	CHARACTER VARYING	YES indicates that the queue table is secure; NO indicates that it is not.

13.43 DBA_ROLE_PRIVS

The DBA_ROLE_PRIVS view provides information on all roles that have been granted to users. A row is created for each role to which a user has been granted.

Name	Type	Description
grantee	TEXT	User name to whom the role was granted.
granted_role	TEXT	Name of the role granted to the grantee.
admin_option	TEXT	YES if the role was granted with the admin option, NO otherwise.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
default_role	TEXT	YES if the role is enabled when the grantee creates a session.

13.44 DBA_ROLES

The DBA_ROLES view provides information on all roles with the NOLOGIN attribute (groups).

Name	Type	Description
role	TEXT	Name of a role having the NOLOGIN attribute – i.e., a group.
password_required	TEXT	Included for compatibility only; always N.

13.45 DBA_SEQUENCES

The DBA_SEQUENCES view provides information about all user-defined sequences.

Name	Type	Description
sequence_owner	TEXT	User name of the sequence's owner.
schema_name	TEXT	The name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	Will always return Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers stored in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

13.46 DBA_SOURCE

The DBA_SOURCE view provides the source code listing of all objects in the database.

Name	Type	Description
owner	TEXT	User name of the program's owner.
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.47 DBA_SUBPART_KEY_COLUMNS

The `DBA_SUBPART_KEY_COLUMNS` view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

Name	Type	Description
`owner`	`TEXT`	The owner of the table.
`schema_name`	`TEXT`	The name of the schema in which the table resides.
`name`	`TEXT`	The name of the table in which the column resides.
`object_type`	`CHARACTER(5)`	For compatibility only; always `TABLE`.
`column_name`	`TEXT`	The name of the column on which the key is defined.
`column_position`	`INTEGER`	`1` for the first column; `2` for the second column, etc.

13.48 DBA_SYNONYMS

The `DBA_SYNONYM` view provides information about all synonyms in the database.

Name	Type	Description
`owner`	`TEXT`	User name of the synonym's owner.
`schema_name`	`TEXT`	Name of the schema in which the synonym belongs.
`synonym_name`	`TEXT`	Name of the synonym.
`table_owner`	`TEXT`	User name of the table's owner on which the synonym is defined.
`table_schema_name`	`TEXT`	The name of the schema in which the table resides.
`table_name`	`TEXT`	Name of the table on which the synonym is defined.
`db_link`	`TEXT`	Name of any associated database link.

13.49 DBA_TAB_COLUMNS

The `DBA_TAB_COLUMNS` view provides information about all columns in the database.

Name	Type	Description
`owner`	`CHARACTER VARYING`	User name of the owner of the table or view in which the column resides.
`schema_name`	`CHARACTER VARYING`	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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
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 within the table or view.
data_default	CHARACTER VARYING	Default value assigned to the column.

13.50 DBA_TAB_PARTITIONS

The DBA_TAB_PARTITIONS view provides information about all of the partitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the partition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of high partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0 .
pct_used	NUMERIC	Included for compatibility only; always 0 .
ini_trans	NUMERIC	Included for compatibility only; always 0 .
max_trans	NUMERIC	Included for compatibility only; always 0 .
initial_extent	NUMERIC	Included for compatibility only; always NULL .
next_extent	NUMERIC	Included for compatibility only; always NULL .
min_extent	NUMERIC	Included for compatibility only; always 0 .
max_extent	NUMERIC	Included for compatibility only; always 0 .
pct_increase	NUMERIC	Included for compatibility only; always 0 .
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL .
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES .
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE .
num_rows	NUMERIC	Same as pg_class.reltuples .
blocks	INTEGER	Same as pg_class.relpages .
empty_blocks	NUMERIC	Included for compatibility only; always NULL .

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
avg_space	NUMERIC	Included for compatibility only; always NULL.
chain_cnt	NUMERIC	Included for compatibility only; always NULL.
avg_row_len	NUMERIC	Included for compatibility only; always NULL.
sample_size	NUMERIC	Included for compatibility only; always NULL.
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL.
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL.
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES.
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO.
backing_table	REGCLASS	Name of the partition backing table.

13.51 DBA_TAB_SUBPARTITIONS

The DBA_TAB_SUBPARTITIONS view provides information about all of the subpartitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the subpartition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0.
pct_used	NUMERIC	Included for compatibility only; always 0.
ini_trans	NUMERIC	Included for compatibility only; always 0.
max_trans	NUMERIC	Included for compatibility only; always 0.
initial_extent	NUMERIC	Included for compatibility only; always NULL.
next_extent	NUMERIC	Included for compatibility only; always NULL.
min_extent	NUMERIC	Included for compatibility only; always 0.
max_extent	NUMERIC	Included for compatibility only; always 0.
pct_increase	NUMERIC	Included for compatibility only; always 0.
freelists	NUMERIC	Included for compatibility only; always NULL.
freelist_groups	NUMERIC	Included for compatibility only; always NULL.
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES.
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE.
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
avg_space	NUMERIC	Included for compatibility only; always NULL.
chain_cnt	NUMERIC	Included for compatibility only; always NULL.
avg_row_len	NUMERIC	Included for compatibility only; always NULL.
sample_size	NUMERIC	Included for compatibility only; always NULL.
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL.
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL.
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES.
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO.
backing_table	REGCLASS	Name of the subpartition backing table.

13.52 DBA_TABLES

The DBA_TABLES view provides information about all tables in the database.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID.
temporary	CHARACTER(1)	Y if the table is temporary; N if the table is permanent.

13.53 DBA_TRIGGERS

The DBA_TRIGGERS view provides information about all triggers in the database.

Name	Type	Description
owner	TEXT	User name of the trigger's owner.
schema_name	TEXT	The name of the schema in which the trigger resides.
trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW, BEFORE STATEMENT, AFTER ROW, AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_names	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

13.54 DBA_TYPES

The DBA_TYPES view provides information about all object types in the database.

Name	Type	Description
owner	TEXT	The owner of the object type.
schema_name	TEXT	The name of the schema in which the type is defined.
type_name	TEXT	The name of the type.
type_oid	OID	The object identifier (OID) of the type.
typecode	TEXT	The typecode of the type. Possible values are: OBJECT, COLLECTION, OTHER
attributes	INTEGER	The number of attributes in the type.

13.55 DBA_USERS

The DBA_USERS view provides information about all users of the database.

Name	Type	Description
username	TEXT	User name of the user.
user_id	OID	ID number of the user.
password	CHARACTER VARYING (30)	The password (encrypted) of the user.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
account_status	CHARACTER VARYING(32)	The current status of the account. Possible values are: OPEN, EXPIRED, EXPIRED(GRACE), EXPIRED & LOCKED, EXPIRED & LOCKED(TIMED), EXPIRED(GRACE) & LOCKED, EXPIRED(GRACE) & LOCKED(TIMED), LOCKED, LOCKED(TIMED). Use the edb_get_role_status(role_id) function to get the current status of the account.
lock_date	TIMESTAMP WITHOUT TIME ZONE	If the account status is LOCKED, lock_date displays the date and time the account was locked.
expiry_date	TIMESTAMP WITHOUT TIME ZONE	The expiration date of the password. Use the edb_get_password_expiry_date(role_id) function to get the current password expiration date.
default_tablespace	TEXT	The default tablespace associated with the account.
temporary_tablespace	CHARACTER VARYING(30)	Included for compatibility only. The value will always be '' (an empty string).
created	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only. The value is always NULL.
profile	CHARACTER VARYING(30)	The profile associated with the user.
initial_rsrc_consumer_group	CHARACTER VARYING(30)	Included for compatibility only. The value is always NULL.
external_name	CHARACTER VARYING(4000)	Included for compatibility only. The value is always NULL.

13.56 DBA_VIEW_COLUMNS

The DBA_VIEW_COLUMNS view provides information on all columns in the database.

Name	Type	Description
owner	CHARACTER VARYING	User name of the view's owner.
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
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 within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

13.57 DBA_VIEWS

The DBA_VIEWS view provides information about all views in the database.

Name	Type	Description
owner	TEXT	User name of the view's owner.
schema_name	TEXT	Name of the schema in which the view belongs.
view_name	TEXT	Name of the view.
text	TEXT	The text of the SELECT statement that defines the view.

13.58 USER_ALL_TABLES

The USER_ALL_TABLES view provides information about all tables owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

13.59 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

13.60 USER_CONSTRAINTS

The USER_CONSTRAINTS view provides information about all constraints placed on tables that are owned by the current user.

Name	Type	Description
owner	TEXT	The name of the owner of the constraint.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	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
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade; R - restrict; N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.61 USER_DB_LINKS

The `USER_DB_LINKS` view provides information about all database links that are owned by the current user.

Name	Type	Description
`db_link`	`TEXT`	The name of the database link.
`type`	`CHARACTER VARYING`	Type of remote server. Value will be either `REDWOOD` or `EDB`.
`username`	`TEXT`	User name of the user logging in.
`password`	`TEXT`	Password used to authenticate on the remote server.
`host`	`TEXT`	Name or IP address of the remote server.

13.62 USER_IND_COLUMNS

The `USER_IND_COLUMNS` view provides information about all columns referred to in indexes on tables that are owned by the current user.

Name	Type	Description
`schema_name`	`TEXT`	Name of the schema in which the index belongs.
`index_name`	`TEXT`	The name of the index.
`table_name`	`TEXT`	The name of the table to which the index belongs.
`column_name`	`TEXT`	The name of the column.
`column_position`	`SMALLINT`	The position of the column within the index.
`column_length`	`SMALLINT`	The length of the column (in bytes).
`char_length`	`NUMERIC`	The length of the column (in characters).
`descend`	`CHARACTER(1)`	Always set to `Y` (descending); included for compatibility only.

13.63 USER_INDEXES

The `USER_INDEXES` view provides information about all indexes on tables that are owned by the current user.

Name	Type	Description
`schema_name`	`TEXT`	Name of the schema in which the index belongs.
`index_name`	`TEXT`	The name of the index.
`index_type`	`TEXT`	Included for compatibility only. The index type is always `BTREE`.
`table_owner`	`TEXT`	User name of the owner of the indexed table.
`table_name`	`TEXT`	The name of the indexed table.
`table_type`	`TEXT`	Included for compatibility only. Always set to `TABLE`.
`uniqueness`	`TEXT`	Indicates if the index is `UNIQUE` or `NONUNIQUE`.
`compression`	`CHARACTER(1)`	Included for compatibility only. Always set to `N` (not compressed).

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Included for compatibility only. Always set to LOGGING.
status	TEXT	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHARACTER(3)	Included for compatibility only. Always set to NO.
temporary	CHARACTER(1)	Included for compatibility only. Always set to N.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

13.64 USER_JOBS

The USER_JOBS view provides information about all jobs owned by the current user.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date.
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

13.65 USER_OBJECTS

The USER_OBJECTS view provides information about all objects that are owned by the current user.

Name	Type	Description

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW
status	CHARACTER VARYING	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the object is temporary; N if the object is not temporary.

13.66 USER_PART_KEY_COLUMNS

The USER_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

13.67 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.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The partitioning type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE.
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT.
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL.

13.68 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.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the object resides.
object_name	TEXT	Name of the object on which the policy applies.
policy_group	TEXT	Name of the policy group. Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Whether or not the policy is static. Included for compatibility only; always set to NO.
policy_type	TEXT	Policy type. Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

13.69 USER_QUEUES

The USER_QUEUES view provides information about any queue on which the current user has usage privileges.

Name	Type	Description
name	TEXT	The name of the queue.
queue_table	TEXT	The name of the queue table in which the queue resides.
qid	OID	The system-assigned object ID of the queue.
queue_type	CHARACTER VARYING	The queue type; may be EXCEPTION_QUEUE, NON_PERSISTENT_QUEUE, or NORMAL_QUEUE.
max_retries	NUMERIC	The maximum number of dequeue attempts.
retrydelay	NUMERIC	The maximum time allowed between retries.
enqueue_enabled	CHARACTER VARYING	YES if the queue allows enqueuing; NO if the queue does not.
dequeue_enabled	CHARACTER VARYING	YES if the queue allows dequeuing; NO if the queue does not.
retention	CHARACTER VARYING	The number of seconds that a processed message is retained in the queue.
user_comment	CHARACTER VARYING	A user-specified comment.
network_name	CHARACTER VARYING	The name of the network on which the queue resides.
sharded	CHARACTER VARYING	YES if the queue resides on a sharded network; NO if the queue does not.

13.70 USER_QUEUE_TABLES

The USER_QUEUE_TABLES view provides information about all of the queue tables accessible by the current user.

Name	Type	Description
queue_table	TEXT	The user-specified name of the queue table.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
type	CHARACTER VARYING	The type of data stored in the queue table.
object_type	TEXT	The user-defined payload type.
sort_order	CHARACTER VARYING	The order in which the queue table is sorted.
recipients	CHARACTER VARYING	Always SINGLE.
message_grouping	CHARACTER VARYING	Always NONE.
compatible	CHARACTER VARYING	The release number of the Advanced Server release with which this queue table is compatible.
primary_instance	NUMERIC	Always 0.
secondary_instance	NUMERIC	Always 0.
owner_instance	NUMERIC	The instance number of the instance that owns the queue table.
user_comment	CHARACTER VARYING	The user comment provided when the table was created.
secure	CHARACTER VARYING	YES indicates that the queue table is secure; NO indicates that it is not.

13.71 USER_ROLE_PRIVS

The USER_ROLE_PRIVS view provides information about the privileges that have been granted to the current user. A row is created for each role to which a user has been granted.

Name	Type	Description
username	TEXT	The name of the user to which the role was granted.
granted_role	TEXT	Name of the role granted to the grantee.
admin_option	TEXT	YES if the role was granted with the admin option, NO otherwise.
default_role	TEXT	YES if the role is enabled when the grantee creates a session.
os_granted	CHARACTER VARYING(3)	Included for compatibility only; always NO.

13.72 USER_SEQUENCES

The USER_SEQUENCES view provides information about all user-defined sequences that belong to the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	Included for compatibility only; always Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

13.73 USER_SOURCE

The USER_SOURCE view provides information about all programs owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

13.74 USER_SUBPART_KEY_COLUMNS

The USER_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that belong to the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

13.75 USER_SYNONYMS

The USER_SYNONYMS view provides information about all synonyms owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the synonym resides.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the table's owner on which the synonym is defined.
table_schema_name	TEXT	The name of the schema in which the table resides.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
table_name	TEXT	Name of the table on which the synonym is defined.
db_link	TEXT	Name of any associated database link.

13.76 USER_TAB_COLUMNS

The USER_TAB_COLUMNS view displays information about all columns in tables and views owned by the current user.

Name	Type	Description
schema_name	CHARACTER VARYING	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 within the table.
data_default	CHARACTER VARYING	Default value assigned to the column.

13.77 USER_TAB_PARTITIONS

The USER_TAB_PARTITIONS view provides information about all of the partitions that are owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
pct_free	NUMERIC	Included for compatibility only; always 0.
pct_used	NUMERIC	Included for compatibility only; always 0.
ini_trans	NUMERIC	Included for compatibility only; always 0.
max_trans	NUMERIC	Included for compatibility only; always 0.
initial_extent	NUMERIC	Included for compatibility only; always NULL.
next_extent	NUMERIC	Included for compatibility only; always NULL.
min_extent	NUMERIC	Included for compatibility only; always 0.
max_extent	NUMERIC	Included for compatibility only; always 0.
pct_increase	NUMERIC	Included for compatibility only; always 0.
freelists	NUMERIC	Included for compatibility only; always NULL.
freelist_groups	NUMERIC	Included for compatibility only; always NULL.
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES.
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE.
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL.
avg_space	NUMERIC	Included for compatibility only; always NULL.
chain_cnt	NUMERIC	Included for compatibility only; always NULL.
avg_row_len	NUMERIC	Included for compatibility only; always NULL.
sample_size	NUMERIC	Included for compatibility only; always NULL.
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL.
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL.
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES.
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO.
backing_table	REGCLASS	Name of the partition backing table.

13.78 USER_TAB_SUBPARTITIONS

The USER_TAB_SUBPARTITIONS view provides information about all of the subpartitions owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0.
pct_used	NUMERIC	Included for compatibility only; always 0.
ini_trans	NUMERIC	Included for compatibility only; always 0.
max_trans	NUMERIC	Included for compatibility only; always 0.
initial_extent	NUMERIC	Included for compatibility only; always NULL.
next_extent	NUMERIC	Included for compatibility only; always NULL.
min_extent	NUMERIC	Included for compatibility only; always 0.
max_extent	NUMERIC	Included for compatibility only; always 0.
pct_increase	NUMERIC	Included for compatibility only; always 0.
freelists	NUMERIC	Included for compatibility only; always NULL.
freelist_groups	NUMERIC	Included for compatibility only; always NULL.
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES.
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE.
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL.
avg_space	NUMERIC	Included for compatibility only; always NULL.
chain_cnt	NUMERIC	Included for compatibility only; always NULL.
avg_row_len	NUMERIC	Included for compatibility only; always NULL.
sample_size	NUMERIC	Included for compatibility only; always NULL.
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL.
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL.
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES.
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO.
backing_table	REGCLASS	Name of the partition backing table.

13.79 USER_TABLES

The USER_TABLES view displays information about all tables owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID.
temporary	CHARACTER(1)	Y if the table is temporary; N if the table is not temporary.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

13.80 USER_TRIGGERS

The `USER_TRIGGERS` view displays information about all triggers on tables owned by the current user.

Name	Type	Description
`schema_name`	`TEXT`	The name of the schema in which the trigger resides.
`trigger_name`	`TEXT`	The name of the trigger.
`trigger_type`	`TEXT`	The type of the trigger. Possible values are: `BEFORE ROW`, `BEFORE STATEMENT`, `AFTER ROW`, `AFTER STATEMENT`
`triggering_event`	`TEXT`	The event that fires the trigger.
`table_owner`	`TEXT`	The user name of the owner of the table on which the trigger is defined.
`base_object_type`	`TEXT`	Included for compatibility only. Value will always be `TABLE`.
`table_name`	`TEXT`	The name of the table on which the trigger is defined.
`referencing_names`	`TEXT`	Included for compatibility only. Value will always be `REFERENCING NEW AS NEW OLD AS OLD`.
`status`	`TEXT`	Status indicates if the trigger is enabled `(VALID)` or disabled `(NOTVALID)`.
`description`	`TEXT`	Included for compatibility only.
`trigger_body`	`TEXT`	The body of the trigger.
`action_statement`	`TEXT`	The SQL command that executes when the trigger fires.

13.81 USER_TYPES

The `USER_TYPES` view provides information about all object types owned by the current user.

Name	Type	Description
`schema_name`	`TEXT`	The name of the schema in which the type is defined.
`type_name`	`TEXT`	The name of the type.
`type_oid`	`OID`	The object identifier (OID) of the type.
`typecode`	`TEXT`	The typecode of the type. Possible values are: `OBJECT`, `COLLECTION`, `OTHER`
`attributes`	`INTEGER`	The number of attributes in the type.

13.82 USER_USERS

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `USER_USERS` view provides information about the current user.

Name	Type	Description
`username`	`TEXT`	User name of the user.
`user_id`	`OID`	ID number of the user.
`account_status`	`CHARACTER VARYING(32)`	The current status of the account. Possible values are: `OPEN`, `EXPIRED`, `EXPIRED(GRACE)`, `EXPIRED & LOCKED`, `EXPIRED & LOCKED(TIMED)`, `EXPIRED(GRACE) & LOCKED`, `EXPIRED(GRACE) & LOCKED(TIMED)`, `LOCKED`, `LOCKED(TIMED)`. Use the `edb_get_role_status(role_id)` function to get the current status of the account.
`lock_date`	`TIMESTAMP WITHOUT TIME ZONE`	If the account status is `LOCKED`, `lock_date` displays the date and time the account was locked.
`expiry_date`	`TIMESTAMP WITHOUT TIME ZONE`	The expiration date of the account.
`default_tablespace`	`TEXT`	The default tablespace associated with the account.
`temporary_tblespace`	`CHARACTER VARYING(30)`	Included for compatibility only. The value will always be '' (an empty string).
`created`	`TIMESTAMP WITHOUT TIME ZONE`	Included for compatibility only. The value will always be `NULL`.
`initial_rsrc_consumer_group`	`CHARACTER VARYING(30)`	Included for compatibility only. The value will always be `NULL`.
`external_name`	`CHARACTER VARYING(4000)`	Included for compatibility only; always set to `NULL`.

13.83 USER_VIEW_COLUMNS

The `USER_VIEW_COLUMNS` view provides information about all columns in views owned by the current user.

Name	Type	Description
`schema_name`	`CHARACTER VARYING`	Name of the schema in which the view belongs.
`view_name`	`CHARACTER VARYING`	Name of the view.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
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 within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

13.84 USER_VIEWS

The USER_VIEWS view provides information about all views owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the view resides.
view_name	TEXT	Name of the view.
text	TEXT	The SELECT statement that defines the view.

13.85 V$VERSION

The V$VERSION view provides information about product compatibility.

Name	Type	Description
banner	TEXT	Displays product compatibility information.

13.86 PRODUCT_COMPONENT_VERSION

The PRODUCT_COMPONENT_VERSION view provides version information about product version compatibility.

Name	Type	Description
product	CHARACTER VARYING(74)	The name of the product.
version	CHARACTER VARYING(74)	The version number of the product.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name	Type	Description
`status`	`CHARACTER VARYING(74)`	Included for compatibility; always `Available`.

14 Database Compatibility Stored Procedural Language Guide

Advanced Server's stored procedural language (SPL) is a highly productive, procedural programming language for writing custom procedures, functions, triggers, and packages for Advanced Server that provides:

- full procedural programming functionality to complement the SQL language
- a single, common language to create stored procedures, functions, triggers, and packages for the Advanced Server database
- a seamless development and testing environment
- the use of reusable code
- ease of use

This guide describes the basic elements of an SPL program, before providing an overview of the organization of an SPL program and how it is used to create a procedure or a function. Guides about the other compatibility features (such as triggers or built-in functions) that might be used in conjunction with the SPL language are available at the EDB website.

14.1 Basic SPL Elements

This section discusses the basic programming elements of an SPL program.

14.1.1 Case Sensitivity

Keywords and user-defined identifiers that are used in an SPL program are case insensitive. So for example, the statement `DBMS_OUTPUT.PUT_LINE('Hello World');` is interpreted to mean the same thing 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 as well as any data retrieved from the Advanced Server database or data obtained from other external sources. The statement `DBMS_OUTPUT.PUT_LINE('Hello World!');` produces the following output:

```
Hello World!
```

However the statement `DBMS_OUTPUT.PUT_LINE('HELLO WORLD!');` produces the output:

```
HELLO WORLD!
```

14.1.2 Identifiers

Identifiers are user-defined names that are used to identify various 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 must not be the same as an SPL keyword or a keyword of the SQL language. The following are some examples of valid identifiers:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 x
last___name
a_$_Sign
Many$$$$$$$$signs_____
THIS_IS_AN_EXTREMELY_LONG_NAME
A1
```

14.1.3 Qualifiers

A *qualifier* is a name that specifies the owner or context of an entity that is the object of the qualification. A qualified object is specified as the qualifier name followed by a dot with no intervening white space, followed by 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`. It is possible to have a chain of qualifications where the preceding qualifier owns the entity identified by the subsequent qualifier(s) and object.

Almost any identifier can be qualified. What an identifier is qualified by depends upon what the identifier represents and the context of its usage.

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, wherever a name appears in the syntax of an SPL statement, its qualified name can be used as well. Typically a qualified name would only be used if there is some ambiguity associated with the name. For example, if two procedures with the same name belonging to two different schemas are invoked from within a program or if the same name is used for a table column and SPL variable within the same program.

You should avoid using qualified names if at all possible. In this chapter, the following conventions are adopted 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 do not have any special prefix conventions. E.g., column `empno` in table `emp`

14.1.4 Constants

Constants or *literals* are fixed values that can be used in SPL programs to represent values of various types - e.g., numbers, strings, dates, etc. Constants come in the following types:

- Numeric (Integer and Real)
- Character and String
- Date/time

14.1.5 User-Defined PL/SQL Subtypes

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Advanced Server supports user-defined PL/SQL subtypes and (subtype) aliases. 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>
```

Where:

`subtype_name`

`subtype_name` specifies the name of the subtype.

`type_name`

`type_name` specifies the name of the original type on which the subtype is based. `type_name` may be:

- The name of any of the type supported by 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`

`precision` specifies the total number of digits permitted in a value of the subtype.

`scale`

`scale` specifies the number of fractional digits permitted in a value of the subtype.

`length`

`length` specifies the total length permitted in a value of `CHARACTER, VARCHAR`, or `TEXT` base types.

Include the `NOT NULL` clause to specify that `NULL` values may not be stored in a column of the specified subtype.

Note that a subtype that is based on a column will inherit the column size constraints, but the subtype will not 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 is a subtype of another subtype:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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` may not be `NULL`.

Constrained Subtypes

Include a `length` value when creating a subtype that is based on a character type to define the maximum length of the subtype. For example:

```
SUBTYPE acct_name IS VARCHAR (15);
```

This example creates a subtype named `acct_name` that is based on a `VARCHAR` data type, but is limited to 15 characters in length.

Include values for `precision` (to specify the maximum number of digits in a value of the subtype) and optionally, `scale` (to specify the number of digits to the right of the decimal point) when constraining a numeric base type. For example:

```
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 may not exceed 3 digits to the left of the decimal point and 2 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 within the body of the function or procedure.

If a formal argument is declared as a constrained subtype:

- Advanced Server does not enforce subtype constraints when assigning an actual argument to a formal argument when invoking a function.
- Advanced Server enforces subtype constraints when assigning an actual argument to a formal argument when invoking a procedure.

Using the %TYPE Operator

You can use `%TYPE` notation to declare a subtype anchored to a column. For example:

```
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 is based on a column will share the column size constraints; `NOT NULL` and `CHECK` constraints are not 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.

A variable of a constrained subtype may be interchanged with a variable of the base type without conversion, but a variable of the base type may only be interchanged with a constrained subtype if it complies with the constraints of the subtype. A variable of a constrained subtype may be implicitly converted to another subtype if it is based on the same subtype, and the constraint values are within the values of the subtype to which it is being converted.

14.1.6 Character Set

SPL programs are written using the following set of characters:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Uppercase letters `A` thru `Z` and lowercase letters `a` thru `z`
- Digits `0` thru `9`
- Symbols `( ) + - * / < > = ! ~ ^ ; : . ' @ % , " # $ & _ | { } ? [ ]`
- White space characters tabs, spaces, and carriage returns

Identifiers, expressions, statements, control structures, etc. that comprise the SPL language are written using these characters.

> **Note**
>
> The data that can be manipulated by an SPL program is determined by the character set supported by the database encoding.

14.2 SPL Programs

SPL is a procedural, block-structured language. There are four different types of programs that can be created using SPL, namely *procedures*, *functions*, *triggers*, and *packages*.

In addition, SPL is used to create subprograms. A *subprogram* refers to a *subprocedure* or a *subfunction*, which are nearly identical in appearance to procedures and functions, but differ in that procedures and functions are *standalone programs*, which are individually stored in the database and can be invoked by other SPL programs or from PSQL. Subprograms can only be invoked from within the standalone program within which they have been created.

14.2.1 SPL Block Structure

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 within the keywords, `BEGIN` and `END`.

The optional declaration section is used to declare variables, cursors, types, and subprograms that are used by the statements within the executable and exception sections. Declarations appear just prior to the `BEGIN` keyword of the executable section. Depending upon the context of where the block is used, the declaration section may begin with the keyword `DECLARE`.

You can include an exception section within 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 within the block, program control goes to the exception section where the thrown exception may or may not be handled depending upon 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, they must be declared after all other variable, cursor, and type declarations. Each declaration must be terminated by a semicolon. The use of the keyword `DECLARE` depends upon the context in which the block appears.

`statements` are one or more SPL statements. Each statement must be terminated by a semicolon. The end of the block denoted by the keyword `END`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

must also be terminated by 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 may be one or more `WHEN exception_condition` clauses, each followed by `statements`.

> **Note**
>
> A `BEGIN/END` block in itself, is considered a statement; thus, blocks may be nested. The exception section may also contain nested blocks.

The following is the simplest possible block consisting of the `NULL` statement within 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, storing the results in a third variable which is then displayed. If executed, the output would be:

```
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 within the exception section is executed as a result of the division by zero.

```
An exception occurred
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.2.2 Anonymous Blocks

Blocks are typically written as part of a procedure, function, subprogram, or trigger. Procedure, function, and trigger programs are named and stored in the database for re-use. For quick (one-time) execution (such as testing), you can simply enter the block without providing a name or storing it in the database.

A block of this type is called an *anonymous block*. An anonymous block is unnamed and is not stored in the database. Once the block has been executed and erased from the application buffer, it cannot be re-executed unless the block code is re-entered into the application.

Typically, the same block of code will be re-executed many times. In order to run a block of code repeatedly without the necessity of re-entering the code each time, with some simple modifications, an anonymous block can be turned into a procedure or function. The following sections discuss how to create a procedure or function that can be stored in the database and invoked repeatedly by another procedure, function, or application program.

14.2.3 Procedures Overview

Procedures are standalone SPL programs that are invoked or called as an individual SPL program statement. When called, procedures may optionally receive values from the caller in the form of input parameters and optionally return values to the caller in the form of output parameters.

14.2.3.1 Creating a Procedure

The `CREATE PROCEDURE` command defines and names a standalone procedure that will be stored in the database.

If a schema name is included, then the procedure is created in the specified schema. Otherwise it is 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 may share a name (this is called *overloading*). (Overloading of procedures is an Advanced Server feature - overloading of stored, standalone procedures is not compatible with Oracle databases.)

To update the definition of an existing procedure, use `CREATE OR REPLACE PROCEDURE`. It is not possible to change the name or argument types of a procedure this way (if you tried, you would actually be creating a new, distinct procedure). When using `OUT` parameters, you cannot change the types of any `OUT` parameters except by dropping the procedure.

```
CREATE [OR REPLACE] PROCEDURE <name> [ (<parameters>) ]
   [
         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; ]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    [ <declarations> ]
  BEGIN
    <statements>
  END [ <name> ];
```

Where:

`name`

`name` is the identifier of the procedure.

`parameters`

`parameters` is a list of formal parameters.

`declarations`

`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`

`statements` are SPL program statements (the `BEGIN - END` block may 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 cannot modify the database and always reaches the same result when given the same argument values; it does not 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.
- `STABLE` indicates that the procedure cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for procedures that depend on database lookups, parameter variables (such as the current time zone), etc.
- `VOLATILE` indicates that the procedure value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

`DETERMINISTIC`

`DETERMINISTIC` is a synonym for `IMMUTABLE`. A `DETERMINISTIC` procedure cannot modify the database and always reaches the same result when given the same argument values; it does not 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`RETURNS NULL ON NULL INPUT`

`STRICT`

- `CALLED ON NULL INPUT` (the default) indicates that the procedure will 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` whenever any of its arguments are `NULL`. If these clauses are specified, the procedure is not executed when there are `NULL` arguments; instead a `NULL` result is assumed automatically.

`[ EXTERNAL ] SECURITY DEFINER`

`SECURITY DEFINER` specifies that the procedure will 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 that the procedure will 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 procedure owner are used to 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 procedure are used to 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 procedure cannot be executed in parallel mode. The presence of such a procedure forces a serial execution plan. This is the default setting if the `PARALLEL` clause is omitted.

- When set to `RESTRICTED`, the procedure 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 can be executed 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 that the planner should expect the procedure to return. This is only allowed when the procedure is declared to return a set. The default assumption is 1000 rows.

`SET configuration_parameter { TO value | = value | FROM CURRENT }`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 be applied 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 would do for a previous `SET LOCAL` command, with the effects of such a command persisting 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 Advanced Server and are not 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

The following is an example of a simple procedure that takes no parameters.

```
 CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('That''s all folks!');
END simple_procedure;
```

The procedure is stored in the database by entering the procedure code in Advanced Server.

The following example demonstrates 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 is 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 will not be affected by these settings.

In this example, the `AUTHID DEFINER` clause temporarily grants privileges to a role that might otherwise not be allowed to execute the statements within 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.

14.2.3.2 Calling a Procedure

A procedure can be invoked from another SPL program by simply specifying the procedure name followed by its parameters, if any, followed by a semicolon.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
<name> [ ([ <parameters> ]) ];
```

Where:

`name` is the identifier of the procedure.

`parameters` is a list of actual parameters.

> **Note**
>
> - If there are no actual parameters to be passed, the procedure may be called with an empty parameter list, or the opening and closing parenthesis may be omitted entirely.
> - 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 the *Database Compatibility for Oracle Developers SQL Guide* for information about the `EXEC` command.

The following is an example of calling the procedure from an anonymous block:

```
BEGIN
   simple_procedure;
END;

That's all folks!
```

> **Note**
>
> Each application has its own unique way to call a procedure. For example, in a Java application, the application programming interface, JDBC, is used.

14.2.3.3 Deleting a Procedure

A procedure can be deleted from the database using the `DROP PROCEDURE` command.

```
DROP PROCEDURE [ IF EXISTS ] <name> [ (<parameters>) ]
    [ CASCADE | RESTRICT ];
```

Where `name` is the name of the procedure to be dropped.

> **Note**
>
> - The specification of the parameter list is required in Advanced Server under certain circumstances such as if this is an overloaded procedure. Oracle requires that the parameter list always be omitted.
> - Usage of `IF EXISTS, CASCADE`, or `RESTRICT` is not compatible with Oracle databases. See the `DROP PROCEDURE` command in the *Database Compatibility for Oracle Developers SQL Guide* for information on these options.

The previously created procedure is dropped in this example:

```
DROP PROCEDURE simple_procedure;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.2.4 Functions Overview

Functions are standalone SPL programs that are invoked as expressions. When evaluated, a function returns a value that is substituted in the expression in which the function is embedded. Functions may optionally take values from the calling program in the form of input parameters. In addition to the fact that the function, itself, returns a value, a function may optionally return additional values to the caller in the form of output parameters. The use of output parameters in functions, however, is not an encouraged programming practice.

14.2.4.1 Creating a Function

The `CREATE FUNCTION` command defines and names a standalone function that will be stored in the database.

If a schema name is included, then the function is created in the specified schema. Otherwise it is 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 may share a name (this is called *overloading*). (Overloading of functions is an Advanced Server feature - overloading of stored, standalone functions is not compatible with Oracle databases).

To update the definition of an existing function, use `CREATE OR REPLACE FUNCTION`. It is not possible to change the name or argument types of a function this way (if you tried, you would actually be creating a new, distinct function). Also, `CREATE OR REPLACE FUNCTION` will not let you change the return type of an existing function. To do that, you must drop and recreate the function. Also when using `OUT` parameters, you cannot change the types of any `OUT` parameters except by dropping the function.

```
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`

`name` is the identifier of the function.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`parameters`

`parameters` is a list of formal parameters.

`data_type`

`data_type` is the data type of the value returned by the function's `RETURN` statement.

`declarations`

`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`

`statements` are SPL program statements (the `BEGIN - END` block may contain an `EXCEPTION` section).

`IMMUTABLE`

`STABLE`

`VOLATILE`

These attributes inform the query optimizer about the behavior of the function; you can specify only one choice. `VOLATILE` is the default behavior.

- `IMMUTABLE` indicates that the function cannot modify the database and always reaches the same result when given the same argument values; it does not 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 cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for function that depend on database lookups, parameter variables (such as the current time zone), etc.
- `VOLATILE` indicates that the function value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

`DETERMINISTIC`

`DETERMINISTIC` is a synonym for `IMMUTABLE`. A `DETERMINISTIC` function cannot modify the database and always reaches the same result when given the same argument values; it does not 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.

`[ 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 that the procedure will 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `RETURNS NULL ON NULL INPUT` or `STRICT` indicates that the procedure always returns `NULL` whenever any of its arguments are `NULL`. If these clauses are specified, the procedure is not executed when there are `NULL` arguments; instead a `NULL` result is assumed automatically.

`[ EXTERNAL ] SECURITY DEFINER`

`SECURITY DEFINER` specifies that the function will 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 that the function will 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 are used to 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 are used to 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 function cannot be executed in parallel mode. The presence of such a function in a SQL statement forces a serial execution plan. This is the default setting if the `PARALLEL` clause is omitted.
- When set to `RESTRICTED`, the 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 function can be executed 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.

`ROWS result_rows`

`result_rows` is a positive number giving the estimated number of rows that the planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption is 1000 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 be applied 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, much as it would do for a previous `SET LOCAL` command, with the effects of such a command persisting after procedure exit, unless the current transaction is rolled back.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 Advanced Server and are not supported by Oracle.

Examples

The following is an example of a simple function that takes no parameters.

```
CREATE OR REPLACE FUNCTION simple_function
    RETURN VARCHAR2
IS
BEGIN
    RETURN 'That''s All Folks!';
END simple_function;
```

The following function takes two input parameters. Parameters are discussed in more detail in subsequent sections.

```
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;
```

The following example demonstrates using 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 = 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 will not execute.

The `dept_salaries` function executes with the privileges of the role that is calling the function. If the current user does not have sufficient privileges to perform the `SELECT` statement querying the `emp` table (to display employee salaries), the function will report 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.

14.2.4.2 Calling a Function

A function can be used anywhere an expression can appear within an SPL statement. A function is invoked by simply specifying its name followed by its parameters enclosed in parenthesis, if any.

```
<name> [ ([ <parameters> ]) ]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`name` is the name of the function.

`parameters` is a list of actual parameters.

> **Note**
>
> If there are no actual parameters to be passed, the function may be called with an empty parameter list, or the opening and closing parenthesis may be omitted entirely.

The following shows how the function can be called from another SPL program.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(simple_function);
END;

That's All Folks!
```

A function is typically used within a SQL statement as shown in the following.

```
SELECT empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM",
    emp_comp(sal, comm) "YEARLY COMPENSATION" FROM emp;

 EMPNO | ENAME  |   SAL   |  COMM   | YEARLY COMPENSATION
-------+--------+---------+---------+---------------------
  7369 | SMITH  |  800.00 |         |            19200.00
  7499 | ALLEN  | 1600.00 |  300.00 |            45600.00
  7521 | WARD   | 1250.00 |  500.00 |            42000.00
  7566 | JONES  | 2975.00 |         |            71400.00
  7654 | MARTIN | 1250.00 | 1400.00 |            63600.00
  7698 | BLAKE  | 2850.00 |         |            68400.00
  7782 | CLARK  | 2450.00 |         |            58800.00
  7788 | SCOTT  | 3000.00 |         |            72000.00
  7839 | KING   | 5000.00 |         |           120000.00
  7844 | TURNER | 1500.00 |    0.00 |            36000.00
  7876 | ADAMS  | 1100.00 |         |            26400.00
  7900 | JAMES  |  950.00 |         |            22800.00
  7902 | FORD   | 3000.00 |         |            72000.00
  7934 | MILLER | 1300.00 |         |            31200.00
(14 rows)
```

14.2.4.3 Deleting a Function

A function can be deleted from the database using the `DROP FUNCTION` command.

```
DROP FUNCTION [ IF EXISTS ] <name> [ (<parameters>) ]
    [ CASCADE | RESTRICT ];
```

Where `name` is the name of the function to be dropped.

> **Note**
>
> - The specification of the parameter list is required in Advanced Server under certain circumstances such as if this is an overloaded function. Oracle requires that the parameter list always be omitted.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Usage of `IF EXISTS, CASCADE`, or `RESTRICT` is not compatible with Oracle databases. See the `DROP FUNCTION` command in the *Database Compatibility for Oracle Developers SQL Guide* for information on these options.

The previously created function is dropped in this example:

```
DROP FUNCTION simple_function;
```

14.2.5 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. This is accomplished by using *parameters*.

Parameters are declared in the procedure or function definition, enclosed within parenthesis following the procedure or function name. Parameters declared in the procedure or function definition are known as *formal parameters*. When the procedure or function is invoked, the calling program supplies the actual data that is to be used in the called program's processing as well as the variables that are to 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.

```
(<name> [ IN | OUT | IN OUT ] <data_type> [ DEFAULT <value> ])
```

`name` is an identifier assigned to the formal parameter. 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. Whether a parameter is `IN, OUT`, or `IN OUT` is referred to as the parameter's *mode*. `data_type` defines the data type of the parameter. `value` is a default value assigned to an `IN` parameter in the called program if an actual parameter is not specified in the call.

The following is an example of a procedure that takes parameters:

```
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 previous example, no maximum length was specified on the `VARCHAR2` parameters and no precision and scale were specified on the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> `NUMBER` parameters. It is illegal to specify a length, precision, scale or other constraints on parameter declarations. These constraints are automatically 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. The following is an example of another SPL program that calls `emp_query`.

```
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);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
END;
```

In this example, `v_deptno, v_empno, v_ename, v_job, v_hiredate`, and `v_sal` are the actual parameters.

The output from the preceding example is shown as follows:

```
 Department : 30
Employee No: 7900
Name       : JAMES
Job        : CLERK
Hire Date  : 03-DEC-81
Salary     : 950
```

14.2.5.1 Positional vs. 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 is not significant.

To 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.

A simple example that demonstrates using positional and named parameter notation follows:

```
CREATE OR REPLACE PROCEDURE emp_info (
    p_deptno        IN     NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
)
IS
BEGIN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
```

To call the procedure using positional notation, pass the following:

```
emp_info(30, 7455, 'Clark');
```

To call the procedure using named notation, pass the following:

```
emp_info(p_ename =>'Clark', p_empno=>7455, p_deptno=>30);
```

Using named notation can alleviate the need to re-arrange a procedure's parameter list if the parameter list changes, if the parameters are reordered or if a new optional parameter is added.

In a case where you have a default value for an argument and the argument is not a trailing argument, you must use named notation to call the procedure or function. The following case demonstrates a procedure with two, leading, default arguments.

```
CREATE OR REPLACE PROCEDURE check_balance (
    p_customerID  IN NUMBER DEFAULT NULL,
    p_balance     IN NUMBER DEFAULT NULL,
    p_amount      IN NUMBER
)
IS
DECLARE
    balance NUMBER;
BEGIN
   IF (p_balance IS NULL AND p_customerID IS NULL) THEN
      RAISE_APPLICATION_ERROR
          (-20010, 'Must provide balance or customer');
   ELSEIF (p_balance IS NOT NULL AND p_customerID IS NOT NULL) THEN
      RAISE_APPLICATION_ERROR
          (-20020,'Must provide balance or customer, not both');
   ELSEIF (p_balance IS NULL) THEN
      balance := getCustomerBalance(p_customerID);
   ELSE
      balance := p_balance;
   END IF;

   IF (amount > balance) THEN
      RAISE_APPLICATION_ERROR
        (-20030, 'Balance insufficient');
   END IF;
END;
```

You can only omit non-trailing argument values (when you call this procedure) by using named notation; when using positional notation, only trailing arguments are allowed to default. You can call this procedure with the following arguments:

```
check_balance(p_customerID => 10, p_amount = 500.00)

check_balance(p_balance => 1000.00, p_amount = 500.00)
```

You can use a combination of positional and named notation (mixed notation) to specify parameters. A simple example that demonstrates using mixed parameter notation follows:

```
CREATE OR REPLACE PROCEDURE emp_info (
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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(30, p_ename =>'Clark', p_empno=>7455);
```

If you do use mixed notation, remember that named arguments cannot precede positional arguments.

14.2.5.2 Parameter Modes

As previously discussed, a parameter has one of three possible modes - `IN, OUT` , or `IN OUT` . The following characteristics of a formal parameter are dependent upon its mode:

- Its initial value when the procedure or function is called.
- Whether or not 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.

Mode Property	IN	IN OUT	OUT
Formal parameter initialized to:	Actual parameter value	Actual parameter value	Actual parameter value
Formal parameter modifiable by the called program?	No	Yes	Yes
Actual parameter contains: (after normal called program termination)	Original actual parameter value prior to the call	Last value of the formal parameter	Last value of the formal parameter
Actual parameter contains: (after a handled exception in the called program)	Original actual parameter value prior to the call	Last value of the formal parameter	Last value of the formal parameter
Actual parameter contains: (after an unhandled exception in the called program)	Original actual parameter value prior to the call	Original actual parameter value prior to the call	Original actual parameter value prior to the call

As shown by the table, an `IN` formal parameter is initialized to the actual parameter with which it is called unless it was explicitly initialized with a default value. The `IN` parameter may be referenced within the called program, however, the called program may not assign a new value to the `IN` parameter. After control returns to the calling program, the actual parameter always contains the same value as it was set to prior to the call.

The `OUT` formal parameter is initialized to the actual parameter with which it is called. The called program may reference and assign new values to the formal parameter. If the called program terminates 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.

Like an `IN` parameter, an `IN OUT` formal parameter is initialized to the actual parameter with which it is called. Like an `OUT` parameter, an `IN OUT` formal parameter is modifiable by the called program and the last value in the formal parameter is passed to the calling program's actual parameter if the called program terminates 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.2.5.3 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.

The general form of a formal parameter declaration is:

```
(<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 do not include a `DEFAULT` clause, the caller must provide a value for the parameter.

The default value is evaluated every time the function or procedure is invoked. 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.

The following simple procedure demonstrates using the assignment operator to set a default value of `SYSDATE` into the parameter, `hiredate:`

```
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_empno)` . The third parameter `(p_hiredate)` defaults to the value of `SYSDATE:`

```
hire_emp (7575, Clark)
```

If you do include a value for the actual parameter when you call the procedure, that value takes precedence over the default value:

```
hire_emp (7575, Clark, 15-FEB-2010)
```

Adds a new 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:

```
CREATE OR REPLACE PROCEDURE hire_emp (
    p_empno         NUMBER,
    p_ename         VARCHAR2,
    p_hiredate      DATE DEFAULT SYSDATE
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
)
IS
BEGIN
    INSERT INTO emp(empno, ename, hiredate)
                   VALUES(p_empno, p_ename, p_hiredate);

    DBMS_OUTPUT.PUT_LINE('Hired!');
END hire_emp;
```

14.2.6 Subprograms – Subprocedures and Subfunctions

The capability and functionality of SPL procedure and function programs can be used in an advantageous manner to build well-structured and maintainable programs by organizing the SPL code into subprocedures and subfunctions.

The same SPL code can be invoked multiple times from different locations within a relatively large SPL program by declaring subprocedures and subfunctions within the SPL program.

Subprocedures and subfunctions have the following characteristics:

- The syntax, structure, and functionality of subprocedures and subfunctions are practically 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, these identifiers cannot be accessed nor altered 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. Within 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 cannot be done for lower level child subprograms in the hierarchy.

Subprocedures and subfunctions can be declared and invoked from within 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 within any of the preceding programs

The rules regarding subprocedure and subfunction structure and access are discussed in more detail in the next sections.

14.2.6.1 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 within the declaration section of that block and is therefore accessible by the SPL code within the executable section and optional exception section of that block.

Subprocedures can only be declared after all other variable, cursor, and type declarations included in the declaration section. (That is, subprograms must be the last set of declarations.)

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
PROCEDURE <name> [ (<parameters>) ]
{ IS | AS }
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]
    [ <declarations> ]
  BEGIN
    <statements>
  END [ <name> ];
```

Where:

`name`

`name` is the identifier of the subprocedure.

`parameters`

`parameters` is a list of formal parameters.

`PRAGMA AUTONOMOUS_TRANSACTION`

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the subprocedure as an autonomous transaction.

`declarations`

`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`

`statements` are SPL program statements (the `BEGIN - END` block may contain an `EXCEPTION` section).

Examples

The following example is a subprocedure within an anonymous block.

```
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;
BEGIN
    list_emp;
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
```

The following example is a subprocedure within a trigger.

```
 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;
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 enterprisedb added dept on 2016-07-26
```

14.2.6.2 Creating a Subfunction

The `FUNCTION` clause specified in the declaration section defines and names a subfunction local to that block.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 within the declaration section of that block and is therefore accessible by the SPL code within the executable section and optional exception section of that block.

```
 FUNCTION <name> [ (<parameters>) ]
RETURN <data_type>
{ IS | AS }
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]
    [ <declarations> ]
  BEGIN
    <statements>
  END [ <name> ];
```

Where:

`name`

`name` is the identifier of the subfunction.

`parameters`

`parameters` is a list of formal parameters.

`data_type`

`data_type` is the data type of the value returned by the function's `RETURN` statement.

`PRAGMA AUTONOMOUS_TRANSACTION`

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the subfunction as an autonomous transaction.

`declarations`

`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`

`statements` are SPL program statements (the `BEGIN - END` block may contain an `EXCEPTION` section).

Examples

The following example shows the use of a recursive subfunction:

```
DECLARE
    FUNCTION factorial (
        n           BINARY_INTEGER
    ) RETURN BINARY_INTEGER
    IS
    BEGIN
        IF n = 1 THEN
            RETURN n;
        ELSE
            RETURN n * factorial(n-1);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        END IF;
    END factorial;
BEGIN
    FOR i IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE(i || '! = ' || factorial(i));
    END LOOP;
END;
```

The output from the example is the following:

```
 1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

14.2.6.3 Block Relationships

This section describes the terminology of the relationship between blocks that can be declared in an SPL program. The ability to invoke subprograms and access identifiers declared within a block depends upon this relationship.

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 is declared within the declaration section of the given block. Such local identifiers are accessible from the executable section and optional exception section of the block.
- The *parent block* contains the declaration of another block (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 (or 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 within the declaration section of this procedure are all at the same level, for example call it level 1. Additional subprograms within the declaration section of the subprograms declared in the standalone procedure are at the next level, which 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 also always at the same level relative to each other.

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. There are three hierarchical paths 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` .

```
 CREATE PROCEDURE block_0
IS
        .
    +---- PROCEDURE block_1a    ------- Local to block_0
    |     IS
    |         .                          |
    |         .                          |
    |         .                          |
    |     +-- PROCEDURE block_2a   ---- Local to block_1a and descendant
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    |     |   IS                          of block_0
    |     |       .                      |
    |     |       .                      |
    |     |       .                      |
    |     |      PROCEDURE block_3   -- Local to block_2a and descendant
    |     |      IS                       of block_1a, and block_0
    | Siblings        .                  |
    |     |           .                  |
    |     |           .                  |
    |     |       END block_3;           |
    |     |   END block_2a;              |
    |     +-- PROCEDURE block_2b   ---- Local to block_1a and descendant
    |     |   IS                          of block_0
 Siblings |       ,                      |
    |     |       .                      |
    |     |       .                      |
    |     +-- END block_2b;              |
    |                                    |
    |      END block_1a;        ----------+
    +----  PROCEDURE block_1b;   ------- Local to block_0
    |      IS
    |          .                         |
    |          .                         |
    |          .                         |
    |          PROCEDURE block_2b   ---- Local to block_1b and descendant
    |          IS                          of block_0
    |              .                     |
    |              .                     |
    |              .                     |
    |          END block_2b;             |
    |                                    |
    +----  END block_1b;        ----------+
BEGIN
      .
      .
      .
END block_0;
```

The rules for invoking subprograms based upon block location is described starting with Invoking Subprograms. The rules for accessing variables based upon block location is described in Accessing Subprogram Variables.

14.2.6.4 Invoking Subprograms

A subprogram is invoked in the same manner as a standalone procedure or function by specifying its name and any actual parameters.

The subprogram may be invoked with none, one, or more qualifiers, which are the names of the parent subprograms or labeled anonymous blocks forming the ancestor hierarchy from where the subprogram has been declared.

The invocation is specified as a dot-separated list of qualifiers ending with the subprogram name and any of its arguments as shown by the following:

```
[[<qualifier_1>.][...]<qualifier_n>.]<subprog> [(<arguments>)]
```

If specified, `qualifier_n` is the subprogram in which `subprog` has been declared in its declaration section. The preceding list of qualifiers must reside in a continuous path up the hierarchy from `qualifier_n` to `qualifier_1. qualifier_1` may be any ancestor subprogram in the path as well as any of the following:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Standalone procedure name containing the subprogram
- Standalone function name containing subprogram
- Package name containing the subprogram
- Object type name containing the subprogram within 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_1` may not be a schema name, otherwise an error is thrown upon invocation of the subprogram. This Advanced Server restriction is not compatible with Oracle databases, which allow use of the schema name as a qualifier.

`arguments` is the list of actual parameters to be passed to the subprocedure or subfunction.

Upon invocation, 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 is invoked when the code is observed from top to bottom. (An exception to this requirement can be accomplished using a forward declaration. See Using Forward Declarations for information on forward declarations.)
- If the invocation list does not 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, never comparing the descendants of the sibling blocks.
- This comparison process continues up the hierarchies until the first complete match is found in which case the located subprogram is invoked. Note that the formal parameter list of the matched subprogram must comply with the actual parameter list specified for the invoked subprogram, otherwise an error occurs upon invocation of the subprogram.
- If no match is found after searching up to the standalone program, then an error is thrown upon invocation of the subprogram.

> **Note**
>
> The Advanced Server search algorithm for subprogram invocation is not quite 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 Advanced Server, a match is not found unless all qualifiers, the subprogram name, and the subprogram type of the invocation match the hierarchy content. If such an exact match is not initially found, Advanced Server continues the search progressing up the hierarchy.

The location of subprograms relative to the block from where the invocation is made can be accessed as follows:

- Subprograms declared in the local block can be invoked from the executable section or the exception section of the same block.
- Subprograms declared in the parent or other ancestor blocks can be invoked from the child block of the parent or other ancestors.
- Subprograms declared in sibling blocks can be called from a sibling block or from any descendent block of the sibling.

However, the following location of subprograms cannot be accessed 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.

The following examples illustrate the various conditions previously described.

Invoking Locally Declared Subprograms

The following example contains a single hierarchy of blocks contained within standalone procedure `level_0`. Within the executable section of procedure `level_1a`, the means of invoking the local procedure `level_2a` are shown, both with and without qualifiers.

Also note that access to the descendant of local procedure `level_2a`, which is procedure `level_3a`, is not permitted, with or without qualifiers. These calls are commented out in the example.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 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');
        level_2a;                               -- Local block called
        level_1a.level_2a;                      -- Qualified local block called
        level_0.level_1a.level_2a;              -- Double qualified local block called
--        level_3a;                             -- Error - Descendant of local block
--        level_2a.level_3a;                    -- 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, which indicates that procedure `level_2a` is successfully invoked from the calls in the executable section of 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 were to attempt to run procedure `level_0` with any of the calls to the descendent block uncommented, then an error occurs.

Invoking Subprograms Declared in Ancestor Blocks

The following example shows how subprograms can be invoked 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 procedure `level_2a`, which is its parent block. (Note that `v_cnt` is used to avoid an infinite loop.)

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 CREATE OR REPLACE PROCEDURE level_0
IS
  v_cnt           NUMBER(2) := 0;
  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;
```

The following is the resulting output:

```
 BEGIN
    level_0;
END;

BLOCK level_0
.. BLOCK level_1a
...... BLOCK level_2a
........ BLOCK level_3a
...... BLOCK level_2a
........ BLOCK level_3a
........ END BLOCK level_3a
...... END BLOCK level_2a
........ END BLOCK level_3a
...... END BLOCK level_2a
.. END BLOCK level_1a
END BLOCK level_0
```

In a similar example, the executable section of procedure `level_3a` invokes procedure `level_1a`, which is further up the ancestor hierarchy. (Note that `v_cnt` is used to avoid an infinite loop.)

```
 CREATE OR REPLACE PROCEDURE level_0
IS
    v_cnt           NUMBER(2) := 0;
    PROCEDURE level_1a
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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_1a;                 -- Ancestor 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;
```

The following is the resulting 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
........ END BLOCK level_3a
...... END BLOCK level_2a
.. END BLOCK level_1a
........ END BLOCK level_3a
...... END BLOCK level_2a
.. END BLOCK level_1a
END BLOCK level_0
```

Invoking Subprograms Declared in Sibling Blocks

The following examples show how subprograms can be invoked 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 procedure `level_1b` invokes procedure `level_1a`, which is its sibling block. Both are local to standalone procedure `level_0`.

Note that invocation of `level_2a` or equivalently, `level_1a.level_2a` from within procedure `level_1b` is commented out as this call would

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

result in an error. Invoking a descendent subprogram `(level_2a)` of sibling block `(level_1a)` is not permitted.

```
 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('.. END BLOCK level_1a');
    END level_1a;
    PROCEDURE level_1b
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
        level_1a;                           -- Sibling block called
--      level_2a;                           -- Error - Descendant of sibling block
--      level_1a.level_2a;                  -- Error - Descendant of sibling block
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
    END level_1b;
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    level_1b;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
```

The following is the resulting output:

```
 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
```

In the following example, procedure `level_1a`, which is the sibling of procedure `level_1b`, which is an ancestor of procedure `level_3b` is successfully invoked.

```
 CREATE OR REPLACE PROCEDURE level_0
IS
    PROCEDURE level_1a
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
    END level_1a;
    PROCEDURE level_1b
    IS
        PROCEDURE level_2b
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        IS
            PROCEDURE level_3b
            IS
            BEGIN
                DBMS_OUTPUT.PUT_LINE('........ BLOCK level_3b');
                level_1a;                       -- Ancestor's sibling block called
                level_0.level_1a;               -- Qualified ancestor's sibling block
                DBMS_OUTPUT.PUT_LINE('........ END BLOCK level_3b');
            END level_3b;
        BEGIN
            DBMS_OUTPUT.PUT_LINE('...... BLOCK level_2b');
            level_3b;                           -- Local block called
            DBMS_OUTPUT.PUT_LINE('...... END BLOCK level_2b');
        END level_2b;
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
        level_2b;                               -- Local block called
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
    END level_1b;
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    level_1b;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END level_0;
```

The following is the resulting output:

```
 BEGIN
    level_0;
END;

BLOCK level_0
.. BLOCK level_1b
...... BLOCK level_2b
........ BLOCK level_3b
.. BLOCK level_1a
.. END BLOCK level_1a
.. BLOCK level_1a
.. END BLOCK level_1a
........ END BLOCK level_3b
...... END BLOCK level_2b
.. END BLOCK level_1b
END BLOCK level_0
```

14.2.6.5 Using Forward Declarations

As discussed so far, when a subprogram is to be invoked, it must have been declared somewhere in the hierarchy of blocks within the standalone program, but prior to where it is invoked. In other words, when scanning the SPL code from beginning to end, the subprogram declaration must be found before its invocation.

However, there is a method of constructing the SPL code so that the full declaration of the subprogram (that is, its optional declaration section, its mandatory executable section, and optional exception section) appears in the SPL code after the point in the code where it is invoked.

This is accomplished 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 is a subfunction.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The full subprogram specification consisting of the optional declaration section, the executable section, and the optional exception section must be specified in the same declaration section as the forward declaration, but may appear following other subprogram declarations that invoke this subprogram with the forward declaration.

Typical usage of a forward declaration is when two subprograms invoke each other as shown by the following:

```
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`, so 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:

```
 Increase by 1
Increase by 1
Final value is 5
```

14.2.6.6 Overloading Subprograms

Generally, subprograms of the same type (subprocedure or subfunction) with the same name, and same formal parameter specification can appear multiple times within the same standalone program as long as they are not sibling blocks (that is, the subprograms are not declared in the same local block).

Each subprogram can be individually invoked depending upon the use of qualifiers and the location where the subprogram invocation is made as discussed in the previous sections.

It is however 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) is 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*.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Thus, the determination of which particular overloaded subprogram is to be invoked 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 are not aliases. Data type aliases are discussed later in this section.

Note that the following differences alone do not 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

As previously indicated, one of the differences allowing overloaded subprograms are different data types.

However, certain data types have alternative names referred to as *aliases*, which can be used for the table definition.

For example, there are fixed length character data types that can be specified as `CHAR` or `CHARACTER`. There are variable length character data types that can be specified 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 Advanced Server, seeData types.

Thus, when attempting to create overloaded subprograms, the formal parameter data types are not considered different if the specified data types are aliases of each other.

It can be determined if certain data types are aliases of other types by displaying the table definition containing the data types in question.

For example, the following table definition contains some data types and their aliases.

```
CREATE TABLE data_type_aliases (
    dt_BLOB             BLOB,
    dt_LONG_RAW         LONG RAW,
    dt_RAW              RAW(4),
    dt_BYTEA            BYTEA,
    dt_INTEGER          INTEGER,
    dt_BINARY_INTEGER   BINARY_INTEGER,
    dt_PLS_INTEGER      PLS_INTEGER,
    dt_REAL             REAL,
    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(4),
    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 upon its data type in the table definition:

```
\d data_type_aliases
       Column        |         Type         | Modifiers
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
--------------------+----------------------+-----------
 dt_blob            | bytea                |
 dt_long_raw        | bytea                |
 dt_raw             | bytea(4)             |
 dt_bytea           | bytea                |
 dt_integer         | integer              |
 dt_binary_integer  | integer              |
 dt_pls_integer     | integer              |
 dt_real            | real                 |
 dt_double_precision | double precision    |
 dt_float           | double precision     |
 dt_number          | numeric              |
 dt_decimal         | numeric              |
 dt_numeric         | numeric              |
 dt_char            | character(1)         |
 dt_character       | character(1)         |
 dt_varchar2        | character varying(4) |
 dt_char_varying    | character varying(4) |
 dt_varchar         | character varying(4) |
```

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 would not be sufficient to allow subprogram overloading. Thus, parameters with data types `INTEGER` and `PLS_INTEGER` cannot overload a pair of subprograms, but data types `INTEGER` and `REAL`, or `INTEGER` and `FLOAT`, or `INTEGER` and `NUMBER` can overload the subprograms.

> **Note**
>
> The overloading rules based upon formal parameter data types are not compatible with Oracle databases. Generally, the Advanced Server rules are more flexible, and certain combinations are allowed in Advanced Server that would result in an error when attempting to create the procedure or function in Oracle databases.

For certain pairs of data types used for overloading, casting of the arguments specified by the subprogram invocation may be required to avoid an error encountered during runtime of the subprogram. Invocation of a subprogram must include the actual parameter list that can specifically identify the data types. Certain pairs of overloaded data types may require the `CAST` function to explicitly identify data types. For example, pairs of overloaded data types that may require casting during the invocation are `CHAR` and `VARCHAR2`, or `NUMBER` and `REAL`.

The following example shows a group of overloaded subfunctions invoked from within 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.

```
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;
    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_2,999.9999);
    END add_it;
    FUNCTION add_it (
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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;
    FUNCTION add_it (
        p_add_1     IN DOUBLE PRECISION,
        p_add_2     IN DOUBLE PRECISION
    ) RETURN VARCHAR2
    IS
    BEGIN
        RETURN 'add_it DOUBLE PRECISION: ' || TO_CHAR(p_add_1 + p_add_2,9999.9999);
    END add_it;
BEGIN
    DBMS_OUTPUT.PUT_LINE(add_it (25, 50));
    DBMS_OUTPUT.PUT_LINE(add_it (25.3333, 50.3333));
    DBMS_OUTPUT.PUT_LINE(add_it (TO_NUMBER(25.3333), TO_NUMBER(50.3333)));
    DBMS_OUTPUT.PUT_LINE(add_it (CAST('25.3333' AS REAL), CAST('50.3333' AS REAL)));
    DBMS_OUTPUT.PUT_LINE(add_it (CAST('25.3333' AS DOUBLE PRECISION),
        CAST('50.3333' AS DOUBLE PRECISION)));
END;
```

The following is the output displayed from the anonymous block:

```
 add_it BINARY_INTEGER:    75.0000
add_it NUMBER:   75.6666
add_it NUMBER:   75.6666
add_it REAL:    75.6666
add_it DOUBLE PRECISION:    75.6666
```

14.2.6.7 Accessing Subprogram Variables

Variable declared in blocks such as subprograms or anonymous blocks can be accessed from the executable section or the exception section of other blocks depending upon their relative location.

Accessing a variable means being able to reference it within a SQL statement or an SPL statement as is done with any local variable.

> **Note**
>
> If the subprogram signature contains formal parameters, these may be accessed in the same manner as local variables of the subprogram. In this section, all discussion related to variables of a subprogram also applies to formal parameters of the subprogram.

Access of variables not only includes those defined as a data type, but also includes others such as record types, collection types, and cursors.

The variable may be accessed by at most one qualifier, which is the name of the subprogram or labeled anonymous block in which the variable has been locally declared.

The syntax to reference a variable is shown by the following:

```
[<qualifier>.]<variable>
```

If specified, `qualifier` is the subprogram or labeled anonymous block in which `variable` has been declared in its declaration section (that is, it is a local variable).

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> **Note**
>
> In Advanced Server, there is only one circumstance where two qualifiers are permitted. This scenario is for accessing public variables of packages where the reference can be specified in the following format:

```
schema_name.package_name.public_variable_name
```

For more information about supported package syntax, see the *Database Compatibility for Oracle Developers Built-In Package Guide*.

The following summarizes how variables can be accessed:

- Variables can be accessed as long as the block in which the variable has been locally declared is within 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 does not 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 is not found, then an error occurs upon invocation of the subprogram.
- If a reference to a qualified variable is made, the same search process is performed as described in the previous bullet point, 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 is not found, then an error occurs upon invocation of the subprogram.

The following location of variables cannot be accessed relative to the block from where the reference to the variable is made:

- Variables declared in a descendent block cannot be accessed,
- Variables declared in a sibling block, a sibling block of an ancestor block, or any descendants within the sibling block cannot be accessed.

> **Note**
>
> The Advanced Server process for accessing variables is not compatible with Oracle databases. For Oracle, any number of qualifiers can be specified and the search is based upon the first match of the first qualifier in a similar manner to the Oracle matching algorithm for invoking subprograms.

The following example displays how variables in various blocks are accessed, with and without qualifiers. The lines that are commented out illustrate attempts to access variables that would result in an error.

```
 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
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1a');
        level_2a;
--        DBMS_OUTPUT.PUT_LINE('.... v_level_2a: ' || v_level_2a);
--                             Error - Descendent block ----^
--        DBMS_OUTPUT.PUT_LINE('.... level_2a.v_level_2a: ' || level_2a.v_level_2a);
--                             Error - Descendent block ---------------^
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
    END level_1a;
    PROCEDURE level_1b
    IS
        v_level_1b  VARCHAR2(20) := 'Value from level_1b';
    BEGIN
        DBMS_OUTPUT.PUT_LINE('.. BLOCK level_1b');
        DBMS_OUTPUT.PUT_LINE('.... v_level_1b: ' || v_level_1b);
        DBMS_OUTPUT.PUT_LINE('.... v_level_0 : ' || v_level_0);
--        DBMS_OUTPUT.PUT_LINE('.... level_1a.v_level_1a: ' || level_1a.v_level_1a);
--                             Error - Sibling block -----------------^
--        DBMS_OUTPUT.PUT_LINE('.... level_2a.v_level_2a: ' || level_2a.v_level_2a);
--                             Error - Sibling block descendant ------^
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
    END level_1b;
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    DBMS_OUTPUT.PUT_LINE('.. v_level_0: ' || v_level_0);
    level_1a;
    level_1b;
    DBMS_OUTPUT.PUT_LINE('END 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_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
........ level_1a.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
.. END BLOCK level_1a
.. BLOCK level_1b
.... v_level_1b: Value from level_1b
.... v_level_0 : Value from level_0
.. END BLOCK level_1b
END BLOCK level_0
```

The following example shows similar access attempts when all variables in all blocks have the same name:

```
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');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
            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);
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
    END level_1b;
BEGIN
    DBMS_OUTPUT.PUT_LINE('BLOCK level_0');
    DBMS_OUTPUT.PUT_LINE('.. v_common: ' || v_common);
    level_1a;
    level_1b;
    DBMS_OUTPUT.PUT_LINE('END 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_0
.. v_common: Value from level_0
.. BLOCK level_1a
.... v_common: Value from level_1a
.... level_0.v_common: Value from level_0
...... 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_0
```

As previously discussed, the labels on anonymous blocks can also be used to qualify access to variables. The following example shows variable access within a set of nested anonymous blocks:

```
DECLARE
    v_common        VARCHAR2(20) := 'Value from level_0';
BEGIN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1a');
    END;
    <<level_1b>>
    DECLARE
        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_1b.v_common: ' || level_1b.v_common);
        DBMS_OUTPUT.PUT_LINE('.. END BLOCK level_1b');
    END;
    DBMS_OUTPUT.PUT_LINE('END BLOCK level_0');
END;
```

The following is the output showing the content of each variable when the anonymous block is invoked:

```
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_0
```

The following example is an object type whose object type method, `display_emp`, contains record type `emp_typ` and subprocedure `emp_sal_query`. Record variable `r_emp` declared locally to `emp_sal_query` is able to access the record type `emp_typ` declared in the parent block `display_emp`.

```
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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
        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;
    BEGIN
        emp_sal_query(SELF.empno);
    END;
END;
```

The following is the output displayed when an instance of the object type is created and procedure `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 #     : 30
Employee's salary does not exceed the department average of 1566.67
```

The following 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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
    )
    IS
        TYPE emp_rec_typ IS RECORD (
            empno           emp.empno%TYPE,
            ename           emp.ename%TYPE
        );
        TYPE emp_arr_typ IS TABLE OF emp_rec_typ INDEX BY BINARY_INTEGER;
        TYPE emp_cur_type IS REF CURSOR RETURN emp_rec_typ;
        PROCEDURE emp_by_dept (
            p_deptno        emp.deptno%TYPE
        )
        IS
            emp_arr         emp_arr_typ;
            emp_refcur      emp_cur_type;
            i               BINARY_INTEGER := 0;
            PROCEDURE display_emp_arr
            IS
            BEGIN
                DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
                DBMS_OUTPUT.PUT_LINE('-----    -------');
                FOR j IN emp_arr.FIRST .. emp_arr.LAST LOOP
                    DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
                        emp_arr(j).ename);
                END LOOP;
            END display_emp_arr;
        BEGIN
            OPEN emp_refcur FOR SELECT empno, ename FROM emp WHERE deptno = p_deptno;
            LOOP
                i := i + 1;
                FETCH emp_refcur INTO emp_arr(i).empno, emp_arr(i).ename;
                EXIT WHEN emp_refcur%NOTFOUND;
            END LOOP;
            CLOSE emp_refcur;
            display_emp_arr;
        END emp_by_dept;
    BEGIN
        emp_by_dept(p_deptno);
    END;
END;
```

The following is the output displayed when the top level package procedure is invoked:

```
BEGIN
    emp_dept_pkg.display_emp(20);
END;

EMPNO    ENAME
-----    -------
7369     SMITH
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
7566     JONES
7788     SCOTT
7876     ADAMS
7902     FORD
```

14.2.7 Compilation Errors in Procedures and Functions

When the Advanced Server parsers compile a procedure or function, they confirm that both the `CREATE` statement and the program body (that portion of the program that follows the `AS` keyword) conforms to the grammar rules for SPL and SQL constructs. By default, the server will terminate the compilation process if a parser detects an error. Note that the parsers detect syntax errors in expressions, but not semantic errors (i.e. an expression referencing a non-existent column, table, or function, or a value of incorrect type).

`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 may occur before the server halts the compilation process. For example:

```
SET spl.max_error_count = 6
```

The example instructs the server to continue past the first five SPL errors it encounters. When the server encounters the sixth error it will stop validating, and print six detailed error messages, and one error summary.

To save time when developing new code, or when importing existing code from another source, you may want to set the `spl.max_error_count` configuration parameter to a relatively high number of errors.

Please note that 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 may still be errors in any SPL or SQL code that follows the erroneous 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
LINE 4:     bonus := baseSalary * 1.10;
            ^
ERROR:  "total" is not a known variable
LINE 5:     total := bonus + 100;
            ^
ERROR:  compilation of SPL function/procedure "computebonus" failed due to 2 errors
```

The following example adds a `SELECT` statement to the previous example. The error in the `SELECT` statement masks the other errors that follow:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE FUNCTION computeBonus(employeeName number) RETURN number AS
BEGIN
    SELECT salary INTO baseSalary FROM emp
      WHERE ename = employeeName;

    bonus := baseSalary * 1.10;
    total := bonus + 100;

    RETURN bonus;

END;

ERROR:  "basesalary" is not a known variable
LINE 3:     SELECT salary INTO baseSalary FROM emp WHERE ename = emp...
```

14.2.8 Program Security

Security over what user may execute an SPL program and what database objects an SPL program may access for any given user executing the program is controlled by the following:

- Privilege to execute a program.
- Privileges granted on the database objects (including other SPL programs) which a program attempts to access.
- Whether the program is defined with definer's rights or invoker's rights.

These aspects are discussed in the following sections.

14.2.8.1 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, or
- The current user has been granted `EXECUTE` privilege on the SPL program, or
- The current user inherits `EXECUTE` privilege on the SPL program by virtue of being a member of a group which does have such privilege, or
- `EXECUTE` privilege has been granted to the `PUBLIC` group.

Whenever an SPL program is created in Advanced Server, `EXECUTE` privilege is automatically granted to the `PUBLIC` group by default, therefore, any user can immediately execute the program.

This default privilege can be removed by using the `REVOKE EXECUTE` command. The following is an example:

```
REVOKE EXECUTE ON PROCEDURE list_emp FROM PUBLIC;
```

Explicit `EXECUTE` privilege on the program can then be granted 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 do not meet any of the conditions listed at the beginning of this section cannot.

Once a program begins execution, the next aspect of security is what privilege checks occur if the program attempts to perform an action on any database object including:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- 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.

Note that it is possible for a database to have more than one object of the same type with the same name, but each such object belonging to a different schema in the database. If this is the case, which object is being referenced by an SPL program? This is the topic of the next section.

14.2.8.2 Database Object Name Resolution

A database object inside an SPL program may either be referenced 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 does not have the `schema.` portion. When a reference is made to a qualified name, there is absolutely no ambiguity as to exactly which database object is intended – it either does or does not 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 which are searched in left-to-right order for locating an unqualified database object reference. The object is considered non-existent if it can't be found in any of the schemas in the search path. The default search path can be displayed in PSQL using the `SHOW search_path` command.

```
edb=# SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)
```

`$user` in the above search path is a generic placeholder that refers to the current user so if the current user of the above session is `enterprisedb`, an unqualified database object would be searched for in the following schemas in this order – first, `enterprisedb`, then `public`.

Once an unqualified name has been resolved in the search path, it can be determined if the current user has the appropriate privilege to perform the desired action on that specific object.

> **Note**
>
> The concept of the search path is not compatible with Oracle databases. For an unqualified reference, Oracle simply looks in the schema of the current user for the named database object. It also important to note that in Oracle, a user and his or her schema is the same entity while in Advanced Server, a user and a schema are two distinct objects.

14.2.8.3 Database Object Privileges

Once an SPL program begins execution, any attempt to access a database object from within the program results in a check to ensure the current user has the authorization to perform the intended action against the referenced object. Privileges on database objects are bestowed and removed using the `GRANT` and `REVOKE` commands, respectively. If the current user attempts unauthorized access on a database object, then the program will throw an exception. See Exception Handling for information about exception handling.

14.2.8.4 Definer's vs. Invokers Rights

When an SPL program is about to begin execution, a determination is made as to what user is to be associated with this process. This user is referred to as

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

the *current user*. The current user's database object privileges are used to determine whether or not access to database objects referenced in the program will be permitted. The current prevailing search path in effect when the program is invoked will be used to resolve any unqualified object references.

The selection of the current user is influenced by whether the SPL program was created with definer's right or invoker's rights. The `AUTHID` clause determines that selection. Appearance of the clause `AUTHID DEFINER` gives the program definer's rights. This is also the default if the `AUTHID` clause is omitted. Use of the clause `AUTHID CURRENT_USER` gives the program invoker's rights. The difference between the two is summarized as follows:

- If a program has *definer's rights*, then the owner of the program becomes the current user when program execution begins. The program owner's database object privileges are used to determine if access to a referenced object is permitted. In a definer's rights program, it is irrelevant as to which user actually invoked the program.
- If a program has *invoker's rights*, then the current user at the time the program is called remains the current user while the program is executing (but not necessarily within called subprograms – see the following bullet points). When an invoker's rights program is invoked, the current user is typically the user that started the session (i.e., made the database connection) although it is possible to change the current user after the session has started using the SET ROLE command. In an invoker's rights program, it is irrelevant as to which user actually owns the program.

From the previous definitions, the following observations can be made:

- If a definer's rights program calls a definer's rights program, the current user changes from the owner of the calling program to the owner of the called program during execution of the called program.
- If a definer's rights program calls an invoker's rights program, the owner of the calling program remains the current user during execution of both the calling and called programs.
- If an invoker's rights program calls an invoker's rights program, the current user of the calling program remains the current user during execution of the called program.
- If an invokers' rights program calls a definer's rights program, the current user switches to the owner of the definer's rights program during execution of the called program.

The same principles apply if the called program in turn calls another program in the cases cited above.

14.2.8.5 Security Example

In the following example, a new database will be created along with two users – `hr_mgr` who will own a copy of the entire sample application in schema, `hr_mgr`; and `sales_mgr` who will own a schema named, `sales_mgr`, that will have 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` will be used in this example. All of the default privileges that are granted upon installation of the sample application will be removed and then be explicitly re-granted so as to present a more secure environment in this example.

Programs `list_emp` and `hire_clerk` will be changed from the default of definer's rights to invoker's rights. It will be then illustrated that when `sales_mgr` runs these programs, they act upon the `emp` table in `sales_mgr`'s schema since `sales_mgr`'s search path and privileges will be used for name resolution and authorization checking.

Programs `get_dept_name` and `hire_emp` in the `emp_admin` package will then be executed by `sales_mgr`. In this case, the `dept` table and `emp` table in `hr_mgr`'s schema will be 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:

```
CREATE DATABASE hr;
```

Switch to the hr database and create the users:

```
\c hr enterprisedb
CREATE USER hr_mgr IDENTIFIED BY password;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE USER sales_mgr IDENTIFIED BY password;
```

Step 2 – Create the Sample Application

Create the entire sample application, owned by `hr_mgr`, in `hr_mgr`'s schema.

```
\c - hr_mgr
\i /usr/edb/as11/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 `sales_mgr`'s schema.

```
\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';
```

In the above example, the `GRANT USAGE ON SCHEMA` command is given to allow `sales_mgr` access into `hr_mgr`'s schema to make a copy of `hr_mgr`'s `emp` table. This step is required in Advanced Server and is not compatible with Oracle databases since Oracle does not have the concept of a schema that is distinct from its user.

Step 4 – Remove Default Privileges

Remove all privileges to later illustrate the minimum required privileges needed.

```
\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;
REVOKE EXECUTE ON FUNCTION hire_clerk(VARCHAR2,NUMBER) FROM PUBLIC;
REVOKE EXECUTE ON PACKAGE emp_admin 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 Advanced Server. When performing this step, be sure you are logged on as `hr_mgr`, otherwise the modified program may wind up in the `public` schema instead of in `hr_mgr`'s schema.

```
CREATE OR REPLACE PROCEDURE list_emp
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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;
```

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` in order 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 are logged on as `hr_mgr`, otherwise the modified program may wind up in the `public` schema instead of in `hr_mgr`'s schema.

```
 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 empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
```

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. Note that 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
Employee No: 8000
Name       : JONES
Job        : CLERK
Manager    : 7782
Hire Date  : 08-NOV-07 00:00:00
Salary     : 950.00
*** After new employee added ***
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
8000     JONES
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The table and sequence accessed by the programs of the anonymous block are illustrated 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 within parenthesis in bold red font.

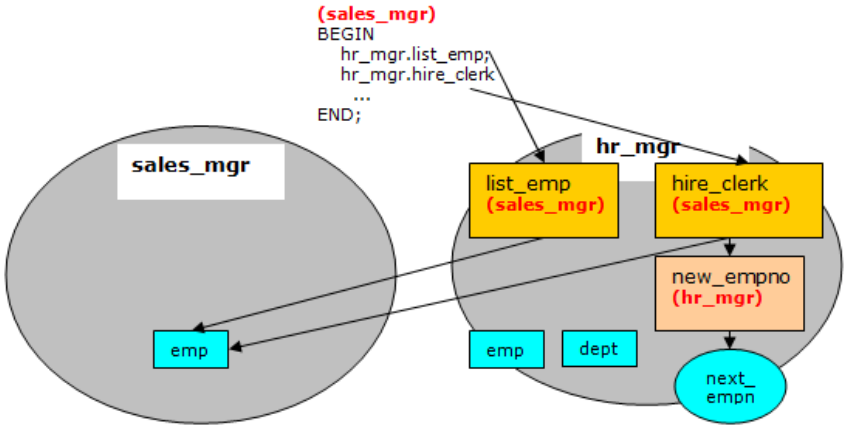


Fig. 1: Invokers Rights Programs

Selecting from `sales_mgr`'s `emp` table shows that the update was made in this table.

```
 SELECT empno, ename, hiredate, sal, deptno,
hr_mgr.emp_admin.get_dept_name(deptno) FROM sales_mgr.emp;

 empno | ename  |      hiredate       |   sal   | deptno | get_dept_name
-------+--------+---------------------+---------+--------+---------------
  7499 | ALLEN  | 20-FEB-81 00:00:00 | 1600.00 |     30 | SALES
  7521 | WARD   | 22-FEB-81 00:00:00 | 1250.00 |     30 | SALES
  7654 | MARTIN | 28-SEP-81 00:00:00 | 1250.00 |     30 | SALES
  7844 | TURNER | 08-SEP-81 00:00:00 | 1500.00 |     30 | SALES
  8000 | JONES  | 08-NOV-07 00:00:00 |  950.00 |     40 | OPERATIONS
(5 rows)
```

The following diagram shows that the `SELECT` command references the `emp` table in the `sales_mgr` schema, but 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.

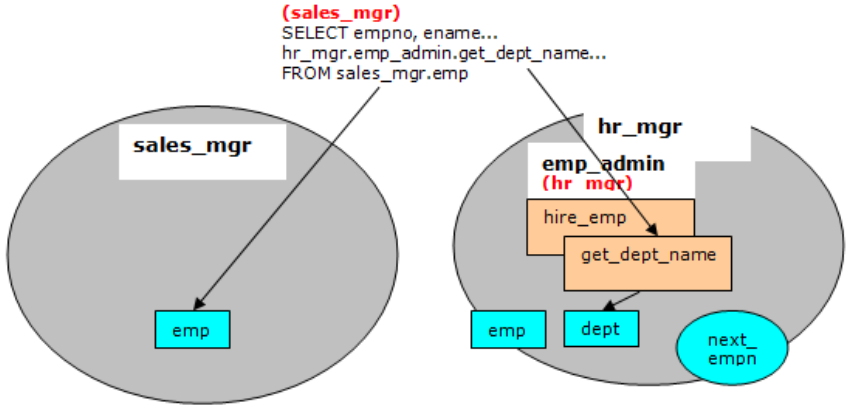


Fig. 2: Definer's Rights Package

Step 9 – Run Program hire_emp in the emp_admin Package

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 illustrates that the `hire_emp` procedure in the `emp_admin` definer's rights package updates the `emp` table belonging to `hr_mgr` since 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`.

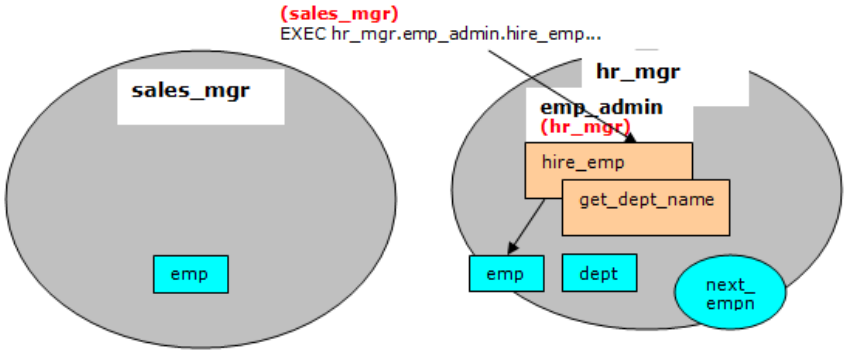


Fig. 3: Definer's Rights Package

Now connect as user, `hr_mgr`. The following `SELECT` command verifies that the new employee was added to `hr_mgr`'s `emp` table since the `emp_admin` package has definer's rights and `hr_mgr` is `emp_admin`'s owner.

```
 \c - hr_mgr
SELECT empno, ename, hiredate, sal, deptno,
hr_mgr.emp_admin.get_dept_name(deptno) FROM hr_mgr.emp;

empno  | ename  |      hiredate       |   sal   | deptno | get_dept_name
-------+--------+---------------------+---------+--------+---------------
  7369 | SMITH  | 17-DEC-80 00:00:00 |  800.00 |     20 | RESEARCH
  7499 | ALLEN  | 20-FEB-81 00:00:00 | 1600.00 |     30 | SALES
  7521 | WARD   | 22-FEB-81 00:00:00 | 1250.00 |     30 | SALES
  7566 | JONES  | 02-APR-81 00:00:00 | 2975.00 |     20 | RESEARCH
  7654 | MARTIN | 28-SEP-81 00:00:00 | 1250.00 |     30 | SALES
  7698 | BLAKE  | 01-MAY-81 00:00:00 | 2850.00 |     30 | SALES
  7782 | CLARK  | 09-JUN-81 00:00:00 | 2450.00 |     10 | ACCOUNTING
  7788 | SCOTT  | 19-APR-87 00:00:00 | 3000.00 |     20 | RESEARCH
  7839 | KING   | 17-NOV-81 00:00:00 | 5000.00 |     10 | ACCOUNTING
  7844 | TURNER | 08-SEP-81 00:00:00 | 1500.00 |     30 | SALES
  7876 | ADAMS  | 23-MAY-87 00:00:00 | 1100.00 |     20 | RESEARCH
  7900 | JAMES  | 03-DEC-81 00:00:00 |  950.00 |     30 | SALES
  7902 | FORD   | 03-DEC-81 00:00:00 | 3000.00 |     20 | RESEARCH
  7934 | MILLER | 23-JAN-82 00:00:00 | 1300.00 |     10 | ACCOUNTING
  9001 | ALICE  | 08-NOV-07 00:00:00 | 8000.00 |     40 | OPERATIONS
(15 rows)
```

14.3 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 can be used in SPL statements contained in the block.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.3.1 Declaring a Variable

Generally, all variables used in a block must be declared in the declaration section of the block. A variable declaration consists of a name that is assigned to the variable and its data type. Optionally, the variable can be initialized to a default value in the variable declaration.

The general syntax of a variable declaration is:

```
<name> <type> [ { := | DEFAULT } { <expression> | NULL } ];
```

`name` is an identifier assigned to the variable.

`type` 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 is not 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.

The following procedure illustrates some variable declarations that utilize defaults consisting of string and numeric expressions.

```
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        INTEGER := 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 above procedure shows that default values in the variable declarations are indeed 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
```

14.3.2 Using %TYPE in Variable Declarations

Often, variables will be declared in SPL programs that will be used to hold values from tables in the database. In order to ensure compatibility between the table columns and the SPL variables, the data types of the two should be the same.

However, as quite often happens, a change might be made to the table definition. If the data type of the column is changed, the corresponding change may be required to the variable in the SPL program.

Instead of coding the specific column data type into the variable declaration the column attribute, `%TYPE`, can be used instead. A qualified column name

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

in dot notation or the name of a previously declared variable must be specified 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 will be associated with the variable without the need to modify the declaration code.

> **Note**
>
> The `%TYPE` attribute can be used with formal parameter declarations as well.

```
<name> { { <table> | <view> }.<column> | <variable> }%TYPE;
```

`name` is the identifier assigned to the variable or formal parameter that is 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 does not inherit any of the column's other attributes such as might be specified on the column with the `NOT NULL` clause or the `DEFAULT` clause.

In the following 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, and then compares the chosen employee's salary with the department average.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN NUMBER
)
IS
    v_ename         VARCHAR2(10);
    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);

    SELECT AVG(sal) INTO v_avgsal
        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;
```

Instead of the above, the procedure could be written as follows without explicitly coding the `emp` table data types into the declaration section of the procedure.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_avgsal        v_sal%TYPE;
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);

    SELECT AVG(sal) INTO v_avgsal
        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;
```

> **Note**
>
> `p_empno` shows an example of a formal parameter defined using `%TYPE`.

`v_avgsal` illustrates the usage 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
```

14.3.3 Using %ROWTYPE in Record Declarations

The `%TYPE` attribute provides an easy way to create a variable dependent upon 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 do not inherit any of the columns' other attributes such as might be 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, and must be referenced using dot notation with the record name as its qualifier.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

```
<record> <table>%ROWTYPE;
```

`record` is an identifier assigned to the record. `table` is the name of a table (or view) whose columns are to define the fields in the record. The following example shows how the `emp_sal_query` procedure from the prior section can be modified to use `emp%ROWTYPE` to create a record named `r_emp` instead of declaring individual variables for the columns in `emp`.

```
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;
```

14.3.4 User-Defined Record Types and Record Variables

Records can be declared based upon a table definition using the `%ROWTYPE` attribute as shown in Using %ROWTYPE in Record Declarations. This section describes how a new record structure can be defined that is not tied to any particular table definition.

The `TYPE IS RECORD` statement is used to create the definition of a record type. A *record type* is a definition of a record comprised of one or more identifiers and their corresponding data types. A record type cannot, by itself, be used to manipulate data.

The syntax for a `TYPE IS RECORD` statement is:

```
TYPE <rec_type> IS RECORD ( <fields> )
```

Where `fields` is a comma-separated list of one or more field definitions of the following form:

```
<field_name> <data_type> [NOT NULL][{:= | DEFAULT} <default_value>]
```

Where:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`rec_type`

`rec_type` is an identifier assigned to the record type.

`field_name`

`field_name` is the identifier assigned to the field of the record type.

`data_type`

`data_type` specifies the data type of `field_name`.

`DEFAULT default_value`

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 no default is specified, then the default is `NULL`.

A *record variable* or simply put, a *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.

```
<record> <rectype>
```

`record` is an identifier assigned to the record variable. `rectype` is the identifier of a previously defined record type. Once declared, a record can then be used to hold data.

Dot notation is used to make reference to the fields in the record.

```
<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.

The `emp_sal_query` is again modified – this time using a user-defined record type and record variable.

```
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);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
```

Note that instead of specifying data type names, the `%TYPE` attribute can be used for the field data types in the record type definition.

The following is the output from executing this stored 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
```

14.4 Basic Statements

This section begins the discussion of the programming statements that can be used in an SPL program.

14.4.1 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 compatible data type with that of `variable`.

The following 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
```

14.4.2 DELETE

The `DELETE` command (available in the SQL language) can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL `DELETE` command. Thus, SPL variables and parameters can be used to supply values to the delete operation.

```
CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno         IN 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.

The following shows the deletion of an employee using this procedure.

```
EXEC emp_delete(9503);

Deleted Employee # : 9503

SELECT * FROM emp WHERE empno = 9503;

 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 rows)
```

> **Note**
>
> The `DELETE` command can be included in a `FORALL` statement. A `FORALL` statement allows a single `DELETE` command to delete multiple

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

rows from values supplied in one or more collections. See Using the FORALL Statement for more information on the `FORALL` statement.

14.4.3 INSERT

The `INSERT` command available in the SQL language can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL `INSERT` command. Thus, SPL variables and parameters can be used to supply values to the insert operation.

The following is an example of a procedure that performs an insert of a new employee using data passed from a calling program.

```
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);
    DBMS_OUTPUT.PUT_LINE('-----------------------');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('OTHERS exception on INSERT of employee # '
            || p_empno);
        DBMS_OUTPUT.PUT_LINE('SQLCODE : ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM : ' || SQLERRM);
END;
```

If an exception occurs all database changes made in the procedure are automatically 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 on exception handling.

The following shows the output when this procedure is executed.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 00:00:00
Salary     : 5000
Dept #     : 40
----------------------

SELECT * FROM emp WHERE empno = 9503;

 empno | ename  |  job    | mgr  |      hiredate     |   sal   | comm | deptno
-------+--------+---------+------+-------------------+---------+------+-------
  9503 |PETERSON| ANALYST| 7902 | 31-MAR-05 00:00:00| 5000.00 |      |   40
(1 row)
```

> **Note**
>
> The `INSERT` command can be included 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 on the `FORALL` statement.

14.4.4 NULL

The simplest statement is the `NULL` statement. This statement is an executable statement that does nothing.

```
NULL;
```

The following is the simplest, possible valid SPL program.

```
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:

```
CREATE OR REPLACE PROCEDURE divide_it (
    p_numerator     IN  NUMBER,
    p_denominator   IN  NUMBER,
    p_result        OUT NUMBER
)
IS
BEGIN
    IF p_denominator = 0 THEN
        NULL;
    ELSE
        p_result := p_numerator / p_denominator;
    END IF;
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.4.5 Using the RETURNING INTO Clause

The `INSERT, UPDATE`, and `DELETE` commands may be appended by 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.

The following is the 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 `*` is specified, 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. (Note that the use of `*` is an Advanced Server extension and is not 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**
>
> There is a variation of `RETURNING INTO` using the `BULK COLLECT` clause that allows a result set of more than one row that is returned into a collection. See Using the BULK COLLECT Clause for more information on the `BULK COLLECT` clause.

The following example is a modification of the `emp_comp_update` procedure introduced in UPDATE, with the addition of the `RETURNING INTO` clause.

```
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_comm          emp.comm%TYPE;
    v_deptno        emp.deptno%TYPE;
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
    RETURNING
        empno,
        ename,
        job,
        sal,
        comm,
        deptno
    INTO
        v_empno,
        v_ename,
        v_job,
        v_sal,
        v_comm,
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        v_deptno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || v_ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || v_job);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || v_deptno);
        DBMS_OUTPUT.PUT_LINE('New Salary         : ' || v_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission     : ' || v_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

The following is the output from this procedure (assuming employee `9503` created by the `emp_insert` procedure still exists within the table).

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Department         : 40
New Salary         : 6540.00
New Commission     : 1200.00
```

The following example is a modification of the `emp_delete` procedure, with the addition of the `RETURNING INTO` clause using record types.

```
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);
        DBMS_OUTPUT.PUT_LINE('Salary             : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Commission         : ' || r_emp.comm);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || r_emp.deptno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

The following is the output from this procedure.

```
EXEC emp_delete(9503);

Deleted Employee # : 9503
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
Name                : PETERSON
Job                 : ANALYST
Manager             : 7902
Hire Date           : 31-MAR-05 00:00:00
Salary              : 6540.00
Commission          : 1200.00
Department          : 40
```

14.4.6 SELECT INTO

The `SELECT INTO` statement is an SPL variation of the SQL `SELECT` command, the differences being:

- That `SELECT INTO` is designed to assign 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 the above, all of the clauses of the `SELECT` command such as `WHERE, ORDER BY, GROUP BY, HAVING`, etc. are valid for `SELECT INTO`. The following are the 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 has previously been declared.

If the query returns zero rows, null values are assigned to the target(s). If the query returns multiple rows, the first row is assigned to the target(s) and the rest are discarded. (Note that "the first row" is not well-defined unless you've used `ORDER BY.`)

> **Note**
>
> - In either cases, 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 is returned into a collection. See SELECT BULK COLLECT for more information on using the `BULK COLLECT` clause with the `SELECT INTO` statement.

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. Also note the addition of the `EXCEPTION` block containing the `WHEN NO_DATA_FOUND` conditional expression.

```
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 * INTO r_emp
        FROM emp WHERE empno = p_empno;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
END;
```

If the query is executed with a non-existent employee number the results appear as follows.

```
EXEC emp_sal_query(0);

Employee # 0 not found
```

Another conditional clause of use 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 an exception is thrown by SPL.

When the following block is executed, the `TOO_MANY_ROWS` exception is thrown since there are many employees in the specified department.

```
DECLARE
    v_ename         emp.ename%TYPE;
BEGIN
    SELECT ename INTO v_ename FROM emp WHERE deptno = 20 ORDER BY ename;
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee found');
        DBMS_OUTPUT.PUT_LINE('First employee returned is ' || v_ename);
END;

More than one employee found
First employee returned is ADAMS
```

> **Note**
>
> See Exception Handling for more information on exception handling.

14.4.7 UPDATE

The `UPDATE` command available in the SQL language can also be used in SPL programs.

An expression in the SPL language can be used wherever an expression is allowed in the SQL `UPDATE` command. Thus, SPL variables and parameters can be used to supply values to the update operation.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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;
```

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.

The following shows the update on the employee using this procedure.

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
New Salary         : 6540
New Commission     : 1200

SELECT * FROM emp WHERE empno = 9503;

empno | ename  |  job    | mgr  |      hiredate     |  sal    | comm  | deptno
------+--------+---------+------+-------------------+---------+-------+-------
 9503 |PETERSON| ANALYST| 7902 | 31-MAR-05 00:00:00|6540.00 |1200.00|  40
(1 row)
```

> **Note**
>
> The `UPDATE` command can be included 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 on the `FORALL` statement.

14.4.8 Obtaining the Result Status

There are several attributes that can be used 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.

The following anonymous block inserts a row and then displays the fact that the row has been inserted.

```
BEGIN
    INSERT INTO emp (empno,ename,job,sal,deptno) VALUES (
        9001, 'JONES', 'CLERK', 850.00, 40);
    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Row has been inserted');
    END IF;
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
Row has been 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;

# rows updated: 1
```

`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 = 9000;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows were updated');
    END IF;
END;

No rows were updated
```

14.5 Control Structures

The programming statements in SPL that make it a full procedural complement to SQL are described in this section.

14.5.1 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`

14.5.1.1 IF-THEN

```
IF boolean-expression THEN
  <statements>
END IF;
```

`IF-THEN` statements are the simplest form of `IF`. The statements between `THEN` and `END IF` will be executed if the condition is `TRUE`. Otherwise, they are skipped.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

In the following example an `IF-THEN` statement is used to test and display employees who have 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.

```
 EMPNO    COMM
-----    -------
7499     $300.00
7521     $500.00
7654    $1400.00
```

14.5.1.2 IF-THEN-ELSE

```
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 that should be executed if the condition evaluates to false.

The previous example is modified so an `IF-THEN-ELSE` statement is used to display the text `Non-commission` if the employee does not 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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'));
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || 'Non-commission');
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following is the output from this program.

```
 EMPNO    COMM
-----    -------
7369     Non-commission
7499  $   300.00
7521  $   500.00
7566     Non-commission
7654  $  1400.00
7698     Non-commission
7782     Non-commission
7788     Non-commission
7839     Non-commission
7844     Non-commission
7876     Non-commission
7900     Non-commission
7902     Non-commission
7934     Non-commission
```

14.5.1.3 IF-THEN-ELSE IF

`IF` statements can be nested so that alternative `IF` statements can be invoked once it is determined whether or not the conditional of an outer `IF` statement is `TRUE` or `FALSE`.

In the following example the outer `IF-THEN-ELSE` statement tests whether or not an employee has a commission. The inner `IF-THEN-ELSE` statements then test whether the employee's total compensation exceeds or is less than the company average.

```
 DECLARE
    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'));
    OPEN emp_cursor;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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 IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

> **Note**
>
> The logic in this program can be simplified considerably by calculating the employee's yearly compensation using the `NVL` function within the `SELECT` command of the cursor declaration, however, the purpose of this example is to demonstrate how `IF` statements can be used.

The following is the output from this program.

```
 Average Yearly Compensation: $  53,528.57
EMPNO    YEARLY COMP
-----    -----------
7369  $  19,200.00 Below Average
7499  $  45,600.00 Below Average
7521  $  42,000.00 Below Average
7566  $  71,400.00 Exceeds Average
7654  $  63,600.00 Exceeds Average
7698  $  68,400.00 Exceeds Average
7782  $  58,800.00 Exceeds Average
7788  $  72,000.00 Exceeds Average
7839  $ 120,000.00 Exceeds Average
7844  $  36,000.00 Below Average
7876  $  26,400.00 Below Average
7900  $  22,800.00 Below Average
7902  $  72,000.00 Exceeds Average
7934  $  31,200.00 Below Average
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

When you use this form, you are 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`.

14.5.1.4 IF-THEN-ELSIF-ELSE

```
IF boolean-expression THEN
  <statements>
[ ELSIF boolean-expression THEN
  <statements>
[ ELSIF boolean-expression THEN
  <statements> ] ...]
[ ELSE
  <statements> ]
END IF;
```

`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.

The following example uses an `IF-THEN-ELSIF-ELSE` statement to count the number of employees by compensation ranges of $25,000.

```
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;
    v_ge_100K       SMALLINT := 0;
    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;
        ELSIF v_comp < 50000 THEN
            v_25K_50K := v_25K_50K + 1;
        ELSIF v_comp < 75000 THEN
            v_50K_75K := v_50K_75K + 1;
        ELSIF v_comp < 100000 THEN
            v_75K_100K := v_75K_100K + 1;
        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 - 49,9999 : ' || v_25K_50K);
    DBMS_OUTPUT.PUT_LINE('50,000 - 74,9999 : ' || v_50K_75K);
    DBMS_OUTPUT.PUT_LINE('75,000 - 99,9999 : ' || v_75K_100K);
    DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;
```

The following is the output from this program.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,9999 : 5
50,000 - 74,9999 : 6
75,000 - 99,9999 : 0
100,000 and over : 1
```

14.5.2 RETURN Statement

The `RETURN` statement terminates the current function, procedure or anonymous block and returns control to the caller.

There are two forms of the `RETURN` Statement. The first form of the `RETURN` statement is used to terminate a procedure or function that returns `void`. The syntax of the first form is:

```
RETURN;
```

The second form of `RETURN` returns a value to the caller. The syntax of the second form of the `RETURN` statement is:

```
RETURN <expression>;
```

`expression` must evaluate to the same data type as the return type of the function.

The following example uses the `RETURN` statement returns a value to the caller:

```
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;
```

14.5.3 GOTO Statement

The `GOTO` statement causes the point of execution to jump to the statement with the specified label. The syntax of a `GOTO` statement is:

```
GOTO <label>
```

`label` is a name assigned to an executable statement. `label` must be unique within the scope of the function, procedure or anonymous block.

To label a statement, use the syntax:

```
<<label>> <statement>
```

`statement` is the point of execution that the program jumps to.

You can label assignment statements, any SQL statement (like `INSERT, UPDATE, CREATE`, etc.) and selected procedural language statements. The

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

procedural language statements that can be labeled are:

- `IF`
- `EXIT`
- `RETURN`
- `RAISE`
- `EXECUTE`
- `PERFORM`
- `GET DIAGNOSTICS`
- `OPEN`
- `FETCH`
- `MOVE`
- `CLOSE`
- `NULL`
- `COMMIT`
- `ROLLBACK`
- `GOTO`
- `CASE`
- `LOOP`
- `WHILE`
- `FOR`

Please note that `exit` is considered a keyword, and cannot be used as the name of a label.

`GOTO` statements cannot transfer control *into* a conditional block or sub-block, but can transfer control *from* a conditional block or sub-block.

The following 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 is not valid.

```
 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;
    <<invalid_emp>> DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' is not a valid employee.');
END;
```

`GOTO` statements have the following restrictions:

- A `GOTO` statement cannot jump to a declaration.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- A `GOTO` statement cannot transfer control to another function or procedure.
- A `label` should not be placed at the end of a block, function or procedure.

14.5.4 CASE Expression

The `CASE` expression returns a value that is substituted where the `CASE` expression is located within an expression.

There are two formats of the `CASE` expression - one that is called a *searched* `CASE` and the other that uses a *selector*.

14.5.4.1 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.

```
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 within 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.

The following example uses a selector `CASE` expression to assign the department name to a variable based upon the department number.

```
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 v_deptno
                WHEN 10 THEN 'Accounting'
                WHEN 20 THEN 'Research'
                WHEN 30 THEN 'Sales'
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
                WHEN 40 THEN 'Operations'
                ELSE 'unknown'
            END;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || v_dname);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following is the output from this program.

```
 EMPNO     ENAME      DEPTNO     DNAME
 -----     -------    ------     ----------
 7369      SMITH        20       Research
 7499      ALLEN        30       Sales
 7521      WARD         30       Sales
 7566      JONES        20       Research
 7654      MARTIN       30       Sales
 7698      BLAKE        30       Sales
 7782      CLARK        10       Accounting
 7788      SCOTT        20       Research
 7839      KING         10       Accounting
 7844      TURNER       30       Sales
 7876      ADAMS        20       Research
 7900      JAMES        30       Sales
 7902      FORD         20       Research
 7934      MILLER       10       Accounting
```

14.5.4.2 Searched CASE Expression

A searched `CASE` expression uses one or more Boolean expressions to determine the resulting value to return.

```
 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 within 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.

The following example uses a searched `CASE` expression to assign the department name to a variable based upon the department number.

```
 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 v_deptno = 10 THEN 'Accounting'
                WHEN v_deptno = 20 THEN 'Research'
                WHEN v_deptno = 30 THEN 'Sales'
                WHEN v_deptno = 40 THEN 'Operations'
                ELSE 'unknown'
            END;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || v_dname);
    END LOOP;
    CLOSE emp_cursor;
END;
```

The following is the output from this program.

```
 EMPNO    ENAME     DEPTNO    DNAME
-----    -------   ------    ----------
7369     SMITH       20      Research
7499     ALLEN       30      Sales
7521     WARD        30      Sales
7566     JONES       20      Research
7654     MARTIN      30      Sales
7698     BLAKE       30      Sales
7782     CLARK       10      Accounting
7788     SCOTT       20      Research
7839     KING        10      Accounting
7844     TURNER      30      Sales
7876     ADAMS       20      Research
7900     JAMES       30      Sales
7902     FORD        20      Research
7934     MILLER      10      Accounting
```

14.5.5 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 stand-alone statement in itself while the previously discussed `CASE` expression must appear as part of an expression.

There are two formats of the `CASE` statement - one that is called a *searched* `CASE` and the other that uses a *selector*.

14.5.5.1 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 CASE <selector-expression>
WHEN <match-expression> THEN
  <statements>
[ 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 within 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 statement(s) in the corresponding `THEN` clause are executed and control continues following the `END CASE` keywords. If there are no matches, the statement(s) following `ELSE` are executed. If there are no matches and there is no `ELSE` clause, an exception is thrown.

The following example uses a selector `CASE` statement to assign a department name and location to a variable based upon the department number.

```
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     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE v_deptno
            WHEN 10 THEN v_dname := 'Accounting';
                         v_loc   := 'New York';
            WHEN 20 THEN v_dname := 'Research';
                         v_loc   := 'Dallas';
            WHEN 30 THEN v_dname := 'Sales';
                         v_loc   := 'Chicago';
            WHEN 40 THEN v_dname := 'Operations';
                         v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                         v_loc   := '';
        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.

```
 EMPNO     ENAME      DEPTNO     DNAME           LOC
-----     -------    ------     ----------      ---------
7369      SMITH        20       Research        Dallas
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
7934     MILLER      10      Accounting     New York
```

14.5.5.2 Searched CASE Statement

A searched `CASE` statement uses one or more Boolean expressions to determine the resulting set of statements to execute.

```
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 within the `CASE` statement. When the first `boolean-expression` is encountered that evaluates to `TRUE`, the statement(s) in the corresponding `THEN` clause are executed and control continues following the `END CASE` keywords. If none of `boolean-expression` evaluates to `TRUE`, the statement(s) following `ELSE` are executed. If none of `boolean-expression` evaluates to `TRUE` and there is no `ELSE` clause, an exception is thrown.

The following example uses a searched `CASE` statement to assign a department name and location to a variable based upon the department number.

```
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     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE
            WHEN v_deptno = 10 THEN v_dname := 'Accounting';
                                    v_loc   := 'New York';
            WHEN v_deptno = 20 THEN v_dname := 'Research';
                                    v_loc   := 'Dallas';
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
            WHEN v_deptno = 30 THEN v_dname := 'Sales';
                                    v_loc   := 'Chicago';
            WHEN v_deptno = 40 THEN v_dname := 'Operations';
                                    v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                                    v_loc   := '';
        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.

```
 EMPNO     ENAME      DEPTNO     DNAME           LOC
-----     -------    ------     ----------      ---------
7369      SMITH        20       Research        Dallas
7499      ALLEN        30       Sales           Chicago
7521      WARD         30       Sales           Chicago
7566      JONES        20       Research        Dallas
7654      MARTIN       30       Sales           Chicago
7698      BLAKE        30       Sales           Chicago
7782      CLARK        10       Accounting      New York
7788      SCOTT        20       Research        Dallas
7839      KING         10       Accounting      New York
7844      TURNER       30       Sales           Chicago
7876      ADAMS        20       Research        Dallas
7900      JAMES        30       Sales           Chicago
7902      FORD         20       Research        Dallas
7934      MILLER       10       Accounting      New York
```

14.5.6 Loops

With the `LOOP, EXIT, CONTINUE, WHILE` , and `FOR` statements, you can arrange for your SPL program to repeat a series of commands.

14.5.6.1 LOOP

```
LOOP
    <statements>
END LOOP;
```

`LOOP` defines an unconditional loop that is repeated indefinitely until terminated by an `EXIT` or `RETURN` statement.

14.5.6.2 EXIT

```
EXIT [ WHEN <expression> ];
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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`.

`EXIT` can be used to cause early exit from all types of loops; it is not limited to use with unconditional loops.

The following is a simple example of a loop that iterates ten times and then uses the `EXIT` statement to terminate.

```
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 program.

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

14.5.6.3 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 is begun, 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 re-evaluated.

If the `WHEN` clause is used, then the next iteration of the loop is begun only if the specified expression in the `WHEN` clause evaluates to `TRUE`. Otherwise, control is passed to the next statement following the `CONTINUE` statement.

The `CONTINUE` statement may not be used outside of a loop.

The following is a variation of the previous example that uses the `CONTINUE` statement to skip the display of the odd numbers.

```
DECLARE
    v_counter       NUMBER(2);
BEGIN
    v_counter := 0;
    LOOP
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 10;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        CONTINUE WHEN MOD(v_counter,2) = 1;
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || v_counter);
    END LOOP;
END;
```

The following is the output from above program.

```
 Iteration # 2
Iteration # 4
Iteration # 6
Iteration # 8
Iteration # 10
```

14.5.6.4 WHILE

```
 WHILE <expression> LOOP
    <statements>
END LOOP;
```

The `WHILE` statement repeats a sequence of statements so long as the condition expression evaluates to `TRUE`. The condition is checked just before each entry to the loop body.

The following example contains the same logic as in the previous example except the `WHILE` statement is used to take the place of the `EXIT` statement to determine when to exit the loop.

> **Note**
>
> The conditional expression used to determine when to exit the loop must be altered. The `EXIT` statement terminates the loop when its conditional expression is true. The `WHILE` statement terminates (or never begins the loop) when its conditional expression is false.

```
 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 same result is generated by this example as in the prior example.

```
 Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.5.6.5 FOR (integer variant)

```
FOR <name> IN [REVERSE] <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 automatically defined as 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 and `name` begins with the value of `expression` to the left of .. and terminates once `name` exceeds the value of `expression` to the right of ... Thus the two expressions take on the following roles: `start-value.. end-value.`

The optional `REVERSE` clause specifies that the loop should 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`.

The following example simplifies the `WHILE` loop example even further by using 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;
```

Here is the output 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 is not executed at all. No error is raised as shown by the following example.

```
BEGIN
    FOR i IN 10 .. 1 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
    END LOOP;
END;
```

There is no output from this example as the loop body is never executed.

> **Note**
>
> SPL also supports `CURSOR` FOR loops (see Cursor FOR Loop).

14.5.7 Exception Handling

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

By default, any error occurring in an SPL program aborts execution of the program. You can trap errors and recover from them by using a `BEGIN` block with an `EXCEPTION` section. The syntax is an extension of the normal syntax for a `BEGIN` block:

```
[ DECLARE
    <declarations> ]
  BEGIN
    <statements>
  EXCEPTION
    WHEN <condition> [ OR <condition> ]... THEN
      <handler_statements>
  [ WHEN <condition> [ OR <condition> ]... THEN
      <handler_statements> ]...
END;
```

If no error occurs, this form of block simply executes all the `statements`, and then control passes to the next statement after `END`. If an error occurs within 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 that occurred. 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 were not there at all. The error can be caught by an enclosing block with `EXCEPTION`; if there is no enclosing block, it aborts processing of the subprogram.

The special condition name `OTHERS` matches every error type. Condition names are not case-sensitive.

If a new error occurs within the selected `handler_statements`, it cannot be caught by this `EXCEPTION` clause, but is propagated out. A surrounding `EXCEPTION` clause could catch it.

The following table lists the condition names that may be used:

Condition Name	Description
`CASE_NOT_FOUND`	The application has encountered a situation where none of the cases in `CASE` statement evaluates to `TRUE` and there is no `ELSE` condition.
`COLLECTION_IS_NULL`	The application has attempted to invoke a collection method on a null collection such as an uninitialized nested table.
`CURSOR_ALREADY_OPEN`	The application has attempted to open a cursor that is already open.
`DUP_VAL_ON_INDEX`	The application has attempted to store a duplicate value that currently exists within a constrained column.
`INVALID_CURSOR`	The application has attempted to access an unopened cursor.
`INVALID_NUMBER`	The application has 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 has encountered an exception that hasn't been caught by a prior condition in the exception section.
`SUBSCRIPT_BEYOND_COUNT`	The application has attempted to reference a subscript of a nested table or varray beyond its initialized or extended size.
`SUBSCRIPT_OUTSIDE_LIMIT`	The application has attempted to reference a subscript or extend a varray beyond its maximum size limit.
`TOO_MANY_ROWS`	The application has encountered more than one row that satisfies the selection criteria (where only one row is allowed to be returned).
`VALUE_ERROR`	The application has encountered a data exception (equivalent to SQLSTATE class code 22). `VALUE_ERROR` is an alias for `INVALID_NUMBER`.
`ZERO_DIVIDE`	The application has tried to divide by zero.
`User-defined Exception`	See User-defined Exceptions.

> **Note**

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> Condition names `INVALID_NUMBER` and `VALUE_ERROR` are not compatible with Oracle databases for which these condition names are for exceptions resulting only from a failed conversion of a string to a numeric literal. In addition, for Oracle databases, an `INVALID_NUMBER` exception is applicable only to SQL statements while a `VALUE_ERROR` exception is applicable only to procedural statements.

14.5.8 User-defined Exceptions

Any number of errors (referred to in PL/SQL as *exceptions*) can occur during program execution. 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* may be a pre-defined error that is generated by the server, or may be a logical error that raises a user-defined exception.

User-defined exceptions are never raised by the server; 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 will provoke a user-defined exception.

You can define exceptions in functions, procedures, packages or anonymous blocks. While you cannot declare the same exception twice in the same block, you can declare the same exception in two different blocks.

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:

```
DECLARE
    <exception_name> EXCEPTION;

BEGIN
    ...
    RAISE <exception_name>;
    ...
END;
```

`exception_name` is the name of the exception.

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 nested blocks within 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 the block name `block_name.exception_name`.

Conversely, outer blocks cannot reference exceptions declared in nested blocks.

The scope of a declaration is limited to the block in which it is declared *unless* it is 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`.

The following example demonstrates declaring a user-defined exception in a package. The user-defined exception does not require a package-qualifier when it is 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  BEGIN
      IF (p_amount > p_balance) THEN
        RAISE overdrawn;
      END IF;
   END;
```

The following 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` is not defined within the `ar` package.

```
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;
```

When `ar.check_balance` raises an exception, execution jumps to the exception handler defined in `purchase:`

```
EXCEPTION
     WHEN ar.overdrawn THEN
       raise_credit_limit(customerid, amount*1.5);
```

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`.

14.5.9 PRAGMA EXCEPTION_INIT

`PRAGMA EXCEPTION_INIT` associates a user-defined error code with an exception. A `PRAGMA EXCEPTION_INIT` declaration may be included in any block, sub-block or package. You can only assign an error code to an exception (using `PRAGMA EXCEPTION_INIT`) after declaring the exception. The format of a `PRAGMA EXCEPTION_INIT` declaration is:

```
PRAGMA EXCEPTION_INIT(<exception_name>,
                      {<exception_number> | <exception_code>})
```

Where:

`exception_name` is the name of the associated exception.

`exception_number` is a user-defined error code associated with the pragma. If you specify an unmapped `exception_number`, the server will return a warning.

`exception_code` is the name of a pre-defined exception. For a complete list of valid exceptions, see the Postgres core documentation available at:

https://www.postgresql.org/docs/11/static/errcodes-appendix.html

The previous section (*User-defined Exceptions*) included an example that demonstrates declaring a user-defined exception in a package. The following example uses the same basic structure, but adds a `PRAGMA EXCEPTION_INIT` declaration:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PRAGMA EXCEPTION_INIT (overdrawn, -20100);
  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;
```

The following 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.

```
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;
```

When `ar.check_balance` raises an exception, execution jumps to the exception handler defined in `purchase`.

```
EXCEPTION
     WHEN ar.overdrawn THEN
      DBMS_OUTPUT.PUT_LINE ('This account is overdrawn.');
      DBMS_OUTPUT.PUT_LINE ('SQLCode :'||SQLCODE||' '||SQLERRM );
```

The exception handler returns an error message, followed by `SQLCODE` information:

```
This account is overdrawn.
SQLCODE: -20100 User-Defined Exception
```

The following example demonstrates using a pre-defined exception. The code creates a more meaningful name for the `no_data_found exception`; if the given customer does not exist, the code catches the exception, calls `DBMS_OUTPUT.PUT_LINE` to report the error, and then re-raises the original exception:

```
CREATE OR REPLACE PACKAGE ar AS
  overdrawn EXCEPTION;
  PRAGMA EXCEPTION_INIT (unknown_customer, no_data_found);
  PROCEDURE check_balance(p_customer_id NUMBER);
END;

CREATE OR REPLACE PACKAGE BODY ar AS
   PROCEDURE check_balance(p_customer_id NUMBER)
   IS
   DECLARE
     v_balance NUMBER;
   BEGIN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
```

14.5.10 RAISE_APPLICATION_ERROR

The procedure, `RAISE_APPLICATION_ERROR`, allows a developer to intentionally abort processing within an SPL program from which it is called 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 can then be used to identify the exception.

```
RAISE_APPLICATION_ERROR(<error_number>, <message>);
```

Where:

`error_number` is an integer value or expression that is returned in a variable named `SQLCODE` when the procedure is executed. `error_number` must be a value between `-20000` and `-20999`.

`message` is a string literal or expression that is returned in a variable named `SQLERRM`.

For additional information on the `SQLCODE` and `SQLERRM` variables, see Errors and Messages.

The following example uses the `RAISE_APPLICATION_ERROR` procedure to display a different code and message depending upon the information missing from an employee.

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno         NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_mgr           emp.mgr%TYPE;
    v_hiredate      emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, mgr, hiredate
        INTO v_ename, v_job, v_mgr, 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_mgr 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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.

```
 EXEC verify_emp(7839);

SQLCODE: -20030
SQLERRM: EDB-20030: No manager for 7839
```

14.6 Transaction Control

There may be circumstances where it is desired that all updates to a database are to occur successfully, or none are to occur at all if any error occurs. A set of database updates that are to all occur successfully as a single unit, or are not to occur at all, is said to be 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 otherwise the bank's books will be out of balance. The deposit and withdrawal are one transaction.

An SPL application can be created that uses a style of transaction control compatible with Oracle databases if the following conditions are met:

- The `edb_stmt_level_tx` parameter must be set to `TRUE`. This prevents the action of unconditionally rolling back all database updates within the `BEGIN/END` block if any exception occurs.
- The application must not be running in autocommit mode. If autocommit mode is on, each successful database update is immediately committed and cannot be undone. The manner in which autocommit mode is turned on or off is application dependent.

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 which 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 which case the effect of all database updates made during the transaction become permanent.
- A `ROLLBACK` command is encountered in which 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, PSQL, etc.) in which case the action of the application determines whether the transaction is committed or rolled back unless the transaction is within a block in which `PRAGMA AUTONOMOUS_TRANSACTION` has been declared in which case the commitment or rollback of the transaction occurs independently of the calling program.

> **Note**
>
> Unlike Oracle, DDL commands such as `CREATE TABLE` do not implicitly occur within their own transaction. Therefore, DDL commands do not automatically cause an immediate database commit as in Oracle, and DDL commands may be rolled back just like DML commands.

A transaction may span one or more `BEGIN/END` blocks, or a single `BEGIN/END` block may contain one or more transactions.

The following sections discuss the `COMMIT` and `ROLLBACK` commands in more detail.

14.6.1 COMMIT

The `COMMIT` command makes all database updates made during the current transaction permanent, and ends the current transaction.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
COMMIT [ WORK ];
```

The `COMMIT` command may be used within anonymous blocks, stored procedures, or functions. Within an SPL program, it may appear in the executable section and/or the exception section.

In the following example, the third `INSERT` command in the anonymous block results in an error. The effect of the first two `INSERT` commands are 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.

> **Note**
>
> The `edb_stmt_level_tx` configuration parameter shown in the example below can be set for the entire database using the `ALTER DATABASE` command, or it can be set for the entire database server by changing it in the `postgresql.conf` file.

```
\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 (70, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

SQLERRM: value too long for type character varying(14)
SQLCODE: 22001

SELECT * FROM dept;

deptno  |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | FINANCE    | DALLAS
     60 | MARKETING  | CHICAGO
(6 rows)

ROLLBACK;

SELECT * FROM dept;

deptno  |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | FINANCE    | DALLAS
     60 | MARKETING  | CHICAGO
(6 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.6.2 ROLLBACK

The `ROLLBACK` command undoes all database updates made during the current transaction, and ends the current transaction.

```
ROLLBACK [ WORK ];
```

The `ROLLBACK` command may be used within anonymous blocks, stored procedures, or functions. Within an SPL program, it may appear in the executable section and/or the exception section.

In the following example, the exception section contains a `ROLLBACK` command. Even though the first two `INSERT` commands are executed successfully, the third results in an exception that results in the rollback of all the `INSERT` commands in the anonymous block.

```
 \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');
    INSERT INTO dept VALUES (70, 'HUMAN RESOURCES', 'CHICAGO');
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

SQLERRM: value too long for type character varying(14)
SQLCODE: 22001

SELECT * FROM dept;

deptno  |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

The following is a more complex example using both `COMMIT` and `ROLLBACK`. First, the following stored procedure is created which 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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);
    DBMS_OUTPUT.PUT_LINE('----------------------');
END;
```

Note that this procedure has no exception section so any error that may occur is propagated up to the calling program.

The following anonymous block is run. Note the use of the `COMMIT` command 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 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)
```

Now, 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
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 yields the following:

```
 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)
```

The `ROLLBACK` command in the exception section successfully undoes the insert of employee Harrison. Also note that 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 will throw an error if there is an Oracle-style SPL procedure on the runtime stack.

14.6.3 PRAGMA AUTONOMOUS_TRANSACTION

An SPL program can be declared as an autonomous transaction by specifying the following directive in the declaration section of the SPL block:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

An *autonomous transaction* is an independent transaction started by a calling program. A commit or rollback of SQL commands within 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.

The following SPL programs can include `PRAGMA AUTONOMOUS_TRANSACTION:`

- Standalone procedures and functions
- Anonymous blocks
- Procedures and functions declared as subprograms within 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 is in progress. In some cases, this may mean that the `max_connections` parameter in the `postgresql.conf` file should be raised.
- In most respects, an autonomous transaction behaves exactly as if it was a completely separate session, but GUCs (that is, 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 within a single session.
- Parallel query is not supported within autonomous transactions.
- The Advanced Server implementation of autonomous transactions is not entirely compatible with Oracle databases in that the Advanced Server autonomous transaction does not produce an ERROR if there is an uncommitted transaction at the end of an SPL block.

The following set of examples illustrates the usage of autonomous transactions. This first set of scenarios show the default behavior when there are no autonomous transactions.

Before each scenario, the `dept` table is reset to the following initial values:

```
SELECT * FROM dept;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

Scenario 1a – No autonomous transactions with only a final COMMIT

This first set of scenarios show the insertion of three rows starting just after the initial `BEGIN` command of the transaction, then from within an anonymous block within the starting transaction, and finally from a stored procedure executed from within the anonymous block.

The stored procedure is the following:

```
CREATE OR REPLACE PROCEDURE insert_dept_70 IS
BEGIN
    INSERT INTO dept VALUES (70,'MARKETING','LOS ANGELES');
END;
```

The PSQL session is the following:

```
BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
BEGIN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
    insert_dept_70;
END;
COMMIT;
```

After the final commit, all three rows are inserted:

```
 SELECT * FROM dept ORDER BY 1;

 deptno |   dname    |     loc
--------+------------+-------------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | HR         | DENVER
     60 | FINANCE    | CHICAGO
     70 | MARKETING  | LOS ANGELES
(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:

```
 BEGIN;
INSERT INTO dept VALUES (50,'HR','DENVER');
BEGIN
    INSERT INTO dept VALUES (60,'FINANCE','CHICAGO');
    insert_dept_70;
END;
ROLLBACK;

SELECT * FROM dept ORDER BY 1;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

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;

 deptno |   dname    |   loc
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)
```

This 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;
```

Now, the `PRAGMA AUTONOMOUS_TRANSACTION` is given with the anonymous block along with the `COMMIT` command at the end of the anonymous block.

```
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;
```

After the `ROLLBACK` at the end of the transaction, only the first row insertion at the very beginning of the transaction is discarded. The other two row insertions within the anonymous block with `PRAGMA AUTONOMOUS_TRANSACTION` have been independently committed.

```
SELECT * FROM dept ORDER BY 1;

 deptno |   dname    |     loc
--------+------------+-------------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     60 | FINANCE    | CHICAGO
     70 | MARKETING  | LOS ANGELES
(6 rows)
```

Scenario 2b – Autonomous transaction anonymous block with COMMIT including procedure with ROLLBACK, but not an autonomous transaction procedure

Now, the procedure has the `ROLLBACK` command at the end. Note, however, that the `PRAGMA ANONYMOUS_TRANSACTION` is not included in this procedure.

```
CREATE OR REPLACE PROCEDURE insert_dept_70 IS
BEGIN
    INSERT INTO dept VALUES (70,'MARKETING','LOS ANGELES');
    ROLLBACK;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
END;
```

Now, the rollback within the procedure removes the two rows inserted within the anonymous block (`deptno` 60 and 70) before the final `COMMIT` command within the anonymous block.

```
 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;
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 within the enclosed procedure has no effect on the insertion that occurs before the anonymous block is executed.

```
 SELECT * FROM dept ORDER by 1;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | HR         | DENVER
(5 rows)
```

Scenario 2c – Autonomous transaction anonymous block with COMMIT including procedure with ROLLBACK that is also an autonomous transaction procedure

Now, the procedure with the `ROLLBACK` command at the end also has `PRAGMA ANONYMOUS_TRANSACTION` included. This isolates the effect of the `ROLLBACK` command within the procedure.

```
 CREATE OR REPLACE PROCEDURE insert_dept_70 IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO dept VALUES (70,'MARKETING','LOS ANGELES');
    ROLLBACK;
END;
```

Now, the rollback within the procedure removes the row inserted by the procedure, but not the other row inserted within the anonymous block.

```
 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;
COMMIT;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 within the procedure.

```
SELECT * FROM dept ORDER by 1;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     50 | HR         | DENVER
     60 | FINANCE    | CHICAGO
(6 rows)
```

The following sections now show examples of `PRAGMA AUTONOMOUS_TRANSACTION` in a couple of other SPL program types.

Autonomous Transaction Trigger

The following 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. Note the inclusion of `PRAGMA AUTONOMOUS_TRANSACTION` 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 within 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SELECT * FROM emp WHERE empno > 9000;

 empno | ename |   job   | mgr  |      hiredate       | sal | comm | deptno
-------+-------+---------+------+---------------------+-----+------+--------
  9001 | SMITH | ANALYST | 7782 | 23-AUG-18 07:12:27 |     |      |     10
  9002 | JONES | CLERK   | 7782 | 23-AUG-18 07:12:27 |     |      |     10
(2 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)
```

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 as 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

The following 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 within 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    END;
END;
```

In the following anonymous block, an insert is performed into the `dept` table, followed by invocation of the `insert_dept` method of the object, 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;
```

Since `insert_dept` has been declared as an autonomous transaction, its insert of department number 60 remains in the table, but the rollback removes the insertion of department 50.

```
 SELECT * FROM dept ORDER BY 1;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
     60 | FINANCE    | CHICAGO
(5 rows)
```

14.7 Dynamic SQL

Dynamic SQL is a technique that provides the ability to execute SQL commands that are not known until the commands are about to be executed. Up to this point, the SQL commands that have been illustrated in SPL programs have been static SQL - the full command (with the exception of variables) must be known and coded into the program before the program, itself, can begin to execute. Thus 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 within an SPL program.

Note, however, that the runtime performance of dynamic SQL will be slower than static SQL.

The `EXECUTE IMMEDIATE` command is used to run SQL commands dynamically.

```
EXECUTE IMMEDIATE '<sql_expression>;'
  [ INTO { <variable> [, ...] | <record> } ]
  [ USING <expression> [, ...] ]
```

`sql_expression` is a string expression containing the SQL command to be dynamically executed. `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, order, and be type-compatible with the fields of the result set. Alternatively, a `record` can be specified as long as the record's fields match the number, 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 within the SQL command in `sql_expression` where variables may be used. Placeholders are denoted by an identifier with a colon (:) prefix - `:name`. The number, order, and resultant data types of the evaluated expressions must match the number, order and be type-compatible with the placeholders in

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`sql_expression`. Note that placeholders are not declared anywhere in the SPL program – they only appear in `sql_expression`.

The following example shows basic dynamic SQL commands as string literals.

```
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'')';
    EXECUTE IMMEDIATE v_sql;
    v_sql := 'INSERT INTO job VALUES (200, ''CLERK'')';
    EXECUTE IMMEDIATE v_sql;
END;
```

The following example illustrates 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;
```

The following example shows both the `INTO` and `USING` clauses. Note 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 100;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 200;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 300;
    DBMS_OUTPUT.PUT_LINE(v_jobno || '      ' || v_jname);
    EXECUTE IMMEDIATE v_sql INTO v_jobno, v_jname USING 400;
    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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following is the output from the previous anonymous block:

```
 JOBNO     JNAME
-----     -------
100       ANALYST
200       CLERK
300       MANAGER
400       SALESMAN
500       PRESIDENT
```

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 information about using the `BULK COLLECT` clause.

14.8 Static Cursors

Rather than executing a whole query at once, it is possible to set up a *cursor* that encapsulates the query, and then read the query result set one row at a time. This allows the creation of 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. A conditional test should be included in the SPL logic that detects when the end of the result set has been reached so the program can exit the loop.

14.8.1 Declaring a Cursor

In order to use a cursor, it must first be declared in the declaration section of the SPL program. A cursor declaration appears as follows:

```
CURSOR <name> IS <query>;
```

`name` is an identifier that will be 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. This is discussed in more detail in Parameterized Cursors.

The following are some examples of cursor declarations:

```
 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;
```

14.8.2 Opening a Cursor

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Before a cursor can be used to retrieve rows, it must first be opened. This is accomplished with the `OPEN` statement.

```
OPEN <name>;
```

`name` is the identifier of a cursor that has been previously declared in the declaration section of the SPL program. The `OPEN` statement must not be executed on a cursor that has already been, and still is open.

The following shows an `OPEN` statement with its corresponding cursor declaration.

```
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;
```

14.8.3 Fetching Rows From a Cursor

Once a cursor has been opened, rows can be retrieved from the cursor's result set by using the `FETCH` statement.

```
FETCH <name> INTO { <record> | <variable> [, <variable_2> ]... };
```

`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 will 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**
>
> There is a variation of `FETCH INTO` using the `BULK COLLECT` clause that can return multiple rows at a time into a collection. See Section Using the BULK COLLECT Clause for more information on using the `BULK COLLECT` clause with the `FETCH INTO` statement.

The following shows the `FETCH` statement.

```
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, `%TYPE` can be used instead. In this way, if the data type of the database column is changed, the target variable declaration in the SPL program does not have to be changed. `%TYPE` will automatically pick up the new data type of the specified column.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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, `%ROWTYPE` can be used to define a record into which the `FETCH` statement will place the retrieved data. Each field within the record can then be accessed using dot notation.

```
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);
        ...
END;
```

14.8.4 Closing a Cursor

Once all the desired rows have been retrieved from the cursor result set, the cursor must be closed. Once closed, the result set is no longer accessible. The `CLOSE` statement appears as follows:

```
CLOSE <name>;
```

`name` is the identifier of a cursor that is currently open. Once a cursor is closed, it must not be closed again. However, once the cursor is closed, the `OPEN` statement can be issued again on the closed cursor and the query result set will be rebuilt after which the `FETCH` statement can then be used to retrieve the rows of the new result set.

The following example illustrates the use of 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 when invoked. Employee number `7369, SMITH` is the first row of the result set.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
EXEC cursor_example;

Employee Number: 7369
Employee Name  : SMITH
```

14.8.5 Using %ROWTYPE With Cursors

Using the `%ROWTYPE` attribute, a record can be defined 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 within the current scope.

The following 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;

SMITH works in department 20
ALLEN works in department 30
WARD works in department 30
JONES works in department 20
MARTIN works in department 30
BLAKE works in department 30
CLARK works in department 10
SCOTT works in department 20
KING works in department 10
TURNER works in department 30
ADAMS works in department 20
JAMES works in department 30
FORD works in department 20
MILLER works in department 10
```

14.8.6 Cursor Attributes

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Each cursor has a set of attributes associated with it that allows the program to test the state of the cursor. These attributes are `%ISOPEN, %FOUND, %NOTFOUND`, and `%ROWCOUNT`. These attributes are described in the following sections.

14.8.6.1 %ISOPEN

The `%ISOPEN` attribute is used to test whether or not a cursor is open.

```
<cursor_name>%ISOPEN
```

`cursor_name` is the name of the cursor for which a `BOOLEAN` data type of `TRUE` will be returned if the cursor is open, `FALSE` otherwise.

The following is an example of using `%ISOPEN`.

```
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;
    END IF;
    FETCH emp_cur_1 INTO ...
        ...
END;
```

14.8.6.2 %FOUND

The `%FOUND` attribute is used to test whether or not 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` will be returned if a row is retrieved from the result set of the cursor after a `FETCH`.

After the last row of the result set has been fetched the next `FETCH` results in `%FOUND` returning `FALSE`. `FALSE` is also returned after the first `FETCH` if there are no rows in the result set to begin with.

Referencing `%FOUND` on a cursor before it is opened or after it is closed results in an `INVALID_CURSOR` exception being thrown.

`%FOUND` returns `null` if it is referenced when the cursor is open, but before the first `FETCH`.

The following example uses `%FOUND`.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
```

When the previous procedure is invoked, the output appears as follows:

```
 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
```

14.8.6.3 %NOTFOUND

The `%NOTFOUND` attribute is the logical opposite of `%FOUND`.

```
 <cursor_name>%NOTFOUND
```

`cursor_name` is the name of the cursor for which a `BOOLEAN` data type of `FALSE` will be returned if a row is retrieved from the result set of the cursor after a `FETCH`.

After the last row of the result set has been fetched the next `FETCH` results in `%NOTFOUND` returning `TRUE`. `TRUE` is also returned after the first `FETCH` if there are no rows in the result set to begin with.

Referencing `%NOTFOUND` on a cursor before it is opened or after it is closed, results in an `INVALID_CURSOR` exception being thrown.

`%NOTFOUND` returns `null` if it is referenced when the cursor is open, but before the first `FETCH`.

The following example uses `%NOTFOUND`.

```
 CREATE OR REPLACE PROCEDURE cursor_example
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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;
    CLOSE emp_cur_1;
END;
```

Similar to the prior example, this procedure produces the same output when invoked.

```
 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
```

14.8.6.4 %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 has been retrieved, `%ROWCOUNT` remains set to the total number of rows returned until the cursor is closed at which point `%ROWCOUNT` will throw an `INVALID_CURSOR` exception if referenced.

Referencing `%ROWCOUNT` on a cursor before it is opened or after it is closed, results in an `INVALID_CURSOR` exception being thrown.

`%ROWCOUNT` returns `0` if it is referenced when the cursor is open, but before the first `FETCH`. `%ROWCOUNT` also returns `0` after the first `FETCH` when there are no rows in the result set to begin with.

The following example uses `%ROWCOUNT`.

```
 CREATE OR REPLACE PROCEDURE cursor_example
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
**********************
14 rows were retrieved
```

14.8.6.5 Summary of Cursor States and Attributes

The following table summarizes the possible cursor states and the values returned by the cursor attributes.

Cursor State	%ISOPEN	%FOUND	%NOTFOUND	%ROWCOUNT
Before `OPEN`	False	`INVALID_CURSOR` Exception	`INVALID_CURSOR` Exception	`INVALID_CURSOR` Exception
After `OPEN` & Before 1st `FETCH`	True	Null	Null	0
After 1st Successful `FETCH`	True	True	False	1
After `nth` Successful `FETCH` (last row)	True	True	False	n
After n+1st `FETCH` (after last row)	True	False	True	n

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Cursor State	%ISOPEN	%FOUND	%NOTFOUND	%ROWCOUNT
After `CLOSE`	False	`INVALID_CURSOR` Exception	`INVALID_CURSOR` Exception	`INVALID_CURSOR` Exception

14.8.7 Cursor FOR Loop

In the cursor examples presented so far, the programming logic required to process the result set of a cursor included 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 finally a statement to close the cursor. The *cursor FOR loop* is a loop construct that eliminates the need to individually code the statements just listed.

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;
```

`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.

The following example shows the example from %NOTFOUND, modified to use 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 same results are achieved as shown in the output below.

```
 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
7934     MILLER
```

14.8.8 Parameterized Cursors

A user can also declare a static cursor that accepts parameters, and can pass values for those parameters when opening that cursor. In the following example we have created a parameterized cursor which will display the name and salary of all employees from the `emp` table that have a salary less than a specified value which is passed as a parameter.

```
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;
```

So for example if we pass the value 2000 as `max_wage`, then we will only be shown the name and salary of all employees that have a salary less than 2000. The result of the above query is the following:

```
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
```

14.9 REF CURSORs and Cursor Variables

This section discusses another type of cursor that provides far greater flexibility than the previously discussed static cursors.

14.9.1 REF CURSOR Overview

A *cursor variable* is a cursor that actually contains a pointer to a query result set. The result set is determined by the execution of the `OPEN FOR` statement using the cursor variable.

A cursor variable is not tied to a single particular query like a static cursor. The same cursor variable may be opened 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 via the cursor variable.

`REF CURSOR` types may be passed as parameters to or from stored procedures and functions. The return type of a function may also be a `REF CURSOR` type. This provides the capability to modularize the operations on a cursor into separate programs by passing a cursor variable between programs.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.9.2 Declaring a Cursor Variable

SPL supports the declaration of a cursor variable using both the `SYS_REFCURSOR` built-in data type as well as creating a type of `REF CURSOR` and then declaring a variable of that 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`.

Only the declaration of `SYS_REFCURSOR` and user-defined `REF CURSOR` variables are different. The remaining usage like opening the cursor, selecting into the cursor and closing the cursor is the same across both the cursor types. For the rest of this chapter our examples will primarily be making use of the `SYS_REFCURSOR` cursors. All you need to change in the examples to make them work for user defined `REF CURSORs` is the declaration section.

> **Note**
>
> *Strongly-typed* `REF CURSORs` require the result set to conform to a declared number and order of fields with compatible data types and can also optionally return a result set.

14.9.2.1 Declaring a SYS_REFCURSOR Cursor Variable

The following is the syntax for declaring a `SYS_REFCURSOR` cursor variable:

```
<name> SYS_REFCURSOR;
```

`name` is an identifier assigned to the cursor variable.

The following is an example of a `SYS_REFCURSOR` variable declaration.

```
DECLARE
    emp_refcur      SYS_REFCURSOR;
        ...
```

14.9.2.2 Declaring a User Defined REF CURSOR Type Variable

You must perform two distinct declaration steps in order 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 is as follows:

```
TYPE <cursor_type_name> IS REF CURSOR [RETURN <return_type>];
```

The following is an example of a cursor variable declaration.

```
DECLARE
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    my_rec emp_cur_type;
        ...
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.9.3 Opening a Cursor Variable

Once a cursor variable is declared, it must be opened with an associated `SELECT` command. The `OPEN FOR` statement specifies the `SELECT` command to be used to create the result set.

```
OPEN <name> FOR query;
```

`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.

In the following example, the result set is a list of employee numbers and names from a selected department. Note that a variable or parameter can be used in the `SELECT` command anywhere an expression can normally appear. In this case a parameter is used in the equality test for department number.

```
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;
        ...
```

14.9.4 Fetching Rows From a Cursor Variable

After a cursor variable is opened, rows may be retrieved from the result set using the `FETCH` statement. See Fetching Rows From a Cursor for details on using the `FETCH` statement to retrieve rows from a result set.

In the example below, a `FETCH` statement has been added to the previous example so now the result set is returned into two variables and then displayed. Note that the cursor attributes used to determine cursor state of static cursors can also be used with cursor variables. See Cursor Attributes for details on cursor attributes.

```
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;
        ...
```

14.9.5 Closing a Cursor Variable

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Use the `CLOSE` statement described in Closing a Cursor to release the result set.

> **Note**
>
> Unlike static cursors, a cursor variable does not have to be closed before it can be re-opened again. The result set from the previous open will be lost.

The example is completed with the addition of the `CLOSE` statement.

```
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;
    CLOSE emp_refcur;
END;
```

The following is the output when this procedure is executed.

```
EXEC emp_by_dept(20)

EMPNO    ENAME
-----    -------
7369     SMITH
7566     JONES
7788     SCOTT
7876     ADAMS
7902     FORD
```

14.9.6 Usage Restrictions

The following are restrictions on cursor variable usage.

- Comparison operators cannot be used to test cursor variables for equality, inequality, null, or not null
- Null cannot be assigned to a cursor variable
- The value of a cursor variable cannot be stored in a database column
- Static cursors and cursor variables are not interchangeable. For example, a static cursor cannot be used in an `OPEN FOR` statement.

In addition the following table shows the permitted parameter modes for a cursor variable used as a procedure or function parameter depending upon the operations on the cursor variable within the procedure or function.

Operation	IN	IN OUT	OUT

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Operation	IN	IN OUT	OUT
`OPEN`	`No`	`Yes`	`No`
`FETCH`	`Yes`	`Yes`	`No`
`CLOSE`	`Yes`	`Yes`	`No`

So 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.

14.9.7 Examples

The following examples demonstrate cursor variable usage.

14.9.7.1 Returning a REF CURSOR From a Function

In the following example the cursor variable is opened with a query that selects employees with a given job. Note that the cursor variable is specified in this function's `RETURN` statement so the result set is made available to the caller of the function.

```
 CREATE OR REPLACE FUNCTION emp_by_job (p_job VARCHAR2)
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.

```
 DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE := 'SALESMAN';
    v_emp_refcur    SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES WITH JOB ' || v_job);
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    v_emp_refcur := emp_by_job(v_job);
    LOOP
        FETCH v_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN v_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE v_emp_refcur;
END;
```

The following is the output when the anonymous block is executed.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 EMPLOYEES WITH JOB SALESMAN
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
```

14.9.7.2 Modularizing Cursor Operations

The following example illustrates how the various operations on cursor variables can be modularized into separate programs.

The following procedure opens the given cursor variable with a `SELECT` command that retrieves all rows.

```
 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, but of a given department.

```
 CREATE OR REPLACE PROCEDURE open_emp_by_dept (
    p_emp_refcur    IN OUT SYS_REFCURSOR,
    p_deptno        emp.deptno%TYPE
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp
        WHERE deptno = p_deptno;
END;
```

This third variation opens the given cursor variable with a `SELECT` command that retrieves all rows, but from a different table. Also note that the function's return value is the opened cursor variable.

```
 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.

```
 CREATE OR REPLACE PROCEDURE fetch_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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.

```
 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.

```
 CREATE OR REPLACE PROCEDURE close_refcur (
    p_refcur        IN OUT SYS_REFCURSOR
)
IS
BEGIN
    CLOSE p_refcur;
END;
```

The following anonymous block executes all the previously described programs.

```
 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('*****************');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    close_refcur(gen_refcur);
END;
```

The following is the output from the anonymous block.

```
 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
****************
EMPLOYEES IN DEPT #10
EMPNO    ENAME
-----    -------
7782     CLARK
7839     KING
7934     MILLER
****************
DEPARTMENTS
DEPT   DNAME
----   ---------
10     ACCOUNTING
20     RESEARCH
30     SALES
40     OPERATIONS
*****************
```

14.9.8 Dynamic Queries With REF CURSORs

Advanced Server also supports dynamic queries via the `OPEN FOR USING` statement. A string literal or string variable is supplied in the `OPEN FOR USING` statement to the `SELECT` command.

```
OPEN <name> FOR <dynamic_string>
  [ USING <bind_arg> [, <bind_arg_2> ] ...];
```

`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 are used to pass variables to corresponding placeholders in the `SELECT` command when the cursor variable is opened. The placeholders are identifiers prefixed by a colon character.

The following is an example of a dynamic query using a string literal.

```
 CREATE OR REPLACE PROCEDURE dept_query
IS
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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 when the procedure is executed.

```
 EXEC dept_query;

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

In the next example, the previous query is modified to use bind arguments to pass the query parameters.

```
 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;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following is the resulting output.

```
 EXEC dept_query(30, 1500);

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Finally, a string variable is used to pass the `SELECT` providing the most flexibility.

```
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);

EMPNO    ENAME
-----    -------
7566     JONES
7788     SCOTT
7902     FORD
```

14.10 Collections

A *collection* is a set of ordered data items with the same data type. Generally, the data item is a scalar field, but may also be a user-defined type such as a record type or an object type as long as the structure and the data types that comprise each field of the user-defined type are the same for each element in the set. Each particular data item in the set is referenced by using subscript notation within a pair of parentheses.

> **Note**
>
> Multilevel collections (that is, where the data item of a collection is another collection) are not supported.

The most commonly known type of collection is an array. In Advanced Server, the supported collection types are *associative arrays* (formerly called *index-by-tables* in Oracle), *nested tables*, and *varrays*.

The general steps for using a collection are the following:

- A collection of the desired type must be defined. This can be done in the declaration section of an SPL program, which results in a *local type* that is accessible only within that program. For nested table and varray types this can also be done using the `CREATE TYPE` command, which creates a persistent, *standalone type* that can be referenced by any SPL program in the database.
- Variables of the collection type are declared. The collection associated with the declared variable is said to be *uninitialized* at this point if there is no value assignment made as part of the variable declaration.
- Uninitialized collections of nested tables and varrays are null. A *null collection* does not yet exist. Generally, a `COLLECTION_IS_NULL` exception is thrown if a collection method is invoked on a null collection.
- Uninitialized collections of associative arrays exist, but have no elements. An existing collection with no elements is called an *empty 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*.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- To add elements to an empty associative array, you can simply 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, the `EXTEND` method is then used to grow the collection beyond its initial size established by the constructor.

The specific process for each collection type is described in the following sections.

14.10.1 Associative Arrays

An *associative array* is a type of collection that associates a unique key with a value. The key does not have to be numeric, but can be character data as well.

An associative array has the following characteristics:

- An *associative array type* must be defined after which *array variables* can be declared 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 - just start assigning values to key values.
- The key can be any negative integer, positive integer, or zero if `INDEX BY BINARY_INTEGER` or `PLS_INTEGER` is specified.
- The key can be character data if `INDEX BY VARCHAR2` is specified.
- There is no pre-defined limit on the number of elements in the array - it grows dynamically as elements are added.
- The array can be sparse - there may be gaps in the assignment of values to keys.
- An attempt to reference an array element that has not been assigned a value will result in an exception.

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>) };
```

`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.

In order to make use of the array, a *variable* must be declared with that array type. The following is the syntax for declaring an array variable.

```
<array> <assoctype>
```

`array` is an identifier assigned to the associative array. `assoctype` is the identifier of a previously defined array type.

An element of the array is referenced 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 within the record type or attribute within the object type from which the array type is defined. Alternatively, the entire record can be referenced by omitting `[.field ]`.

The following example reads the first ten employee names from the `emp` table, stores them in an array, 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END;
```

The above example produces the following output:

```
 SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

The previous example is now modified to use a record type in the array definition.

```
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;
```

The following is the output from this anonymous block.

```
 EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `emp%ROWTYPE` attribute could be used to define `emp_arr_typ` instead of using the `emp_rec_typ` record type as shown in the following.

```
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 in the prior example.

Instead of assigning each field of the record individually, a record level assignment can be made from `r_emp` to `emp_arr`.

```
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) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END;
```

The key of an associative array can be character data as shown in the following example.

```
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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;

ANALYST  : 100
CLERK    : 200
MANAGER  : 300
SALESMAN : 400
PRESIDENT: 500
```

14.10.2 Nested Tables

A *nested table* is a type of collection that associates a positive integer with a value. A nested table has the following characteristics:

- A *nested table type* must be defined after which *nested table variables* can be declared of that nested table type. Data manipulation occurs using the nested table variable, or simply, "table" for short.
- When a nested table variable is declared, the nested table initially does not exist (it is a null collection). The null table must be initialized 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 additional elements to the table. See Collection Methods for information on collection methods. **Note:** Usage of the constructor to establish the number of elements in the table and usage of the `EXTEND` method to add additional elements to the table are mandatory in Oracle, but optional in SPL.
- The table can be sparse - there may be gaps in the assignment of values to keys.
- An attempt to reference a table element beyond its initialized or extended size will result in a `SUBSCRIPT_BEYOND_COUNT` exception.

The `TYPE IS TABLE` statement is used to define a nested table type within the declaration section of an SPL program.

```
TYPE <tbltype> IS TABLE OF { <datatype> | <rectype> | <objtype> };
```

`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 is available to all SPL programs in the database. See the *Database Compatibility for Oracle Developers SQL Guide* for more information about the `CREATE TYPE` command.

In order to make use of the table, a *variable* must be declared of that nested table type. The following is the syntax for declaring a table variable.

```
<table> <tbltype>
```

`table` is an identifier assigned to the nested table. `tbltype` is the identifier of a previously defined nested table type.

A nested table is initialized using the nested table type's constructor.

```
<tbltype> ([ { <expr1> | NULL } [, { <expr2> | NULL } ] [, ...] ])
```

`tbltype` is the identifier of the nested table type's constructor, which has the same name as the nested table type. `expr1, expr2, …` are expressions that are type-compatible with the element type of the table. If `NULL` is specified, the corresponding element is set to null. If the parameter list

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

is empty, then an empty nested table is returned, which means there are no elements in the table. 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 a collection method other than `EXISTS` is applied to an uninitialized nested table, a `COLLECTION_IS_NULL` exception is thrown. See Collection Methods for information on collection methods.

The following is an example of a constructor for a nested table:

```
DECLARE
    TYPE nested_typ IS TABLE OF CHAR(1);
    v_nested        nested_typ := nested_typ('A','B');
```

An element of the table is referenced using the following syntax.

```
<table>(<n>)[<.element> ]
```

`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 within the record type or attribute within the object type from which the nested table type is defined. Alternatively, the entire record or object can be referenced by omitting `[.element ]`.

The following is an example of a nested table where it is known that there will be four elements.

```
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;
    DBMS_OUTPUT.PUT_LINE('DNAME');
    DBMS_OUTPUT.PUT_LINE('----------');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(dname_tbl(j));
    END LOOP;
END;
```

The above example produces the following output:

```
DNAME
----------
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```

The following example reads the first ten employee names from the `emp` table, stores them in a nested table, then displays the results from the table. The SPL code is written to assume that the number of employees to be returned is not known beforehand.

```
DECLARE
    TYPE emp_rec_typ IS RECORD (
        empno       NUMBER(4),
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    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;
```

Note the creation of an empty table with the constructor `emp_tbl_typ()` 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 for information on `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
```

The following example shows how a nested table of an object type can be used. First, an object type is created with attributes for the department name and location.

```
CREATE TYPE dept_obj_typ AS OBJECT (
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
);
```

The following 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.

```
DECLARE
    TYPE dept_tbl_typ IS TABLE OF dept_obj_typ;
    dept_tbl        dept_tbl_typ;
    CURSOR dept_cur IS SELECT dname, loc FROM dept ORDER BY dname;
    i               INTEGER := 0;
BEGIN
    dept_tbl := dept_tbl_typ(
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        dept_obj_typ(NULL,NULL),
        dept_obj_typ(NULL,NULL),
        dept_obj_typ(NULL,NULL),
        dept_obj_typ(NULL,NULL)
    );
    FOR r_dept IN dept_cur LOOP
        i := i + 1;
        dept_tbl(i).dname := r_dept.dname;
        dept_tbl(i).loc   := r_dept.loc;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('DNAME          LOC');
    DBMS_OUTPUT.PUT_LINE('----------     ----------');
    FOR j IN 1..i LOOP
        DBMS_OUTPUT.PUT_LINE(RPAD(dept_tbl(j).dname,14) || ' ' ||
            dept_tbl(j).loc);
    END LOOP;
END;
```

The parameters comprising 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:

```
 DNAME          LOC
----------     ----------
ACCOUNTING     NEW YORK
OPERATIONS     BOSTON
RESEARCH       DALLAS
SALES          CHICAGO
```

14.10.3 Varrays

A *varray* or *variable-size array* is a type of collection that associates a positive integer with a value. In many respects, it is similar to a nested table.

A varray has the following characteristics:

- A *varray type* must be defined along with a maximum size limit. After the varray type is defined, *varray variables* can be declared of that varray type. Data manipulation occurs using the varray variable, or simply, "varray" for short. The number of elements in the varray cannot exceed the maximum size limit established in the varray type definition.
- When a varray variable is declared, the varray initially does not exist (it is a null collection). The null varray must be initialized 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 establishes the number of elements in the varray, which must not exceed the maximum size limit. The `EXTEND` method can add additional elements to the varray up to the maximum size limit. See Collection Methods for information on collection methods.
- Unlike a nested table, a varray cannot be sparse - there are no gaps in the assignment of values to keys.
- An attempt to reference a varray element beyond its initialized or extended size, but within the maximum size limit will result 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 will result in a `SUBSCRIPT_OUTSIDE_LIMIT` exception.

The `TYPE IS VARRAY` statement is used to define a varray type within the declaration section of an SPL program.

```
TYPE <varraytype> IS { VARRAY | VARYING ARRAY }(<maxsize>)
  OF { <datatype> | <objtype> };
```

`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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `CREATE TYPE` command can be used to define a varray type that is available to all SPL programs in the database. In order to make use of the varray, a *variable* must be declared of that varray type. The following is the syntax for declaring a varray variable:

```
<varray> <varraytype>
```

`varray` is an identifier assigned to the varray. `varraytype` is the identifier of a previously defined varray type.

A varray is initialized using the varray type's constructor.

```
<varraytype> ([ { <expr1> | NULL } [, { <expr2> | NULL } ]
  [, ...] ])
```

`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 `NULL` is specified, 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 a collection method other than `EXISTS` is applied to an uninitialized varray, a `COLLECTION_IS_NULL` exception is thrown. See Collection Methods for information on 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');
```

An element of the varray is referenced using the following syntax.

```
<varray>(<n>)[<.element> ]
```

`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 within the object type from which the varray type is defined. Alternatively, the entire object can be referenced by omitting `[.element ].`

The following is an example of a varray where it is known that there will be four elements.

```
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 above example produces the following output:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 DNAME
----------
ACCOUNTING
OPERATIONS
RESEARCH
SALES
```

14.11 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. The following sections discuss the collection methods supported by Advanced Server.

14.11.1 COUNT

`COUNT` is a method that returns the number of elements in a collection. The syntax for using `COUNT` is as follows:

```
<collection>.COUNT
```

`collection` is the name of a collection.

For a varray, `COUNT` always equals `LAST`.

The following example shows that an associative array can be sparsely populated (i.e., there are "gaps" in the sequence of assigned elements). `COUNT` includes only the elements that have been 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`.

```
COUNT: 5
```

14.11.2 DELETE

The `DELETE` method deletes entries from a collection. You can call the `DELETE` method in three different ways.

Use the first form of the `DELETE` method to remove all entries from a collection:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
<collection>.DELETE
```

Use the second form of the `DELETE` method to remove the specified entry from a collection:

```
<collection>.DELETE(<subscript>)
```

Use the third form of the `DELETE` method to remove the entries that are within the range specified by `first_subscript` and `last_subscript` (including the entries for the `first_subscript` and the `last_subscript`) from a collection.

```
<collection>.DELETE(<first_subscript>, <last_subscript>)
```

If `first_subscript` and `last_subscript` refer to non-existent elements, 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.

Note that when you delete an entry, the subscript remains in the collection; you can re-use the subscript with an alternate entry. If you specify a subscript that does not exist in the call to the `DELETE` method, `DELETE` does not raise an exception.

The following example demonstrates using the `DELETE` method to remove the element with subscript `0` from the collection:

```
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 `5` elements in the collection; after the `DELETE` method was invoked, the collection contains `4` elements.

14.11.3 EXISTS

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `EXISTS` method verifies that a subscript exists within a collection. `EXISTS` returns `TRUE` if the subscript exists; if the subscript does not exist, `EXISTS` returns `FALSE`. The method takes a single argument; the `subscript` that you are testing for. The syntax is:

```
<collection>.EXISTS(<subscript>)
```

`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`.

The following example verifies that subscript number `10` exists within 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('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 does not exist within the specified collection. Rather than raising an error, the `EXISTS` method returns a value of `FALSE`.

14.11.4 EXTEND

The `EXTEND` method increases the size of a collection. There are three variations of the `EXTEND` method. The first variation appends a single `NULL` element to a collection; the syntax for the first variation is:

```
<collection>.EXTEND
```

`collection` is the name of a collection.

The following example demonstrates using the `EXTEND` method to append a single, null element to a 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;
    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) || ' ';
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || v_results);
END;

COUNT: 5
COUNT: 6
Results: -100 -10 0 10 100 NULL
```

`COUNT` indicates that before the `EXTEND` method, there were `5` elements in the collection; after the `EXTEND` method was invoked, the collection contains `6` elements.

The second variation of the `EXTEND` method appends a specified number of elements to the end of a collection.

```
<collection>.EXTEND(<count>)
```

`collection` is the name of a collection.

`count` is the number of null elements added to the end of the collection.

The following example demonstrates using the `EXTEND` method to append multiple null elements to a 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);
    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 `5` elements in the collection; after the `EXTEND` method was invoked, the collection contains `8` elements.

The third 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>)
```

`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 is being copied to the collection.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example demonstrates using 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 -10 0 10 100 -10 -10 -10
```

`COUNT` indicates that before the `EXTEND` method, there were `5` elements in the collection; after the `EXTEND` method was invoked, the collection contains `8` elements.

> **Note**
>
> The `EXTEND` method cannot be used on a null or empty collection.

14.11.5 FIRST

`FIRST` is a method that returns the subscript of the first element in a collection. The syntax for using `FIRST` is as follows:

```
<collection>.FIRST
```

`collection` is the name of a collection.

The following example displays the first 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.11.6 LAST

`LAST` is a method that returns the subscript of the last element in a collection. The syntax for using `LAST` is as follows:

```
<collection>.LAST
```

`collection` is the name of a collection.

The following 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('LAST element: ' || sparse_arr(sparse_arr.LAST));
END;

LAST element: 100
```

14.11.7 LIMIT

`LIMIT` is a method that returns the maximum number of elements permitted in a collection. `LIMIT` is applicable only to varrays. The syntax for using `LIMIT` is as follows:

```
<collection>.LIMIT
```

`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 is a null varray), an exception is thrown.

For an associative array or an initialized nested table, `LIMIT` returns `NULL`. If the nested table is uninitialized (that is, it is a null nested table), an exception is thrown.

14.11.8 NEXT

`NEXT` is a method that returns the subscript that follows a specified subscript. The method takes a single argument; the `subscript` that you are testing for.

```
<collection>.NEXT(<subscript>)
```

`collection` is the name of the collection.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

If the specified subscript is less than the first subscript in the collection, the function returns the first subscript. If the subscript does not have a successor, `NEXT` returns `NULL`. If you specify a `NULL` subscript, `PRIOR` does not return a value.

The following example demonstrates using `NEXT` to return the subscript that follows subscript `10` 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('NEXT element: ' || sparse_arr.next(10));
END;

NEXT element: 100
```

14.11.9 PRIOR

The `PRIOR` method returns the subscript that precedes a specified subscript in a collection. The method takes a single argument; the `subscript` that you are testing for. The syntax is:

```
<collection>.PRIOR(<subscript>)
```

`collection` is the name of the collection.

If the subscript specified does not 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` does not return a value.

The following 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;

PRIOR element: 10
```

14.11.10 TRIM

The `TRIM` method removes an element or elements from the end of a collection. The syntax for the `TRIM` method is:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
<collection>.TRIM[(<count>)]
```

`collection` is the name of a collection.

`count` is the number of elements removed from the end of the collection. Advanced Server will return an error if `count` is less than `0` or greater than the number of elements in the collection.

The following example demonstrates using 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(-100,-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: 5
COUNT: 4
```

`COUNT` indicates that before the `TRIM` method, there were `5` elements in the collection; after the `TRIM` method was invoked, the collection contains `4` elements.

You can also specify the number of elements to remove from the end of the collection with the `TRIM` method:

```
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.TRIM(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: 3
Results: -100 -10 0
```

`COUNT` indicates that before the `TRIM` method, there were `5` elements in the collection; after the `TRIM` method was invoked, the collection contains `3` elements.

14.12 Working with Collections

Collection operators allow you to transform, query and manipulate the contents of a collection.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.12.1 TABLE()

Use the `TABLE()` function to transform the members of an array into a set of rows. The signature is:

```
TABLE(<collection_value>)
```

Where:

`collection_value`

`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` (a set of values of any type). For example, if the argument passed to this function is an array of `dates, TABLE()` will return a `SETOF dates`. If the argument passed to this function is an array of `paths, TABLE()` will return a `SETOF paths`.

You can use the `TABLE()` function to expand the contents of a collection into table form:

```
postgres=# SELECT * FROM TABLE(monthly_balance(445.00, 980.20, 552.00));

 monthly_balance
-----------------
  445.00
  980.20
  552.00
(3 rows)
```

14.12.2 Using the MULTISET UNION Operator

The `MULTISET UNION` operator combines two collections to form a third collection. The signature is:

```
<coll_1> MULTISET UNION [ ALL | DISTINCT] <coll_2>
```

Where `coll_1` and `coll_2` specify the names of the collections to combine.

Include the `ALL` keyword to specify that duplicate elements (elements that are present in both `coll_1` and `coll_2`) should be represented in the result, once for each time they are present in the original collections. This is the default behavior of `MULTISET UNION`.

Include the `DISTINCT` keyword to specify that duplicate elements should be included in the result only once.

The following example demonstrates using the `MULTISET UNION` operator to combine two collections (`collection_1` and `collection_2`) into a third collection `(collection_3)`:

```
DECLARE
    TYPE int_arr_typ IS TABLE OF NUMBER(2);
    collection_1    int_arr_typ;
    collection_2    int_arr_typ;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;

COUNT: 5
Results: 10 20 30 30 40
```

The resulting collection includes one entry for each element in `collection_1` and `collection_2`. If the `DISTINCT` keyword is used, the results are as follows:

```
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 40
```

The resulting collection includes only those members with distinct values. Note in the following example that the `MULTISET UNION DISTINCT` operator also removes duplicate entries that are stored within the same collection:

```
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);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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
```

14.12.3 Using the FORALL Statement

Collections can be used to more efficiently process DML commands 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 rather than re-iteratively invoking the DML command with new values. The DML command to be processed in such a manner is specified with the `FORALL` statement. In addition, one or more collections are given in the DML command where different values are to be substituted each time the command is executed.

```
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.`

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 automatically rolled back. This behavior is not compatible with Oracle databases. Oracle allows explicit use of the `COMMIT` or `ROLLBACK` commands to control whether or not 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` within the loop to select a member of a collection). The number of iterations is controlled by the `lower_bound .. upper_bound` clause. The loop is executes once for each integer between the `lower_bound` and `upper_bound` (inclusive) and the index is incremented by one for each iteration. For example:

```
FORALL i IN 2 .. 5
```

Creates a loop that executes four times – in the first iteration, the `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.

The following example creates a table `(emp_copy)` that is an empty copy of the `emp` table. The example declares a type `(emp_tbl)` that is an array where 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.

`t_emp` is an associative array, of type `emp_tbl`. The `SELECT` statement uses the `BULK COLLECT INTO` command to populate the `t_emp` array. After the `t_emp` array is populated, the `FORALL` statement iterates through the values `(i)` in the `t_emp` array index and inserts a row for each record into `emp_copy`.

```
CREATE TABLE emp_copy(LIKE emp);

DECLARE

    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
```

The following 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;
    t_sal           SAL_TBL;
BEGIN
    t_empno(1)  := 9001;
    t_sal(1)    := 3350.00;
    t_empno(2)  := 9002;
    t_sal(2)    := 2000.00;
    t_empno(3)  := 9003;
    t_sal(3)    := 4100.00;
    FORALL i IN t_empno.FIRST..t_empno.LAST
        UPDATE emp SET sal = t_sal(i) WHERE empno = t_empno(i);
END;

SELECT * FROM emp WHERE empno > 9000;

 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)
```

The following 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;

SELECT * FROM emp WHERE empno > 9000;

 empno | ename | job | mgr | hiredate | sal | comm | deptno
-------+-------+-----+-----+----------+-----+------+--------
(0 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.12.4 Using the BULK COLLECT Clause

SQL commands that return a result set consisting of a large number of rows may not be operating as efficiently as possible due to the constant context switching that must occur between the database server and the client in order to transfer the entire result set. This inefficiency can be mitigated by using a collection to gather the entire result set in memory which the client can then access. The `BULK COLLECT` clause is used to specify the aggregation of the result set into a collection.

The `BULK COLLECT` clause can be used with the `SELECT INTO, FETCH INTO` and `EXECUTE IMMEDIATE` commands, and with the `RETURNING INTO` clause of the `DELETE, INSERT`, and `UPDATE` commands. Each of these is illustrated in the following sections.

14.12.4.1 SELECT BULK COLLECT

The `BULK COLLECT` clause can be used with the `SELECT INTO` statement as follows. (Refer to SELECT INTO for additional information on the `SELECT INTO` statement.)

```
SELECT <select_expressions> BULK COLLECT INTO <collection>
  [, ...] FROM ...;
```

If a single collection is specified, then `collection` may be a collection of a single field, or it may be a collection of a record type. If more than one collection is specified, then each `collection` must consist of a single field. `select_expressions` must match in number, order, and type-compatibility all fields in the target collections.

The following example shows the use of the `BULK COLLECT` clause where the target collections are associative arrays consisting of a single field.

```
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 job_tbl      IS TABLE OF emp.job%TYPE      INDEX BY BINARY_INTEGER;
    TYPE hiredate_tbl IS TABLE OF emp.hiredate%TYPE INDEX BY BINARY_INTEGER;
    TYPE sal_tbl      IS TABLE OF emp.sal%TYPE      INDEX BY BINARY_INTEGER;
    TYPE comm_tbl     IS TABLE OF emp.comm%TYPE     INDEX BY BINARY_INTEGER;
    TYPE deptno_tbl   IS TABLE OF emp.deptno%TYPE   INDEX BY BINARY_INTEGER;
    t_empno           EMPNO_TBL;
    t_ename           ENAME_TBL;
    t_job             JOB_TBL;
    t_hiredate        HIREDATE_TBL;
    t_sal             SAL_TBL;
    t_comm            COMM_TBL;
    t_deptno          DEPTNO_TBL;
BEGIN
    SELECT empno, ename, job, hiredate, sal, comm, deptno BULK COLLECT
       INTO t_empno, t_ename, t_job, t_hiredate, t_sal, t_comm, t_deptno
       FROM emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
        'SAL        ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..t_empno.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_empno(i) || '   ' ||
            RPAD(t_ename(i),8) || ' ' ||
            RPAD(t_job(i),10) || ' ' ||
            TO_CHAR(t_hiredate(i),'DD-MON-YY') || ' ' ||
            TO_CHAR(t_sal(i),'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_comm(i),0),'99,999.99') || '  ' ||
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
            t_deptno(i));
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL        COMM      DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80     800.00        .00  20
7499   ALLEN    SALESMAN   20-FEB-81   1,600.00     300.00  30
7521   WARD     SALESMAN   22-FEB-81   1,250.00     500.00  30
7566   JONES    MANAGER    02-APR-81   2,975.00        .00  20
7654   MARTIN   SALESMAN   28-SEP-81   1,250.00   1,400.00  30
7698   BLAKE    MANAGER    01-MAY-81   2,850.00        .00  30
7782   CLARK    MANAGER    09-JUN-81   2,450.00        .00  10
7788   SCOTT    ANALYST    19-APR-87   3,000.00        .00  20
7839   KING     PRESIDENT  17-NOV-81   5,000.00        .00  10
7844   TURNER   SALESMAN   08-SEP-81   1,500.00        .00  30
7876   ADAMS    CLERK      23-MAY-87   1,100.00        .00  20
7900   JAMES    CLERK      03-DEC-81     950.00        .00  30
7902   FORD     ANALYST    03-DEC-81   3,000.00        .00  20
7934   MILLER   CLERK      23-JAN-82   1,300.00        .00  10
```

The following example produces the same result, but uses an associative array on a record type defined with the `%ROWTYPE` attribute.

```
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp           EMP_TBL;
BEGIN
    SELECT * BULK COLLECT INTO t_emp FROM emp;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
        'SAL        ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            RPAD(t_emp(i).job,10) || ' ' ||
            TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || '  ' ||
            t_emp(i).deptno);
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL        COMM      DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80     800.00        .00  20
7499   ALLEN    SALESMAN   20-FEB-81   1,600.00     300.00  30
7521   WARD     SALESMAN   22-FEB-81   1,250.00     500.00  30
7566   JONES    MANAGER    02-APR-81   2,975.00        .00  20
7654   MARTIN   SALESMAN   28-SEP-81   1,250.00   1,400.00  30
7698   BLAKE    MANAGER    01-MAY-81   2,850.00        .00  30
7782   CLARK    MANAGER    09-JUN-81   2,450.00        .00  10
7788   SCOTT    ANALYST    19-APR-87   3,000.00        .00  20
7839   KING     PRESIDENT  17-NOV-81   5,000.00        .00  10
7844   TURNER   SALESMAN   08-SEP-81   1,500.00        .00  30
7876   ADAMS    CLERK      23-MAY-87   1,100.00        .00  20
7900   JAMES    CLERK      03-DEC-81     950.00        .00  30
7902   FORD     ANALYST    03-DEC-81   3,000.00        .00  20
7934   MILLER   CLERK      23-JAN-82   1,300.00        .00  10
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.12.4.2 FETCH BULK COLLECT

The `BULK COLLECT` clause can be used with a `FETCH` statement. (See Fetching Rows From a Cursor for information on the `FETCH` statement.) Instead of returning a single row at a time from the result set, the `FETCH BULK COLLECT` will return all rows at once from the result set into the specified collection unless restricted by the `LIMIT` clause.

```
FETCH <name> BULK COLLECT INTO <collection> [, ...] [ LIMIT <n> ];
```

If a single collection is specified, then `collection` may be a collection of a single field, or it may be a collection of a record type. If more than one collection is specified, then each `collection` must consist of a single field. The expressions in the `SELECT` list of the cursor identified by `name` must match in number, order, and type-compatibility all fields in the target collections. If `LIMIT n` is specified, the number of rows returned into the collection on each `FETCH` will not exceed `n`.

The following example uses the `FETCH BULK COLLECT` statement to retrieve rows into an associative array.

```
DECLARE
    TYPE emp_tbl IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
    t_emp           EMP_TBL;
    CURSOR emp_cur IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur;
    FETCH emp_cur BULK COLLECT INTO t_emp;
    CLOSE emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME    JOB        HIREDATE    ' ||
        'SAL        ' || 'COMM      DEPTNO');
    DBMS_OUTPUT.PUT_LINE('-----  -------  ---------  ---------   ' ||
        '--------   ' || '--------  ------');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            RPAD(t_emp(i).job,10) || ' ' ||
            TO_CHAR(t_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(t_emp(i).comm,0),'99,999.99') || '  ' ||
            t_emp(i).deptno);
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL        COMM      DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80     800.00        .00  20
7499   ALLEN    SALESMAN   20-FEB-81   1,600.00     300.00  30
7521   WARD     SALESMAN   22-FEB-81   1,250.00     500.00  30
7566   JONES    MANAGER    02-APR-81   2,975.00        .00  20
7654   MARTIN   SALESMAN   28-SEP-81   1,250.00   1,400.00  30
7698   BLAKE    MANAGER    01-MAY-81   2,850.00        .00  30
7782   CLARK    MANAGER    09-JUN-81   2,450.00        .00  10
7788   SCOTT    ANALYST    19-APR-87   3,000.00        .00  20
7839   KING     PRESIDENT  17-NOV-81   5,000.00        .00  10
7844   TURNER   SALESMAN   08-SEP-81   1,500.00        .00  30
7876   ADAMS    CLERK      23-MAY-87   1,100.00        .00  20
7900   JAMES    CLERK      03-DEC-81     950.00        .00  30
7902   FORD     ANALYST    03-DEC-81   3,000.00        .00  20
7934   MILLER   CLERK      23-JAN-82   1,300.00        .00  10
```

14.12.4.3 EXECUTE IMMEDIATE BULK COLLECT

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `BULK COLLECT` clause can be used with a `EXECUTE IMMEDIATE` statement to specify a collection to receive the returned rows.

```
EXECUTE IMMEDIATE '<sql_expression>;'
   BULK COLLECT INTO <collection> [,...]
   [USING {[<bind_type>] <bind_argument>} [, ...]}];
```

`collection` specifies the name of a collection.

`bind_type` specifies the parameter mode of the `bind_argument`.

- A `bind_type` of `IN` specifies that the `bind_argument` contains a value that is passed to the `sql_expression`.
- A `bind_type` of `OUT` specifies that the `bind_argument` receives a value from the `sql_expression`.
- A `bind_type` of `IN OUT` specifies that the `bind_argument` is passed to `sql_expression`, and then stores the value returned by `sql_expression`.

`bind_argument` specifies a parameter that contains a value that is either passed to the `sql_expression` (specified with a `bind_type` of `IN`), or that receives a value from the `sql_expression` (specified with a `bind_type` of `OUT`), or both (specified with a `bind_type` of `IN OUT`).

If a single collection is specified, then `collection` may be a collection of a single field, or a collection of a record type; if more than one collection is specified, each `collection` must consist of a single field.

14.12.4.4 RETURNING BULK COLLECT

The `BULK COLLECT` clause can be added to the `RETURNING INTO` clause of a `DELETE, INSERT`, or `UPDATE` command. (See Using the RETURNING INTO Clause for information on the `RETURNING INTO` clause.)

```
{ <insert> | <update> | <delete> }
  RETURNING { * | <expr_1> [, <expr_2> ] ...}
    BULK COLLECT INTO <collection> [, ...];
```

`insert, update`, and `delete` are the `INSERT, UPDATE`, and `DELETE` commands as described in INSERT, UPDATE, and DELETE, respectively. If a single collection is specified, then `collection` may be a collection of a single field, or it may be a collection of a record type. If more than one collection is specified, then each `collection` must consist of a single field. The expressions following the `RETURNING` keyword must match in number, order, and type-compatibility all fields in the target collections. If `*` is specified, then all columns in the affected table are returned. (Note that the use of `*` is an Advanced Server extension and is not compatible with Oracle databases.)

The `clerkemp` table created by copying the `emp` table is used in the remaining examples in this section as shown below.

```
CREATE TABLE clerkemp AS SELECT * FROM emp WHERE job = 'CLERK';

SELECT * FROM clerkemp;

 empno | ename  |  job  | mgr  |      hiredate       |   sal   | comm | deptno
-------+--------+-------+------+---------------------+---------+------+-------
-
  7369 | SMITH  | CLERK | 7902 | 17-DEC-80 00:00:00 |  800.00 |      |    20
  7876 | ADAMS  | CLERK | 7788 | 23-MAY-87 00:00:00 | 1100.00 |      |    20
  7900 | JAMES  | CLERK | 7698 | 03-DEC-81 00:00:00 |  950.00 |      |    30
  7934 | MILLER | CLERK | 7782 | 23-JAN-82 00:00:00 | 1300.00 |      |    10
(4 rows)
```

The following example increases everyone's salary by 1.5, stores the employees' numbers, names, and new salaries in three associative arrays, and finally, displays the contents of these arrays.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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        ');
    DBMS_OUTPUT.PUT_LINE('-----  -------    --------   ');
    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;

EMPNO  ENAME      SAL
-----  -------    --------
7369   SMITH      1,200.00
7876   ADAMS      1,650.00
7900   JAMES      1,425.00
7934   MILLER     1,950.00
```

The following example performs the same functionality as the previous example, but uses a single collection defined with a record type to store the employees' numbers, names, and new salaries.

```
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        ');
    DBMS_OUTPUT.PUT_LINE('-----  -------    --------   ');
    FOR i IN 1..t_emp.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(t_emp(i).empno || '   ' ||
            RPAD(t_emp(i).ename,8) || ' ' ||
            TO_CHAR(t_emp(i).sal,'99,999.99'));
    END LOOP;
END;

EMPNO  ENAME      SAL
-----  -------    --------
7369   SMITH      1,200.00
7876   ADAMS      1,650.00
7900   JAMES      1,425.00
7934   MILLER     1,950.00
```

The following example deletes all rows from the `clerkemp` table, and returns information on the deleted rows into an associative array, which is then displayed.

```
DECLARE
    TYPE emp_rec IS RECORD (
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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    ' ||
        'SAL        ' || 'COMM      DEPTNO');
    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) || ' ' ||
            RPAD(r_emp(i).job,10) || ' ' ||
            TO_CHAR(r_emp(i).hiredate,'DD-MON-YY') || ' ' ||
            TO_CHAR(r_emp(i).sal,'99,999.99') || ' ' ||
            TO_CHAR(NVL(r_emp(i).comm,0),'99,999.99') || '  ' ||
            r_emp(i).deptno);
    END LOOP;
END;

EMPNO  ENAME    JOB        HIREDATE    SAL        COMM      DEPTNO
-----  -------  ---------  ---------   --------   --------  ------
7369   SMITH    CLERK      17-DEC-80   1,200.00        .00  20
7876   ADAMS    CLERK      23-MAY-87   1,650.00        .00  20
7900   JAMES    CLERK      03-DEC-81   1,425.00        .00  30
7934   MILLER   CLERK      23-JAN-82   1,950.00        .00  10
```

14.12.5 Errors and Messages

Use the `DBMS_OUTPUT.PUT_LINE` statement to report messages.

```
DBMS_OUTPUT.PUT_LINE ( <message> );
```

`message` is any expression evaluating to a string.

This example displays the message on the user's output display:

```
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.

14.13 Triggers

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

This chapter describes Advanced Server triggers. As with procedures and functions, triggers are written in the SPL language.

14.13.1 Overview

A trigger is a named SPL code block that is associated with a table and stored in the database. When a specified event occurs on the associated table, the SPL code block is executed. The trigger is said to be *fired* when the code block is executed.

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`, or `DELETE`, command the trigger is directly fired assuming that the corresponding insert, update, or delete event is defined as a *triggering event*. The events that fire the trigger are defined in the `CREATE TRIGGER` command.

A trigger can be fired indirectly if a triggering event occurs on the table as a result of an event initiated on another table. For example, if 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, all children of the parent would be deleted as well. If deletion is a triggering event on the child table, deletion of the children will cause the trigger to fire.

14.13.2 Types of Triggers

Advanced Server supports both *row-level* and *statement-level* triggers. A row-level trigger fires once for each row that is affected by a triggering event. For example, if deletion is defined as a triggering event on a table and a single `DELETE` command is issued that deletes five rows from the table, then the trigger will fire five times, once for each row.

In contrast, a statement-level trigger fires once per triggering statement regardless of the number of rows affected by the triggering event. In the prior example of a single `DELETE` command deleting five rows, a statement-level trigger would fire only once.

The sequence of actions can be defined regarding whether the trigger code block is executed before or after the triggering statement, itself, in the case of statement-level triggers; or before or after each row is affected by the triggering statement in the case of row-level triggers.

In a *before* row-level trigger, the trigger code block is executed before the triggering action is carried out on each affected row. In a *before* statement-level trigger, the trigger code block is executed before the action of the triggering statement is carried out.

In an *after* row-level trigger, the trigger code block is executed after the triggering action is carried out on each affected row. In an *after* statement-level trigger, the trigger code block is executed after the action of the triggering statement is carried out.

14.13.3 Creating Triggers

The `CREATE TRIGGER` command defines and names a trigger that will be stored in the database.

Name

`CREATE TRIGGER` -- define a simple trigger.

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER <name>
  { BEFORE | AFTER | INSTEAD OF }
  { INSERT | UPDATE | DELETE }
      [ OR { INSERT | UPDATE | DELETE } ] [, ...]
    ON <table>
  [ REFERENCING { OLD AS <old> | NEW AS <new> } ...]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
[ FOR EACH ROW ]
[ WHEN <condition> ]
[ DECLARE
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]
    <declaration>; [, ...] ]
  BEGIN
    <statement>; [, ...]
[ EXCEPTION
  { WHEN <exception> [ OR <exception> ] [...] THEN
      <statement>; [, ...] } [, ...]
]
  END
```

Description

`CREATE TRIGGER` defines a new trigger. `CREATE OR REPLACE TRIGGER` will either create a new trigger, or replace an existing definition.

If you are using the `CREATE TRIGGER` keywords to create a new trigger, the name of the new trigger must not match any existing trigger defined on the same table. New triggers will be created in the same schema as the table on which the triggering event is defined.

If you are 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.

`INSERT | UPDATE | DELETE`

Defines the triggering event.

`table`

The name of the table on which the triggering event occurs.

`condition`

`condition` is a Boolean expression that determines if the trigger will actually be executed; if `condition` evaluates to `TRUE`, the trigger will fire.

- If the simple trigger definition includes the `FOR EACH ROW` keywords, the `WHEN` clause can refer to columns of the old and/or new row values by writing `OLD.column_name` or `NEW.column_name` respectively. `INSERT` triggers cannot refer to `OLD` and `DELETE` triggers cannot refer to `NEW`.

- If the trigger includes the `INSTEAD OF` keywords, it may not include a `WHEN` clause. A `WHEN` clause cannot contain subqueries.

`REFERENCING { OLD AS old | NEW AS new } ...`

`REFERENCING` clause to reference old rows and new rows, but restricted in that `old` may only be replaced by an identifier named `old` or any

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

equivalent that is saved in all lowercase (for example, `REFERENCING OLD AS old, REFERENCING OLD AS OLD`, or `REFERENCING OLD AS "old")`. Also, `new` may only be replaced by an identifier named `new` or any equivalent that is saved in all lowercase (for example, `REFERENCING NEW AS new, REFERENCING NEW AS NEW`, or `REFERENCING NEW AS "new")`.

Either one, or both phrases `OLD AS old` and `NEW AS new` may be specified in the `REFERENCING` clause (for example, `REFERENCING NEW AS New OLD AS Old)`. This clause is not compatible with Oracle databases in that identifiers other than `old` or `new` may not be used.

`FOR EACH ROW`

Determines whether the trigger should be fired once for every row affected by the triggering event, or just 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, they must be declared after all other variable, type, and `REF CURSOR` declarations.

`statement`

An SPL program statement. Note that a `DECLARE - BEGIN - END` block is considered an SPL statement unto itself. Thus, the trigger body may contain nested blocks.

`exception`

An exception condition name such as `NO_DATA_FOUND, OTHERS`, etc.

14.13.4 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 is not applicable in statement-level triggers and in delete operations of row-level triggers.

Its usage is: `:NEW.column` where `column` is the name of a column in the table on which the trigger is defined.

The initial content of `:NEW.column` is the value in the named column of the new row to be inserted or of the new row that is to replace the old one when used in a before row-level trigger. When used in an after row-level trigger, this value has already been stored in the table since the action has already occurred on the affected row.

In the trigger code block, `:NEW.column` can be used 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 will be 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 is not applicable in statement-level triggers and in insert operations of row-level triggers.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 be deleted or of the old row that is to be replaced by 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 has already occurred on the affected row.

In the trigger code block, `:OLD.column` can be used 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`.

14.13.5 Transactions and Exceptions

A trigger is always executed as part of the same transaction within which the DML statement is executing. When no exceptions occur within the trigger code block, the effects of any DML commands within the trigger are committed if and only if the transaction containing the triggering statement is committed. Therefore, if the transaction is rolled back, the effects of any triggering commands within the trigger are also rolled back.

If an exception does occur within the trigger code block, but it is caught and handled in an exception section, the effects of any DML commands within the trigger are still rolled back nonetheless. The triggering statement itself, however, is not rolled back unless the application forces a roll back of the encapsulating transaction.

If an unhandled exception occurs within the trigger code block, the transaction that encapsulates the trigger is aborted and rolled back. Therefore, the effects of any DML commands within the trigger and the triggering statement, itself are all rolled back.

14.13.6 Trigger Examples

The following sections illustrate an example of each type of trigger.

14.13.6.1 Before Statement-Level Trigger

The following is an example of a simple before statement-level trigger that displays a message prior to an insert operation on the `emp` table.

```
CREATE OR REPLACE TRIGGER emp_alert_trig
    BEFORE INSERT ON emp
BEGIN
    DBMS_OUTPUT.PUT_LINE('New employees are about to be added');
END;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

```
INSERT INTO emp (empno, ename, deptno) SELECT empno + 1000, ename, 40
    FROM emp WHERE empno BETWEEN 7900 AND 7999;
New employees are about to be added

SELECT empno, ename, deptno FROM emp WHERE empno BETWEEN 8900 AND 8999;

     EMPNO ENAME          DEPTNO
---------- ---------- ----------
      8900 JAMES              40
      8902 FORD               40
      8934 MILLER             40
```

The message, `New employees are about to be added`, is displayed once by the firing of the trigger even though the result is the addition of three new rows.

14.13.6.2 After Statement-Level Trigger

The following is an example of an after statement-level trigger. Whenever 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.

```
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. The `sal` and `comm` columns of both rows are updated with one `UPDATE` command. Finally, both rows are deleted with one `DELETE` command.

```
INSERT INTO emp VALUES (9001,'SMITH','ANALYST',7782,SYSDATE,NULL,NULL,10);

INSERT INTO emp VALUES (9002,'JONES','CLERK',7782,SYSDATE,NULL,NULL,10);

UPDATE emp SET sal = 4000.00, comm = 1200.00 WHERE empno IN (9001, 9002);

DELETE FROM emp WHERE empno IN (9001, 9002);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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
------------------ -------------------- --------------------
31-MAR-05 14:59:48 SYSTEM               Added employee(s)
31-MAR-05 15:00:07 SYSTEM               Added employee(s)
31-MAR-05 15:00:19 SYSTEM               Updated employee(s)
31-MAR-05 15:00:34 SYSTEM               Deleted employee(s)
```

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) and once for the deletion (even though two rows were deleted).

14.13.6.3 Before Row-Level Trigger

The following example is a before row-level trigger that calculates the commission of every new employee belonging to department 30 that is inserted into the `emp` table.

```
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.

```
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);

 EMPNO ENAME  JOB       MGR    HIREDATE       SAL       COMM       DEPTNO
------ ------ --------  ------ ---------- --------- ---------- -------------
 9005  ROBERS SALESMAN  7782   01-APR-05      3000       1200         30
 9006  ALLEN  SALESMAN  7782   01-APR-05      4500       1800         30
```

14.13.6.4 After Row-Level Trigger

The following example is an after row-level trigger. When a new employee row is inserted, the trigger adds a new 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.

Finally, trigger adds a row to the `empchglog` table with a description of the action.

```
CREATE TABLE empchglog (
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
        INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
            :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, 'New Hire');
    ELSIF UPDATING THEN
        v_action := 'Updated';
        v_empno := :NEW.empno;
        v_deptno := :NEW.deptno;
        v_chgdesc := '';
        IF NVL(:OLD.ename, '-null-') != NVL(:NEW.ename, '-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, NULL,
            :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, v_chgdesc);
    ELSIF DELETING THEN
        v_action := 'Deleted';
        v_empno := :OLD.empno;
        v_deptno := :OLD.deptno;
    END IF;

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;
```

In the first sequence of commands shown below, two employees are added using two separate `INSERT` commands and then both are updated using a single `UPDATE` command. The contents of the `jobhist` table shows 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,5000.00,NULL,40);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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);

SELECT * FROM jobhist WHERE empno IN (9003, 9004);

   EMPNO STARTDATE ENDDATE   JOB              SAL       COMM     DEPTNO   CHGDESC
---------- --------- --------- --------- ---------- ---------- ---------- -------------
    9003 31-MAR-05 31-MAR-05 ANALYST         5000                    40   New Hire
    9004 31-MAR-05 31-MAR-05 ANALYST         4500                    40   New Hire
    9003 31-MAR-05           ANALYST         5000       5500         40   Changed commission
    9004 31-MAR-05           ANALYST         4500       4950         40   Changed commission

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
```

Finally, both employees are deleted with a single `DELETE` command. The `empchglog` table now shows the trigger was fired twice, once for each deleted employee.

```
 DELETE FROM emp WHERE empno IN (9003, 9004);

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
31-MAR-05 Deleted employee # 9003
31-MAR-05 Deleted employee # 9004
```

14.14 Packages

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 – 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 is dependent upon one privilege granted to the entire package. All of the package programs must be referenced with a common name.
- Certain functions, procedures, variables, types, etc. 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 - the program names, parameters if any, and return types of functions. The SPL code of these functions and procedures is not accessible to others, therefore applications that utilize a package are dependent only upon the information available in the signature – not in the procedural logic itself.
- Other functions, procedures, variables, types, etc. in the package can be declared as *private*. Private entities can be referenced and used by function and procedures within the package, but not by other external applications. Private entities are for use only by programs within the package.
- Function and procedure names can be overloaded within a package. One or more functions/procedures can be defined with the same name, but with different signatures. This provides the capability to create identically named programs that perform the same job, but on different types of input.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

For more information about the package support provided by Advanced Server, see the*Database Compatibility for Oracle Developers Built-in Package Guide*, available at:

https://www.enterprisedb.com/docs

For a list of built-in packages, see the Table of Contents, of "Built-In Packages" of the *Database Compatibility for Oracle Developers Built-in Package Guide*.

14.15 Object Types and Objects

This chapter discusses how object-oriented programming techniques can be exploited in SPL. Object-oriented programming as seen in programming languages such as Java and C++ centers on the concept of objects. An *object* is a representation of 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, or equivalently, *instances* of the object type, person, or simply, person objects.

> **Note**
>
> - The terms “database objects” and “objects” that have been used in this document up to this point should not be confused with an object type and object as used in this chapter. The previous usage of these terms relates to the entities that can be created in a database such as tables, views, indexes, users, etc. Within the context of this chapter, 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) is used to describe object types in PL/SQL. The SPL implementation of object types is intended to be compatible with Oracle abstract data types.
> - Advanced Server has not yet implemented support for some features of object-oriented programming languages. This chapter documents only those features that have been implemented.

14.15.1 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 might be name, address, gender, date of birth, height, weight, eye color, occupation, etc.
- *Methods* – programs that perform some type of function or operation on, or related to an object. For a person object, examples might be calculating the person’s age, displaying the person’s attributes, changing the values assigned to the person’s attributes, etc.

The following sections elaborate on some basic object concepts.

14.15.1.1 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, VARCHAR2`, etc.
- 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 may be null) when an object instance is initially created. Each object instance has its own set of attribute values.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.15.1.2 Methods

Methods are SPL procedures or functions defined within an object type. Methods can be categorized into three general types:

- *Member Methods* – procedures or functions that operate within the context of an object instance. Member methods have access to, and can change the attributes of the object instance on which they are operating.
- *Static Methods* – procedures or functions that operate independently of any particular object instance. Static methods do not have access to, and cannot 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.

14.15.1.3 Overloading Methods

In an object type it is permissible to define two or more identically named methods of the same type (this is, either a procedure or function), 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.

14.15.2 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 is the public interface specifying the attributes and method signatures of the object type.
- The *object type body* - This contains the implementation of the methods specified in the object type specification.

The following sections describe the commands used to create the object type specification and the object type body.

14.15.2.1 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> | <collecttype> } }
    [, ...]
  [ <method_spec> ] [, ...]
  [ <constructor> ] [, ...]
) [ [ NOT ] { FINAL | INSTANTIABLE } ] ...;
```

where `method_spec` is the following:

```
[ [ NOT ] { FINAL | INSTANTIABLE } ] ...
[ OVERRIDING ]
  <subprogram_spec>
```

where `subprogram_spec` is the following:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 { 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 | IN OUT | OUT ] <datatype1>
                [ DEFAULT <value1> ] ]
      [, <parm2> [ IN | IN OUT | OUT ] <datatype2>
                 [ DEFAULT <value2> ]
      ] ...)
  ]
RETURN <return_type>;
```

Note

- The `OR REPLACE` option cannot be currently used to add, delete, or modify the attributes of an existing object type. Use the `DROP TYPE` command to first delete the existing object type. The `OR REPLACE` option can be used to add, delete, or modify the methods in an existing object type.
- The PostgreSQL form of the `ALTER TYPE ALTER ATTRIBUTE` command can be used to change the data type of an attribute in an existing object type. However, the `ALTER TYPE` command cannot add or delete attributes in the object type.

`name` is an identifier (optionally schema-qualified) assigned to the object type.

If the `AUTHID` clause is omitted or `DEFINER` is specified, the rights of the object type owner are used to determine access privileges to database objects. If `CURRENT_USER` is specified, the rights of the current user executing a method in the object are used to determine access privileges.

`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 or not 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> **Note**
>
> Even though the specification of `NOT FINAL` is accepted in the `CREATE TYPE` command, SPL does not currently support the creation of subtypes.

Following the closing parenthesis of the `CREATE TYPE` definition, `[ NOT ] INSTANTIABLE` specifies whether or not an object instance can be created of this object type. `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 to be defined. If `NOT INSTANTIABLE` is specified, then `NOT FINAL` must be specified as well. If any method in the object type contains the `NOT INSTANTIABLE` qualifier, then the object type, itself, must be defined with `NOT INSTANTIABLE` and `NOT FINAL`.

> **Note**
>
> Even though the specification of `NOT INSTANTIABLE` is accepted in the `CREATE TYPE` command, SPL does not currently support the creation of subtypes.

`method_spec` denotes the specification of a member method or static method.

Prior to the definition of a method, `[ NOT ] FINAL` specifies whether or not the method can be overridden in a subtype. `NOT FINAL` is the default meaning the method can be overridden in a subtype.

Prior to the definition of 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 the same return type (if the method is a function) as defined in the supertype.

Prior to the definition of a method, `[ NOT ] INSTANTIABLE` specifies whether or not the object type definition provides an implementation for the method. If `INSTANTIABLE` is specified, then the `CREATE TYPE BODY` command for the object type must specify the implementation of the method. If `NOT INSTANTIABLE` is specified, 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 itself, must specify `NOT INSTANTIABLE` and `NOT FINAL` following the closing parenthesis of the object type specification. The default is `INSTANTIABLE`.

`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 is not invoked with respect to any object instance.

`proc_name` is an identifier of a procedure. If the `SELF` parameter is specified, `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. If none are specified, the default is `IN`. `value1, value2, …` are default values that may be specified for `IN` parameters.

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. If none are specified, the default is `IN`. `value1, value2, …` are default values that may be specified for `IN` parameters. `return_type` is the data type of the value the function returns.

The following points should be noted about an object type specification:

- There must be at least one attribute defined in the object type.
- There may be none, one, or more methods defined in the object type.
- A static method cannot be overridden (`OVERRIDING` and `STATIC` cannot be specified together in `method_spec`).
- A static method must be instantiable (`NOT INSTANTIABLE` and `STATIC` cannot be specified together in `method_spec`).

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

14.15.2.2 Object Type Body Syntax

The following is the syntax of the object type body:

```
CREATE [ OR REPLACE ] TYPE BODY <name>
  { IS | AS }
  <method_spec> [...]
  [<constructor>] [...]
END;
```

where `method_spec` is the following:

`subprogram_spec`

and `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> ]
        ] ...)
    ]
{ IS | AS }
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  [ <declarations> ]
  BEGIN
    <statement>; ...
[ EXCEPTION
    WHEN ... THEN
      <statement>; ...]
  END;
|
  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>
{ IS | AS }
  [ PRAGMA AUTONOMOUS_TRANSACTION; ]
  [ <declarations> ]
  BEGIN
    <statement>; ...
[ EXCEPTION
    WHEN ... THEN
      <statement>; ...]
  END;
```

where `constructor` is:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 }
[ <declarations> ]
BEGIN
  <statement>; ...
[ EXCEPTION
  WHEN ... THEN
    <statement>; ...]
END;
```

`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 was 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, and must be specified in the `CREATE TYPE BODY` command in the same manner as specified 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 specified in the `CREATE TYPE` command. `return_type` is the data type of the value the function returns and must match `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.

14.15.3 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. This section provides some examples using 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:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 is not required. This example creates a composite type, which allows you to treat related objects as a single attribute.

14.15.3.1 Member Methods

A member method is a function or procedure that is defined within an object type and can only be invoked through an instance of that type. Member methods have access to, and can change the attributes of, the object instance on which they are operating.

The following object type specification creates the `emp_obj_typ` object type:

```
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 is invoking the method. A `SELF` parameter is the first parameter in a member procedure or function *regardless* of whether it is explicitly declared in the parameter list.

The following code snippet defines an object type body for `emp_obj_typ:`

```
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);
        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. To illustrate how the `SELF` parameter would be used in the `CREATE TYPE BODY` command, the preceding object type body could be written as follows:

```
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);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        DBMS_OUTPUT.PUT_LINE('Name          : ' || SELF.ename);
        DBMS_OUTPUT.PUT_LINE('Street        : ' || SELF.addr.street);
        DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || SELF.addr.city || ', ' ||
            SELF.addr.state || ' ' || LPAD(SELF.addr.zip,5,'0'));
    END;
END;
```

Both versions of the `emp_obj_typ` body are completely equivalent.

14.15.3.2 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 within the `emp_obj_type type`, you can write:

```
emp_obj_type.get_count();
```

A static method does not have access to, and cannot change the attributes of an object instance, and does not 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:`

```
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:`

```
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;
```

Within the static function `get_dname`, there can be no references to `SELF`. Since a static function is invoked independently of any object instance, it has no implicit access to any object attribute.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Member procedure `display_dept` can access the `deptno` attribute of the object instance passed in the `SELF` parameter. It is not 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`).

14.15.3.3 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 may 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 may overload a constructor by creating one or more different constructor functions with the same name, but with different argument types.

The SPL compiler will provide 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 will provide 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 simply sets each member to `NULL`.

To create a custom constructor, declare the constructor function (using the keyword constructor) in the `CREATE TYPE` command and define the construction function in the `CREATE TYPE BODY` command. For example, you may wish to create a custom constructor for the `address` type which 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
)
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
   DBMS_OUTPUT.PUT_LINE(cust_addr.city);  -- displays Boston
   DBMS_OUTPUT.PUT_LINE(cust_addr.state); -- displays MA
END;
```

Custom constructor functions are typically used to compute member values when given incomplete information. The preceding example computes the values for `city` and `state` when given a postal code.

Custom constructor functions are 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 would set `payment.amount` to `NULL` so you must create a custom constructor (whose signature matches the default constructor) to prohibit `NULL` amounts.

14.15.4 Creating Object Instances

To create an instance of an object type, you must first declare a variable of the object type, and then initialize the declared object variable. The syntax for declaring an object variable is:

```
<object> <obj_type>
```

`object` is an identifier assigned to the object variable.

`obj_type` is the identifier of a previously defined object type.

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} ] [, ...])
```

`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, etc. If an

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

The following 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_type` 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.

The following 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');
    emp  EMP_OBJ_TYPE;
BEGIN
    emp := NEW EMP_OBJ_TYPE (9003,'RAY');
END;
```

> **Note**
>
> In Advanced Server, the following alternate syntax can be used in place of the constructor method.

```
[ ROW ] ({ <expr1> | NULL } [, { <expr2> | NULL } ] [, ...])
```

`ROW` is an optional keyword if two or more terms are specified within the parenthesis-enclosed, comma-delimited list. If only one term is specified, then specification of the `ROW` keyword is mandatory.

14.15.5 Referencing an Object

Once an object variable is created and initialized, individual attributes can be referenced using dot notation of the form:

```
<object>.<attribute>
```

`object` is the identifier assigned to the object variable. `attribute` is the identifier of an object type attribute.

If `attribute`, itself, is of an object type, then the reference must take the form:

```
<object>.<attribute>.<attribute_inner>
```

`attribute_inner` is an identifier belonging to the object type to which `attribute` references in its definition of `object`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example expands upon the previous anonymous block to display the values assigned to the `emp_obj_typ` object.

```
DECLARE
    v_emp          EMP_OBJ_TYP;
BEGIN
    v_emp := emp_obj_typ(9001,'JONES',
        addr_obj_typ('123 MAIN STREET','EDISON','NJ',08817));
    DBMS_OUTPUT.PUT_LINE('Employee No   : ' || v_emp.empno);
    DBMS_OUTPUT.PUT_LINE('Name          : ' || v_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Street        : ' || v_emp.addr.street);
    DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || v_emp.addr.city || ', ' ||
        v_emp.addr.state || ' ' || LPAD(v_emp.addr.zip,5,'0'));
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
```

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>
```

`object` is the identifier assigned to the object variable. `prog_name` is the identifier of the procedure or function.

Static procedures or functions are not called utilizing an object variable. Instead the procedure or function is called utilizing the object type name:

```
<object_type>.<prog_name>
```

`object_type` is the identifier assigned to the object type. `prog_name` is the identifier of the procedure or function.

The results of the previous anonymous block can be duplicated by calling the member procedure `display_emp:`

```
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
```

The following anonymous block creates an instance of `dept_obj_typ` and calls the member procedure `display_dept:`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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
```

The static function defined in `dept_obj_typ` can be called directly by qualifying it by the object type name as follows:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(dept_obj_typ.get_dname(20));
END;

RESEARCH
```

14.15.6 Dropping an Object Type

The syntax for deleting an object type is as follows.

```
DROP TYPE <objtype>;
```

`objtype` is the identifier of the object type to be dropped. If the definition of `objtype` contains attributes that are themselves object types or collection types, these nested object types or collection types must be dropped last.

If an object type body is defined for the object type, the `DROP TYPE` command deletes the object type body as well as the object type specification. In order to recreate the complete object type, both the `CREATE TYPE` and `CREATE TYPE BODY` commands must be reissued.

The following example drops the `emp_obj_typ` and the `addr_obj_typ` object types created earlier in this chapter. `emp_obj_typ` must be dropped first since it contains `addr_obj_typ` within its definition as an attribute.

```
DROP TYPE emp_obj_typ;
DROP TYPE addr_obj_typ;
```

The syntax for deleting an object type body, but not the object type specification is as follows.

```
DROP TYPE BODY <objtype>;
```

The object type body can be recreated by issuing the `CREATE TYPE BODY` command.

The following example drops only the object type body of the `dept_obj_typ.`

```
DROP TYPE BODY dept_obj_typ;
```

15 Database Compatibility for Oracle Developers SQL Guide

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

This guide provides a summary of the SQL commands compatible with Oracle databases that are supported by Advanced Server. The SQL commands in this section will work on both an Oracle database and an Advanced Server database.

Note the following points:

- Advanced Server supports other commands that are not listed here. These commands may have no Oracle equivalent or they may provide the similar or same functionality as an Oracle SQL command, but with different syntax.
- The SQL commands in this section do not necessarily represent the full syntax, options, and functionality available for each command. In most cases, syntax, options, and functionality that are not compatible with Oracle databases have been omitted from the command description and syntax.
- The Advanced Server documentation set documents command functionality that may not be compatible with Oracle databases.

15.1 ALTER INDEX

Name

`ALTER INDEX` -- modify an existing index.

Synopsis

Advanced Server supports two 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
```

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 at:

https://www.postgresql.org/docs/11/static/sql-reindex.html

`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.

Examples

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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;
```

See Also

CREATE INDEX, DROP INDEX

15.2 ALTER PROCEDURE

Name

`ALTER PROCEDURE`

Synopsis

```
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`

`procedure_name` specifies the (possibly schema-qualified) name of a stored procedure.

`options` may be:

- `[EXTERNAL] SECURITY DEFINER`

  Specify `SECURITY DEFINER` to instruct the server 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 instruct the server to execute the procedure with the privileges of the user that is invoking the procedure. The `EXTERNAL` keyword is accepted for compatibility, but ignored.

The `RESTRICT` keyword is accepted for compatibility, but ignored.

Examples

The following command specifies that the `update_balance` procedure should execute with the privileges of the user invoking the procedure:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
ALTER PROCEDURE update_balance SECURITY INVOKER;
```

See Also

CREATE PROCEDURE, DROP PROCEDURE

15.3 ALTER PROFILE

Name

`ALTER PROFILE` -- alter an existing profile.

Synopsis

```
ALTER PROFILE <profile_name> RENAME TO <new_name>;

ALTER PROFILE <profile_name>
      LIMIT {<parameter value>}[...];
```

Description

Use the `ALTER PROFILE` command to modify a user-defined profile; 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 Advanced Server, or use `ALTER PROFILE…RENAME TO` to change the name of a profile.

Parameters

`profile_name`

The name of the profile.

`new_name`

`new_name` specifies the new name of the profile.

`parameter`

`parameter` specifies the attribute limited by the profile.

`value`

`value` specifies the parameter limit.

Advanced Server supports the `value` shown below for each `parameter`:

`FAILED_LOGIN_ATTEMPTS` specifies the number of failed login attempts that a user may make before the server locks the user out of their account for the length of time specified by `PASSWORD_LOCK_TIME`. Supported values are:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- An `INTEGER` value greater than `0`.
- `DEFAULT` - the value of `FAILED_LOGIN_ATTEMPTS` specified in the `DEFAULT` profile.
- `UNLIMITED` – the connecting user may 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 has been locked because of `FAILED_LOGIN_ATTEMPTS`. Supported values are:

- A `NUMERIC` value greater than 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_LOCK_TIME` specified in the `DEFAULT` profile.
- `UNLIMITED` – the account is locked until it is manually unlocked by a database superuser.

`PASSWORD_LIFE_TIME` specifies the number of days that the current password may 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 that will pass after the password expires before connections by the role are rejected. If `PASSWORD_GRACE_TIME` is not specified, the password will expire on the day specified by the default value of `PASSWORD_GRACE_TIME`, and the user will not be allowed to execute any command until a new password is provided. Supported values are:

- A `NUMERIC` value greater than 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 does not 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 will be allowed to connect, but will not be allowed to execute any command until they update their expired password. Supported values are:

- A `NUMERIC` value greater than 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_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 re-using a password. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used 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 greater than 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_REUSE_TIME` specified in the `DEFAULT` profile.
- `UNLIMITED` – The password can be re-used without restrictions.

`PASSWORD_REUSE_MAX` specifies the number of password changes that must occur before a password can be reused. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used 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 or equal to `0`.
- `DEFAULT` - the value of `PASSWORD_REUSE_MAX` specified in the `DEFAULT` profile.
- `UNLIMITED` – The password can be re-used 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 to be allowed for use or not. If you specify the value as `TRUE`, the system allows

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 in order to be validated effectively, else an error will be 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` is not an Oracle-compatible parameter.

Examples

The following example modifies a profile named `acctg_profile`:

```
ALTER PROFILE acctg_profile
       LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

`acctg_profile` will count failed connection attempts when a login role attempts to connect to the server. The profile specifies that if a user has not authenticated with the correct password in three attempts, the account will be locked for one day.

The following example changes the name of `acctg_profile` to `payables_profile`:

```
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

See Also

CREATE PROFILE, DROP PROFILE

15.4 ALTER QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the `ALTER QUEUE SQL` command. This syntax can be used in association 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. The first form of this command changes the name of a queue.

```
ALTER QUEUE <queue_name> RENAME TO <new_name>
```

Parameters

`queue_name`

The name (optionally schema-qualified) of an existing queue.

`RENAME TO`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Include the `RENAME TO` clause and a new name for the queue to rename the queue.

`new_name`

New name for the queue.

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> ] ]
```

Parameters

`queue_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 the option or options to be associated with the new queue and the corresponding value of the option. If you provide duplicate option names, the server will return an error.

- If `option_name` is `retries`, provide an integer that represents the number of times a dequeue may be attempted.
- 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.

Use the third form of the `ALTER QUEUE` command to enable or disable enqueuing and/or dequeuing on a particular queue:

```
ALTER QUEUE <queue_name> ACCESS { START | STOP } [ FOR { enqueue | dequeue } ] [ NOWAIT ]
```

Parameters

`queue_name`

The name (optionally schema-qualified) of an existing queue.

`ACCESS`

Include the `ACCESS` keyword to enable or disable enqueuing and/or dequeuing on a particular 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 enqueueing or dequeueing activity on the specified queue.

`NOWAIT`

Include the `NOWAIT` keyword to specify that the server should not wait for the completion of outstanding transactions before changing the state of the queue. The `NOWAIT` keyword can only be used when specifying an `ACCESS` value of `STOP`. The server will return an error if `NOWAIT` is specified with

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

an `ACCESS` value of `START`.

Use the fourth form to `ADD` or `DROP` callback details for a particular queue.

```
ALTER QUEUE <queue_name> { ADD | DROP } CALL TO <location_name> [ WITH <callback_option> ]
```

Parameters

`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`

`location_name` specifies the name of the callback procedure.

`callback_option`

`callback_option` can be `context`; specify a `RAW` value when including this clause.

Examples

The following example changes the name of a queue from `work_queue_east` to `work_order`:

```
ALTER QUEUE work_queue_east RENAME TO work_order;
```

The following example modifies a queue named `work_order`, setting the number of retries to `100`, the delay between retries to `2` seconds, and the length of time that the queue will retain dequeued messages to `10` seconds:

```
ALTER QUEUE work_order SET (retries 100, retrydelay 2, retention 10);
```

The following 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;
```

The following 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

15.5 ALTER QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the `ALTER QUEUE SQL` command. This syntax can be used in association 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

To change 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

15.6 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>]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Description

A role without the `CREATEROLE` privilege may use this command to change their own password. An unprivileged role must include the `REPLACE` clause and their previous password if `PASSWORD_VERIFY_FUNCTION` is not `NULL` in their profile. When the `REPLACE` clause is used by a non-superuser, the server will compare the password provided to the existing password and raise an error if the passwords do not 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 will not 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 is not a superuser.

Parameters

`role_name`

The name of the role whose password is to be altered.

`password`

The role's new password.

`prev_password`

The role's previous password.

Examples

To change a role's password:

```
ALTER ROLE john IDENTIFIED BY xyRP35z REPLACE 23PJ74a;
```

15.7 ALTER ROLE - Managing Database Link and DBMS_RLS Privileges

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 may create a private database link. The following `ALTER ROLE` command grants privileges to an Advanced Server role that allow the specified role to create a private database link:

```
ALTER ROLE role_name
    WITH [CREATEDBLINK | CREATE DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT CREATE DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
ALTER ROLE role_name
  WITH [NOCREATEDBLINK | NO CREATE DATABASE LINK]
```

> **Note**
>
> The `CREATEDBLINK` and `NOCREATEDBLINK` keywords should be considered deprecated syntax; 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 may create a public database link. The following `ALTER ROLE` command grants privileges to an Advanced Server role that allow the specified role to create a public database link:

```
ALTER ROLE role_name
  WITH [CREATEPUBLICDBLINK | CREATE PUBLIC DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT CREATE PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
  WITH [NOCREATEPUBLICDBLINK | NO CREATE PUBLIC DATABASE LINK]
```

> **Note**
>
> The `CREATEPUBLICDBLINK` and `NOCREATEPUBLICDBLINK` keywords should be considered deprecated syntax; 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 may drop a public database link. The following `ALTER ROLE` command grants privileges to an Advanced Server role that allow the specified role to drop a public database link:

```
ALTER ROLE role_name
  WITH [DROPPUBLICDBLINK | DROP PUBLIC DATABASE LINK]
```

This command is the functional equivalent of:

```
GRANT DROP PUBLIC DATABASE LINK to role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
  WITH [NODROPPUBLICDBLINK | NO DROP PUBLIC DATABASE LINK]
```

> **Note**
>
> The `DROPPUBLICDBLINK` and `NODROPPUBLICDBLINK` keywords should be considered deprecated syntax; we recommend using the `DROP PUBLIC DATABASE LINK` and `NO DROP PUBLIC DATABASE LINK` syntax options.

EXEMPT ACCESS POLICY

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 will be able to view or modify any row in a table constrained by a `DBMS_RLS` policy. The following `ALTER ROLE` command grants privileges to an Advanced Server role that exempt the specified role from any defined `DBMS_RLS` policies:

```
ALTER ROLE role_name
    WITH [POLICYEXEMPT | EXEMPT ACCESS POLICY]
```

This command is the functional equivalent of:

```
GRANT EXEMPT ACCESS POLICY TO role_name
```

Use the following command to revoke the privilege:

```
ALTER ROLE role_name
  WITH [NOPOLICYEXEMPT | NO EXEMPT ACCESS POLICY]
```

> **Note**
>
> The `POLICYEXEMPT` and `NOPOLICYEXEMPT` keywords should be considered deprecated syntax; we recommend using the `EXEMPT ACCESS POLICY` and `NO EXEMPT ACCESS POLICY` syntax options.

See Also

CREATE ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE

15.8 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 be altered.

`increment`

The clause `INCREMENT BY increment` is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

unspecified, the old increment value will be maintained.

`minvalue`

The optional clause `MINVALUE minvalue` determines the minimum value a sequence can generate. If not specified, the current minimum value will be maintained. Note that the key words, `NO MINVALUE`, may be used to set this behavior back to the defaults of 1 and -2^{63}-1 for ascending and descending sequences, respectively, however, this term is not compatible with Oracle databases.

`maxvalue`

The optional clause `MAXVALUE maxvalue` determines the maximum value for the sequence. If not specified, the current maximum value will be maintained. Note that the key words, `NO MAXVALUE`, may be used to set this behavior back to the defaults of 2^{63}-1 and -1 for ascending and descending sequences, respectively, however, this term is not compatible with Oracle databases.

`cache`

The optional clause `CACHE cache` specifies how many sequence numbers are to be preallocated and stored 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 will be maintained.

`CYCLE`

The `CYCLE` option allows the sequence to wrap around when the `maxvalue` or `minvalue` has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the `minvalue` or `maxvalue`, respectively. If not specified, the old cycle behavior will be maintained. Note that the key words, `NO CYCLE`, may be used to alter the sequence so that it does not recycle, however, this term is not compatible with Oracle databases.

Notes

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, `ALTER SEQUENCE` is never rolled back; the changes take effect immediately and are not reversible.

`ALTER SEQUENCE` will not immediately affect `NEXTVAL` results in backends, other than the current one, that have pre-allocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence parameters. The current backend will be affected immediately.

Examples

Change the increment and cache value of sequence, `serial`.

```
ALTER SEQUENCE serial INCREMENT BY 2 CACHE 5;
```

See Also

CREATE SEQUENCE, DROP SEQUENCE

15.9 ALTER SESSION

Name

`ALTER SESSION` -- change a runtime parameter.

Synopsis

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
ALTER SESSION SET <name> = <value>
```

Description

The `ALTER SESSION` command changes runtime configuration parameters. `ALTER SESSION` only affects the value used by the current session. Some of these parameters are provided solely for compatibility with Oracle syntax and have no effect whatsoever on the runtime behavior of Advanced Server. Others will alter a corresponding Advanced Server database server runtime configuration parameter.

Parameters

`name`

Name of a settable runtime parameter. Available parameters are listed below.

`value`

New value of parameter.

Configuration Parameters

The following configuration parameters can be modified 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 Advanced Server `datestyle` runtime configuration parameter.

- `NLS_LANGUAGE (string)`

  Sets the language in which messages are displayed. Has the same effect as setting the 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 for syntax compatibility only and has no effect in the 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 Advanced Server.

- `QUERY_REWRITE_ENABLED (string)`

  Valid values are `TRUE`, `FALSE`, and `FORCE`. The default is `FALSE`. This parameter is provided for syntax compatibility only and has no effect in Advanced Server.

- `QUERY_REWRITE_INTEGRITY (string)`

  Valid values are `ENFORCED`, `TRUSTED`, and `STALE_TOLERATED`. The default is `ENFORCED`. This parameter is provided for syntax compatibility only and has no effect in Advanced Server.

Examples

Set the language to U.S. English in UTF-8 encoding. Note that in this example, the value, `en_US.UTF-8`, is in the format that must be specified for

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Advanced Server. This form is not 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';
```

15.10 ALTER SYNONYM

Name

`ALTER SYNONYM` -- change ownership of synonym object

Synopsis

```
ALTER SYNONYM <syn_name> OWNER TO <new_owner>
```

Description

The `ALTER SYNONYM` command changes 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 a synonym for table t
edb=> create public synonym x for  t;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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          |
```

15.11 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>
```

where `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 new 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 will be automatically dropped as well.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `ADD table_constraint`

  This form adds a new constraint to a table; for details, see CREATE TABLE.

- `DROP CONSTRAINT`

  This form drops constraints on a table. Currently, constraints on tables are not required to have unique names, so there may be more than one constraint matching the specified name. All matching constraints will be dropped.

`RENAME`

The `RENAME` forms change the name of a table (or an index, sequence, or view) or the name of an individual column in a table. There is no effect on the stored data.

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.

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 will require the entire table to be rewritten. This may take a significant amount of time for a large table; and it will

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

temporarily require 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 does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated.

Changing any part of a system catalog table is not 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;
```

See Also

CREATE TABLE, DROP TABLE

15.12 ALTER TABLESPACE

Name

`ALTER TABLESPACE` -- change the definition of a tablespace.

Synopsis

```
ALTER TABLESPACE <name> RENAME TO <newname>
```

Description

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 cannot begin with `pg_`, as such names are reserved for system tablespaces.

Examples

Rename tablespace `empspace` to `employee_space`:

```
ALTER TABLESPACE empspace RENAME TO employee_space;
```

See Also

DROP TABLESPACE

15.13 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 may use this command to change their own password. An unprivileged role must include the `REPLACE` clause and their previous password if `PASSWORD_VERIFY_FUNCTION` is not `NULL` in their profile. When the `REPLACE` clause is used by a non-superuser, the server will compare the password provided to the existing password and raise an error if the passwords do not 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 will not 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 is not a superuser.

Parameters

`role_name`

The name of the role whose password is to be altered.

`password`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

15.14 ALTER USER|ROLE... PROFILE MANAGEMENT CLAUSES

Name

`ALTER USER|ROLE`

Synopsis

```
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 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 Advanced Server, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/sql-commands.html

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 pre-defined profile with a role, or to change which pre-defined profile is associated with a user.
- Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to specify that the user account should be placed 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

indicated by the `PASSWORD_LOCK_TIME` parameter of the profile assigned to this role. If `LOCK TIME` is used with the `ACCOUNT LOCK` clause, the role can only be unlocked by a database superuser 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 will expire. If you omit the `AT 'timestamp'` keywords, the password will expire 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 may only have one profile. To discover the profile that is currently associated with a login role, query the `profile` column of the `DBA_USERS` view.

Parameters

`name`

The name of the role with which the specified profile will be associated.

`password`

The password associated with the role.

`profile_name`

The name of the profile that will be associated with the role.

`timestamp`

The date and time at which the clause will be 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 available at:

https://www.postgresql.org/docs/11/static/sql-alterrole.html

Examples

The following 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;
```

See Also

CREATE USER|ROLE... PROFILE MANAGEMENT CLAUSES

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

15.15 CALL

Name

`CALL`

Synopsis

```
CALL <procedure_name> '('[<argument_list>]')'
```

Description

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 is invoking.

Parameters

`procedure_name`

`procedure_name` is the (optionally schema-qualified) procedure name.

`argument_list`

`argument_list` specifies a comma-separated list of arguments required by the procedure. Note that each member of `argument_list` corresponds to a formal argument expected by the procedure. Each formal argument may be an `IN` parameter, an `OUT` parameter, or an `INOUT` parameter.

Examples

The `CALL` statement may take one of several forms, depending on the arguments required by the procedure:

```
CALL update_balance();
CALL update_balance(1,2,3);
```

15.16 COMMENT

Name

`COMMENT` -- define or change the comment of an object.

Synopsis

```
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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

string is stored for each object. To remove a comment, specify the empty string (two consecutive single quotes with no intervening space) for `text`. Comments are automatically dropped when the object is dropped.

Parameters

`table_name`

The name of the table to be commented. The table name may be schema-qualified.

`table_name.column_name`

The name of a column within `table_name` to be commented. The table name may be schema-qualified.

`text`

The new comment.

Notes

There is presently 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). *Do not put security-critical information in a comment*.

Examples

Attach a comment to the table `emp`:

```
COMMENT ON TABLE emp IS 'Current employee information';
```

Attach a comment to the `empno` column of the `emp` table:

```
COMMENT ON COLUMN emp.empno IS 'Employee identification number';
```

Remove these comments:

```
COMMENT ON TABLE emp IS '';
COMMENT ON COLUMN emp.empno IS '';
```

15.17 COMMIT

Name

`COMMIT` -- commit the current transaction.

Synopsis

```
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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameters

`WORK`

Optional key word - has no effect.

Notes

Use `ROLLBACK` to abort a transaction. Issuing `COMMIT` when not inside a transaction does no harm.

> **Note**
>
> Executing a `COMMIT` in a plpgsql procedure will throw an error if there is an Oracle-style SPL procedure on the runtime stack.

Examples

To commit the current transaction and make all changes permanent:

```
COMMIT;
```

See Also

ROLLBACK, ROLLBACK TO SAVEPOINT

15.18 CREATE DATABASE

Name

`CREATE DATABASE` -- create a new database.

Synopsis

```
CREATE DATABASE <name>
```

Description

`CREATE DATABASE` creates a new database.

To create a database, you must be a superuser or have the special `CREATEDB` privilege. Normally, the creator becomes the owner of the new database. Non-superusers with `CREATEDB` privilege can only create databases owned by them.

The new database will be created by cloning the standard system database `template1`.

Parameters

`name`

The name of the database to be created.

Notes

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`CREATE DATABASE` cannot be executed 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 new database:

```
CREATE DATABASE employees;
```

15.19 CREATE PUBLIC DATABASE LINK

Name

`CREATE [PUBLIC] DATABASE LINK` -- create a new database link.

Synopsis

```
CREATE [ PUBLIC ] DATABASE LINK <name>
  CONNECT TO { CURRENT_USER |
               <username> IDENTIFIED BY '<password>'}
  USING { postgres_fdw '<fdw_connection_string>' |
        [ oci ] '<oracle_connection_string>' }
```

Description

`CREATE DATABASE LINK` creates a new database link. A database link is an object that allows a reference to a table or view in a remote database within a `DELETE`, `INSERT`, `SELECT` or `UPDATE` command. A database link is referenced 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 can be used by any user. A *private database link* can be used only by the database link's owner. Specification of the `PUBLIC` option creates a public database link. If omitted, a private database link is created.

When the `CREATE DATABASE LINK` command is given, the database link name and the given connection attributes are stored in the 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 within 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.

A SQL command containing a reference to a database link must be issued while connected to the local database. When the SQL command is executed, 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**
>
> A database link cannot be used to access a remote database within a standby database server. Standby database servers are used for high availability, load balancing, and replication.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

For information about high availability, load balancing, and replication for Postgres database servers, see the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/high-availability.html

> **Note**
>
> - For Advanced Server 11, 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), and Oracle version 12c Release 1 (12.1).
> - The `edb_dblink_oci.rescans` GUC can be set to `SCROLL` or `SERIALIZABLE` at the server level in `postgresql.conf` file. It can also be set at session level using the `SET` command, but the setting will not be applied to existing dblink connections due to dblink connection caching.
> - When executing `SELECT` on LOB data of more than 4000 characters, it is advisable to use `edb_dblink_oci.rescans=serializable` to free up the temporary PGA memory and avoid exceeding the `PGA_AGGREGATE_LIMIT`.

The `edb_dblink_oci` supports both types of rescans: `SCROLL` and `SERIALIZABLE`. By default it is 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 as a single PostgreSQL statement can lead to multiple Oracle queries and thereby uses a serializable isolation level to provide consistent results.
- A serialization failure may occur due to a table modification 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` rescan will be quick, but with each iteration will reset the current row position to `1`. A `SERIALIZABLE` rescan has performance benefits over a `SCROLL` rescan.

Parameters

`PUBLIC`

Create a public database link that can be used by any user. If omitted, then the database link is private and can only be used by the database link's owner.

`name`

The name of the database link.

`username`

The username to be used for connecting to the remote database.

`CURRENT_USER`

Include `CURRENT_USER` to specify that Advanced Server should use the user mapping associated with the role that is 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 Advanced Server database. If `postgres_fdw` has not been installed on the database, use the `CREATE EXTENSION` command to install `postgres_fdw`. For more information, see the `CREATE EXTENSION` command in

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

the PostgreSQL Core documentation at: https://www.postgresql.org/docs/11/static/sql-createextension.html

`fdw_connection_string`

Specify the connection information for the `postgres_fdw` foreign data wrapper.

`oci`

Specifies a connection to a remote Oracle database. This is Advanced Server's default behavior.

`oracle_connection_string`

Specify the connection information for an oci connection.

> **s**
>
> To create a non-public database link you must have the `CREATE DATABASE LINK` privilege. To create a public database link you must have the `CREATE PUBLIC DATABASE LINK` privilege.

Setting up an Oracle Instant Client for oci-dblink

In order to use oci-dblink, an Oracle instant client must be downloaded and installed on the host running the Advanced Server database in which the database link is to be created.

An instant client can be downloaded from the following site:

http://www.oracle.com/technetwork/database/features/instant-client/index-097480.html

Oracle Instant Client for Linux

The following instructions apply to Linux hosts running Advanced Server.

Be sure the `libaio` library (the Linux-native asynchronous I/O facility) has already been installed on the Linux host running Advanced Server.

The `libaio` library can be installed with the following command:

```
yum install libaio
```

If the Oracle instant client that you've downloaded does not include the file specifically named `libclntsh.so` without a version number suffix, you must create a symbolic link named `libclntsh.so` that points to the downloaded version of the library file. 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
```

When you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server must know where the Oracle instant client library resides on the Advanced Server host.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `LD_LIBRARY_PATH` environment variable must include the path to the Oracle client installation directory containing the `libclntsh.so` file. For example, assuming the installation directory containing `libclntsh.so` is `/tmp/instantclient`:

```
export LD_LIBRARY_PATH=/tmp/instantclient:$LD_LIBRARY_PATH
```

Alternatively, you can also set the value of the `oracle_home` configuration parameter in the `postgresql.conf` file. So, 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 paramters.

The `ORACLE_HOME` environment variable is must to be set and should include the path to the Oracle home directory. For example,

```
export ORACLE_HOME=/opt/product/version/dbhomeXE
```

> **Note**
>
> Either the `LD_LIBRARY_PATH` or the `oracle_home` configuration parameter must be set along with the `ORACLE_HOME` environment variable before the `pg_ctl` utility is executed to start or restart Advanced Server.

If you are running the current session as the user account (for example, `enterprisedb`) that will directly invoke `pg_ctl` to start or restart Advanced Server, then be sure to 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 within the `.bash_profile` file under the home directory of the `enterprisedb` user account (that is, set `LD_LIBRARY_PATH` and `ORACLE_HOME` within file `~enterprisedb/.bash_profile`). This ensures that `LD_LIBRARY_PATH` and `ORACLE_HOME` will be set when you log in as `enterprisedb`.

If you are using a Linux service script with the `systemctl` or `service` command to start or restart Advanced Server, you must set `LD_LIBRARY_PATH` and `ORACLE_HOME` so it is in effect when the script invokes the `pg_ctl` utility.

For example, to set an environment variable for Advanced Server, you can create a file named `/etc/systemd/system/edb-as-11.service`; include `/lib/systemd/system/edb-as-11.service` within the file.

Assuming the `LD_LIBRARY_PATH=/tmp/instantclient` you can now include the environment variable by specifying:

```
[Service]
Environment=LD_LIBRARY_PATH=/tmp/instantclient:$LD_LIBRARY_PATH
Environment=ORACLE_HOME=/tmp/instantclient
```

Then, use the following command to reload `systemd`:

```
systemctl daemon-reload
```

Then, restart the Advanced Server service with the following command:

```
systemctl restart edb-as-11
```

The particular script file that needs to be modified to include the `LD_LIBRARY_PATH` setting depends upon the Advanced Server version, the Linux system on which it was installed, and whether it was installed with the graphical installer or an RPM package.

See the appropriate version of the *EDB Postgres Advanced Server Installation Guide* to determine the service script that affects the startup environment. The installation guides can be found at the following location:

EDB docs website

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Oracle Instant Client for Windows

The following instructions apply to Windows hosts running Advanced Server.

When you are executing a SQL command that references a database link to a remote Oracle database, Advanced Server must know where the Oracle instant client library resides on the Advanced Server host.

Set the Windows `PATH` system environment variable to include the Oracle client installation directory that contains the `oci.dll` file.

As an alternative 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 will override the Windows `PATH` environment variable.

To set the `oracle_home` configuration parameter in the `postgresql.conf` file, edit the file, adding 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 will be established with a secondary server (usually a backup server). Later, when the primary server resumes, the client will retain their connection to a secondary server until a new session is established. The new client connections will automatically be established with the primary server. If the primary and secondary servers are out-of-sync, then there is a possibility that the clients that have established a connection to the secondary server and the clients which later connected to the primary server can see a different database view.

Examples

Creating an oci-dblink Database Link

The following example demonstrates using the `CREATE DATABASE LINK` command to create a database link (named `chicago`) that connects an instance of Advanced Server to an Oracle server via an oci-dblink connection. The connection information tells Advanced Server to log in to Oracle as user `admin`, whose password is `mypassword`. Including the `oci` option tells Advanced Server that this is an oci-dblink 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example demonstrates using the `CREATE DATABASE LINK` command to create a database link (named `bedford`) that connects an instance of Advanced Server to another Advanced Server instance via a `postgres_fdw` foreign data wrapper connection. The connection information tells Advanced Server to log in as user `admin`, whose password is `mypassword`. Including the `postgres_fdw` option tells 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

The following examples demonstrate using a database link with Advanced Server to connect to an Oracle database. The examples assume that a copy of the Advanced Server sample application's `emp` table has been created in an Oracle database and a second 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 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 database link, `oralink`.

```
 SELECT * FROM emp@oralink;

 empno | ename  |    job    | mgr  |      hiredate       | sal  | comm | deptno
-------+--------+-----------+------+---------------------+------+------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800 |      |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600 |  300 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250 |  500 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975 |      |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250 | 1400 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850 |      |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450 |      |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000 |      |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000 |      |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500 |    0 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100 |      |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950 |      |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000 |      |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300 |      |     10
(14 rows)
```

Create a private database link named, `fdwlink`, to the Advanced Server database named, `edb`, located on host `192.168.2.22` running on port `5444`. Connect to the Advanced Server database with 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';
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Display attributes of database links, `oralink` and `fdwlink`, from the local `edb_dblink` system table:

```
SELECT lnkname, lnkuser, lnkconnstr FROM pg_catalog.edb_dblink;

 lnkname |   lnkuser    |               lnkconnstr
---------+--------------+----------------------------------------
 oralink | edb          | //127.0.0.1:1521/xe
 fdwlink | enterprisedb |
(2 rows)
```

Perform a join of the `emp` table from the Oracle database with the `dept` table from the 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     | 1300 |
     20 | RESEARCH   |  7369 | SMITH  | CLERK     |  800 |
     20 | RESEARCH   |  7566 | JONES  | MANAGER   | 2975 |
     20 | RESEARCH   |  7788 | SCOTT  | ANALYST   | 3000 |
     20 | RESEARCH   |  7876 | ADAMS  | CLERK     | 1100 |
     20 | RESEARCH   |  7902 | FORD   | ANALYST   | 3000 |
     30 | SALES      |  7499 | ALLEN  | SALESMAN  | 1600 |  300
     30 | SALES      |  7521 | WARD   | SALESMAN  | 1250 |  500
     30 | SALES      |  7654 | MARTIN | SALESMAN  | 1250 | 1400
     30 | SALES      |  7698 | BLAKE  | MANAGER   | 2850 |
     30 | SALES      |  7844 | TURNER | SALESMAN  | 1500 |    0
     30 | SALES      |  7900 | JAMES  | CLERK     |  950 |
(14 rows)
```

Pushdown for an oci Database Link

When the oci-dblink is used to execute SQL statements on a remote Oracle database, there are certain circumstances where pushdown of the processing occurs on the foreign server.

Pushdown refers to the occurrence of processing on the foreign (that is, the remote) server instead of the local client where the SQL statement was issued. Pushdown can result in performance improvement since the data is processed on the remote server before being returned to the local client.

Pushdown applies to statements with the standard SQL join operations (inner join, left outer join, right outer join, and full outer join). Pushdown still occurs even when a sort is specified on the resulting data set.

In order for pushdown to occur, certain 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 (that is, the same database link defined with the `CREATE DATABASE LINK` command).

In order to determine if pushdown is to be used for a SQL statement, display the execution plan by using the `EXPLAIN` command.

For information about the `EXPLAIN` command, see the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/11/static/sql-explain.html

The following examples use the database link created as shown by the following:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password'
USING '//192.168.2.23:1521/xe';
```

The following 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)
```

Note that 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)
```

The following shows the execution plan of a left outer join:

```
 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:

```
 deptno |   dname    | empno | ename
--------+------------+-------+--------
     10 | ACCOUNTING |  7782 | CLARK
     10 | ACCOUNTING |  7839 | KING
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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
    40 | OPERATIONS |       |
(15 rows)
```

The following example shows a case where the entire processing is not pushed down because the `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 e ON d.deptno = e.deptno ORDER BY 1, 3;

                           QUERY PLAN
-----------------------------------------------------------------
 Sort
   Output: d.deptno, d.dname, e.empno, e.ename
   Sort Key: d.deptno, e.empno
   ->  Hash Left Join
         Output: d.deptno, d.dname, e.empno, e.ename
         Hash Cond: (d.deptno = e.deptno)
         ->  Foreign Scan on _dblink_dept_1 d
               Output: d.deptno, d.dname, d.loc
               Remote Query: SELECT deptno, dname, NULL FROM dept
         ->  Hash
               Output: e.empno, e.ename, e.deptno
               ->  Seq Scan on public.emp e
                     Output: e.empno, e.ename, e.deptno
(13 rows)
```

The output of this join is the same as the previous left outer join example.

Creating a Foreign Table from a Database Link

> **Note**
>
> The procedure described in this section is not compatible with Oracle databases.

After you have created a database link, you can create a foreign table based upon this database link. The foreign table can then be used 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.

This technique can be used for either an oci-dblink connection for remote Oracle access, or a `postgres_fdw` connection for remote Postgres access.

The following example shows the creation of a foreign table to access a remote Oracle table.

First, create a database link as previously described. The following is the creation of a database link named `oralink` for connecting to the Oracle database.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE PUBLIC DATABASE LINK oralink CONNECT TO edb IDENTIFIED BY 'password'
USING '//127.0.0.1:1521/xe';
```

The following query shows the database link:

```
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, Advanced Server creates a corresponding foreign server. The following query displays the foreign server:

```
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` command in the PostgreSQL Core documentation at:

https://www.postgresql.org/docs/11/static/sql-createserver.html

Create the foreign table as shown by the following:

```
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'
);
```

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 as displayed 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 may not be accepted or enforced on the foreign table as 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 at:

https://www.postgresql.org/docs/11/static/sql-createforeigntable.html

The following is a query on the foreign table:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SELECT * FROM emp_ora;

 empno | ename  |    job    | mgr  |      hiredate       |   sal   |  comm   | deptno
-------+--------+-----------+------+---------------------+---------+---------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |         |    20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-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 | 02-APR-81 00:00:00 | 2975.00 |         |    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 | 2850.00 |         |    30
  7782 | CLARK  | MANAGER   | 7839 | 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  | 7698 | 08-SEP-81 00:00:00 | 1500.00 |    0.00 |    30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |         |    20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950.00 |         |    30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |         |    20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |         |    10
(14 rows)
```

In contrast, the following is a query on the same remote table, but using the database link instead of the foreign table:

```
SELECT * FROM emp@oralink;

 empno | ename  |    job    | mgr  |      hiredate       | sal  | comm | deptno
-------+--------+-----------+------+---------------------+------+------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800 |      |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600 |  300 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250 |  500 |     30
  7566 | JONES  | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975 |      |     20
  7654 | MARTIN | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250 | 1400 |     30
  7698 | BLAKE  | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850 |      |     30
  7782 | CLARK  | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450 |      |     10
  7788 | SCOTT  | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000 |      |     20
  7839 | KING   | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000 |      |     10
  7844 | TURNER | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500 |    0 |     30
  7876 | ADAMS  | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100 |      |     20
  7900 | JAMES  | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950 |      |     30
  7902 | FORD   | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000 |      |     20
  7934 | MILLER | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300 |      |     10
(14 rows)
```

> **Note**
>
> For backward compatibility reasons, it is still possible to write `USING libpq` rather than `USING postgres_fdw`. However, the `libpq` connector is missing many important optimizations which are present in the `postgres_fdw` connector. Therefore, the `postgres_fdw` connector should be used whenever possible. The `libpq` option is deprecated and may be removed entirely in a future Advanced Server release.
>
> See Also
>
> DROP DATABASE LINK

15.20 CREATE DIRECTORY

Name

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`CREATE DIRECTORY` -- create an alias for a file system directory path.

Synopsis

```
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 the alias is specified 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 given alias.

Parameters

`name`

The directory alias name.

`pathname`

The fully-qualified directory path represented by the alias name. The `CREATE DIRECTORY` command does not create the operating system directory. The physical directory must be created independently using the appropriate operating system commands.

Notes

The operating system user id, `enterprisedb`, must have the appropriate read and/or write privileges on the directory if the `UTL_FILE` package is to be used to create and/or read files using the directory.

The directory alias is stored in the `pg_catalog.edb_dir` system catalog table. Note that `edb_dir` is not a table compatible with Oracle databases.

The directory alias can also be viewed from the Oracle catalog views `SYS.ALL_DIRECTORIES` and `SYS.DBA_DIRECTORIES`, which are compatible with Oracle databases.

Use the `DROP DIRECTORY` command to delete the directory alias. When a directory alias is deleted, the corresponding physical file system directory is not affected. The file system directory must be deleted using the appropriate operating system commands.

In a Linux system, the directory name separator is a forward slash `(/)`.

In a Windows system, the directory name separator can be specified as a forward slash `(/)` or two consecutive backslashes `(\\)`.

Examples

Create an alias named `empdir` for directory `/tmp/empdir` on Linux:

```
CREATE DIRECTORY empdir AS '/tmp/empdir';
```

Create an alias named `empdir` for directory `C:\TEMP\EMPDIR` on Windows:

```
CREATE DIRECTORY empdir AS 'C:/TEMP/EMPDIR';
```

View all of the directory aliases:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SELECT * FROM pg_catalog.edb_dir;

dirname | dirowner |    dirpath     | diracl
--------+----------+----------------+--------
 empdir |       10 | C:/TEMP/EMPDIR |
(1 row)
```

View the directory aliases using a view compatible with Oracle databases:

```
SELECT * FROM SYS.ALL_DIRECTORIES;

    owner     | directory_name | directory_path
--------------+----------------+----------------
 ENTERPRISEDB | EMPDIR         | C:/TEMP/EMPDIR
(1 row)
```

See Also

DROP DIRECTORY

15.21 CREATE FUNCTION

Name

`CREATE FUNCTION` -- define a new function.

Synopsis

```
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> ];
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Description

`CREATE FUNCTION` defines a new function. `CREATE OR REPLACE FUNCTION` will either create a new function, or replace an existing definition.

If a schema name is included, then the function is created in the specified schema. Otherwise it is 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 may share a name (this is called *overloading*). (Overloading of functions is an Advanced Server feature - overloading of stored, standalone functions is not compatible with Oracle databases.)

To update the definition of an existing function, use `CREATE OR REPLACE FUNCTION`. It is not possible to change the name or argument types of a function this way (if you tried, you would actually be creating a new, distinct function). Also, `CREATE OR REPLACE FUNCTION` will not let you change the return type of an existing function. To do that, you must drop and recreate the function. Also when using `OUT` parameters, you cannot change the types of any `OUT` parameters except by dropping the function.

The user that creates the function becomes the owner of the function.

Parameters

`name`

`name` is the identifier of the function.

`parameters`

`parameters` is a list of formal parameters.

`data_type`

`data_type` is the data type of the value returned by the function's `RETURN` statement.

`declarations`

`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`

`statements` are SPL program statements (the `BEGIN - END` block may contain an `EXCEPTION` section).

`IMMUTABLE`

`STABLE`

`VOLATILE`

These attributes inform the query optimizer about the behavior of the function; you can specify only one choice. `VOLATILE` is the default behavior.

- `IMMUTABLE` indicates that the function cannot modify the database and always reaches the same result when given the same argument values; it does not 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 cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for function that depend on database lookups, parameter variables (such as the current time zone), etc.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `VOLATILE` indicates that the function value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

`DETERMINISTIC`

`DETERMINISTIC` is a synonym for `IMMUTABLE`. A `DETERMINISTIC` function cannot modify the database and always reaches the same result when given the same argument values; it does not 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.

`[ 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 that the procedure will 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` whenever any of its arguments are `NULL`. If these clauses are specified, the procedure is not executed when there are `NULL` arguments; instead a `NULL` result is assumed automatically.

`[ EXTERNAL ] SECURITY DEFINER`

`SECURITY DEFINER` specifies that the function will 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 that the function will 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 are used to 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 are used to 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 function cannot be executed in parallel mode. The presence of such a function in a SQL statement forces a serial execution plan. This is the default setting if the `PARALLEL` clause is omitted.

- When set to `RESTRICTED`, the 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- When set to `SAFE`, the function can be executed 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.

`ROWS result_rows`

`result_rows` is a positive number giving the estimated number of rows that the planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption is `1000` 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 be applied 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, much as it would do for a previous `SET LOCAL` command, with the effects of such a command persisting 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 Advanced Server and are not supported by Oracle.

Notes

Advanced Server allows function overloading; that is, the same name can be used for several different functions so long as they have distinct 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 illustrates use of the function.

```
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;

  Name   | Salary  | Commission | Total Compensation
---------+---------+------------+--------------------
 SMITH   |  800.00 |            |           19200.00
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 ALLEN  | 1600.00 |     300.00 |           45600.00
 WARD   | 1250.00 |     500.00 |           42000.00
 JONES  | 2975.00 |            |           71400.00
 MARTIN | 1250.00 |    1400.00 |           63600.00
 BLAKE  | 2850.00 |            |           68400.00
 CLARK  | 2450.00 |            |           58800.00
 SCOTT  | 3000.00 |            |           72000.00
 KING   | 5000.00 |            |          120000.00
 TURNER | 1500.00 |       0.00 |           36000.00
 ADAMS  | 1100.00 |            |           26400.00
 JAMES  |  950.00 |            |           22800.00
 FORD   | 3000.00 |            |           72000.00
 MILLER | 1300.00 |            |           31200.00
(14 rows)
```

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 arguments' default values for the first two calls.

```
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 '
        || '$2000.00 and $3000.00: ' || sal_range(2000.00, 3000.00));

END;

Number of employees with a salary: 14
Number of employees with a salary of at least $2000.00: 6
Number of employees with a salary between $2000.00 and $3000.00: 5
```

The following example demonstrates using 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 = 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 will not execute.

The `dept_salaries` function executes with the privileges of the role that is calling the function. If the current user does not have sufficient privileges to perform the `SELECT` statement querying the `emp` table (to display employee salaries), the function will report 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Other Pragmas (declared within a package specification)

`PRAGMA RESTRICT_REFERENCES`

Advanced Server accepts but ignores syntax referencing `PRAGMA RESTRICT_REFERENCES` .

See Also

DROP FUNCTION

15.22 CREATE INDEX

Name

`CREATE INDEX` -- define a new index.

Synopsis

```
CREATE [ UNIQUE ] INDEX <name> ON <table>
  ( { <column> | ( <expression> ) } )
  [ TABLESPACE <tablespace> ]
```

Description

`CREATE INDEX` constructs an index, `name` , on the specified table. Indexes are primarily used to enhance database performance (though inappropriate use will result in slower performance).

The key field(s) for the index are specified as column names, constants, or as expressions written in parentheses. Multiple fields can be specified to create multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on `UPPER(col)` would allow the clause `WHERE UPPER(col) = 'JIM'` to use an index.

Advanced Server provides the B-tree index method. The B-tree index method is an implementation of Lehman-Yao high-concurrency B-trees.

Indexes are not used for `IS NULL` clauses by default.

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 remember to mark the function immutable when you create it.

If you create an index on a partitioned table, the `CREATE INDEX` command does propagate indexes to the table's subpartitions.

Parameters

`UNIQUE`

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

`name`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table.

`table`

The name (possibly schema-qualified) of the table to be indexed.

`column`

The name of a column in the table.

`expression`

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses may be omitted if the expression has the form of a function call.

`tablespace`

The tablespace in which to create the index. If not specified, `default_tablespace` is used, or the database's default tablespace if `default_tablespace` is an empty string.

Notes

Up to 32 fields may be specified in a multicolumn index.

Examples

To create a B-tree index on the column, `ename`, in the table, `emp`:

```
CREATE INDEX name_idx ON emp (ename);
```

To create the same index as above, but have it reside in the `index_tblspc` tablespace:

```
CREATE INDEX name_idx ON emp (ename) TABLESPACE index_tblspc;
```

See Also

ALTER INDEX, DROP INDEX

15.23 CREATE MATERIALIZED VIEW

Name

`CREATE MATERIALIZED VIEW` -- define a new materialized view.

Synopsis

```
CREATE MATERIALIZED VIEW <name>
  [<build_clause>][<create_mv_refresh>] AS subquery

    Where <build_clause> is:
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
BUILD {IMMEDIATE | DEFERRED}

Where <create_mv_refresh> is:

REFRESH [COMPLETE] [ON DEMAND]
```

Description

`CREATE MATERIALIZED VIEW` defines a view of a query that is not 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 the population of the view.

A materialized view may be schema-qualified; if you specify a schema name when invoking the `CREATE MATERIALIZED VIEW` command, the view will be created in the specified schema. The view name must be distinct 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 be created.

`subquery`

A `SELECT` statement that specifies the contents of the view. Refer to `SELECT` for more information about valid queries.

`build_clause`

Include a `build_clause` to specify when the view should be populated. Specify `BUILD IMMEDIATE`, or `BUILD DEFERRED`:

- `BUILD IMMEDIATE` instructs the server to populate the view immediately. This is the default behavior.
- `BUILD DEFERRED` instructs the server to populate the view at a later time (during a `REFRESH` operation).

`create_mv_refresh`

Include the `create_mv_refresh` clause to specify when the contents of a materialized view should be updated. The clause contains the `REFRESH` keyword followed by `COMPLETE` and/or `ON DEMAND`, where:

- `COMPLETE` instructs the server to discard the current content and reload the materialized view by executing the view's defining query when the materialized view is refreshed.

- `ON DEMAND` instructs the server to refresh the materialized view on demand by calling the `DBMS_MVIEW` package or by calling the Postgres `REFRESH MATERIALIZED VIEW` statement. This is the default behavior.

Notes

Materialized views are read only - the server will not allow an `INSERT`, `UPDATE`, or `DELETE` on a view.

Access to tables referenced in the view is determined by permissions of the view owner; the user of a view must have permissions to call all functions used by the view.

For more information about the Postgres `REFRESH MATERIALIZED VIEW` command, see the PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/11/static/sql-refreshmaterializedview.html

Examples

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following statement creates a materialized view named `dept_30`:

```
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`.

15.24 CREATE PACKAGE

Name

`CREATE PACKAGE` -- define a new package specification.

Synopsis

```
CREATE [ OR REPLACE ] PACKAGE <name>
[ AUTHID { DEFINER | CURRENT_USER } ]
{ IS | AS }
  [ <declaration>; ] [, ...]
  [ { PROCEDURE <proc_name>
      [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT <value> ]
        [, ...]) ];
      [ PRAGMA RESTRICT_REFERENCES(<name>,
        { RNDS | RNPS | TRUST | WNDS | WNPS } [, ... ] ); ]
    |
      FUNCTION <func_name>
      [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT <value> ]
        [, ...]) ]
      RETURN <rettype> [ DETERMINISTIC ];
      [ PRAGMA RESTRICT_REFERENCES(<name>,
        { RNDS | RNPS | TRUST | WNDS | WNPS } [, ... ] ); ]
    }
  ] [, ...]
  END [ <name> ]
```

Description

`CREATE PACKAGE` defines a new package specification. `CREATE OR REPLACE PACKAGE` will either create a new package specification, or replace an existing specification.

If a schema name is included, then the package is created in the specified schema. Otherwise it is created in the current schema. The name of the new package must not match any existing package in the same schema unless the intent is to update the definition of an existing package, in which 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Specifies whether the privileges of the package owner (`DEFINER`) or the privileges of the current user executing a program in the package (`CURRENT_USER`) are to be used to determine whether or not 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 type(s) of the program's arguments.

`DEFAULT value`

Default value of an input argument.

`func_name`

The name of a public function.

`rettype`

The return data type.

`DETERMINISTIC`

`DETERMINISTIC` is a synonym for `IMMUTABLE` . A `DETERMINISTIC` procedure cannot modify the database and always reaches the same result when given the same argument values; it does not 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.

`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.

```
CREATE OR REPLACE PACKAGE empinfo
IS
    emp_name        VARCHAR2(10);
    PROCEDURE get_name (
        p_empno     NUMBER
    );
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    FUNCTION display_counter
    RETURN INTEGER;
END;
```

See Also

DROP PACKAGE

15.25 CREATE PACKAGE BODY

Name

`CREATE PACKAGE BODY` -- define a new package body.

Synopsis

```
 CREATE [ OR REPLACE ] PACKAGE BODY <name>
{ IS | AS }
  [ declaration; ] [, ...]
  [ { PROCEDURE <proc_name>
      [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT <value> ] [, ...]) ]
      [ STRICT ]
      [ LEAKPROOF ]
      [ PARALLEL { UNSAFE | RESTRICTED | SAFE } ]
      [ COST <execution_cost> ]
      [ ROWS <result_rows> ]
      [ SET <config_param> { TO <value> | = <value> | FROM CURRENT } ]
    { IS | AS }
        <program_body>
      END [ <proc_name> ];
    |
      FUNCTION <func_name>
      [ (<argname> [ IN | IN OUT | OUT ] <argtype> [ DEFAULT <value> ] [, ...]) ]
      RETURN <rettype> [ DETERMINISTIC ]
      [ STRICT ]
      [ LEAKPROOF ]
      [ PARALLEL { UNSAFE | RESTRICTED | SAFE } ]
      [ COST <execution_cost> ]
      [ ROWS <result_rows> ]
      [ SET <config_param> { TO <value> | = <value> | FROM CURRENT } ]
    { IS | AS }
        <program_body>
      END [ <func_name> ];
    }
  ] [, ...]
  [ BEGIN
      <statement>; [, ...] ]
  END [ <name> ]
```

Description

`CREATE PACKAGE BODY` defines a new package body. `CREATE OR REPLACE PACKAGE BODY` will either create a new package body, or replace an existing body.

If a schema name is included, then the package body is created in the specified schema. Otherwise it is created in the current schema. The name of the new

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

package body must match an existing package specification in the same schema. The new package body name must not match any existing package body in the same schema unless the intent is to update the definition of an existing package body, in which case use `CREATE OR REPLACE PACKAGE BODY`.

Parameters

`name`

The name (optionally schema-qualified) of the package body to create.

`declaration`

A private variable, type, cursor, or `REF CURSOR` declaration.

`proc_name`

The name of a public or private procedure. If `proc_name` exists in the package specification with an identical signature, then it is public, otherwise it is private.

`argname`

The name of an argument.

`IN | IN OUT | OUT`

The argument mode.

`argtype`

The data type(s) of the program's arguments.

`DEFAULT value`

Default value of an input argument.

`STRICT`

The `STRICT` keyword specifies that the function will not be executed if called with a `NULL` argument; instead the function will return `NULL`.

`LEAKPROOF`

The `LEAKPROOF` keyword specifies that the function will not 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 cannot be executed in parallel mode. The presence of such a procedure or function forces a serial execution plan. This is the default setting if the `PARALLEL` clause is omitted.
- 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 with no restriction.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 that the query planner should 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 may be `PRAGMA AUTONOMOUS_TRANSACTION` to set the function or procedure as an autonomous transaction.

The declarations may include variable, type, `REF CURSOR`, or subprogram declarations. If subprogram declarations are included, they must be declared 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 is public, otherwise it is private.

`rettype`

The return data type.

`DETERMINISTIC`

Include `DETERMINISTIC` to specify that the function will 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 `DETERMINISTIC` is specified for a public function in the package body, it must also be specified for the function declaration in the package specification. For private functions, there is no function declaration in the package specification.

`statement`

An SPL program statement. Statements in the package initialization section are executed once per session the first time the package is referenced.

> **Note**
>
> The `STRICT`, `LEAKPROOF`, `PARALLEL`, `COST`, `ROWS` and `SET` keywords provide extended functionality for Advanced Server and are not supported by Oracle.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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;
```

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

BEGIN
    empinfo.get_name(7900);
    DBMS_OUTPUT.PUT_LINE('Employee Name    : ' || empinfo.emp_name);
    DBMS_OUTPUT.PUT_LINE('Number of queries: ' || empinfo.display_counter);
END;

Employee Name    : JAMES
Number of queries: 2
```

See Also

CREATE PACKAGE, DROP PACKAGE

15.26 CREATE PROCEDURE

Name

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`CREATE PROCEDURE` -- define a new stored procedure.

Synopsis

```
CREATE [OR REPLACE] PROCEDURE <name> [ (<parameters>) ]
   [
         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> ];
```

Description

`CREATE PROCEDURE` defines a new stored procedure. `CREATE OR REPLACE PROCEDURE` will either create a new procedure, or replace an existing definition.

If a schema name is included, then the procedure is created in the specified schema. Otherwise it is 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 may share a name (this is called *overloading*). (Overloading of procedures is an Advanced Server feature - overloading of stored, standalone procedures is not compatible with Oracle databases.)

To update the definition of an existing procedure, use `CREATE OR REPLACE PROCEDURE` . It is not possible to change the name or argument types of a procedure this way (if you tried, you would actually be creating a new, distinct procedure). When using `OUT` parameters, you cannot change the types of any `OUT` parameters except by dropping the procedure.

Parameters

`name`

`name` is the identifier of the procedure.

`parameters`

`parameters` is a list of formal parameters.

`declarations`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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`

`statements` are SPL program statements (the `BEGIN - END` block may 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 cannot modify the database and always reaches the same result when given the same argument values; it does not 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.
- `STABLE` indicates that the procedure cannot modify the database, and that within a single table scan, it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for procedures that depend on database lookups, parameter variables (such as the current time zone), etc.
- `VOLATILE` indicates that the procedure value can change even within a single table scan, so no optimizations can be made. Please note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away.

`DETERMINISTIC`

`DETERMINISTIC` is a synonym for `IMMUTABLE`. A `DETERMINISTIC` procedure cannot modify the database and always reaches the same result when given the same argument values; it does not 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 will 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` whenever any of its arguments are `NULL`. If these clauses are specified, the procedure is not executed when there are `NULL` arguments; instead a `NULL` result is assumed automatically.

`[ EXTERNAL ] SECURITY DEFINER`

`SECURITY DEFINER` specifies that the procedure will 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `SECURITY INVOKER` clause indicates that the procedure will 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 procedure owner are used to 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 procedure are used to 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 procedure cannot be executed in parallel mode. The presence of such a procedure forces a serial execution plan. This is the default setting if the `PARALLEL` clause is omitted.
- When set to `RESTRICTED`, the procedure 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 can be executed 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 that the planner should expect the procedure to return. This is only allowed when the procedure is declared to return a set. The default assumption is 1000 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 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 be applied 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 would do for a previous `SET LOCAL` command, with the effects of such a command persisting 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 Advanced Server and are not supported by Oracle.
> - The `IMMUTABLE`, `STABLE`, `STRICT`, `LEAKPROOF`, `COST`, `ROWS` and `PARALLEL { UNSAFE | RESTRICTED | SAFE }`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

attributes are only supported for EDB SPL procedures.

- By default, stored procedures are created as `SECURITY DEFINERS`; stored procedures defined in plpgsql are created as `SECURITY INVOKERS`.

Examples

The following procedure lists the employees in the `emp` table:

```
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;

EXEC list_emp;

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
```

The following procedure uses `IN OUT` and `OUT` arguments to return an employee's number, name, and job based upon a search using first, the given employee number, and if that is not found, then using 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    p_job   := v_job;
    DBMS_OUTPUT.PUT_LINE('Found employee # ' || p_empno);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        BEGIN
            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);
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('Could not find an employee with ' ||
                    'number, ' || p_empno || ' nor name, '  || p_ename);
                p_empno := NULL;
                p_ename := NULL;
                p_job   := NULL;
        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);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
END;

Found employee CLARK
Employee No: 7782
Name       : CLARK
Job        : MANAGER
```

The following example demonstrates 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 is 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 will not be affected by these settings.

In this example, the `AUTHID DEFINER` clause temporarily grants privileges to a role that might otherwise not be allowed to execute the statements within 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.

See Also

DROP PROCEDURE, ALTER PROCEDURE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

15.27 CREATE PROFILE

Name

`CREATE PROFILE` -- create a new profile.

Synopsis

```
CREATE PROFILE <profile_name>
       [LIMIT {<parameter value>} ... ];
```

Description

`CREATE PROFILE` creates a new profile. Include the `LIMIT` clause and one or more space-delimited `parameter/value` pairs to specify the rules enforced by Advanced Server.

Advanced Server creates a default profile named `DEFAULT`. When you use the `CREATE ROLE` command to create a new role, the new role is automatically associated with the `DEFAULT` profile. If you upgrade from a previous version of Advanced Server to Advanced Server 10, the upgrade process will automatically create 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 Advanced Server.

Parameters

`profile_name`

The name of the profile.

`parameter`

The password attribute that will be monitored by the rule.

`value`

The value the `parameter` must reach before an action is taken by the server.

Advanced Server supports the `value` shown below for each `parameter`:

`FAILED_LOGIN_ATTEMPTS` specifies the number of failed login attempts that a user may make before the server locks the user 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 may 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 has been locked because of `FAILED_LOGIN_ATTEMPTS`. Supported values are:

- A `NUMERIC` value greater than 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_LOCK_TIME` specified in the `DEFAULT` profile.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `UNLIMITED` – the account is locked until it is manually unlocked by a database superuser.

`PASSWORD_LIFE_TIME` specifies the number of days that the current password may 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 that will pass after the password expires before connections by the role are rejected. If `PASSWORD_GRACE_TIME` is not specified, the password will expire on the day specified by the default value of `PASSWORD_GRACE_TIME`, and the user will not be allowed to execute any command until a new password is provided. Supported values are:

- A `NUMERIC` value greater than 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 does not 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 will be allowed to connect, but will not be allowed to execute any command until they update their expired password. Supported values are:

- A `NUMERIC` value greater than 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_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 re-using a password. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used 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 greater than 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_REUSE_TIME` specified in the `DEFAULT` profile.
- `UNLIMITED` – The password can be re-used without restrictions.

`PASSWORD_REUSE_MAX` specifies the number of password changes that must occur before a password can be reused. The `PASSWORD_REUSE_TIME` and `PASSWORD_REUSE_MAX` parameters are intended to be used 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 or equal to `0`.
- `DEFAULT` - the value of `PASSWORD_REUSE_MAX` specified in the `DEFAULT` profile.
- `UNLIMITED` – The password can be re-used 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 to be allowed for use or not. 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 in order to be validated effectively, else an error will be 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` is not an Oracle-compatible parameter.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Notes

Use `DROP PROFILE` command to remove the profile.

Examples

The following command creates a profile named `acctg`. The profile specifies that if a user has not authenticated with the correct password in five attempts, the account will be 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 has not changed their password before the 90 days specified in the profile has passed, they will be issued a warning at login. After a grace period of 3 days, their account will not be allowed to invoke any commands until they change their password.

The following command creates a profile named `accts`. The profile specifies that a user cannot re-use a password within 180 days of the last use of the password, and must change their password at least 5 times before re-using 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 will verify that the password provided meets their standards for complexity:

```
CREATE PROFILE resources LIMIT
       PASSWORD_VERIFY_FUNCTION password_rules;
```

See Also

ALTER PROFILE, DROP PROFILE

15.28 CREATE QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the `CREATE QUEUE SQL` command. This syntax can be used in association with `DBMS_AQADM`.

Name

`CREATE QUEUE` -- create a queue.

Synopsis

Use `CREATE QUEUE` to define a new queue:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE QUEUE <name> QUEUE TABLE <queue_table_name> [ ( { <option_name option_value> } [, ... ] ) ]
```

where `option_name` and the corresponding `option_value` can be:

```
TYPE [normal_queue | exception_queue]
RETRIES [INTEGER]
RETRYDELAY [DOUBLE PRECISION]
RETENTION [DOUBLE PRECISION]
```

Description

The `CREATE QUEUE` command allows a database superuser or any user with the system-defined `aq_administrator_role` privilege to create a new queue in the current database.

If the name of the queue is schema-qualified, the queue is created in the specified schema. If a schema is not included in the `CREATE QUEUE` command, the queue is created in the current schema. A queue may only be created in the schema in which the queue table resides. The name of the queue must be unique from the name of any other queue in the same schema.

Use `DROP QUEUE` to remove a queue.

Parameters

`name`

The name (optionally schema-qualified) of the queue to be created.

`queue_table_name`

The name of the queue table with which this queue is associated.

`option_name option_value`

The name of any options that will be associated with the new queue, and the corresponding value for the option. If the call to `CREATE QUEUE` includes duplicate option names, the server will return an error. The following values are accepted:

- `TYPE` : Specify `normal_queue` to indicate that the queue is a normal queue, or `exception_queue` to indicate that the queue is an exception queue. An exception queue will only accept 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 will wait before retrying a message.
- `RETENTION` : A `DOUBLE PRECISION` value that specifies the number of seconds that a message will be saved in the queue table after dequeueing.

Examples

The following command creates a queue named `work_order` that is associated with a queue table named `work_order_table` :

```
CREATE QUEUE work_order QUEUE TABLE work_order_table (RETRIES 5, RETRYDELAY 2);
```

The server will allow 5 attempts to remove a message from the queue, and enforce a retry delay of 2 seconds between attempts.

See Also

ALTER QUEUE, DROP QUEUE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

15.29 CREATE QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the `CREATE QUEUE TABLE SQL` command. This syntax can be used in association with `DBMS_AQADM`.

Name

`CREATE QUEUE TABLE` -- create a new queue table.

Synopsis

Use `CREATE QUEUE TABLE` to define a new queue table:

```
CREATE QUEUE TABLE <name> OF <type_name> [ ( { <option_name option_value> } [, ... ] ) ]
```

where `option_name` and the corresponding `option_value` can be:

- `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` Where `storage_option` is 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}`.

Please note that 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 the table will be created.

Description

`CREATE QUEUE TABLE` allows a superuser or a user with the `aq_administrator_role` privilege to create a new queue table.

If the call to `CREATE QUEUE TABLE` includes a schema name, the queue table is created in the specified schema. If no schema name is provided, the new queue table is created in the current schema.

The name of the queue table must be unique from the name of any other queue table 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 will be associated with the new queue, and the corresponding value for the option. If the call to `CREATE QUEUE` includes duplicate option names, the server will return an error. The following values are accepted:

- `SORT_LIST` : Use the `SORT_LIST` option to control the dequeueing order of the queue; specify the names of the column(s) that will be used to

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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`

Any other value will return an `ERROR`.

- `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 indicate that each message should be dequeued individually, or `transactional` to indicate that messages that are added to the queue as a result of one transaction should be dequeued as a group.

- `STORAGE_CLAUSE`: Use `STORAGE_CLAUSE` to specify table attributes. `STORAGE_CLAUSE` may be `TABLESPACE tablespace_name`, `PCTFREE integer`, `PCTUSED integer`, `INITRANS integer`, `MAXTRANS integer`, `STORAGE storage_option` Where `storage_option` is 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}`.

Please note that 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 the table will be created.

Examples

You must create a user-defined type before creating a queue table; the type describes the columns and data types within the table. The following command creates a type named `work_order`:

```
CREATE TYPE work_order AS (name VARCHAR2, project TEXT, completed BOOLEAN);
```

The following 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

ALTER QUEUE TABLE, DROP QUEUE TABLE

15.30 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 new role to the 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 it is used. The newly created role does not have the `LOGIN` attribute, so it cannot be used 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 the `CREATE ROLE` command.

If the `IDENTIFIED BY` clause is specified, 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, and so are 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 only of use for roles having the `LOGIN` attribute, but you can nonetheless define one for roles without it.) If you do not plan to use password authentication you can omit this option.

Notes

Use `ALTER ROLE` to change the attributes of a role, and `DROP ROLE` to remove a role. The attributes specified by `CREATE ROLE` can be modified by later `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:

```
CREATE ROLE admins IDENTIFIED BY Rt498zb;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

See Also

ALTER ROLE, DROP ROLE, GRANT, REVOKE, SET ROLE

15.31 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 new schema owned by `username` and populated with one or more objects. The creation of the schema and objects occur within a single transaction so either all objects are created or none of them including the schema. (Please note: if you are using an Oracle database, no new schema is created – `username`, and therefore the schema, must pre-exist.)

A schema is essentially a namespace: it contains named objects (tables, views, etc.) whose names may duplicate those of other objects existing in other schemas. Named objects are accessed either by "qualifying" their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). Unqualified objects are created in the current schema (the one at the front of the search path, which can be determined with the function `CURRENT_SCHEMA`). (The search path concept and the `CURRENT_SCHEMA` function are not compatible with Oracle databases.)

`CREATE SCHEMA` includes subcommands to create objects within the schema. The subcommands are treated essentially the same as separate commands issued after creating the schema. All the created objects will be owned by the specified user.

Parameters

`username`

The name of the user who will own the new schema. The schema will be named the same as `username`. Only superusers may create schemas owned by users other than themselves. (Please note: In Advanced Server the role, `username`, must already exist, but the schema must not exist. In Oracle, the user (equivalently, the schema) must exist.)

`schema_element`

An SQL statement defining an object to be created within the schema. `CREATE TABLE`, `CREATE VIEW`, and `GRANT` are accepted as clauses within `CREATE SCHEMA`. Other kinds of objects may be created in separate commands after the schema is created.

Notes

To create a schema, the invoking user must have the `CREATE` privilege for the current database. (Of course, superusers bypass this check.)

In Advanced Server, there are other forms of the `CREATE SCHEMA` command that are not 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

15.32 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 new sequence number generator. This involves creating and initializing a new special single-row table with the name, `name` . The generator will be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema, otherwise it is created in the current schema. The sequence name must be distinct from the name of any other sequence, table, index, or view in the same schema.

After a sequence is created, use the functions `NEXTVAL` and `CURRVAL` to operate on the sequence. These functions are documented in Sequence Manipulation Functions of *Database Compatibility for Oracle Developers Reference Guide*.

Parameters

`name`

The name (optionally schema-qualified) of the sequence to be created.

`increment`

The optional clause `INCREMENT BY increment` specifies the value to add to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is `1` .

`NOMINVALUE | MINVALUE minvalue`

The optional clause `MINVALUE minvalue` determines the minimum value a sequence can generate. If this clause is not supplied, then defaults will be used. The defaults are 1 and -2^{63}-1 for ascending and descending sequences, respectively. Note that the key words, `NOMINVALUE` , may be used to set this behavior to the default.

`NOMAXVALUE | MAXVALUE maxvalue`

The optional clause `MAXVALUE maxvalue` determines the maximum value for the sequence. If this clause is not supplied, then default values will be used. The defaults are 2^{63}-1 and -1 for ascending and descending sequences, respectively. Note that the key words, `NOMAXVALUE` , may be used 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The optional clause `CACHE cache` specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is `1` (only one value can be generated at a time, i.e., `NOCACHE`), and this is also the default.

`CYCLE`

The `CYCLE` option allows the sequence to wrap around when the `maxvalue` or `minvalue` has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the `minvalue` or `maxvalue`, respectively.

If `CYCLE` is omitted (the default), any calls to `NEXTVAL` after the sequence has reached its maximum value will return an error. Note that the key words, `NO CYCLE`, may be used to obtain the default behavior, however, this term is not compatible with Oracle databases.

Notes

Sequences are based on big integer arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807). On some older platforms, there may be no compiler support for eight-byte integers, in which case sequences use regular `INTEGER` arithmetic (range -2147483648 to +2147483647).

Unexpected results may be obtained if a `cache` setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Each session will allocate and cache successive sequence values during one access to the sequence object and increase the sequence object's last value accordingly. Then, the next `cache-1` uses of `NEXTVAL` within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in "holes" in the sequence.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values may 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=1`, then session B might reserve values 11..20 and return `NEXTVAL=11` before session A has generated `NEXTVAL=2`. Thus, with a `cache` setting of one it is safe to assume that `NEXTVAL` values are generated sequentially; with a `cache` setting greater than one you should only assume that the `NEXTVAL` values are all distinct, not that they are generated purely sequentially. Also, the last value will reflect the latest value reserved by any session, whether or not it has yet been 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 nextval
---------
       1
(1 row)
```

See Also

ALTER SEQUENCE, DROP SEQUENCE

15.33 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. 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`

`syn_name` is the name of the synonym. A synonym name must be unique within a schema.

`schema`

`schema` specifies the name of the schema that the synonym resides in. If you do not specify a schema name, the synonym is created in the first existing schema in your search path.

`object_name`

`object_name` specifies the name of the object.

`object_schema`

`object_schema` specifies the name of the schema that the referenced object resides in.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`dblink_name`

`dblink_name` specifies the name of the database link through which an object is accessed.

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. The `CREATE PUBLIC SYNONYM` command, compatible with Oracle databases, creates a synonym that resides in the `public` schema:

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

This just a shorthand way to write:

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

Notes

Access to the object referenced by the synonym is determined by the permissions of the current user of the synonym; the synonym user must have the appropriate permissions on the underlying database object.

Examples

Create a synonym for the `emp` table in a schema named, `enterprisedb` :

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

See Also

DROP SYNONYM

15.34 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> ]
```

where `column_constraint` is:

```
[ CONSTRAINT <constraint_name> ]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
{ NOT NULL |
  NULL |
  UNIQUE [ USING INDEX TABLESPACE <tablespace> ] |
  PRIMARY KEY [ USING INDEX TABLESPACE <tablespace> ] |
  CHECK (<expression>) |
  REFERENCES <reftable> [ ( <refcolumn> ) ]
    [ ON DELETE <action> ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
  INITIALLY IMMEDIATE ]
```

and `table_constraint` is:

```
 [ CONSTRAINT <constraint_name> ]
{ UNIQUE ( <column_name> [, ...] )
    [ USING INDEX TABLESPACE <tablespace> ] |
  PRIMARY KEY ( <column_name> [, ...] )
    [ USING INDEX TABLESPACE <tablespace> ] |
  CHECK ( <expression> ) |
  FOREIGN KEY ( <column_name> [, ...] )
      REFERENCES <reftable> [ ( <refcolumn> [, ...] ) ]
    [ ON DELETE <action> ] }
[ DEFERRABLE | NOT DEFERRABLE ]
[ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Description

`CREATE TABLE` will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

If a schema name is given (for example, `CREATE TABLE myschema.mytable ...`) then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables exist in a special schema, so a schema name may not be given when creating a temporary table. The table name must be distinct from the name of any other table, sequence, index, or view in the same schema.

`CREATE TABLE` also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

A table cannot 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 an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience if the constraint only affects one column.

Parameters

`GLOBAL TEMPORARY`

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see `ON COMMIT` below). Existing permanent tables with the same name are not visible to the current session while the temporary table exists, unless they are referenced with schema-qualified names. In addition, temporary tables are not visible outside the session in which it was created. (This aspect of global temporary tables is not compatible with Oracle databases.) Any indexes created on a temporary table are automatically temporary as well.

`UNLOGGED`

If specified, the table is created as an unlogged table. The data written to unlogged tables is not written to the write-ahead log (WAL), making them faster than an ordinary table. Indexes created on an unlogged table are automatically unlogged. The contents of an unlogged table are not replicated to a

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

standby server. The unlogged table is automatically truncated after a crash or shutdown.

`table_name`

The name (optionally schema-qualified) of the table to be created.

`column_name`

The name of a column to be created in the new table.

`data_type`

The data type of the column. This may include array specifiers. For more information on the data types included with Advanced Server, refer to Data Types of *Database Compatibility for Oracle Developers Reference Guide*.

`DEFAULT default_expr`

The `DEFAULT` clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is `null`.

`CONSTRAINT constraint_name`

An optional name for a column or table constraint. If not specified, the system generates a name.

`NOT NULL`

The column is not allowed to contain null values.

`NULL`

The column is allowed to contain null values. This is the default.

This clause is only available for compatibility with non-standard SQL databases. Its use is discouraged 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 may contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

For the purpose of a unique constraint, null values are not considered equal.

Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise it would just be the same constraint listed twice.)

`PRIMARY KEY` - column constraint

`PRIMARY KEY ( column_name [, ...] )` - table constraint

The primary key constraint specifies that a column or columns of a table may contain only unique (non-duplicate), non-null values. Technically, `PRIMARY`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`KEY` is merely a combination of `UNIQUE` and `NOT NULL`, but identifying a set of columns as primary key also provides metadata about the design of the schema, as a primary key implies that other tables may rely on this set of columns as a unique identifier for rows.

Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

The primary key constraint should name a set of columns that is 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 which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or "unknown" succeed. Should any row of an insert or update operation produce a FALSE result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint may reference multiple columns.

Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than columns of the current row.

`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 one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If `refcolumn` is omitted, 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 is changed, 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. Referential actions cannot be deferred 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, respectively.

- `SET NULL`

  Set the referencing column(s) to `NULL`.

  If the referenced column(s) are changed frequently, it may be wise to add an index to the foreign key column so that referential actions associated with the foreign key column can be performed more efficiently.

`DEFERRABLE`

`NOT DEFERRABLE`

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable may be postponed 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 are not 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 is checked after each statement. This is the default. If the constraint is `INITIALLY DEFERRED`, it is checked only at the end of the transaction. The constraint check time can be altered with the `SET CONSTRAINTS` command.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`WITH ( ROWIDS [= value ] )`

The `ROWIDS` option for a table include `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. If a value is not specified then 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`

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The two options are:

- `PRESERVE ROWS`

  No special action is taken at the ends of transactions. This is the default behavior. (Note that this aspect is not compatible with Oracle databases. The Oracle default is `DELETE ROWS`.)

- `DELETE ROWS`

  All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic `TRUNCATE` is done at each commit.

`TABLESPACE tablespace`

The `tablespace` is the name of the tablespace in which the new table is to be created. If not specified, `default tablespace` is used, or the database's default tablespace if `default_tablespace` is an empty string.

`USING INDEX TABLESPACE tablespace`

This clause allows selection of the tablespace in which the index associated with a `UNIQUE` or `PRIMARY KEY` constraint will be created. If not specified, `default tablespace` is used, or the database's default tablespace if `default_tablespace` is an empty string.

Notes

Advanced Server automatically creates an index for each unique constraint and primary key constraint to enforce the uniqueness. Thus, it is not necessary to create an explicit index for primary key columns. (See `CREATE INDEX` for more information.)

Examples

Create table `dept` and table `emp`:

```
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)
);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Define a unique table constraint for the table `dept`. Unique table constraints can be defined on one or more columns of the 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)
);
```

Define a check column constraint:

```
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) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
```

Define a check table constraint:

```
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 primary key table constraint for the table `jobhist`. Primary key table constraints can be defined on one or more columns of the 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(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate)
);
```

This assigns a literal constant default value for the column, `job` and makes the default value of `hiredate` be the date at which the row is inserted.

```
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9) DEFAULT 'SALESMAN',
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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`:

```
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14),
    loc             VARCHAR2(13)
) TABLESPACE diskvol1;
```

See Also

ALTER TABLE, DROP TABLE

15.35 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 } ]
  [ TABLESPACE 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` (except that you can override the column names by giving an explicit list of new column names).

`CREATE TABLE AS` bears some resemblance to creating a view, but it is really quite different: it creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query. In contrast, a view re-evaluates its defining `SELECT` statement whenever it is queried.

Parameters

`GLOBAL TEMPORARY`

If specified, the table is created as a temporary table. Refer to `CREATE TABLE` for details.

`table_name`

The name (optionally schema-qualified) of the table to be created.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`column_name`

The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query.

`query`

A query statement (a `SELECT` command). Refer to `SELECT` for a description of the allowed syntax.

15.36 CREATE TRIGGER

Name

`CREATE TRIGGER` -- define a new trigger.

Synopsis

```
CREATE [ OR REPLACE ] TRIGGER <name>
  { BEFORE | AFTER | INSTEAD OF }
  { INSERT | UPDATE | DELETE }
      [ OR { INSERT | UPDATE | DELETE } ] [, ...]
    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
```

Description

`CREATE TRIGGER` defines a new trigger. `CREATE OR REPLACE TRIGGER` will either create a new trigger, or replace an existing definition.

If you are using the `CREATE TRIGGER` keywords to create a new trigger, the name of the new trigger must not match any existing trigger defined on the same table. New triggers will be created in the same schema as the table on which the triggering event is defined.

If you are updating the definition of an existing trigger, use the `CREATE OR REPLACE TRIGGER` keywords.

When you use syntax that is 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Determines whether the trigger is fired before or after the triggering event.

`INSTEAD OF`

`INSTEAD OF` trigger modifies an updatable view; the trigger will execute to update the underlying table(s) appropriately. The `INSTEAD OF` trigger is executed for each row of the view that is updated or modified.

`INSERT | UPDATE | DELETE`

Defines the triggering event.

`table`

The name of the table on which the triggering event occurs.

`condition`

`condition` is a Boolean expression that determines if the trigger will actually be executed; if `condition` evaluates to `TRUE`, the trigger will fire.

- If the trigger definition includes the `FOR EACH ROW` keywords, the `WHEN` clause can refer to columns of the old and/or new row values by writing `OLD.column_name` or `NEW.column_name` respectively. `INSERT` triggers cannot refer to `OLD` and `DELETE` triggers cannot refer to `NEW`.

- If the trigger includes the `INSTEAD OF` keywords, it may not include a `WHEN` clause. A `WHEN` clause cannot contain subqueries.

`REFERENCING { OLD AS old | NEW AS new } ...`

`REFERENCING` clause to reference old rows and new rows, but restricted in that `old` may only be replaced by an identifier named old or any equivalent that is saved in all lowercase (for example, `REFERENCING OLD AS old`, `REFERENCING OLD AS OLD`, or `REFERENCING OLD AS "old"`). Also, `new` may only be replaced by an identifier named new or any equivalent that is saved in all lowercase (for example, `REFERENCING NEW AS new`, `REFERENCING NEW AS NEW`, or `REFERENCING NEW AS "new"`).

Either one, or both phrases `OLD AS old` and `NEW AS new` may be specified in the `REFERENCING` clause (for example, `REFERENCING NEW AS New OLD AS Old`).

This clause is not compatible with Oracle databases in that identifiers other than `old` or `new` may not be used.

`FOR EACH ROW`

Determines whether the trigger should be fired once for every row affected by the triggering event, or just 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, they must be declared after all other variable, type, and `REF CURSOR` declarations.

`statement`

An SPL program statement. Note that a `DECLARE - BEGIN - END` block is considered an SPL statement unto itself. Thus, the trigger body may contain nested blocks.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`exception`

An exception condition name such as `NO_DATA_FOUND`, `OTHERS`, etc.

Examples

The following is a statement-level trigger that fires after the triggering statement (insert, update, or delete on table `emp`) is executed.

```
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;
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
        TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
```

The following is a row-level trigger that fires before each row is either inserted, updated, or deleted on table `emp`.

```
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;
END;
```

See Also

DROP TRIGGER

15.37 CREATE TYPE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name

`CREATE TYPE` -- define a new user-defined type, which 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:

```
[ [ NOT ] { FINAL | INSTANTIABLE } ] ...
[ OVERRIDING ]
  <subprogram_spec>
```

and `subprogram_spec` is:

```
{ MEMBER | STATIC }
{ PROCEDURE <proc_name>
    [ ( [ SELF [ IN | IN OUT ] <name> ]
        [, <argname> [ IN | IN OUT | OUT ] <argtype>
                     [ DEFAULT <value> ]
        ] ...)
    ]
|
  FUNCTION <func_name>
    [ ( [ SELF [ IN | IN OUT ] <name> ]
        [, <argname> [ IN | IN OUT | OUT ] <argtype>
                     [ DEFAULT <value> ]
        ] ...)
    ]
  RETURN <rettype>
}
```

Nested Table Type

```
CREATE [ OR REPLACE ] TYPE <name> { IS | AS } TABLE OF
  { <datatype> | <objtype> | <collecttype> }
```

Varray Type

```
CREATE [ OR REPLACE ] TYPE <name> { IS | AS }
  { VARRAY | VARYING ARRAY } (<maxsize>) OF { <datatype> | <objtype> }
```

Composite Type

```
CREATE [ OR REPLACE ] TYPE <name> { IS | AS }
( [ attribute <datatype> ][, ...]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
)
```

Description

`CREATE TYPE` defines a new, user-defined data type. The types that can be created 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 are not compatible with Oracle databases. However, composite types can be accessed by SPL programs as with other types described in this section.

> **Note**
>
> - For packages only, a composite type can be included in a user-defined record type declared with the `TYPE IS RECORD` statement within the package specification or package body. Such nested structure is not permitted in other SPL programs such as functions, procedures, triggers, etc.
> - In the `CREATE TYPE` command, if a schema name is included, then the type is created in the specified schema, otherwise it is created in the current schema. The name of the new type must not match any existing type in the same schema unless the intent is to update the definition of an existing type, in which case use `CREATE OR REPLACE TYPE` .
> - The `OR REPLACE` option cannot be currently used to add, delete, or modify the attributes of an existing object type. Use the `DROP TYPE` command to first delete the existing object type. The `OR REPLACE` option can be used to add, delete, or modify the methods in an existing object type.
> - The PostgreSQL form of the `ALTER TYPE ALTER ATTRIBUTE` command can be used to change the data type of an attribute in an existing object type. However, the `ALTER TYPE` command cannot add or delete attributes in the object type.

The user that creates the type becomes the owner of the type.

Parameters

`name`

The name (optionally schema-qualified) of the type to create.

`DEFINER | CURRENT_USER`

Specifies whether the privileges of the object type owner (`DEFINER`) or the privileges of the current user executing a method in the object type (`CURRENT_USER`) are to be used to determine whether or not 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 is 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 is 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 is being created.

`FINAL`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`NOT FINAL`

For an object type, specifies whether or not a subtype can be derived from the object type. `FINAL` (subtype cannot be derived from the object type) is the default.

For `method_spec`, specifies whether or not the method may be overridden in a subtype. `NOT FINAL` (method may be overridden in a subtype) is the default.

`INSTANTIABLE`

`NOT INSTANTIABLE`

For an object type, specifies whether or not an object instance can be created of this object type. `INSTANTIABLE` (an instance of this object type can be created) is the default. If `NOT INSTANTIABLE` is specified, then `NOT FINAL` must be specified as well. If `method_spec` for any method in the object type contains the `NOT INSTANTIABLE` qualifier, then the object type, itself, must be defined with `NOT INSTANTIABLE` and `NOT FINAL` following the closing parenthesis of the object type specification.

For `method_spec`, specifies whether or not 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 `NOT INSTANTIABLE` is specified, then the `CREATE TYPE BODY` command for the object type must not contain the implementation of the method.

`OVERRIDING`

If `OVERRIDING` is specified, `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 is currently invoking the method. `SELF` can be explicitly declared 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 `SELF` is not explicitly declared, 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 within the method body.

`argtype`

The data type(s) of the method's arguments. The argument types may be a base data type or a user-defined type such as a nested table or an object type. A length must not be specified for any base type - for example, specify `VARCHAR2`, not `VARCHAR2(10)`.

`DEFAULT value`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Supplies a default value for an input argument if one is not supplied in the method call. `DEFAULT` may not be specified for arguments with modes `IN OUT` or `OUT`.

`func_name`

The name of the function to create.

`rettype`

The return data type, which may be any of the types listed for `argtype`. As for `argtype`, a length must not be specified for `rettype`.

`maxsize`

The maximum number of elements permitted in the varray.

Examples

Creating an Object Type

Create object type `addr_obj_typ`.

```
CREATE OR REPLACE TYPE addr_obj_typ AS OBJECT (
    street          VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip             NUMBER(5)
);
```

Create object type `emp_obj_typ` that includes a member method `display_emp`.

```
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)
);
```

Create object type `dept_obj_typ` that includes a static method `get_dname`.

```
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)`.

```
CREATE OR REPLACE TYPE budget_tbl_typ IS TABLE OF NUMBER(8,2);
```

Creating and Using a Composite Type

The following example shows the usage of a composite type accessed from an anonymous block.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The composite type is created by the following:

```
CREATE OR REPLACE TYPE emphist_typ AS (
    empno           NUMBER(4),
    ename           VARCHAR2(10),
    hiredate        DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2)
);
```

The following is the anonymous block that accesses the composite type:

```
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;

   EMPNO: 9001
   ENAME: SMITH
HIREDATE: 01-AUG-17 00:00:00
     JOB: SALESMAN
     SAL: 8000.00
```

The following example shows the usage of a composite type accessed from a user-defined record type, declared within a package body.

The composite type is created by the following:

```
CREATE OR REPLACE TYPE salhist_typ AS (
    startdate       DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2)
);
```

The package specification is defined by the following:

```
CREATE OR REPLACE PACKAGE emp_salhist
IS
    PROCEDURE fetch_emp (
        p_empno     IN NUMBER
    );
END;
```

The package body is defined by the following:

```
CREATE OR REPLACE PACKAGE BODY emp_salhist
IS
    TYPE emprec_typ IS RECORD (
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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;

    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;
```

Note that 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 as 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` that is defined with the `SALHIST_TYP` composite type.

Invocation of the package procedure that loads the array from a join of the `emp` and `jobhist` tables, then displays the array content is shown by the following:

```
 EXEC emp_salhist.fetch_emp(7788);

EMPNO  ENAME    STARTDATE  JOB         SAL
-----  -------  ---------  ---------   ---------
7788   SCOTT    19-APR-87  CLERK        1,000.00
7788   SCOTT    13-APR-88  CLERK        1,040.00
7788   SCOTT    05-MAY-90  ANALYST      3,000.00

EDB-SPL Procedure successfully completed
```

See Also

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

CREATE TYPE BODY, DROP TYPE

15.38 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
```

and `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` will either create a new object type body, or replace an existing body.

If a schema name is included, then the object type body is created in the specified schema. Otherwise it is 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 must not match any existing object type body in the same schema unless the intent is to update the definition of an existing object type body, in which case use `CREATE OR REPLACE TYPE BODY`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameters

`name`

The name (optionally schema-qualified) of the object type for which a body is to be created.

`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 is currently invoking the method. `SELF` can be explicitly declared 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 `SELF` is not explicitly declared, 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 within the method body.

`argtype`

The data type(s) of the method's arguments. The argument types may be a base data type or a user-defined type such as a nested table or an object type. A length must not be specified for any base type - for example, specify `VARCHAR2`, not `VARCHAR2(10)`.

`DEFAULT value`

Supplies a default value for an input argument if one is not supplied in the method call. `DEFAULT` may not be specified for arguments with modes `IN OUT` or `OUT`.

`program_body`

The pragma, declarations, and SPL statements that comprise the body of the function or procedure. The pragma may be `PRAGMA AUTONOMOUS_TRANSACTION` to set the function or procedure as an autonomous transaction.

`func_name`

The name of the function to create.

`rettype`

The return data type, which may be any of the types listed for `argtype`. As for `argtype`, a length must not be specified for `rettype`.

Examples

Create the object type body for object type `emp_obj_typ` given in the example for the `CREATE TYPE` command.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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);
        DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || addr.city || ', ' ||
            addr.state || ' ' || LPAD(addr.zip,5,'0'));
    END;
END;
```

Create the object type body for object type `dept_obj_typ` given in the example for the `CREATE TYPE` command.

```
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 := 'ACCOUNING';
            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;
```

See Also

CREATE TYPE, DROP TYPE

15.39 CREATE USER

Name

`CREATE USER` -- define a new database user account.

Synopsis

```
CREATE USER <name> [IDENTIFIED BY <password>]
```

Description

`CREATE USER` adds a new user to an Advanced Server database cluster. You must be a database superuser to use this command.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

When the `CREATE USER` command is given, a schema will also be created with the same name as the new user and owned by the new user. Objects with unqualified names created by this user will be created in this schema.

Parameters

`name`

The name of the user.

`password`

The user's password. The password can be changed later using `ALTER USER`.

Notes

The maximum length allowed for the user name and password is `63` characters.

Examples

Create a user named, `john`.

```
CREATE USER john IDENTIFIED BY abc;
```

See Also

DROP USER

15.40 CREATE USER|ROLE... PROFILE MANAGEMENT CLAUSES

Name

`CREATE USER|ROLE`

Synopsis

```
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 non-compatible clauses:

```
|  LOCK TIME  '<timestamp>'
```

For information about the administrative clauses of the `CREATE USER` or `CREATE ROLE` command that are supported by Advanced Server, see the PostgreSQL core documentation available at:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

https://www.postgresql.org/docs/11/static/sql-commands.html

Description

`CREATE ROLE|USER… PROFILE` adds a new role with an associated profile to an 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 pre-defined profile with a role, or to change which pre-defined profile is associated with a user.
- Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to specify that the user account should be placed 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, the role can only be unlocked by a database superuser 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 will expire. If you omit the `AT 'timestamp'` keywords, the password will expire immediately.

Each login role may only have one profile. To discover the profile that is 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 at which the clause will be enforced. When specifying a value for `timestamp` , enclose the value in single-quotes.

Examples

The following example uses `CREATE USER` to create a login role named `john` who is associated with the `acctg_profile` profile:

```
CREATE USER john PROFILE acctg_profile IDENTIFIED BY “1safepwd”;
```

`john` can log in to the server, using the password `1safepwd` .

The following example uses `CREATE ROLE` to create a login role named `john` who is associated with the `acctg_profile` profile:

```
CREATE ROLE john PROFILE acctg_profile LOGIN PASSWORD “1safepwd”;
```

`john` can log in to the server, using the password `1safepwd` .

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

See Also

ALTER USER|ROLE... PROFILE MANAGEMENT CLAUSES

15.41 CREATE VIEW

Name

`CREATE VIEW` -- define a new view.

Synopsis

```
CREATE [ OR REPLACE ] VIEW <name> [ ( <column_name> [, ...] ) ]
  AS <query>
```

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced.

If a schema name is given (for example, `CREATE VIEW myschema.myview ...`) then the view is created in the specified schema. Otherwise it is created in the current schema. The view name must be distinct 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 be created.

`column_name`

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

`query`

A query (that is, a `SELECT` statement) which will provide the columns and rows of the view.

Refer to `SELECT` for more information about valid queries.

Notes

Currently, views are read only - the system will not allow an insert, update, or delete on a view. You can get the effect of an updatable view by creating rules that rewrite inserts, etc. on the view into appropriate actions on other tables.

Access to tables referenced in the view is determined by permissions of the view owner. However, functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call all functions used by the view.

Examples

Create a view consisting of all employees in department `30`:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE VIEW dept_30 AS SELECT * FROM emp WHERE deptno = 30;
```

See Also

DROP VIEW

15.42 DELETE

Name

`DELETE` -- delete rows of a table.

Synopsis

```
DELETE [ <optimizer_hint> ] FROM <table>[@<dblink> ]
  [ WHERE <condition> ]
  [ RETURNING <return_expression> [, ...]
      { INTO { <record> | <variable> [, ...] }
      | BULK COLLECT INTO <collection> [, ...] } ]
```

Description

`DELETE` deletes rows that satisfy the `WHERE` clause from the specified table. If the `WHERE` clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

> **Note**
>
> The `TRUNCATE` command provides a faster mechanism to remove all rows from a table.

The `RETURNING INTO { record | variable [, ...] }` clause may only be specified if the `DELETE` command is used within an SPL program. In addition the result set of the `DELETE` command must not include 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.

The `RETURNING BULK COLLECT INTO collection [, ...]` clause may only be specified if the `DELETE` command is used within an SPL program. If more than one `collection` is specified as the target of the `BULK COLLECT INTO` clause, then each `collection` must consist of a single, scalar field – i.e., `collection` must not be a record. The result set of the `DELETE` command may contain none, one, 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` will be empty.

You must have the `DELETE` privilege on the table to delete from it, as well as the `SELECT` privilege for any table whose values are read in the condition.

Parameters

`optimizer_hint`

Comment-embedded hints to the optimizer for selection of an execution plan.

`table`

The name (optionally schema-qualified) of an existing table.

`dblink`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Database link name identifying a remote database. See the `CREATE DATABASE LINK` command for information on database links.

`condition`

A value expression that returns a value of type `BOOLEAN` that determines the rows which are to be deleted.

`return_expression`

An expression that may include one or more columns from `table`. If a column name from `table` is specified in the `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 the evaluated `return_expression` is to be assigned. The first `return_expression` is assigned to the first field in `record`, the second `return_expression` is assigned to the second field in `record`, etc. 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 the evaluated `return_expression` is to be assigned. 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`, etc. 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`. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be 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

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

15.43 DROP DATABASE LINK

Name

`DROP DATABASE LINK` -- remove a database link.

Synopsis

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 be removed.

`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

15.44 DROP DIRECTORY

Name

`DROP DIRECTORY` -- remove a directory alias for a file system directory path.

Synopsis

```
DROP DIRECTORY <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 a directory alias is deleted, the corresponding physical file system directory is not affected. The file system directory must be deleted using the appropriate operating system commands.

Parameters

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`name`

The name of a directory alias to be removed.

Examples

Remove the directory alias named `empdir`:

```
DROP DIRECTORY empdir;
```

See Also

CREATE DIRECTORY

15.45 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. All input (`IN` , `IN OUT`) argument data types to the function must be specified if this is an overloaded function. (This requirement is not compatible with Oracle databases. In Oracle, only the function name is specified. Advanced Server allows overloading of function names, so the function signature given by the input argument data types is required in the Advanced Server `DROP FUNCTION` command of an overloaded function.)

Usage of `IF EXISTS` , `CASCADE` , or `RESTRICT` is not compatible with Oracle databases and is used only by Advanced Server.

Parameters

`IF EXISTS`

Do not throw an error if the function does not exist. A notice is issued in this case.

`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` . Note that `DROP FUNCTION` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list only the `IN` and `IN OUT` arguments. (Specification of `argmode` is not compatible with Oracle databases and applies only to Advanced Server.)

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`argname`

The name of an argument. Note that `DROP FUNCTION` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity. (Specification of `argname` is not compatible with Oracle databases and applies only to Advanced Server.)

`argtype`

The data type of an argument of the function. (Specification of `argtype` is not compatible with Oracle databases and applies only to Advanced Server.)

`CASCADE`

Automatically drop objects that depend on the function (such as operators or triggers), and in turn all objects that depend on those objects.

`RESTRICT`

Refuse to drop the function if any objects depend on it. This is the default.

Examples

The following command removes the `emp_comp` function.

```
DROP FUNCTION emp_comp(NUMBER, NUMBER);
```

See Also

CREATE FUNCTION

15.46 DROP INDEX

Name

`DROP INDEX` -- remove an index.

Synopsis

```
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 will be given and the index will not be dropped.

Parameters

`name`

The name (optionally schema-qualified) of an index to remove.

Examples

This command will remove the index, `name_idx`:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DROP INDEX name_idx;
```

See Also

CREATE INDEX, ALTER INDEX

15.47 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. If `BODY` is specified, only the package body is removed – the package specification is not dropped. If `BODY` is omitted, both the package specification and body are removed.

Parameters

`name`

The name (optionally schema-qualified) of a package to remove.

Examples

This command will remove the `emp_admin` package:

```
DROP PACKAGE emp_admin;
```

See Also

CREATE PACKAGE, CREATE PACKAGE BODY

15.48 DROP PROCEDURE

Name

`DROP PROCEDURE` -- remove a procedure.

Synopsis

```
DROP PROCEDURE [ IF EXISTS ] <name>
  [ ([ [ <argmode> ] [ <argname> ] <argtype> ] [, ...]) ]
  [ CASCADE | RESTRICT ]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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. All input (`IN` , `IN OUT`) argument data types to the procedure must be specified if this is an overloaded procedure. (This requirement is not compatible with Oracle databases. In Oracle, only the procedure name is specified. Advanced Server allows overloading of procedure names, so the procedure signature given by the input argument data types is required in the Advanced Server `DROP PROCEDURE` command of an overloaded procedure.)

Usage of `IF EXISTS` , `CASCADE` , or `RESTRICT` is not compatible with Oracle databases and is used only by Advanced Server.

Parameters

`IF EXISTS`

Do not throw an error if the procedure does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing procedure.

`argmode`

The mode of an argument: `IN` , `IN OUT` , or `OUT` . If omitted, the default is `IN` . Note that `DROP PROCEDURE` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the procedure's identity. So it is sufficient to list only the `IN` and `IN OUT` arguments. (Specification of `argmode` is not compatible with Oracle databases and applies only to Advanced Server.)

`argname`

The name of an argument. Note that `DROP PROCEDURE` does not actually pay any attention to argument names, since only the argument data types are needed to determine the procedure's identity. (Specification of `argname` is not compatible with Oracle databases and applies only to Advanced Server.)

`argtype`

The data type of an argument of the procedure. (Specification of `argtype` is not compatible with Oracle databases and applies only to Advanced Server.)

`CASCADE`

Automatically drop objects that depend on the procedure, and in turn all objects that depend on those objects.

`RESTRICT`

Refuse to drop the procedure if any objects depend on it. This is the default.

Examples

The following command removes the `select_emp` procedure.

```
DROP PROCEDURE select_emp;
```

See Also

CREATE PROCEDURE, ALTER PROCEDURE

15.49 DROP PROFILE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name

`DROP PROFILE` -- drop a user-defined profile.

Synopsis

```
DROP PROFILE [IF EXISTS] <profile_name> [CASCADE | RESTRICT];
```

Description

Include the `IF EXISTS` clause to instruct the server to not throw an error if the specified profile does not exist. The server will issue a notice if the profile does not 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 to not drop any profile that is associated with a role. This is the default behavior.

Parameters

`profile_name`

The name of the profile being dropped.

Examples

The following example drops a profile named `acctg_profile`:

```
DROP PROFILE acctg_profile CASCADE;
```

The command first re-associates 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`:

```
DROP PROFILE acctg_profile RESTRICT;
```

The `RESTRICT` clause in the command instructs the server to not drop `acctg_profile` if there are any roles associated with the profile.

See Also

CREATE PROFILE, ALTER PROFILE

15.50 DROP QUEUE

Advanced Server includes extra syntax (not offered by Oracle) with the `DROP QUEUE SQL` command. This syntax can be used in association with `DBMS_AQADM`.

Name

`DROP QUEUE` -- drop an existing queue.

Synopsis

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Use `DROP QUEUE` to drop an existing queue:

```
DROP QUEUE [IF EXISTS] <name>
```

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 is being dropped.

`IF EXISTS`

Include the `IF EXISTS` clause to instruct the server to not return an error if the queue does not exist. The server will issue a notice.

Examples

The following example drops a queue named `work_order`:

```
DROP QUEUE work_order;
```

See Also

CREATE QUEUE, ALTER QUEUE

15.51 DROP QUEUE TABLE

Advanced Server includes extra syntax (not offered by Oracle) with the `DROP QUEUE TABLE SQL` command. This syntax can be used in association 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`name`

The name (possibly schema-qualified) of the queue table that will be deleted.

`IF EXISTS`

Include the `IF EXISTS` clause to instruct the server to not return an error if the queue table does not exist. The server will issue a notice.

`CASCADE`

Include the `CASCADE` keyword to automatically delete any objects that depend on the queue table.

`RESTRICT`

Include the `RESTRICT` keyword to instruct the server to refuse to delete the queue table if any objects depend on it. This is the default.

Examples

The following 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

15.52 DROP SYNONYM

Name

`DROP SYNONYM` -- remove a synonym.

Synopsis

```
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`

`syn_name` is the name of the synonym. A synonym name must be unique within a schema.

`schema`

`schema` specifies the name of the schema that the synonym resides in.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Like any other object that can be schema-qualified, you may have two synonyms with the same name in your search path. To disambiguate the name of the synonym that you are dropping, include a schema name. Unless a synonym is schema qualified in the `DROP SYNONYM` command, 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:

```
DROP PUBLIC SYNONYM syn_name;
```

The following example drops the synonym, `personnel` :

```
DROP SYNONYM personnel;
```

See Also

CREATE SYNONYM

15.53 DROP ROLE

Name

`DROP ROLE` -- remove a database role.

Synopsis

```
DROP ROLE <name> [ CASCADE ]
```

Description

`DROP ROLE` removes the specified role. To drop a superuser role, you must be a superuser yourself; to drop non-superuser roles, you must have `CREATEROLE` privilege.

A role cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted.

It is not necessary to remove role memberships involving the role; `DROP ROLE` automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

Alternatively, if the only objects owned by the role belong within a schema that is owned by the role and has the same name as the role, the `CASCADE` option can be specified. In this case the issuer of the `DROP ROLE name CASCADE` command must be a superuser and the named role, the schema, and all objects within the schema will be deleted.

Parameters

`name`

The name of the role to remove.

`CASCADE`

If specified, also drops the schema owned by, and with the same name as the role (and all objects owned by the role belonging to the schema) as long as no other dependencies on the role or the schema exist.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Examples

To drop a role:

```
DROP ROLE admins;
```

See Also

CREATE ROLE, SET ROLE, GRANT, REVOKE

15.54 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

To remove the sequence, `serial`:

```
DROP SEQUENCE serial;
```

See Also

ALTER SEQUENCE, CREATE SEQUENCE

15.55 DROP TABLE

Name

`DROP TABLE` -- remove a table.

Synopsis

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DROP TABLE <name> [CASCADE | RESTRICT | CASCADE CONSTRAINTS]
```

Description

`DROP TABLE` removes tables from the database. Only its owner may destroy a table. To empty a table of rows, without destroying the table, use `DELETE`. `DROP TABLE` always removes any indexes, rules, triggers, and constraints that exist for the target table.

Parameters

`name`

The name (optionally schema-qualified) of the table to drop.

- Include the `RESTRICT` keyword to specify that the server should refuse to drop the table if any objects depend on it. This is the default behavior; the `DROP TABLE` command will report an error if any objects depend on the table.
- Include the `CASCADE` clause to drop any objects that depend on the table.
- Include the `CASCADE CONSTRAINTS` clause to specify that Advanced Server should drop any dependent constraints (excluding other object types) on the specified table.

Examples

The following command drops a table named `emp` that has no dependencies:

```
DROP TABLE emp;
```

The outcome of a `DROP TABLE` command will vary depending on whether the table has any dependencies - you can control the outcome by specifying a *drop behavior*. For example, if you create two tables, `orders` and `items`, where the `items` table is dependent on the `orders` table:

```
CREATE TABLE orders
  (order_id int PRIMARY KEY, order_date date, …);
CREATE TABLE items
  (order_id REFERENCES orders, quantity int, …);
```

Advanced Server will perform 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`, Advanced Server will report an error.
- If you specify `DROP TABLE orders CASCADE`, Advanced Server will drop the `orders` table *and* the `items` table.
- If you specify `DROP TABLE orders CASCADE CONSTRAINTS`, Advanced Server will drop the `orders` table and remove the foreign key specification from the `items` table, but not drop the `items` table.

See Also

CREATE TABLE, ALTER TABLE

15.56 DROP TABLESPACE

Name

`DROP TABLESPACE` -- remove a tablespace.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Synopsis

```
DROP TABLESPACE <tablespacename>
```

Description

`DROP TABLESPACE` removes a tablespace from the system.

A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases may 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

To remove tablespace `employee_space` from the system:

```
DROP TABLESPACE employee_space;
```

See Also

ALTER TABLESPACE

15.57 DROP TRIGGER

Name

`DROP TRIGGER` -- remove a trigger.

Synopsis

```
DROP TRIGGER <name>
```

Description

`DROP TRIGGER` removes a trigger from its associated table. The command must be run by a superuser or the owner of the table on which the trigger is defined.

Parameters

`name`

The name of a trigger to remove.

Examples

Remove a trigger named `emp_salary_trig` :

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
DROP TRIGGER emp_salary_trig;
```

See Also

CREATE TRIGGER

15.58 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 nor to composite types. If `BODY` is specified, only the object type body is removed – the object type specification is not dropped. If `BODY` is omitted, both the object type specification and body are removed.

The type will not be deleted if there are other database objects dependent upon 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

15.59 DROP USER

Name

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`DROP USER` -- remove a database user account.

Synopsis

```
DROP USER <name> [ CASCADE ]
```

Description

`DROP USER` removes the specified user. To drop a superuser, you must be a superuser yourself; to drop non-superusers, you must have `CREATEROLE` privilege.

A user cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the user, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the user has been granted.

However, it is not necessary to remove role memberships involving the user; `DROP USER` automatically revokes any memberships of the target user in other roles, and of other roles in the target user. The other roles are not dropped nor otherwise affected.

Alternatively, if the only objects owned by the user belong within a schema that is owned by the user and has the same name as the user, the `CASCADE` option can be specified. In this case the issuer of the `DROP USER name CASCADE` command must be a superuser and the named user, the schema, and all objects within the schema will be deleted.

Parameters

`name`

The name of the user to remove.

`CASCADE`

If specified, also drops the schema owned by, and with the same name as the user (and all objects owned by the user belonging to the schema) as long as no other dependencies on the user or the schema exist.

Examples

To drop a user account named `john` that owns no objects nor has been granted any privileges on other objects:

```
DROP USER john;
```

To drop user account, `john`, who has not been granted any privileges on any objects, and does not own any objects outside of a schema named, `john`, that is owned by user, `john`:

```
DROP USER john CASCADE;
```

See Also

CREATE USER, ALTER USER

15.60 DROP VIEW

Name

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 will not be deleted if other objects are dependent upon this view (such as a view of a view).

The form of the `DROP VIEW` command compatible with Oracle does not support a `CASCADE` clause; to drop a view and it's dependencies, use the PostgreSQL-compatible form of the `DROP VIEW` command. For more information, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/11/static/sql-dropview.html

Parameters

`name`

The name (optionally schema-qualified) of the view to remove.

Examples

This command will remove the view named `dept_30`:

```
DROP VIEW dept_30;
```

See Also

CREATE VIEW

15.61 EXEC

Name

`EXEC`

Synopsis

```
EXEC function_name ['('[<argument_list>]')']
```

Description

`EXECUTE`

Parameters

`procedure_name`

`procedure_name` is the (optionally schema-qualified) function name.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`argument_list`

`argument_list` specifies a comma-separated list of arguments required by the function. Note that each member of `argument_list` corresponds to a formal argument expected by the function. Each formal argument may be an `IN` parameter, an `OUT` parameter, or an `INOUT` parameter.

Examples

The `EXEC` statement may 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);
```

15.62 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 progname
    ( [ [ argmode ] [ argname ] argtype ] [, ...] )
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON PROCEDURE progname
    [ ( [ [ argmode ] [ argname ] argtype ] [, ...] ) ]
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON PACKAGE packagename
  TO { username | groupname | PUBLIC } [, ...]
  [ WITH GRANT OPTION ]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 }
```

Description

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, and one that grants system privileges. These variants are similar in many ways, but they are different enough to be described separately.

In Advanced Server, the concept of users and groups has been 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 may be used to create a session and connect to an application. A *group* is a role that does not have the `LOGIN` attribute – the role may not be used to create a session or connect to an application.

A role may 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 may "belong" to users, groups may "belong" to groups, and groups may "belong" to users, forming a general multi-level hierarchy of roles. User names and group names share the same namespace therefore it is not necessary to distinguish 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 key word `PUBLIC` indicates that the privileges are to be granted to all roles, including those that may be created later. `PUBLIC` may be thought of as an implicitly defined group that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to `PUBLIC`.

If the `WITH GRANT OPTION` is specified, the recipient of the privilege may in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to `PUBLIC`.

There is no 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 could, however, choose to revoke some of his own privileges for safety.) The right to drop an object or to alter its definition in any way is not described by a grantable privilege; it is inherent in the owner, and cannot 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 may 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 may of course revoke these privileges. (For maximum security, issue the `REVOKE` in the same transaction that creates the object; then there is no window in which another user may use the object.)

The possible privileges are:

`SELECT`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 `INSERT` of a new row into the specified table.

`UPDATE`

Allows `UPDATE` of a column of the specified table. `SELECT ... FOR UPDATE` also requires this privilege (besides the `SELECT` privilege).

`DELETE`

Allows `DELETE` of a row from the specified table.

`REFERENCES`

To create a foreign key constraint, it is necessary to have 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 is applicable to functions, procedures, and packages.

The Advanced Server syntax for granting the `EXECUTE` privilege is not fully compatible with Oracle databases. Advanced Server requires qualification of the program name by one of the keywords, `FUNCTION`, `PROCEDURE`, or `PACKAGE` whereas these keywords must be omitted in Oracle. For functions, 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, all input argument data types must be specified if the procedure has one or more input arguments. In Oracle, function and procedure signatures must be omitted. This is due to the fact that all programs share the same namespace in Oracle, whereas functions, procedures, and packages have their own individual namespace in Advanced Server to allow program name overloading to a certain extent.

`ALL PRIVILEGES`

Grant all of the available privileges at once.

The privileges required by other commands are listed on the reference page of the respective command.

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 the `WITH ADMIN OPTION` is specified, the member may in turn grant membership in the role to others, and revoke membership in the role as well. Without the admin option, ordinary users cannot do that.

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 is not a superuser.

There are three pre-defined roles that have the following meanings:

`CONNECT`

Granting the `CONNECT` role is equivalent to giving the grantee the `LOGIN` privilege. The grantor must have the `CREATEROLE` privilege.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 the grant is given. 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

The `REVOKE` command is used to revoke access privileges.

When a non-owner of an object attempts to `GRANT` privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as a privilege is available, the command will proceed, but it will grant only those privileges for which the user has grant options. The `GRANT ALL PRIVILEGES` forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

It should be noted that 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`, it's unwise to operate as a superuser except when absolutely necessary.

If a superuser chooses to issue 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 via such a command will appear to have been granted by the object owner. (For role membership, the membership appears to have been granted by the containing role itself.)

`GRANT` and `REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the privileges will be recorded as having been granted by the role that actually 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`, but those privileges will appear to have been granted directly by `g1`. Any other member of role `g1` could revoke them later.

If the role executing `GRANT` holds the required privileges indirectly via more than one role membership path, it is unspecified which containing role will be recorded as having done the grant. In such cases it is best practice to use `SET ROLE` to become the specific role you want to do the `GRANT` as.

Currently, Advanced Server does not support granting or revoking privileges for individual columns of a table. One possible 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`:

```
GRANT INSERT ON emp TO PUBLIC;
```

Grant all available privileges to user `mary` on view `salesemp`:

```
GRANT ALL PRIVILEGES ON salesemp TO mary;
```

Note that while the above will indeed grant all privileges if executed by a superuser or the owner of `emp`, when executed by someone else it will only grant those permissions for which the someone else has grant options.

Grant membership in role `admins` to user `joe`:

```
GRANT admins TO joe;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 within a database. System privileges relate to the ability to create or delete certain database objects that are not necessarily within 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. System privileges are not required to drop a private database link. A private database link may be dropped by the link owner or a database superuser.

EXEMPT ACCESS POLICY

The `EXEMPT ACCESS POLICY` privilege allows a role to execute a SQL command without invoking any policy function that may be associated with the target database object. The role is exempt from all security policies in the database.

The `EXEMPT ACCESS POLICY` privilege is not inheritable by membership to a role that has the `EXEMPT ACCESS POLICY` privilege. For example, the following sequence of `GRANT` commands does not result in user `joe` obtaining the `EXEMPT ACCESS POLICY` privilege even though `joe` is granted membership to the `enterprisedb` role, which has been 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Using the ALTER ROLE Command to Assign System Privileges

The 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 functionally equivalent to the respective commands compatible with Oracle databases.

See Also

REVOKE, ALTER ROLE

15.63 INSERT

Name

`INSERT` -- create new rows in a table.

Synopsis

```
INSERT INTO <table>[@<dblink> ] [ ( <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 at a time or several rows as a result of a query.

The columns in the target list may be listed in any order. Each column not present in the target list will be inserted using a default value, either its declared default value or null.

If the expression for each column is not of the correct data type, automatic type conversion will be attempted.

The `RETURNING INTO { record | variable [, ...] }` clause may only be specified when the `INSERT` command is used within an SPL program and only when the `VALUES` clause is used.

The `RETURNING BULK COLLECT INTO collection [, ...]` clause may only be specified if the `INSERT` command is used within an SPL program. If more than one `collection` is specified as the target of the `BULK COLLECT INTO` clause, then each `collection` must consist of a single, scalar field – i.e., `collection` must not be a record. `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` will be empty.

You must have `INSERT` privilege to a table in order to insert into it. If you use the `query` clause to insert rows from a query, you also need to have `SELECT` privilege on any table used in the query.

Parameters

`table`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The name (optionally schema-qualified) of an existing table.

`dblink`

Database link name identifying a remote database. See the `CREATE DATABASE LINK` command 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 will be filled with its default value.

`query`

A query (`SELECT` statement) that supplies the rows to be inserted. Refer to the `SELECT` command for a description of the syntax.

`return_expression`

An expression that may include one or more columns from `table`. If a column name from `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 `INSERT` command, then the assigned value is used in the evaluation of `return_expression`.
- If the column specified in `return_expression` is not assigned a value in the `INSERT` command and there is 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` is not assigned a value in the `INSERT` command and there is 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 the evaluated `return_expression` is to be assigned. The first `return_expression` is assigned to the first field in `record`, the second `return_expression` is assigned to the second field in `record`, etc. 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 the evaluated `return_expression` is to be assigned. 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`, etc. 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`. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Examples

Insert a single row into table `emp`:

```
INSERT INTO emp VALUES (8021,'JOHN','SALESMAN',7698,'22-FEB-07',1250,500,30);
```

In this second example, the column, `comm`, is omitted and therefore it will have the default value of null:

```
INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, deptno)
    VALUES (8022,'PETERS','CLERK',7698,'03-DEC-06',950,30);
```

The third example uses the `DEFAULT` clause for the `hiredate` and `comm` columns rather than specifying a value:

```
INSERT INTO emp VALUES (8023,'FORD','ANALYST',7566,NULL,3000,NULL,20);
```

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:

```
CREATE TABLE deptnames (
    deptname        VARCHAR2(14)
);
INSERT INTO deptnames SELECT dname FROM dept;
```

15.64 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

`LOCK TABLE` obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If `NOWAIT` is specified, `LOCK TABLE` does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the remainder of the current transaction. (There is no `UNLOCK TABLE` command; locks are always released at transaction end.)

When acquiring locks automatically for commands that reference tables, 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 duration of the transaction. To achieve this you could obtain `SHARE` lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because `SHARE` lock mode conflicts with the `ROW EXCLUSIVE` lock acquired by writers, and your `LOCK TABLE` name `IN SHARE MODE` statement will wait 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; furthermore 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

executing any data modification statement. A serializable transaction's view of data will be frozen when its first data modification statement begins. A later `LOCK TABLE` will still prevent concurrent writes - but it won't ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use `SHARE ROW EXCLUSIVE` lock mode instead of `SHARE` mode.

This 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 actually perform their updates. (Note that 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, and if multiple lock modes are involved for a single object, then transactions should always 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`

The lock mode specifies which locks this lock conflicts with.

If no lock mode is specified, then the server uses the most restrictive mode, `ACCESS EXCLUSIVE`. (`ACCESS EXCLUSIVE` is not compatible with Oracle databases. In Advanced Server, this configuration mode ensures that no other transaction can access the locked table in any manner.)

`NOWAIT`

Specifies that `LOCK TABLE` should not wait for any conflicting locks to be released: if the specified lock cannot be immediately acquired without waiting, the transaction is aborted.

Notes

All forms of `LOCK` require `UPDATE` and/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 will be dropped as soon as it is obtained.

`LOCK TABLE` only deals with table-level locks, and so the mode names involving `ROW` are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, `ROW EXCLUSIVE` mode is a sharable table lock. Keep in mind that all the lock modes have identical semantics so far as `LOCK TABLE` is concerned, differing only in the rules about which modes conflict with which.

15.65 REVOKE

Name

`REVOKE` -- remove access privileges.

Synopsis

```
REVOKE { { SELECT | INSERT | UPDATE | DELETE | REFERENCES }
  [,...] | ALL [ PRIVILEGES ] }
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  ON tablename
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { SELECT | ALL [ PRIVILEGES ] }
  ON sequencename
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
  ON FUNCTION progname
    ( [ [ argmode ] [ argname ] argtype ] [, ...] )
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
  ON PROCEDURE progname
    [ ( [ [ argmode ] [ argname ] argtype ] [, ...] ) ]
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { EXECUTE | ALL [ PRIVILEGES ] }
  ON PACKAGE packagename
  FROM { username | groupname | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE role [, ...] FROM { username | groupname | PUBLIC }
  [, ...]
  [ CASCADE | RESTRICT ]

REVOKE { CONNECT | RESOURCE | DBA } [, ...]
  FROM { username | groupname } [, ...]

REVOKE CREATE [ PUBLIC ] DATABASE LINK
  FROM { username | groupname }

REVOKE DROP PUBLIC DATABASE LINK
  FROM { username | groupname }

REVOKE EXEMPT ACCESS POLICY
  FROM { username | groupname }
```

Description

The `REVOKE` command revokes previously granted privileges from one or more roles. The key word `PUBLIC` refers to the implicitly defined group of all roles.

See the description of the `GRANT` command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to `PUBLIC`. Thus, for example, revoking `SELECT` privilege from `PUBLIC` does not necessarily mean that all roles have lost `SELECT` privilege on the object: those who have it granted directly or via another role will still have it.

If the privilege had been granted with the grant option, the grant option for the privilege is revoked as well as the privilege, itself.

If a user holds a privilege with grant option and has 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 being revoked and dependent privileges exist, those dependent privileges are also revoked if `CASCADE` is specified, else the revoke action will fail. This recursive revocation only affects privileges that were granted through a chain of users that is traceable to the user that is the subject of this `REVOKE` command. Thus, the affected users may effectively keep the privilege if it was also granted through other users.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> **Note**
>
> `CASCADE` is not an option compatible with Oracle databases. By default Oracle always cascades dependent privileges, but Advanced Server requires the `CASCADE` keyword to be explicitly given, otherwise the `REVOKE` command will fail.

When revoking membership in a role, `GRANT OPTION` is instead called `ADMIN OPTION`, but the behavior is similar.

Notes

A user can only revoke privileges that were granted directly by that user. If, for example, user `A` has granted a privilege with grant option to user `B`, and user `B` has in turned granted it to user `C`, then user `A` cannot revoke the privilege directly from `C`. Instead, user `A` could revoke the grant option from user `B` and use the `CASCADE` option so that the privilege is in turn revoked from user `C`. For another example, if both `A` and `B` have granted the same privilege to `C`, `A` can revoke his own grant but not `B`'s grant, so `C` will still effectively have the privilege.

When a non-owner of an object attempts to `REVOKE` privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will revoke only those privileges for which the user has grant options. The `REVOKE ALL PRIVILEGES` forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

If a superuser chooses to issue 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 via chains of grant options), it is possible for a superuser to revoke all privileges, but this may require use of `CASCADE` as stated above.

`REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the command is performed as though it were issued by the containing role that actually 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 revoke privileges on `t1` that are recorded as being granted by `g1`. This would include grants made by `u1` as well as by other members of role `g1`.

If the role executing `REVOKE` holds privileges indirectly via more than one role membership path, it is unspecified which containing role will be used to perform the command. In such cases it is best practice to use `SET ROLE` to become the specific role you want to do the `REVOKE` as. Failure to do so may lead to revoking privileges other than the ones you intended, or not revoking anything at all.

> **Note**
>
> The 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`:

```
REVOKE ALL PRIVILEGES ON salesemp FROM mary;
```

Note that this actually means “revoke all privileges that I granted”.

Revoke membership in role `admins` from user `joe`:

```
REVOKE admins FROM joe;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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`:

```
REVOKE EXEMPT ACCESS POLICY FROM joe;
```

See Also

GRANT, SET ROLE

15.66 ROLLBACK

Name

`ROLLBACK` -- abort the current transaction.

Synopsis

```
ROLLBACK [ WORK ]
```

Description

`ROLLBACK` rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

`WORK`

Optional key word - 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 will throw an error if there is an Oracle-style SPL procedure on the runtime stack.

Examples

To abort all changes:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
ROLLBACK;
```

See Also

COMMIT, ROLLBACK TO SAVEPOINT, SAVEPOINT

15.67 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 established. The savepoint remains valid and can be rolled back to again, if needed.

`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 has not been established is an error.

`ROLLBACK TO SAVEPOINT` is not supported within SPL programs.

Examples

To undo the effects of the commands executed 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

15.68 SAVEPOINT

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name

`SAVEPOINT` -- define a new savepoint within the current transaction.

Synopsis

```
SAVEPOINT <savepoint_name>
```

Description

`SAVEPOINT` establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

`savepoint_name`

The name to be given to the savepoint.

Notes

Use `ROLLBACK TO SAVEPOINT` to roll back to a savepoint.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

When another savepoint is established with the same name as a previous savepoint, the old savepoint is kept, though only the more recent one will be used when rolling back.

`SAVEPOINT` is not supported within SPL programs.

Examples

To establish a savepoint and later undo the effects of all commands executed after it 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);
SAVEPOINT emps;
INSERT INTO jobhist VALUES (9001,'17-SEP-07',NULL,'CLERK',800,NULL,50,'New Hire');
INSERT INTO jobhist VALUES (9002,'20-SEP-07',NULL,'CLERK',700,NULL,50,'New Hire');
ROLLBACK TO depts;
COMMIT;
```

The above transaction will commit a row into the `dept` table, but the inserts into the `emp` and `jobhist` tables are rolled back.

See Also

COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT

15.69 SELECT

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Name

`SELECT` -- retrieve rows from a table or view.

Synopsis

```
SELECT [ optimizer_hint ] [ ALL | DISTINCT ]
  * | 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 more than one element is specified in the `FROM` list, they are cross-joined together. (See `FROM` clause, below.)
2. If the `WHERE` clause is specified, all rows that do not satisfy the condition are eliminated from the output. (See `WHERE` clause, below.)
3. If the `GROUP BY` clause is specified, the output is divided into groups of rows that match on one or more values. If the `HAVING` clause is present, it eliminates groups that do not satisfy the given condition. (See `GROUP BY` clause and `HAVING` clause below.)
4. Using the operators `UNION` , `INTERSECT` , and `MINUS` , the output of more than one `SELECT` statement can be combined 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 `ALL` is specified then duplicates are not eliminated. (See `UNION` clause, `INTERSECT` clause, and `MINUS` clause below.)
5. The actual output rows are computed using the `SELECT` output expressions for each selected row. (See `SELECT` list below.)
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 the `ORDER BY` clause is specified, the returned rows are sorted in the specified order. If `ORDER BY` is not given, the rows are returned in whatever order the system finds fastest to produce. (See `ORDER BY clause` below.)
8. `DISTINCT` eliminates duplicate rows from the result. `ALL` (the default) will return all candidate rows, including duplicates. (See `DISTINCT` clause below.)
9. The `FOR UPDATE` clause causes the `SELECT` statement to lock the selected rows against concurrent updates. (See `FOR UPDATE` clause below.)

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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Comment-embedded hints to the optimizer for selection of an execution plan. See *Database Compatibility for Oracle Developers Guide* for information about optimizer hints.

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. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an alias is provided, it completely hides the actual 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. Note that the sub-`SELECT` must be surrounded by parentheses, and an alias must be provided 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, a join condition must be specified, namely exactly one of `NATURAL`, `ON join_condition`, or `USING (join_column [, ...] )`. See below for the meaning. For `CROSS JOIN`, none of these clauses may appear.

A `JOIN` clause combines two `FROM` items. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, `JOINs` nest left-to-right. In any case `JOIN` binds more tightly than the commas separating `FROM` items.

`CROSS JOIN` and `INNER JOIN` produce a simple Cartesian product, the same result as you get from listing the two items at the top level of `FROM`, but restricted by the join condition (if any). `CROSS JOIN` is equivalent to `INNER JOIN ON (TRUE)`, that is, no rows are removed by qualification. These join types are just a notational convenience, since they do nothing you couldn't do with plain `FROM` and `WHERE`.

`LEFT OUTER JOIN` returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its join condition), plus one copy of each row in

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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. Note that 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 just a notational convenience, since you could convert it to a `LEFT OUTER JOIN` by switching the left and right inputs.

`FULL OUTER JOIN` returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

`ON join_condition`

`join_condition` is an expression resulting in a value of type `BOOLEAN` (similar to a `WHERE` clause) that specifies which rows in a join are considered to match.

`USING (join_column [, ...] )`

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 will be included in the join output, not both.

`NATURAL`

`NATURAL` is shorthand for a `USING` list that mentions all columns in the two tables that have the same names.

If multiple sources are specified, 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

The following example selects all of the entries from the `dept` table:

```
 SELECT * FROM dept;
deptno |  dname      |  loc
-------+-------------+-----------
    10 |  ACCOUNTING |  NEW YORK
    20 |  RESEARCH   |  DALLAS
    30 |  SALES      |  CHICAGO
    40 | OPERATIONS  |  BOSTON
 (4 rows)
```

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 does not satisfy this condition will be 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

The following example joins the contents of the `emp` and `dept` tables, `WHERE` the value of the `deptno` column in the `emp` table is equal to the value of the `deptno` column in the `deptno` table:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

 deptno |   dname    | empno | ename  | mgr  |      hiredate
--------+------------+-------+--------+------+---------------------
     10 | ACCOUNTING |  7934 | MILLER | 7782 | 23-JAN-82 00:00:00
     10 | ACCOUNTING |  7782 | CLARK  | 7839 | 09-JUN-81 00:00:00
     10 | ACCOUNTING |  7839 | KING   |      | 17-NOV-81 00:00:00
     20 | RESEARCH   |  7788 | SCOTT  | 7566 | 19-APR-87 00:00:00
     20 | RESEARCH   |  7566 | JONES  | 7839 | 02-APR-81 00:00:00
     20 | RESEARCH   |  7369 | SMITH  | 7902 | 17-DEC-80 00:00:00
     20 | RESEARCH   |  7876 | ADAMS  | 7788 | 23-MAY-87 00:00:00
     20 | RESEARCH   |  7902 | FORD   | 7566 | 03-DEC-81 00:00:00
     30 | SALES      |  7521 | WARD   | 7698 | 22-FEB-81 00:00:00
     30 | SALES      |  7844 | TURNER | 7698 | 08-SEP-81 00:00:00
     30 | SALES      |  7499 | ALLEN  | 7698 | 20-FEB-81 00:00:00
     30 | SALES      |  7698 | BLAKE  | 7839 | 01-MAY-81 00:00:00
     30 | SALES      |  7654 | MARTIN | 7698 | 28-SEP-81 00:00:00
     30 | SALES      |  7900 | JAMES  | 7698 | 03-DEC-81 00:00:00
(14 rows)
```

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` will condense into a single row all selected rows that share the same values for the grouped expressions. `expression` can be an input column name, or the name or ordinal number of an output column (`SELECT` list item), or an arbitrary expression formed from input-column values. In case of ambiguity, a `GROUP BY` name will be 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 (whereas without `GROUP BY` , an aggregate produces a single value computed across all the selected rows). When `GROUP BY` is present, it is not valid for the `SELECT` list expressions to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

Example

The following example computes the sum of the `sal` column in the `emp` table, grouping the results by department number:

```
SELECT deptno, SUM(sal) AS total
    FROM emp
    GROUP BY deptno;

 deptno |  total
--------+----------
     10 |  8750.00
     20 | 10875.00
     30 |  9400.00
(3 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 do not satisfy the specified condition. `HAVING` is different from `WHERE; WHERE` filters individual rows before the application of `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 within an aggregate function.

Example

To sum the column, `sal` of all employees, group the results by department number and show those group totals that are less than 10000:

```
SELECT deptno, SUM(sal) AS total
    FROM emp
    GROUP BY deptno
    HAVING SUM(sal) < 10000;

 deptno |  total
--------+---------
     10 | 8750.00
     30 | 9400.00
(2 rows)
```

SELECT List

The `SELECT` list (between the key words `SELECT` and `FROM`) specifies expressions that form the output rows of the `SELECT` statement. The expressions can (and usually do) refer to columns computed in the `FROM` clause. Using the clause `AS output_name` , another name can be specified for an output column. This name is primarily used to label the column for display. It can also be used to refer to the column's value in `ORDER BY` and `GROUP BY` clauses, but not in the `WHERE` or `HAVING` clauses; there you must write out the expression instead.

Instead of an expression, * can be written in the output list as a shorthand for all the columns of the selected rows.

Example

The `SELECT` list in the following example specifies that the result set should include the `empno` column, the `ename` column, the `mgr` column and the `hiredate` column:

```
SELECT empno, ename, mgr, hiredate FROM emp;

 empno | ename  | mgr  |      hiredate
-------+--------+------+---------------------
  7934 | MILLER | 7782 | 23-JAN-82 00:00:00
  7782 | CLARK  | 7839 | 09-JUN-81 00:00:00
  7839 | KING   |      | 17-NOV-81 00:00:00
  7788 | SCOTT  | 7566 | 19-APR-87 00:00:00
  7566 | JONES  | 7839 | 02-APR-81 00:00:00
  7369 | SMITH  | 7902 | 17-DEC-80 00:00:00
  7876 | ADAMS  | 7788 | 23-MAY-87 00:00:00
  7902 | FORD   | 7566 | 03-DEC-81 00:00:00
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  7521 | WARD   | 7698 | 22-FEB-81 00:00:00
  7844 | TURNER | 7698 | 08-SEP-81 00:00:00
  7499 | ALLEN  | 7698 | 20-FEB-81 00:00:00
  7698 | BLAKE  | 7839 | 01-MAY-81 00:00:00
  7654 | MARTIN | 7698 | 28-SEP-81 00:00:00
  7900 | JAMES  | 7698 | 03-DEC-81 00:00:00
(14 rows)
```

UNION Clause

The `UNION` clause has the form:

```
select_statement UNION [ ALL ] select_statement
```

`select_statement` is any `SELECT` statement without an `ORDER BY` or `FOR UPDATE` clause. (`ORDER BY` can be attached to a sub-expression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to 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 be of compatible data types.

The result of `UNION` does not contain any duplicate rows unless the `ALL` option is specified. `ALL` prevents elimination of duplicates.

Multiple `UNION` operators in the same `SELECT` statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, `FOR UPDATE` may not be specified either for a `UNION` result or for any input of a `UNION` .

INTERSECT Clause

The `INTERSECT` clause has the form:

```
select_statement INTERSECT select_statement
```

`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` does not contain any duplicate rows.

Multiple `INTERSECT` operators in the same `SELECT` statement are evaluated left to right, unless parentheses dictate otherwise. `INTERSECT` binds more tightly than `UNION` . That is, `A UNION B INTERSECT C` will be read as `A UNION (B INTERSECT C)` .

MINUS Clause

The `MINUS` clause has this general form:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
select_statement MINUS select_statement
```

`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` does not contain any duplicate rows.

Multiple `MINUS` operators in the same `SELECT` statement are evaluated left to right, unless parentheses dictate otherwise. `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 }
```

`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.

The following optional clauses may be specified in conjunction 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**
>
> Advanced Server does not 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 ] [, ...]
```

`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 causes the result rows to be sorted according to the specified expressions. If two rows are equal according to the leftmost 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to a result column using the `AS` clause.

It is also possible to use arbitrary expressions in the `ORDER BY` clause, including columns that do not appear in the `SELECT` result list. Thus the following statement is valid:

```
SELECT ename FROM emp ORDER BY empno;
```

A limitation of this feature is that an `ORDER BY` clause applying to the result of a `UNION`, `INTERSECT`, or `MINUS` clause may only specify 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` will interpret it as the result column name. This is the opposite of the choice that `GROUP BY` will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one may add the key word `ASC` (ascending) or `DESC` (descending) after any expression in the `ORDER BY` clause. If not specified, `ASC` is assumed by default.

The null value sorts higher than any other value. In other words, with ascending sort order, null values sort at the end, and 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 `SELECT DISTINCT` is specified 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

The following two examples are identical ways of sorting the individual results according to the contents of the second column (`dname`):

```
SELECT * FROM dept ORDER BY dname;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     40 | OPERATIONS | BOSTON
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
(4 rows)

SELECT * FROM dept ORDER BY 2;

 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     40 | OPERATIONS | BOSTON
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
(4 rows)
```

The following 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),
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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;
 job      | deptno
----------+--------
 CLERK    |     20
 MANAGER  |     20
 SALESMAN |     30
(3 rows)
```

DISTINCT Clause

If a `SELECT` statement specifies `DISTINCT`, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). The `ALL` keyword specifies the opposite: all rows are kept; that is the default.

FOR UPDATE Clause

The `FOR UPDATE` clause takes the form:

```
 FOR UPDATE [WAIT n|NOWAIT|SKIP LOCKED]
```

`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 will be 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` will wait for the first transaction to complete, and will then lock and return the updated row (or no row, if the row was deleted).

`FOR UPDATE` cannot be used in contexts where returned rows cannot be clearly identified 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) that the `SELECT` statement will wait for a row locked by another session. Use a decimal form to specify fractional seconds; for example, `WAIT 1.5` instructs the server to wait one and a half seconds. Specify up to `4` digits to the right of the decimal.
- Include the `NOWAIT` keyword to report an error immediately if a row cannot be locked by the current session.
- Include `SKIP LOCKED` to instruct the server to lock rows if possible, and skip rows that are already locked by another session.

15.70 SET CONSTRAINTS

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 within the current transaction. `IMMEDIATE` constraints are checked at the end of each statement. `DEFERRED` constraints are not checked until transaction commit. Each constraint has its own `IMMEDIATE` or `DEFERRED` mode.

Upon creation, a constraint is given one of three characteristics: `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE`, or `NOT DEFERRABLE`. The third class is always `IMMEDIATE` and is not affected by the `SET CONSTRAINTS` command. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by `SET CONSTRAINTS`.

`SET CONSTRAINTS` with a list of constraint names changes the mode of just those constraints (which must all be deferrable). If there are multiple constraints matching 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 would have been checked at the end of the transaction are instead checked during the execution of the `SET CONSTRAINTS` command. If any such constraint is violated, the `SET CONSTRAINTS` fails (and does not change the constraint mode). Thus, `SET CONSTRAINTS` can be used to force checking of constraints to occur at a specific point in a transaction.

Currently, only foreign key constraints are affected by this setting. Check and unique constraints are always effectively not deferrable.

Notes

This command only alters the behavior of constraints within the current transaction. Thus, if you execute this command outside of a transaction block it will not appear to have any effect.

15.71 SET ROLE

Name

`SET ROLE` -- set the current user identifier of the current session.

Synopsis

```
SET ROLE { rolename | NONE }
```

Description

This command sets the current user identifier of the current SQL session context to be `rolename`. After `SET ROLE`, permissions checking for SQL commands is carried out as though the named role is the one that had logged in originally.

The specified `rolename` must be a role that the current session user is a member of. If the session user is a superuser, any role can be selected.

`NONE` resets the current user identifier to be the current session user identifier. These forms may be executed by any user.

Notes

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

You can use this command, to either add privileges or restrict one's privileges. If the session user role has the `INHERITS` attribute, then it automatically has all the privileges of every role that it could `SET ROLE` to; in this case `SET ROLE` effectively drops all the privileges assigned directly to the session user and to the other roles it is a member of, leaving 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 ROLE` to a non-superuser role, she loses her superuser privileges.

Examples

User `mary` takes on the identity of role `admins`:

```
SET ROLE admins;
```

User `mary` reverts back to her own identity:

```
SET ROLE NONE;
```

15.72 SET TRANSACTION

Name

`SET TRANSACTION` -- set the characteristics of the current transaction.

Synopsis

```
SET TRANSACTION transaction_mode
```

where `transaction_mode` is one of:

```
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 subsequent 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 what data the transaction can see when other transactions are running concurrently:

`READ COMMITTED`

A statement can only see rows committed before it began. This is the default.

`SERIALIZABLE`

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

The transaction isolation level cannot be changed after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, or `FETCH`) of a transaction has been 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 are disallowed: `INSERT`, `UPDATE`, and `DELETE` if the table they would write to is not a

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `COMMENT`, `GRANT`, `REVOKE`, `TRUNCATE`; and `EXECUTE` if the command it would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

15.73 TRUNCATE

Name

`TRUNCATE` -- empty a table.

Synopsis

```
TRUNCATE TABLE <name> [DROP STORAGE]
```

Description

`TRUNCATE` quickly removes all rows from a table. It has the same effect as an unqualified `DELETE` but since it does not actually scan the table, it is faster. This 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 be truncated.

Notes

`TRUNCATE` cannot be used if there are foreign-key references to the table from other tables. Checking validity in such cases would require table scans, and the whole point is not to do one.

`TRUNCATE` will not run any user-defined `ON DELETE` triggers that might exist for the table.

Examples

The following command truncates a table named `accounts`:

```
TRUNCATE TABLE accounts;
```

See Also

DROP VIEW, DELETE

15.74 UPDATE

Name

`UPDATE` -- update rows of a table.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Synopsis

```
UPDATE [ <optimizer_hint> ] <table>[@<dblink> ]
    SET <column> = { <expression> | DEFAULT } [, ...]
  [ WHERE <condition> ]
  [ RETURNING <return_expression> [, ...]
      { INTO { <record> | <variable> [, ...] }
      | BULK COLLECT INTO <collection> [, ...] } ]
```

Description

`UPDATE` changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

The `RETURNING INTO { record | variable [, ...] }` clause may only be specified within 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.

The `RETURNING BULK COLLECT INTO collection [, ...]` clause may only be specified if the `UPDATE` command is used within an SPL program. If more than one `collection` is specified as the target of the `BULK COLLECT INTO` clause, then each `collection` must consist of a single, scalar field – i.e., `collection` must not be a record. The result set of the `UPDATE` command may contain none, one, 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` will be empty.

You must have the `UPDATE` privilege on the table to update it, as well as the `SELECT` privilege to any table whose values are read in `expression` or `condition`.

Parameters

`optimizer_hint`

Comment-embedded hints to the optimizer for selection of an execution plan.

`table`

The name (optionally schema-qualified) of the table to update.

`dblink`

Database link name identifying a remote database. See the `CREATE DATABASE LINK` command for information on database links.

`column`

The name of a column in table.

`expression`

An expression to assign to the column. The expression may use the old values of this and other columns in the table.

`DEFAULT`

Set the column to its default value (which will be null if no specific default expression has been assigned to it).

`condition`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

An expression that returns a value of type `BOOLEAN`. Only rows for which this expression returns true will be updated.

`return_expression`

An expression that may include one or more columns from table. If a column name from 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 in the evaluation of `return_expression`.
- If the column specified in `return_expression` is not assigned a value in the `UPDATE` command, then the column's current value in the affected row is used in the evaluation of `return_expression`.

`record`

A record whose field the evaluated `return_expression` is to be assigned. The first `return_expression` is assigned to the first field in `record`, the second `return_expression` is assigned to the second field in `record`, etc. 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 the evaluated `return_expression` is to be assigned. 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`, etc. 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`. There can be either a single collection which may be a collection of a single field or a collection of a record type, or there may be 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 sal = sal * 1.1, comm = comm + 500 WHERE job = 'SALESMAN';
```

16 Database Compatibility Table Partitioning Guide

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 fits in memory.
- When a query or update accesses a large percentage of a single partition, performance may improve because the server will perform a sequential scan of the partition instead of using an index and random access reads scattered across the whole table.
- A bulk load (or unload) can be implemented by adding or removing partitions, if you plan that requirement into the partitioning design. `ALTER`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`TABLE` is far faster than a bulk operation. It also entirely avoids the `VACUUM` overhead caused by a bulk `DELETE`.
- Seldom-used data can be migrated 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 will benefit from partitioning depends on the application; a good rule of thumb is that the size of the table should exceed the physical memory of the database server.

This document discusses the aspects of table partitioning compatible with Oracle databases that are supported by Advanced Server.

> **Note**
>
> This document and particularly the partitioning presented in this chapter do not describe the *declarative partitioning* feature, which has been introduced with PostgreSQL version 10. Note that PostgreSQL declarative partitioning is supported in Advanced Server 10 in addition to the table partitioning compatible with Oracle databases as described in this chapter. For information about declarative partitioning, see the PostgreSQL core documentation available at: https://www.postgresql.org/docs/11/static/ddl-partitioning.html

The PostgreSQL 9.6 `INSERT... ON CONFLICT DO NOTHING/UPDATE` clause (commonly known as `UPSERT`) is not supported on Oracle-styled partitioned tables. If you include the `ON CONFLICT DO NOTHING/UPDATE` clause when invoking the INSERT command to add data to a partitioned table, the server will return an error.

16.1 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 is stored in each partition. As new rows are added to the partitioned table, the server uses the partitioning rules to decide which partition should contain each row.

Advanced Server can also use the partitioning rules to enforce partition pruning, improving performance when responding to user queries. When selecting a partitioning type and partitioning keys for a table, you should take into consideration how the data that is stored within a table will be queried, and include often-queried columns in the partitioning rules.

> **Note**
>
> Advanced Server does not support the creation of partitioned tables when using `default_with_rowids` set to `ON`.

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 within the row. If the column value matches a value in the partitioning rule, the row is stored in the partition named in the rule.

Range Partitioning

When you create a range-partitioned table, you specify one or more partitioning key columns. When you add a new row to the table, the server compares the value of the partitioning key (or keys) to the corresponding column (or 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.

Hash Partitioning

When you create a hash-partitioned table, you specify one or more partitioning key columns. Data is divided into (approx.) equal-sized partitions amongst 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 column (or columns), and stores the row in a partition according to the hash value.

> **Note**
>
> When upgrading Advanced Server, you must rebuild each hash-partitioned table on the upgraded version server.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Subpartitioning

Subpartitioning breaks a partitioned table into smaller subsets. All subsets must be stored in the same database server cluster. A table is typically subpartitioned by a different set of columns, and may be a different subpartitioning type than the parent partition. If one partition is subpartitioned, then each partition will have at least one subpartition.

If a table is subpartitioned, no data will be stored in any of the partition tables; the data will be stored instead in the corresponding subpartitions.

16.2 Using Partition Pruning

Advanced Server's query planner uses *partition pruning* to compute an efficient plan to locate a row (or rows) that matches 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 is compared to the partitioning key column specified when creating the partitioned table. When querying a:

- list-partitioned table, partition pruning is effective when the `WHERE` clause compares a literal value to the partitioning key using operators like equal (=) or `AND.`
- range-partitioned table, partition pruning is effective when the `WHERE` clause compares a literal value to a partitioning key using operators such as equal (=), less than (<), or greater than (>).
- hash-partitioned table, partition pruning is effective when the `WHERE` clause compares a literal value to the partitioning key using an operator such as equal (=).

The partition pruning mechanism uses two optimization techniques:

- Fast Pruning
- Constraint exclusion

Partition pruning techniques limit the search for data to only those partitions in which the values for which you are searching might reside. Both pruning techniques remove partitions from a query's execution plan, increasing 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, while constraint exclusion does not. For example, when a query searches for a specific value within a list-partitioned table, fast pruning can reason that only a specific partition may hold that value, while 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, while constraint exclusion occurs late in the planning process.

Using Constraint Exclusion

The `constraint_exclusion` parameter controls constraint exclusion. The `constraint_exclusion` parameter may have a value of `on, off,` or `partition`. To enable constraint exclusion, the parameter must be set to *either* `partition` or `on`. By default, the parameter is set to `partition.`

For more information about constraint exclusion, see:

https://www.postgresql.org/docs/11/static/ddl-partitioning.html

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 *does not* contain a `WHERE` clause, the query planner must recommend an execution plan that searches the entire table. When you execute a `SELECT` statement that *does* contain a `WHERE` clause, the query planner determines in which partition that row would be stored, and sends query fragments to that partition, pruning the partitions that could not contain that row from the execution plan. If you are not using partitioned tables, disabling constraint exclusion may improve performance.

Fast Pruning

Like constraint exclusion, fast pruning can only optimize queries that include a `WHERE` (or join) clause, and only when the qualifiers in the `WHERE` clause match a certain form. In both cases, the query planner will avoid searching for data within partitions that cannot possibly hold the data required by the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

query.

Fast pruning is controlled by a boolean configuration parameter named `edb_enable_pruning.` If `edb_enable_pruning` is `ON`, Advanced Server will fast prune certain queries. If `edb_enable_pruning` is `OFF`, the server will disable fast pruning.

> **Note**
>
> Fast pruning cannot optimize queries against subpartitioned tables or optimize queries against range-partitioned tables that are partitioned on more than one column.

For LIST-partitioned tables, 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:

```
 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;
```

Given the first `WHERE` clause, fast pruning would eliminate partitions `europe, asia,` and `others` because those partitions cannot hold rows that satisfy the qualifier: `WHERE country = 'US'.`

Given the second `WHERE` clause, fast pruning would eliminate partitions `americas, europe`, and `asia` because those partitions cannot hold rows where `country IS NULL.`

The operator specified in the `WHERE` clause must be an equal sign (=) or the equality operator appropriate for the data type of the partitioning column.

For range-partitioned tables, Advanced Server can fast prune queries that contain a `WHERE` clause that constrains a partitioning column to a literal value, but the operator may be any of the following:

```
>
```

```
>=
```

```
=
```

```
<=
```

```
<
```

Fast pruning will also reason about more complex expressions involving `AND` and `BETWEEN` operators, such as:

```
 WHERE size > 100 AND size <= 200
WHERE size BETWEEN 100 AND 200
```

Fast pruning cannot prune based on expressions involving `OR` or `IN.` For example, when querying a RANGE-partitioned table, such as:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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)
)
```

Fast pruning can reason about `WHERE` clauses such as:

```
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'
```

In each case, fast pruning requires that the qualifier must refer to a partitioning column and literal value (or `IS NULL/IS NOT NULL).`

> **Note**
>
> Fast pruning can also optimize `DELETE` and `UPDATE` statements containing `WHERE` clauses of the forms described above.

16.2.1 Example - Partition Pruning

The `EXPLAIN` statement displays the execution plan of a statement. You can use the `EXPLAIN` statement to confirm that Advanced Server is pruning partitions from the execution plan of a query.

To demonstrate the efficiency of partition pruning, first create a simple 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 asia VALUES('INDIA', 'PAKISTAN'),
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  PARTITION americas VALUES('US', 'CANADA')
);
```

Then, perform a constrained query that includes the `EXPLAIN` statement:

```
EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'INDIA';
```

The resulting query plan shows that the server will scan only the `sales_asia` table - the table in which a row with a `country` value of `INDIA` would be 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)
```

If you perform a query that searches for a row that matches a value not included in the partitioning key:

```
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'),
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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 cannot possibly contain the desired result set:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'US' AND date = 'Dec 12, 2012';
                                                 QUERY PLAN
------------------------------------------------------------------------------------
-----------------------------------
 Append
   ->  Seq Scan on sales_americas_2012
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12
00:00:00'::timestamp without time zone))
(3 rows)
```

16.3 Partitioning Commands Compatible with Oracle Databases

The following sections provide information about using the table partitioning syntax compatible with Oracle databases supported by Advanced Server.

16.3.1 CREATE TABLE...PARTITION BY

Use the `PARTITION BY` clause of the `CREATE TABLE` command to create a partitioned table with data distributed amongst one or more partitions (and subpartitions). The command syntax comes in the following forms:

List Partitioning Syntax

Use the first form to create a list-partitioned table:

```
CREATE TABLE [ schema. ]<table_name>
   <table_definition>
   PARTITION BY LIST(<column>)
   [SUBPARTITION BY {RANGE|LIST|HASH} (<column>[, <column> ]...)]
   (<list_partition_definition>[, <list_partition_definition>]...)
   [ENABLE ROW MOVEMENT];

Where list_partition_definition is:

      PARTITION [<partition_name>]
        VALUES (<value>[, <value>]...)
        [TABLESPACE <tablespace_name>]
        [(<subpartition>, ...)]
```

Range Partitioning Syntax

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Use the second form to create a range-partitioned table:

```
CREATE TABLE [ schema. ]<table_name>
   <table_definition>
   PARTITION BY RANGE(<column>[, <column> ]...)
   [SUBPARTITION BY {RANGE|LIST|HASH} (<column>[, <column> ]...)]
   (<range_partition_definition>[, <range_partition_definition>]...)
   [ENABLE ROW MOVEMENT];

Where range_partition_definition is:

      PARTITION [<partition_name>]
        VALUES LESS THAN (<value>[, <value>]...)
        [TABLESPACE <tablespace_name>]
        [(<subpartition>, ...)]
```

Hash Partitioning Syntax

Use the third 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> ]...)]
   (<hash_partition_definition>[, <hash_partition_definition>]...)
   [ENABLE ROW MOVEMENT];

Where hash_partition_definition is:

      PARTITION [<partition_name>]
        [TABLESPACE <tablespace_name>]
        [(<subpartition>, ...)]
```

Subpartitioning Syntax

`subpartition` may 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>]

where hash_subpartition is:

      SUBPARTITION [<subpartition_name>]
        [TABLESPACE <tablespace_name>]
```

Description

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `CREATE TABLE… PARTITION BY` command creates a table with one or more partitions; each partition may have one or more subpartitions. There is no upper limit to the number of defined partitions, but if you include the `PARTITION BY` clause, you must specify at least one partitioning rule. The resulting table will be owned by the user that creates it.

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, but there is no upper limit placed on the number of values you may specify. Include a rule that specifies a matching value of `DEFAULT` to direct any un-qualified rows to the given partition; for more information about using the `DEFAULT` keyword, see Handling Stray Values in a LIST or RANGE Partitioned Table.

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 (i.e., 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, 2013 will include only those date values that fall on or before December 31, 2012.

Range partition rules must be specified in ascending order. `INSERT` commands that store rows with values that exceed the top boundary of a range-partitioned table will fail unless the partitioning rules include a boundary rule that specifies a value of `MAXVALUE`. If you do not include a `MAXVALUE` partitioning rule, any row that exceeds the maximum limit specified by the boundary rules will result in an error.

For more information about using the `MAXVALUE` keyword, see Handling Stray Values in a LIST or RANGE Partitioned Table.

Use the `PARTITION BY HASH` clause to create a hash-partitioned table. In a `HASH` partitioned table, data is divided amongst 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 is as close to unique as possible to help ensure that data is evenly distributed amongst the partitions. When selecting a partitioning column (or combination of columns), select a column (or columns) that you frequently search for exact matches for best performance.

> **Note**
>
> If you are upgrading Advanced Server, you must rebuild the hash-partitioned table on the upgraded version server.

Use the `TABLESPACE` keyword to specify the name of a tablespace on which a partition or subpartition will reside; if you do not specify a tablespace, the partition or subpartition will reside in the default tablespace.

If a table definition includes the `SUBPARTITION BY` clause, each partition within that table will have at least one subpartition. Each subpartition may be explicitly defined or system-defined.

If the subpartition is system-defined, the server-generated subpartition will reside in the default tablespace, and the name of the subpartition will be assigned by the server. The server will create:

- A `DEFAULT` subpartition if the `SUBPARTITION BY` clause specifies `LIST`.
- A `MAXVALUE` subpartition if the `SUBPARTITION BY` clause specifies `RANGE`.

The server will generate a subpartition name that is 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.

Use the `ENABLE ROW MOVEMENT` with a list, range, or hash partition tables to support the migration of tables with similar syntax from oracle databases, however it doesnot enable the actual row movement. Also, this syntax is not supported with `ALTER TABLE` command.

Parameters

`table_name`

The name (optionally schema-qualified) of the table to be created.

`table_definition`

The column names, data types, and constraint information as described in the PostgreSQL core documentation for the `CREATE TABLE` statement, available at:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

https://www.postgresql.org/docs/11/static/sql-createtable.html

`partition_name`

The name of the partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`subpartition_name`

The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`column`

The name of a column on which the partitioning rules are based. Each row will be stored in a partition that corresponds to the `value` of the specified column(s).

`(value[, value]...)`

Use `value` to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. `value` may 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 un-matched rows to the given partition. If you do not include a rule that includes a value of `DEFAULT`, any `INSERT` statement that attempts to add a row that does not match the specified rules of at least one partition will fail, and return an error.

When specifying rules for a list-partitioned table, include the `MAXVALUE` keyword in the last partition rule to direct any un-categorized rows to the given partition. If you do not include a `MAXVALUE` partition, any `INSERT` statement that attempts to add a row where the partitioning key is greater than the highest value specified will fail, and return an error.

`tablespace_name`

The name of the tablespace in which the partition or subpartition resides.

16.3.1.1 Example - PARTITION BY LIST

The following example creates a partitioned table `(sales)` using the `PARTITION BY LIST` clause. The `sales` table stores information in three partitions (`europe, asia`, and `americas`).

```
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')
);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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         | FOR VALUES IN ('FRANCE', 'ITALY')
 ASIA           | FOR VALUES IN ('INDIA', 'PAKISTAN')
 AMERICAS       | FOR VALUES IN ('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 would evaluate the following statement against the partitioning rules, and store the row in the `europe` partition.

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');
```

16.3.1.2 Example - PARTITION BY RANGE

The following 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)`.

```
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.

```
edb=# SELECT partition_name, high_value from ALL_TAB_PARTITIONS;
 partition_name | high_value
----------------+-------------------------------------------------------------
 Q1_2012        | FOR VALUES FROM (MINVALUE) TO ('01-APR-12 00:00:00')
 Q2_2012        | FOR VALUES FROM ('01-APR-12 00:00:00') TO ('01-JUL-12 00:00:00')
 Q3_2012        | FOR VALUES FROM ('01-JUL-12 00:00:00') TO ('01-OCT-12 00:00:00')
 Q4_2012        | FOR VALUES FROM ('01-OCT-12 00:00:00') TO ('01-JAN-13 00:00:00')
(4 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Any row with a value in the `date` column before April 1, 2012 is stored in a partition named `q1_2012`.
- Any row with a value in the `date` column before July 1, 2012 is stored in a partition named `q2_2012`.
- Any row with a value in the `date` column before October 1, 2012 is stored in a partition named `q3_2012`.
- Any row with a value in the `date` column before January 1, 2013 is stored in a partition named `q4_2012`.

The server would evaluate the following statement against the partitioning rules and store the row in the `q3_2012` partition.

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');
```

16.3.1.3 Example - PARTITION BY HASH

The following example creates a partitioned table `(sales)` using the `PARTITION BY HASH` clause. The `sales` table stores information in three partitions (`p1, p2` , and `p3`).

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY HASH (part_no)
(
  PARTITION p1,
  PARTITION p2,
  PARTITION p3
);
```

The table is partitioned by the hash value of the value specified in the `part_no` column.

```
edb=# SELECT partition_name, high_value from ALL_TAB_PARTITIONS;
 partition_name |                high_value
----------------+-------------------------------------------
 P1             | FOR VALUES WITH (modulus 3, remainder 0)
 P2             | FOR VALUES WITH (modulus 3, remainder 1)
 P3             | FOR VALUES WITH (modulus 3, remainder 2)
(3 rows)
```

The server will evaluate the hash value of the `part_no` column and distribute the rows into approximately equal partitions.

16.3.1.4 Example - PARTITION BY RANGE, SUBPARTITION BY LIST

The following example creates a partitioned table `(sales)` that is 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.

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
  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')
      )
);
```

This statement creates a table with four partitions; each partition has three subpartitions.

```
edb=# SELECT subpartition_name, high_value, partition_name FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name  |             high_value              | partition_name
--------------------+-------------------------------------+----------------
 Q1_AMERICAS        | FOR VALUES IN ('US', 'CANADA')      | Q1_2012
 Q1_ASIA            | FOR VALUES IN ('INDIA', 'PAKISTAN') | Q1_2012
 Q1_EUROPE          | FOR VALUES IN ('FRANCE', 'ITALY')   | Q1_2012
 Q2_AMERICAS        | FOR VALUES IN ('US', 'CANADA')      | Q2_2012
 Q2_ASIA            | FOR VALUES IN ('INDIA', 'PAKISTAN') | Q2_2012
 Q2_EUROPE          | FOR VALUES IN ('FRANCE', 'ITALY')   | Q2_2012
 Q3_AMERICAS        | FOR VALUES IN ('US', 'CANADA')      | Q3_2012
 Q3_ASIA            | FOR VALUES IN ('INDIA', 'PAKISTAN') | Q3_2012
 Q3_EUROPE          | FOR VALUES IN ('FRANCE', 'ITALY')   | Q3_2012
 Q4_AMERICAS        | FOR VALUES IN ('US', 'CANADA')      | Q4_2012
 Q4_ASIA            | FOR VALUES IN ('INDIA', 'PAKISTAN') | Q4_2012
 Q4_EUROPE          | FOR VALUES IN ('FRANCE', 'ITALY')   | Q4_2012
(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, and the server selects the partition in which the row should reside. 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 will be stored in a subpartition, so the partitions will contain no data.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The server would evaluate the following statement against the partitioning and subpartitioning rules and store the row in the `q3_europe` partition.

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012', '650000');
```

16.3.2 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:

        {<list_subpartition> | <range_subpartition> | <hash_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 PARTITION` command adds a partition to an existing partitioned table. There is 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 partition(s).

You can use the `ALTER TABLE… ADD PARTITION` statement to add a partition to a table with a `DEFAULT` rule as long as there are no conflicting values between existing rows in the table and the values of the partition to be added.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

You cannot 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.

`RANGE` partitions must be specified in ascending order. You cannot 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 will reside. If you do not specify a tablespace, the partition will reside in the default tablespace.

If the table is indexed, the index will be 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

`table_name`

The name (optionally schema-qualified) of the partitioned table.

`partition_name`

The name of the partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`subpartition_name`

The name of the subpartition to be created. Subpartition names must be unique amongst 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 rows will be distributed into partitions. Each partitioning rule must specify at least one `value`, but there is no limit placed on the number of values specified within a rule. `value` may 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 a partition or subpartition resides.

16.3.2.1 Example - Adding a Partition to a LIST Partitioned Table

The example that follows adds a partition to the list-partitioned `sales` table. The table was created using the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The table contains three partitions (`americas, asia` , and `europe`).

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |             high_value
----------------+-------------------------------------
 EUROPE         | FOR VALUES IN ('FRANCE', 'ITALY')
 ASIA           | FOR VALUES IN ('INDIA', 'PAKISTAN')
 AMERICAS       | FOR VALUES IN ('US', 'CANADA')
(3 rows)
```

The following command adds a partition named `east_asia` to the `sales` table.

```
ALTER TABLE sales ADD PARTITION east_asia
  VALUES ('CHINA', 'KOREA');
```

After invoking the command, the table includes the `east_asia` partition.

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |             high_value
----------------+-------------------------------------
 EUROPE         | FOR VALUES IN ('FRANCE', 'ITALY')
 ASIA           | FOR VALUES IN ('INDIA', 'PAKISTAN')
 AMERICAS       | FOR VALUES IN ('US', 'CANADA')
 EAST_ASIA      | FOR VALUES IN ('CHINA', 'KOREA')
(4 rows)
```

16.3.2.2 Example - Adding a Partition to a RANGE Partitioned Table

The example that follows adds a partition to a range-partitioned table named `sales` :

```
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'),
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The table contains four partitions (`q1_2012, q2_2012, q3_2012` , and `q4_2012`).

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name | high_value
----------------+-----------------------------------------------------------------------
 Q1_2012        | FOR VALUES FROM (MINVALUE) TO ('01-APR-12 00:00:00')
 Q2_2012        | FOR VALUES FROM ('01-APR-12 00:00:00') TO ('01-JUL-12 00:00:00')
 Q3_2012        | FOR VALUES FROM ('01-JUL-12 00:00:00') TO ('01-OCT-12 00:00:00')
 Q4_2012        | FOR VALUES FROM ('01-OCT-12 00:00:00') TO ('01-JAN-13 00:00:00')
(4 rows)
```

The following 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 invoking the command, the table includes the `q1_2013` partition.

```
edb=#  SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name | high_value
----------------+-----------------------------------------------------------------------
 Q1_2012        | FOR VALUES FROM (MINVALUE) TO ('01-APR-12 00:00:00')
 Q2_2012        | FOR VALUES FROM ('01-APR-12 00:00:00') TO ('01-JUL-12 00:00:00')
 Q3_2012        | FOR VALUES FROM ('01-JUL-12 00:00:00') TO ('01-OCT-12 00:00:00')
 Q4_2012        | FOR VALUES FROM ('01-OCT-12 00:00:00') TO ('01-JAN-13 00:00:00')
 Q1_2013        | FOR VALUES FROM ('01-JAN-13 00:00:00') TO ('01-APR-13 00:00:00')
(5 rows)
```

16.3.3 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>]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Description

The `ALTER TABLE… ADD SUBPARTITION` command adds a subpartition to an existing partition; the partition must already be subpartitioned. There is 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 subpartition(s).

You can use the `ALTER TABLE… ADD SUBPARTITION` statement to add a subpartition to a table with a `DEFAULT` rule as long as there are no conflicting values between existing rows in the table and the values of the subpartition to be added.

You cannot 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 cannot add a new subpartition that precedes existing subpartitions in a range subpartitioned table; range subpartitions must be specified in ascending order.

Include the `TABLESPACE` clause to specify the tablespace in which the subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be 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 will reside.

`partition_name`

The name of the partition in which the new subpartition will reside.

`subpartition_name`

The name of the subpartition to be created. Subpartition names must be unique amongst 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 will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. `value` may 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.

16.3.3.1 Example - Adding a Subpartition to a LIST/RANGE Partitioned Table

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example adds a `RANGE` subpartition to the list-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 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.

```
edb=#  SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name | high_value
----------------+-------------------+-----------------------------------------------------------------
 EUROPE         | EUROPE_2011       | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 EUROPE         | EUROPE_2012       | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JAN-13 00:00:00')
 ASIA           | ASIA_2011         | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 ASIA           | ASIA_2012         | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01- JAN-13 00:00:00')
 AMERICAS       | AMERICAS_2011     | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 AMERICAS       | AMERICAS_2012     | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01- JAN-13 00:00:00')
(6 rows)
```

The following command adds a subpartition named `europe_2013`.

```
ALTER TABLE sales MODIFY PARTITION europe
  ADD SUBPARTITION europe_2013
  VALUES LESS THAN('2015-Jan-01');
```

After invoking the command, the table includes a subpartition named `europe_2013`.

```
edb=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 partition_name | subpartition_name | high_value
----------------+-------------------+-------------------------------------------------------------
 EUROPE         |  EUROPE_2011      | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 EUROPE         |  EUROPE_2012      | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JAN-13 00:00:00')
 EUROPE         |  EUROPE_2013      | FOR VALUES FROM ('01-JAN-13 00:00:00') TO ('01-JAN-15 00:00:00')
 ASIA           |  ASIA_2011        | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 ASIA           |  ASIA_2012        | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JAN-13 00:00:00')
 AMERICAS       |  AMERICAS_2011    | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 AMERICAS       |  AMERICAS_2012    | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JAN-13 00:00:00')
(7 rows)
```

Note

When adding a new range subpartition, the subpartitioning rules must specify a range that falls *after* any existing subpartitions.

16.3.3.2 Example - Adding a Subpartition to a RANGE/LIST Partitioned Table

The following 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')
    (
      SUBPARTITION asia VALUES ('INDIA', 'PAKISTAN')
    )
);
```

After executing the above command, the `sales` table will have two partitions, named `first_half_2012` and `second_half_2012`. The `first_half_2012 partition` has two subpartitions, named `europe` and `americas`, and the `second_half_2012` partition has one partition, named `asia`.

```
edb=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |             high_value
------------------+-------------------+------------------------------------
 FIRST_HALF_2012  | AMERICAS          | FOR VALUES IN ('US', 'CANADA')
 FIRST_HALF_2012  | EUROPE            | FOR VALUES IN ('ITALY', 'FRANCE')
 SECOND_HALF_2012 | ASIA              | FOR VALUES IN ('INDIA', 'PAKISTAN')
(3 rows)
```

The following command adds a subpartition to the `second_half_2012` partition, named `east_asia`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
ALTER TABLE sales MODIFY PARTITION second_half_2012
  ADD SUBPARTITION east_asia VALUES ('CHINA');
```

After invoking the command, the table includes a subpartition named `east_asia`.

```
edb=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |             high_value
------------------+-------------------+------------------------------------
 FIRST_HALF_2012  | AMERICAS          | FOR VALUES IN ('US', 'CANADA')
 FIRST_HALF_2012  | EUROPE            | FOR VALUES IN ('ITALY', 'FRANCE')
 SECOND_HALF_2012 | EAST_ASIA         | FOR VALUES IN ('CHINA')
 SECOND_HALF_2012 | ASIA              | FOR VALUES IN ('INDIA', 'PAKISTAN')
(4 rows)
```

16.3.4 ALTER TABLE...SPLIT PARTITION

Use the `ALTER TABLE… SPLIT PARTITION` command to divide a single partition into two partitions and redistribute 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>]
    PARTITION <new_part2>
     [TABLESPACE <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>]
    PARTITION <new_part2>
     [TABLESPACE <tablespace_name>]
  );
```

Description

The `ALTER TABLE...SPLIT PARTITION` command adds a partition to an existing `LIST` or `RANGE` partitioned table. Please note that the `ALTER TABLE… SPLIT PARTITION` command cannot add a partition to a `HASH` partitioned table. There is no upper limit to the number of partitions that a table may have.

When you execute an `ALTER TABLE...SPLIT PARTITION` command, Advanced Server creates two new partitions and 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 will reside. If you do not specify a tablespace, the partition will reside in the default tablespace.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

If the table is indexed, the index will be 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 is being split.

`new_part1`

The name of the first new partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_part1` will receive the rows that meet the partitioning constraints specified in the `ALTER TABLE… SPLIT PARTITION` command.

`new_part2`

The name of the second new partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_part2` will receive the rows are not 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 (i.e., 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 will include only those date values that fall on or before December 31, 2009.

`(value[, value]...)`

Use `value` to specify a quoted literal value (or comma-delimited list of literal values) by which rows will be distributed into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule.

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.

16.3.4.1 Example - Splitting a LIST Partition

This example will divide one of the partitions in the list-partitioned `sales` table into two new partitions, and redistribute the contents of the partition between them. The `sales` table is created with the statement:

```
CREATE TABLE sales
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(
  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`). The following 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'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

The rows are distributed amongst the partitions.

```
edb=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date         | amount
----------------+---------+---------+----------+---------------------+--------
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(17 rows)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following command splits the `americas` partition into two partitions named `us` and `canada`.

```
ALTER TABLE sales SPLIT PARTITION americas
  VALUES ('US')
  INTO (PARTITION us, PARTITION canada);
```

A `SELECT` statement confirms that the rows have been redistributed.

```
edb=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date         | amount
---------------+---------+---------+----------+---------------------+--------
 sales_canada  |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00  |  50000
 sales_canada  |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00  |  75000
 sales_canada  |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00  | 120000
 sales_europe  |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00  |  45000
 sales_europe  |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00  |  15000
 sales_europe  |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00  | 650000
 sales_europe  |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00  | 650000
 sales_asia    |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00  |  75000
 sales_asia    |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00  |  37500
 sales_asia    |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00  |   5090
 sales_asia    |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00  | 650000
 sales_asia    |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00  |   5090
 sales_us      |      40 | 9519b   | US       | 12-APR-12 00:00:00  | 145000
 sales_us      |      40 | 4577b   | US       | 11-NOV-12 00:00:00  |  25000
 sales_us      |      40 | 3788a   | US       | 12-MAY-12 00:00:00  |   4950
 sales_us      |      40 | 4788a   | US       | 23-SEP-12 00:00:00  |   4950
 sales_us      |      40 | 4788b   | US       | 09-OCT-12 00:00:00  |  15000
(17 rows)
```

16.3.4.2 Example - Splitting a RANGE Partition

This example divides the `q4_2012` partition (of the range-partitioned `sales` table) into two partitions, and redistribute the partition's contents. Use the following 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')
);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The table definition creates four partitions (`q1_2012, q2_2012, q3_2012` , and `q4_2012`). The following 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'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

A `SELECT` statement confirms that the rows are distributed amongst the partitions as expected.

```
edb=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date         | amount
---------------+---------+---------+----------+---------------------+--------
 sales_q1_2012 |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_q1_2012 |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_q1_2012 |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_q2_2012 |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_q2_2012 |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_q2_2012 |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_q2_2012 |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_q3_2012 |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_q3_2012 |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_q3_2012 |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_q4_2012 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_q4_2012 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_q4_2012 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |   15000
 sales_q4_2012 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_q4_2012 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(17 rows)
```

The following 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 have been redistributed across the new partitions.

```
edb=# SELECT tableoid::regclass, * FROM sales;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
   tableoid      | dept_no | part_no | country  |        date         |amount
-----------------+---------+---------+----------+---------------------+------
 sales_q1_2012   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 | 45000
 sales_q1_2012   |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 | 75000
 sales_q1_2012   |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 | 75000
 sales_q2_2012   |      40 | 9519b   | US       | 12-APR-12 00:00:00 |145000
 sales_q2_2012   |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 | 37500
 sales_q2_2012   |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |120000
 sales_q2_2012   |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |  4950
 sales_q3_2012   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 | 15000
 sales_q3_2012   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |650000
 sales_q3_2012   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |650000
 sales_q3_2012   |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |  5090
 sales_q3_2012   |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |  4950
 sales_q4_2012_p1|      40 | 4577b   | US       | 11-NOV-12 00:00:00 | 25000
 sales_q4_2012_p1|      40 | 4788b   | US       | 09-OCT-12 00:00:00 | 15000
 sales_q4_2012_p1|      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |650000
 sales_q4_2012_p2|      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 | 50000
 sales_q4_2012_p2|      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |  5090
(17 rows)
```

16.3.5 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 is no upper limit to the number of defined subpartitions. When you execute an `ALTER TABLE...SPLIT SUBPARTITION` command, Advanced Server creates two new subpartitions, moving any rows that contain values that are constrained by the specified subpartition rules into `new_subpart1`, and any remaining rows into

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`new_subpart2`.

The new subpartition rules must reference the column specified in the rules that define the existing subpartition(s).

Include the `TABLESPACE` clause to specify a tablespace in which a new subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be 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 is being split.

`new_subpart1`

The name of the first new subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_subpart1` will receive the rows that meet the subpartitioning constraints specified in the `ALTER TABLE… SPLIT SUBPARTITION` command.

`new_subpart2`

The name of the second new subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_subpart2` will receive the rows are not directed to `new_subpart1` by the subpartitioning constraints specified in the `ALTER TABLE… SPLIT SUBPARTITION` command.

`(value[, value]...)`

Use `value` to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. `value` may 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.

16.3.5.1 Example - Splitting a LIST Subpartition

The following example splits a list subpartition, redistributing the subpartition's contents between two new subpartitions. The sample table `(sales)` was created with the command:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 two 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.

```
edb=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |            high_value
------------------+-------------------+----------------------------------
 FIRST_HALF_2012  | P1_AMERICAS       | FOR VALUES IN ('US', 'CANADA')
 FIRST_HALF_2012  | P1_EUROPE         | FOR VALUES IN ('ITALY', 'FRANCE')
 SECOND_HALF_2012 | P2_AMERICAS       | FOR VALUES IN ('US', 'CANADA')
 SECOND_HALF_2012 | P2_EUROPE         | FOR VALUES IN ('ITALY', 'FRANCE')
(4 rows)
```

The following command adds rows to each subpartition.

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000');
```

A `SELECT` statement confirms that the rows are correctly distributed amongst the subpartitions.

```
edb=# SELECT tableoid::regclass, * 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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_p1_europe  |      10 | 4519b   | FRANCE  | 17-JAN-12 00:00:00 |  45000
sales_p2_americas|      40 | 4577b   | US      | 11-NOV-12 00:00:00 |  25000
sales_p2_americas|      30 | 7588b   | CANADA  | 14-DEC-12 00:00:00 |  50000
sales_p2_americas|      40 | 4788a   | US      | 23-SEP-12 00:00:00 |   4950
sales_p2_americas|      40 | 4788b   | US      | 09-OCT-12 00:00:00 |  15000
sales_p2_europe  |      10 | 9519b   | ITALY   | 07-JUL-12 00:00:00 |  15000
sales_p2_europe  |      10 | 9519a   | FRANCE  | 18-AUG-12 00:00:00 | 650000
sales_p2_europe  |      10 | 9519b   | FRANCE  | 18-AUG-12 00:00:00 | 650000
(12 rows)
```

The following 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 invoking the command, the `p2_americas` subpartition has been deleted; in its place, the server has created two new subpartitions `(p2_us` and `p2_canada)`.

```
edb=# SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |            high_value
------------------+------------------+----------------------------------
 FIRST_HALF_2012  | P1_AMERICAS      | FOR VALUES IN ('US', 'CANADA')
 FIRST_HALF_2012  | P1_EUROPE        | FOR VALUES IN ('ITALY', 'FRANCE')
 SECOND_HALF_2012 | P2_CANADA        | FOR VALUES IN ('CANADA')
 SECOND_HALF_2012 | P2_US            | FOR VALUES IN ('US')
 SECOND_HALF_2012 | P2_EUROPE        | FOR VALUES IN ('ITALY', 'FRANCE')
(5 rows)
```

Querying the `sales` table demonstrates that the content of the `p2_americas` subpartition has been redistributed.

```
edb=# SELECT tableoid::regclass, * 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_p1_europe  |      10 | 4519b   | FRANCE  | 17-JAN-12 00:00:00 | 45000
 sales_p2_canada  |      30 | 7588b   | CANADA  | 14-DEC-12 00:00:00 | 50000
 sales_p2_europ   |      10 | 9519b   | ITALY   | 07-JUL-12 00:00:00 | 15000
 sales_p2_europe  |      10 | 9519a   | FRANCE  | 18-AUG-12 00:00:00 |650000
 sales_p2_europe  |      10 | 9519b   | FRANCE  | 18-AUG-12 00:00:00 |650000
 sales_p2_us      |      40 | 4577b   | US      | 11-NOV-12 00:00:00 | 25000
 sales_p2_us      |      40 | 4788a   | US      | 23-SEP-12 00:00:00 |  4950
 sales_p2_us      |      40 | 4788b   | US      | 09-OCT-12 00:00:00 | 15000
(12 rows)
```

16.3.5.2 Example - Splitting a RANGE Subpartition

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example splits a range subpartition, redistributing the subpartition's contents between two new subpartitions. The sample table `(sales)` 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 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 (`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.

```
edb=#  SELECT partition_name, subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name | high_value
----------------+-------------------+-----------------------------------------------------------------------
 EUROPE         | EUROPE_2012       | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JAN-13 00:00:00')
 EUROPE         | EUROPE_2011       | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 ASIA           | ASIA_2012         | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JAN-13 00:00:00')
 ASIA           | ASIA_2011         | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 AMERICAS       | AMERICAS_2012     | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JAN-13 00:00:00')
 AMERICAS       | AMERICAS_2011     | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
(6 rows)
```

The following command adds rows to each subpartition.

```
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'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

A `SELECT` statement confirms that the rows are distributed amongst 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   | 3788b | 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)
```

The following 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 invoking the command, the `americas_2012` subpartition has been deleted; in its place, the server has created two new 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
----------------+-------------------+-----------------------------------------------------------------------
 EUROPE         | EUROPE_2011       | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 EUROPE         | EUROPE_2012       | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JAN-13 00:00:00')
 ASIA           | ASIA_2011         | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 ASIA           | ASIA_2012         | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JAN-13 00:00:00')
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 AMERICAS       | AMERICAS_2011     | FOR VALUES FROM (MINVALUE) TO ('01-JAN-12 00:00:00')
 AMERICAS       | AMERICAS_P1_2012  | FOR VALUES FROM ('01-JAN-12 00:00:00') TO ('01-JUN-12 00:00:00')
 AMERICAS       | AMERICAS_P2_2012  | FOR VALUES FROM ('01-JUN-12 00:00:00') TO ('01-JAN-13 00:00:00')
(7 rows)
```

Querying the `sales` table demonstrates that the subpartition's contents are redistributed.

```
edb=# SELECT tableoid::regclass, * FROM sales;
 tableoid              |dept_no|part_no|country  |      date         | amount
-----------------------+-------+-------+---------+-------------------+-------
 sales_americas_p1_2012|  40   | 9519b | US      | 12-APR-12 00:00:00|145000
 sales_americas_p1_2012|  30   | 9519b | CANADA  | 01-FEB-12 00:00:00|75000
 sales_americas_p1_2012|  30   | 4519b | CANADA  | 08-APR-12 00:00:00|120000
 sales_americas_p1_2012|  40   | 3788a | US      | 12-MAY-12 00:00:00| 4950
 sales_americas_p2_2012|  40   | 4577b | US      | 11-NOV-12 00:00:00|25000
 sales_americas_p2_2012|  30   | 7588b | CANADA  | 14-DEC-12 00:00:00|50000
 sales_americas_p2_2012|  40   | 4788a | US      | 23-SEP-12 00:00:00| 4950
 sales_americas_p2_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   | 3788b | 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)
```

16.3.6 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

When the `ALTER TABLE… EXCHANGE PARTITION` command completes, the data originally located in the `target_partition` will be located in the `source_table`, and the data originally located in the `source_table` will be 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` (both tables must have matching columns and data types), and the data contained within 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, but 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 previously used *matching index* term refers to indexes that have the same attributes such as 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 both `INCLUDING INDEXES` and `EXCLUDING INDEXES` are omitted, then the default action is the `EXCLUDING INDEXES` behavior.

The same behavior as previously 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 be replaced.

`target_subpartition`

The name of the subpartition to be replaced.

`source_table`

The name of the table that will replace the `target_partition` or `target_subpartition`.

16.3.6.1 Example - Exchanging a Table for a Partition

The example that follows demonstrates swapping a table for a partition `(americas)` of the `sales` table. You can create the `sales` table with the following command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Use the following command to add sample data to the `sales` table.

```
INSERT INTO sales VALUES
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that only one row resides in the `americas` partition.

```
edb=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date         | amount
----------------+---------+---------+----------+---------------------+--------
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(10 rows)
```

The following command creates a table `(n_america)` that matches the definition of the `sales` table.

```
CREATE TABLE n_america
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
);
```

The following command adds data to the `n_america` table. The data conforms to the partitioning rules of the `americas` partition.

```
INSERT INTO n_america VALUES
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000');
```

The following command swaps the table into the partitioned table.

```
ALTER TABLE sales
  EXCHANGE PARTITION americas
  WITH TABLE n_america;
```

Querying the `sales` table shows that the contents of the `n_america` table has been exchanged for the content of the `americas` partition.

```
edb=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date         | amount
----------------+---------+---------+----------+---------------------+--------
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(17 rows)
```

Querying the `n_america` table shows that the row that was previously stored in the `americas` partition has been moved to the `n_america` table.

```
edb=# SELECT tableoid::regclass, * FROM n_america;
 tableoid  | dept_no | part_no | country |        date         | amount
-----------+---------+---------+---------+---------------------+--------
 n_america |      40 | 9519b   | US      | 12-APR-12 00:00:00 | 145000
(1 row)
```

16.3.7 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>;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 as previously described 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 be moved.

`subpartition_name`

The name of the subpartition to be moved.

`tablespace_name`

The name of the tablespace to which the partition or subpartition will be moved.

16.3.7.1 Example - Moving a Partition to a Different Tablespace

The following example moves a partition of the `sales` table from one tablespace to another. First, 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
);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Querying the `ALL_TAB_PARTITIONS` view confirms that the partitions reside on the expected servers and tablespaces.

```
edb=# SELECT partition_name, tablespace_name FROM ALL_TAB_PARTITIONS;
 partition_name | tablespace_name
----------------+-----------------
 Q1_2012        |
 Q2_2012        |
 Q3_2012        |
 Q4_2012        | TS_1
 Q1_2013        | TS_2
(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.

```
edb=# SELECT partition_name, tablespace_name FROM ALL_TAB_PARTITIONS;
 partition_name | tablespace_name
----------------+-----------------
 Q1_2012        |
 Q2_2012        |
 Q3_2012        |
 Q4_2012        | TS_1
 Q1_2013        | TS_3
(5 rows)
```

16.3.8 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 `ALTER TABLE… RENAME PARTITION` command renames a partition.

The same behavior as previously described applies for the `subpartition_name` used with the `RENAME SUBPARTITION` clause.

You must own the specified table to invoke `ALTER TABLE… RENAME PARTITION` or `ALTER TABLE… RENAME SUBPARTITION`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 be renamed.

`subpartition_name`

The name of the subpartition to be renamed.

`new_name`

The new name of the partition or subpartition.

16.3.8.1 Example - Renaming a Partition

The following command creates a list-partitioned table named `sales`:

```
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;
 partition_name |             high_value
----------------+-------------------------------------
 EUROPE         | FOR VALUES IN ('FRANCE', 'ITALY')
 ASIA           | FOR VALUES IN ('INDIA', 'PAKISTAN')
 AMERICAS       | FOR VALUES IN ('US', 'CANADA')
(3 rows)
```

The following command renames the `americas` partition to `n_america`.

```
ALTER TABLE sales
 RENAME PARTITION americas TO n_america;
```

Querying the `ALL_TAB_PARTITIONS` view demonstrates that the partition has been successfully renamed.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |             high_value
----------------+-------------------------------------
 EUROPE         | FOR VALUES IN ('FRANCE', 'ITALY')
 ASIA           | FOR VALUES IN ('INDIA', 'PAKISTAN')
 N_AMERICA      | FOR VALUES IN ('US', 'CANADA')
(3 rows)
```

16.3.9 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 be deleted.

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; please note that this command does not work on a `HASH` partitioned table. When you delete a partition, any subpartitions (of that partition) are deleted as well.

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.

16.3.9.1 Example - Deleting a Partition

The example that follows deletes a partition of the `sales` table. Use the following command to 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 asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Querying the `ALL_TAB_PARTITIONS` view displays the partition names.

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |             high_value
----------------+------------------------------------
 EUROPE         | FOR VALUES IN ('FRANCE', 'ITALY')
 ASIA           | FOR VALUES IN ('INDIA', 'PAKISTAN')
 AMERICAS       | FOR VALUES IN ('US', 'CANADA')
(3 rows)
```

To delete the americas partition from the `sales` table, invoke the following command:

```
ALTER TABLE sales DROP PARTITION americas;
```

Querying the `ALL_TAB_PARTITIONS` view demonstrates that the partition has been successfully deleted.

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |             high_value
----------------+------------------------------------
 EUROPE         | FOR VALUES IN ('FRANCE', 'ITALY')
 ASIA           | FOR VALUES IN ('INDIA', 'PAKISTAN')
(2 rows)
```

16.3.10 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 be deleted.

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.

16.3.10.1 Example - Deleting a Subpartition

The example that follows deletes a subpartition of the `sales` table. Use the following command to create the `sales` table:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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 ('CANADA', 'US'),
      SUBPARTITION asia VALUES ('PAKISTAN', 'INDIA')
    ),
    PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
);
```

Querying the `ALL_TAB_SUBPARTITIONS` view displays the subpartition names.

```
edb=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |             high_value
-------------------+------------------------------------
 ASIA              | FOR VALUES IN ('PAKISTAN', 'INDIA')
 AMERICAS          | FOR VALUES IN ('CANADA', 'US')
 EUROPE            | FOR VALUES IN ('ITALY', 'FRANCE')
 SYS0101           | DEFAULT
(4 rows)
```

To delete the `americas` subpartition from the `sales` table, invoke the following command:

```
ALTER TABLE sales DROP SUBPARTITION americas;
```

Querying the `ALL_TAB_SUBPARTITIONS` view demonstrates that the subpartition has been successfully deleted.

```
edb=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |             high_value
-------------------+------------------------------------
 ASIA              | FOR VALUES IN ('PAKISTAN', 'INDIA')
 EUROPE            | FOR VALUES IN ('ITALY', 'FRANCE')
 SYS0101           | DEFAULT
(3 rows)
```

16.3.11 TRUNCATE TABLE

Use the `TRUNCATE TABLE` command to remove the contents of a table, while preserving the table definition. When you truncate a table, any partitions or subpartitions of that table are also truncated. The syntax is:

```
TRUNCATE TABLE <table_name>
```

Parameters

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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. When you delete a table, any partitions or subpartitions (of that table) are deleted as well.

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.

16.3.11.1 Example - Emptying a Table

The example that follows removes the data from the `sales` table. Use the following command to 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 asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Populate the `sales` table with the command:

```
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'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that the partitions are populated with data.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date         | amount
----------------+---------+---------+----------+---------------------+--------
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(17 rows)
```

To delete the contents of the `sales` table, invoke the following command:

```
TRUNCATE TABLE sales;
```

Now, querying the `sales` table shows that the data has been removed but the structure is intact.

```
edb=# SELECT tableoid::regclass, * FROM sales;
 tableoid | dept_no | part_no | country | date | amount
----------+---------+---------+---------+------+--------
(0 rows)
```

For more information about the `TRUNCATE TABLE` command, see the PostgreSQL documentation at:

https://www.postgresql.org/docs/11/static/sql-truncate.html

16.3.12 ALTER TABLE...TRUNCATE PARTITION

Use the `ALTER TABLE… TRUNCATE PARTITION` command to remove the data from the specified partition, leaving 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The name of the partition to be deleted.

Description

Use the `ALTER TABLE… TRUNCATE PARTITION` command to remove the data from the specified partition, leaving the partition structure intact. When you truncate a partition, any subpartitions of that partition are also truncated.

`ALTER TABLE… TRUNCATE PARTITION` will not cause `ON DELETE` triggers that might exist for the table to fire, but it will fire `ON TRUNCATE` triggers. If an `ON TRUNCATE` trigger is defined for the partition, all `BEFORE TRUNCATE` triggers are fired before any truncation happens, and 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 for compatibility only; the clauses are parsed and ignored.

16.3.12.1 Example - Emptying a Partition

The example that follows removes the data from a partition of the `sales` table. Use the following command to 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 asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Populate the `sales` table with the command:

```
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'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Querying the `sales` table shows that the partitions are populated with data.

```
edb=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date         | amount
----------------+---------+---------+----------+---------------------+--------
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(17 rows)
```

To delete the contents of the `americas` partition, invoke the following command:

```
ALTER TABLE sales TRUNCATE PARTITION americas;
```

Now, querying the `sales` table shows that the content of the `americas` partition has been removed.

```
edb=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date         | amount
---------------+---------+---------+----------+---------------------+--------
 sales_europe |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia   |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia   |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia   |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia   |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia   |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(9 rows)
```

While the rows have been removed, the structure of the `americas` partition is still intact.

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |             high_value
----------------+------------------------------------
 EUROPE         | FOR VALUES IN ('FRANCE', 'ITALY')
 ASIA           | FOR VALUES IN ('INDIA', 'PAKISTAN')
 AMERICAS       | FOR VALUES IN ('US', 'CANADA')
(3 rows)
```

16.3.13 ALTER TABLE...TRUNCATE SUBPARTITION

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Use the `ALTER TABLE… TRUNCATE SUBPARTITION` command to remove all of the data from the specified subpartition, leaving 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 be truncated.

Description

The `ALTER TABLE… TRUNCATE SUBPARTITION` command removes all data from a specified subpartition, leaving the subpartition structure intact.

`ALTER TABLE… TRUNCATE SUBPARTITION` will not cause `ON DELETE` triggers that might exist for the table to fire, but it will fire `ON TRUNCATE` triggers. If an `ON TRUNCATE` trigger is defined for the subpartition, all `BEFORE TRUNCATE` triggers are fired before any truncation happens, and 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 for compatibility only; the clauses are parsed and ignored.

16.3.13.1 Example - Emptying a Subpartition

The example that follows removes the data from a subpartition of the `sales` table. Use the following 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) 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')
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  ),
  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')
  )
);
```

Populate the `sales` table with the command:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2011', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2011', '50000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2011', '4950'),
  (20, '3788a', 'US', '04-Apr-2012', '37500'),
  (40, '4577b', 'INDIA', '11-Jun-2011', '25000'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that the rows have been distributed amongst the subpartitions.

```
edb=# SELECT tableoid::regclass, * FROM sales;
    tableoid       | dept_no| part_no | country|        date         | amount
-------------------+--------+---------+--------+---------------------+--------
sales_americas_2011|      30| 7588b   | CANADA | 14-DEC-11 00:00:00 | 50000
sales_americas_2011|      40| 3788a   | US     | 12-MAY-11 00:00:00 | 4950
sales_europe_2011  |      10| 4519b   | FRANCE | 17-JAN-11 00:00:00 | 45000
sales_asia_2011    |      40| 4577b   | INDIA  | 11-JUN-11 00:00:00 | 25000
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| 4519b   | CANADA | 08-APR-12 00:00:00 |120000
sales_americas_2012|      20| 3788a   | US     | 04-APR-12 00:00:00 | 37500
sales_europe_2012  |      10| 9519b   | ITALY  | 07-JUL-12 00:00:00 | 15000
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| 4519b   | INDIA  | 02-DEC-12 00:00:00 | 5090
(12 rows)
```

To delete the contents of the `2012_americas` partition, invoke the following command:

```
ALTER TABLE sales TRUNCATE SUBPARTITION "americas_2012";
```

Now, querying the `sales` table shows that the content of the `americas_2012` partition has been removed.

```
edb=# SELECT tableoid::regclass, * FROM sales;
 tableoid          | dept_no| part_no | country  |        date         |amount
-------------------+--------+---------+----------+---------------------+------
sales_americas_2011|      30| 7588b   | CANADA   | 14-DEC-11 00:00:00 |50000
sales_americas_2011|      40| 3788a   | US       | 12-MAY-11 00:00:00 | 4950
sales_europe_2011  |      10| 4519b   | FRANCE   | 17-JAN-11 00:00:00 |45000
sales_asia_2011    |      40| 4577b   | INDIA    | 11-JUN-11 00:00:00 |25000
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
sales_europe_2012  |      10| 9519b   | ITALY    | 07-JUL-12 00:00:00 |15000
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| 4519b   | INDIA    | 02-DEC-12 00:00:00 | 5090
(8 rows)
```

While the rows have been removed, the structure of the `2012_americas` partition is still intact.

```
edb=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |             high_value
-------------------+------------------------------------
AMERICAS_2011      | FOR VALUES IN ('US', 'CANADA')
ASIA_2011          | FOR VALUES IN ('PAKISTAN', 'INDIA')
EUROPE_2011        | FOR VALUES IN ('ITALY', 'FRANCE')
AMERICAS_2012      | FOR VALUES IN ('US', 'CANADA')
ASIA_2012          | FOR VALUES IN ('PAKISTAN', 'INDIA')
EUROPE_2012        | FOR VALUES IN ('ITALY', 'FRANCE')
AMERICAS_2013      | FOR VALUES IN ('US', 'CANADA')
ASIA_2013          | FOR VALUES IN ('PAKISTAN', 'INDIA')
EUROPE_2013        | FOR VALUES IN ('ITALY', 'FRANCE')
(9 rows)
```

16.3.14 DROP TABLE

Use the PostgreSQL `DROP TABLE` command to remove a partitioned table definition, it's partitions and subpartitions, and delete 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. When you delete a table, any partitions or subpartitions (of that table) are deleted as well.

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, to delete the `sales` table, invoke the following command:

```
DROP TABLE sales;
```

The server will confirm that the table has been dropped.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# drop table sales;
DROP TABLE
edb=#
```

For more information about the `DROP TABLE` command, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/11/static/sql-droptable.html

16.4 Handling Stray Values in a LIST or RANGE Partitioned Table

A `DEFAULT` or `MAXVALUE` partition or subpartition will capture any rows that do not meet the other partitioning rules defined for a table.

Defining a DEFAULT Partition

A `DEFAULT` partition will capture any rows that do not fit into any other partition in a `LIST` partitioned (or subpartitioned) table. If you do not include a `DEFAULT` rule, any row that does not match one of the values in the partitioning constraints will result in an error. Each `LIST` partition or subpartition may 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 will store any rows that do not match the rules specified for other partitions.

The last example created a list partitioned table in which the server decided which partition to store the data based upon the value of the `country` column. If you attempt to add a row in which the value of the `country` column contains a value not listed in the rules, Advanced Server reports an error.

```
edb=# INSERT INTO sales VALUES
edb-#  (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');
ERROR:  no partition of relation "sales_2012" found for row
DETAIL:  Partition key of the failing row contains (country) = (IRELAND).
```

The following example creates the same table, but adds a `DEFAULT` partition. The server will store any rows that do not 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)
);
```

To test the `DEFAULT` partition, add row with a value in the `country` column that does not match one of the countries specified in the partitioning

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

constraints.

```
INSERT INTO sales VALUES
  (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');
```

Querying the contents of the `sales` table confirms that the previously rejected row is now stored in the `sales_others` partition.

```
edb=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date         | amount
----------------+---------+---------+----------+---------------------+--------
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_others   |      40 | 3000x   | IRELAND  | 01-MAR-12 00:00:00 |  45000
(18 rows)
```

Advanced Server provides the following methods to re-assign 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 as long as there are no conflicting values between existing rows in the table and the values of the partition to be added. You can alternatively use the `ALTER TABLE… SPLIT PARTITION` command to split an existing partition. Examples are shown following this bullet point list.
- You can use the `ALTER TABLE… ADD SUBPARTITION` command to add a subpartition to a table with a `DEFAULT` rule as long as there are no conflicting values between existing rows in the table and the values of the subpartition to be added. You can alternatively use the `ALTER TABLE… SPLIT SUBPARTITION` command to split an existing subpartition.

Adding a Partition to a Table with a DEFAULT Partition

Using the table that was created with the `CREATE TABLE sales` command shown at the beginning of this section, the following shows use of the `ALTER TABLE... ADD PARTITION` command assuming there is no conflict of values between the existing rows in the table and the values of the partition to be added.

```
edb=# ALTER TABLE sales ADD PARTITION africa values ('SOUTH AFRICA', 'KENYA');
ALTER TABLE
```

However, the following shows the error when there are conflicting values when the following rows have been inserted into the table.

```
edb=# INSERT INTO sales (dept_no, country) VALUES
(1,'FRANCE'),(2,'INDIA'),(3,'US'),(4,'SOUTH AFRICA'),(5,'NEPAL');
INSERT 0 5
```

Row `(4,'SOUTH AFRICA')` conflicts with the `VALUES` list in the `ALTER TABLE... ADD PARTITION` statement, thus resulting in an error.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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
```

Splitting a DEFAULT Partition

The following example splits a `DEFAULT` partition, redistributing the partition's content between two new partitions. The table was created with the `CREATE TABLE sales` command shown at the beginning of this section.

The following inserts rows into the table including rows into the `DEFAULT` partition.

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (30, '9519b', 'US', '12-Apr-2012', '145000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (40, '4519b', 'SOUTH AFRICA', '08-Apr-2012', '120000'),
  (40, '4519b', 'KENYA', '08-Apr-2012', '120000'),
  (50, '3788a', 'CHINA', '12-May-2012', '4950');
```

The partitions include the `DEFAULT others` partition.

```
edb=# SELECT partition_name, high_value FROM FROM ALL_TAB_PARTITIONS;
 partition_name |             high_value
----------------+-------------------------------------
EUROPE          | FOR VALUES IN ('FRANCE', 'ITALY')
ASIA            | FOR VALUES IN ('INDIA', 'PAKISTAN')
AMERICAS        | FOR VALUES IN ('US', 'CANADA')
OTHERS          | DEFAULT
(4 rows)
```

The following shows the rows distributed amongst the partitions.

```
edb=# SELECT tableoid::regclass, * FROM sales;
 tableoid     | dept_no| part_no |   country    |        date         | amount
--------------+--------+---------+--------------+---------------------+-------
sales_americas|     30 | 9519b   | US           | 12-APR-12 00:00:00 |145000
sales_americas|     30 | 7588b   | CANADA       | 14-DEC-12 00:00:00 | 50000
sales_europe  |     10 | 4519b   | FRANCE       | 17-JAN-12 00:00:00 | 45000
sales_europe  |     10 | 9519b   | ITALY        | 07-JUL-12 00:00:00 | 15000
sales_asia    |     20 | 3788a   | INDIA        | 01-MAR-12 00:00:00 | 75000
sales_asia    |     20 | 3788a   | PAKISTAN     | 04-JUN-12 00:00:00 | 37500
sales_others  |     40 | 4519b   | SOUTH AFRICA | 08-APR-12 00:00:00 |120000
sales_others  |     40 | 4519b   | KENYA        | 08-APR-12 00:00:00 |120000
sales_others  |     50 | 3788a   | CHINA        | 12-MAY-12 00:00:00 |  4950
(9 rows)
```

The following command splits the `DEFAULT others` partition into two 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |              high_value
----------------+-----------------------------------------
 EUROPE         | FOR VALUES IN ('FRANCE', 'ITALY')
 ASIA           | FOR VALUES IN ('INDIA', 'PAKISTAN')
 AMERICAS       | FOR VALUES IN ('US', 'CANADA')
 AFRICA         | FOR VALUES IN ('SOUTH AFRICA', 'KENYA')
 OTHERS         | DEFAULT
(5 rows)
```

The following shows that the rows have been redistributed across the new partitions.

```
edb=# SELECT tableoid::regclass, * FROM sales;
 tableoid      |dept_no | part_no |   country   |        date         | amount
---------------+--------+---------+-------------+---------------------+-------
sales_americas |     30 | 9519b   | US          | 12-APR-12 00:00:00 |145000
sales_americas |     30 | 7588b   | CANADA      | 14-DEC-12 00:00:00 | 50000
sales_europe   |     10 | 4519b   | FRANCE      | 17-JAN-12 00:00:00 | 45000
sales_europe   |     10 | 9519b   | ITALY       | 07-JUL-12 00:00:00 | 15000
sales_asia     |     20 | 3788a   | INDIA       | 01-MAR-12 00:00:00 | 75000
sales_asia     |     20 | 3788a   | PAKISTAN    | 04-JUN-12 00:00:00 | 37500
sales_africa   |     40 | 4519b   | SOUTH AFRICA| 08-APR-12 00:00:00 |120000
sales_africa   |     40 | 4519b   | KENYA       | 08-APR-12 00:00:00 |120000
sales_others_1 |     50 | 3788a   | CHINA       | 12-MAY-12 00:00:00 | 4950
(9 rows)
```

Defining a MAXVALUE Partition

A `MAXVALUE` partition (or subpartition) will capture any rows that do not fit into any other partition in a range-partitioned (or subpartitioned) table. If you do not include a `MAXVALUE` rule, any row that exceeds the maximum limit specified by the partitioning rules will result in an error. Each partition or subpartition may 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 will store any rows that do not match the rules specified for other partitions.

The last example created a range-partitioned table in which the data was partitioned based upon the value of the `date` column. If you attempt to add a row with a `date` that exceeds a date listed in the partitioning constraints, Advanced Server reports an error.

```
edb=# INSERT INTO sales VALUES
edb-#   (40, '3000x', 'IRELAND', '01-Mar-2013', '45000');
ERROR:  no partition of relation "sales" found for row
DETAIL:  Partition key of the failing row contains (date) = (01-MAR-13 00:00:00).
```

The following `CREATE TABLE` command creates the same table, but with a `MAXVALUE` partition. Instead of throwing an error, the server will store any rows that do not match the previous partitioning constraints in the `others` partition.

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
)
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 will store the row in the `others` partition.

```
INSERT INTO sales VALUES
  (40, '3000x', 'IRELAND', '01-Mar-2013', '45000');
```

Querying the contents of the `sales` table confirms that the previously rejected row is now stored in the `sales_others` partition.

```
edb=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date         | amount
---------------+---------+---------+----------+---------------------+--------
 sales_q1_2012 |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_q1_2012 |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_q1_2012 |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_q2_2012 |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_q2_2012 |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_q2_2012 |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_q2_2012 |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_q3_2012 |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_q3_2012 |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_q3_2012 |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_q4_2012 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_q4_2012 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_q4_2012 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_q4_2012 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_q4_2012 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_others  |      40 | 3000x   | IRELAND  | 01-MAR-13 00:00:00 |  45000
(18 rows)
```

Please note that Advanced Server does not have a way to re-assign the contents of a `MAXVALUE` partition or subpartition.

- You cannot use the `ALTER TABLE… ADD PARTITION` statement to add a partition to a table with a `MAXVALUE` rule, but you can use the `ALTER TABLE… SPLIT PARTITION` statement to split an existing partition.
- You cannot use the `ALTER TABLE… ADD SUBPARTITION` statement to add a subpartition to a table with a `MAXVALUE` rule , but you can split an existing subpartition with the `ALTER TABLE… SPLIT SUBPARTITION` statement.

16.5 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 (based on a greater-than or less-than value) on a small set of columns, consider using those columns in `RANGE` partitioning rules.

Specifying Multiple Keys in a Range-Partitioned Table

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Range-partitioned table definitions may 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.

```
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 q1_2012
    VALUES LESS THAN(2012, 4),
  PARTITION q2_2012
    VALUES LESS THAN(2012, 7),
  PARTITION q3_2012
    VALUES LESS THAN(2012, 10),
  PARTITION q4_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.

```
edb=# EXPLAIN SELECT * FROM sales WHERE sale_year = 2012 AND sale_month = 8;
                                QUERY PLAN
---------------------------------------------------------------------------
 Append  (cost=0.00..14.35 rows=1 width=250)
   ->  Seq Scan on sales_q3_2012  (cost=0.00..14.35 rows=1 width=250)
         Filter: ((sale_year = '2012'::numeric) AND (sale_month = '8'::numeric))
(3 rows)
```

Since all rows with a value of `8` in the `sale_month` column and a value of `2012` in the `sale_year` column will be stored in the `q3_2012` partition, Advanced Server searches only that partition.

16.6 Retrieving Information about a Partitioned Table

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 are using the EDB-PSQL client, you can also discover the structure of a view by entering:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
\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, the following code snippet displays the names of a subpartitioned table:

```
edb=# SELECT subpartition_name, partition_name FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name | partition_name
-------------------+----------------
 EUROPE_2011       | EUROPE
 EUROPE_2012       | EUROPE
 ASIA_2011         | ASIA
 ASIA_2012         | ASIA
 AMERICAS_2011     | AMERICAS
 AMERICAS_2012     | AMERICAS
(6 rows)
```

16.6.1 Table Partitioning Views - Reference

Query the following catalog views (compatible with Oracle databases), to review detailed information about your partitioned tables.

16.6.1.1 ALL_PART_TABLES

The following table lists the information available in the `ALL_PART_TABLES` view:

Column	Type	Description
`owner`	`name`	The owner of the table.
`schema_name`	`name`	The schema in which the table resides.
`table_name`	`name`	The name of the table.
`partitioning_type`	`text`	RANGE, LIST, or HASH.
`subpartitioning_type`	`text`	RANGE, LIST, HASH, or NONE.
`partition_count`	`bigint`	The number of partitions.
`def_subpartition_count`	`integer`	The default subpartition count - this will always be 0.
`partitioning_key_count`	`integer`	The number of columns listed in the partition by clause.
`subpartitioning_key_count`	`integer`	The number of columns in the subpartition by clause.
`status`	`character varying(8)`	This column will always be VALID.
`def_tablespace_name`	`character varying(30)`	This column will always be NULL.
`def_pct_free`	`numeric`	This column will always be NULL.
`def_pct_used`	`numeric`	This column will always be NULL.
`def_ini_trans`	`numeric`	This column will always be NULL.
`def_max_trans`	`numeric`	This column will always be NULL.
`def_initial_extent`	`character varying(40)`	This column will always be NULL.
`def_next_extent`	`character varying(40)`	This column will always be NULL.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	Description
def_min_extents	character varying(40)	This column will always be NULL.
def_max_extents	character varying(40)	This column will always be NULL.
def_pct_increase	character varying(40)	This column will always be NULL.
def_freelists	numeric	This column will always be NULL.
def_freelist_groups	numeric	This column will always be NULL.
def_logging	character varying(7)	This column will always be YES.
def_compression	character varying(8)	This column will always be NONE.
def_buffer_pool	character varying(7)	This column will always be DEFAULT.
ref_ptn_constraint_name	character varying(30)	This column will always be NULL.
interval	character varying(1000)	This column will always be NULL.

16.6.1.2 ALL_TAB_PARTITIONS

The following table lists the information available in the ALL_TAB_PARTITIONS view:

Column	Type	Description
table_owner	name	The owner of the table.
schema_name	name	The schema in which the table resides.
table_name	name	The name of the table.
composite	text	YES if the table is subpartitioned; NO if it is not subpartitioned.
partition_name	name	The name of the partition.
subpartition_count	bigint	The number of subpartitions for this partition.
high_value	text	The partition limit for RANGE partitions, or the partition value for LIST partitions.
high_value_length	integer	The length of high partitioning value.
partition_position	integer	This column will always be NULL
tablespace_name	name	The tablespace in which this partition resides.
pct_free	numeric	This column will always be 0.
pct_used	numeric	This column will always be 0.
ini_trans	numeric	This column will always be 0.
max_trans	numeric	This column will always be 0.
initial_extent	numeric	This column will always be NULL.
next_extent	numeric	This column will always be NULL.
min_extent	numeric	This column will always be 0.
max_extent	numeric	This column will always be 0.
pct_increase	numeric	This column will always be 0.
freelists	numeric	This column will always be NULL.
freelist_groups	numeric	This column will always be NULL.
logging	character varying(7)	This column will always be YES.
compression	character varying(8)	This column will always be NONE.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	Description
num_rows	numeric	The approx. number of rows in this partition.
blocks	integer	The approx. number of blocks in this partition.
empty_blocks	numeric	This column will always be NULL.
avg_space	numeric	This column will always be NULL.
chain_cnt	numeric	This column will always be NULL.
avg_row_len	numeric	This column will always be NULL.
sample_size	numeric	This column will always be NULL.
last_analyzed	timestamp without time zone	This column will always be NULL.
buffer_pool	character varying(7)	This column will always be NULL.
global_stats	character varying(3)	This column will always be YES.
user_stats	character varying(3)	This column will always be NO.
backing_table	regclass	OID of the backing table for this partition.

16.6.1.3 ALL_TAB_SUBPARTITIONS

The following table lists the information available in the ALL_TAB_SUBPARTITIONS view:

Column	Type	Description
table_owner	name	The name of the owner of the table.
schema_name	name	The name of the schema in which the table resides.
table_name	name	The name of the table.
partition_name	name	The name of the partition.
subpartition_name	name	The name of the subpartition.
high_value	text	The subpartition limit for RANGE subpartitions, or the subpartition value for LIST subpartitions.
high_value_length	integer	The length of high partitioning value.
subpartition_position	integer	This column will always be NULL
tablespace_name	name	The tablespace in which this subpartition resides.
pct_free	numeric	This column will always be 0.
pct_used	numeric	This column will always be 0.
ini_trans	numeric	This column will always be 0.
max_trans	numeric	This column will always be 0.
initial_extent	numeric	This column will always be NULL.
next_extent	numeric	This column will always be NULL.
min_extent	numeric	This column will always be 0.
max_extent	numeric	This column will always be 0.
pct_increase	numeric	This column will always be 0.
freelists	numeric	This column will always be NULL.
freelist_groups	numeric	This column will always be NULL.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	Description
logging	character varying(7)	This column will always be YES.
compression	character varying(8)	This column will always be NONE.
num_rows	numeric	The approx. number of rows in this subpartition.
blocks	integer	The approx. number of blocks in this subpartition.
empty_blocks	numeric	This column will always be NULL.
avg_space	numeric	This column will always be NULL.
chain_cnt	numeric	This column will always be NULL.
avg_row_len	numeric	This column will always be NULL.
sample_size	numeric	This column will always be NULL.
last_analyzed	timestamp without time zone	This column will always be NULL.
buffer_pool	character varying(7)	This column will always be NULL.
global_stats	character varying(3)	This column will always be YES.
user_stats	character varying(3)	This column will always be NO.
backing_table	regclass	OID of the backing table for this subpartition.

16.6.1.4 ALL_PART_KEY_COLUMNS

The following table lists the information available in the ALL_PART_KEY_COLUMNS view:

Column	Type	Description
owner	name	The name of the table owner.
schema_name	name	The name of the schema on which the table resides.
name	name	The name of the table.
object_type	character (5)	This column will always be TABLE.
column_name	name	The name of the partitioning key column.
column_position	integer	The position of this column within the partitioning key (the first column has a column position of 1, the second column has a column position of 2...)

16.6.1.5 ALL_SUBPART_KEY_COLUMNS

The following table lists the information available in the ALL_SUBPART_KEY_COLUMNS view:

Column	Type	Description
owner	name	The name of the table owner.
schema_name	name	The name of the schema on which the table resides.
name	name	The name of the table.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	Description
object_type	character(5)	This column will always be TABLE.
column_name	name	The name of the partitioning key column.
column_position	integer	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...)

17 ECPGPlus Guide

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) 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 (Advanced Server) database. When you use ECPGPlus to compile an application, the SQL code is syntax-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 do not need to include the BEGIN DECLARE SECTION and END DECLARE SECTION directives.

PostgreSQL Compatibility

While most ECPGPlus statements will 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.

17.1 ECPGPlus - Overview

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) to create ECPGPlus. ECPGPlus is a Pro*C-compatible version of the PostgreSQL C pre-compiler. 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.

The following diagram charts the path of a program containing embedded SQL statements as it is compiled into an executable:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

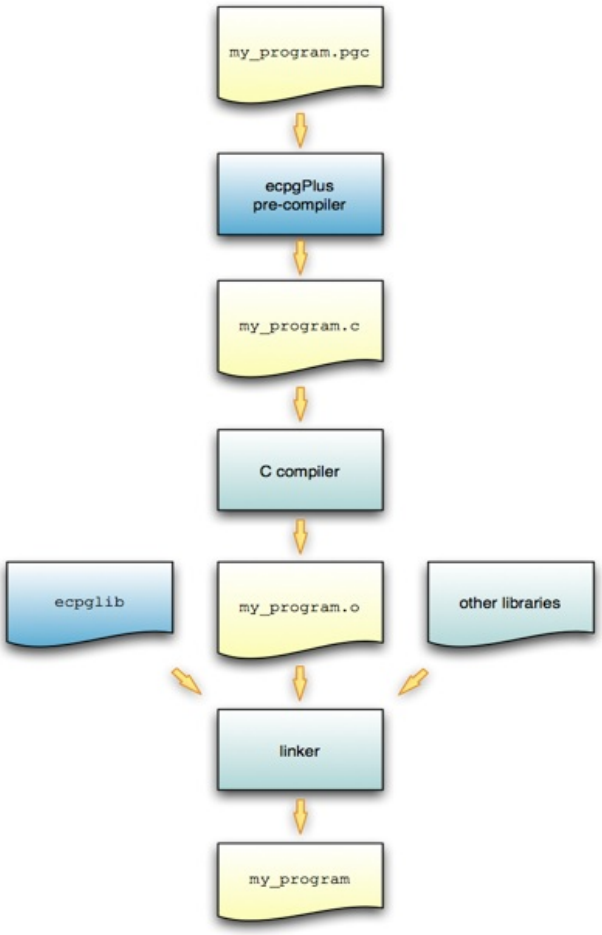


Fig. 1: Compilation of a program containing embedded SQL statements

To produce an executable from a C program that contains embedded SQL statements, pass the program (`my_program.pgc` in the diagram above) to the ECPGPlus pre-compiler. 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`). Then, pass the C program to a C compiler; the C compiler generates an object file (`my_program.o`). Finally, 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 cannot validate the *semantics*. For example, the preprocessor will confirm that an `INSERT` statement is syntactically correct, but it cannot confirm that the table mentioned in the `INSERT` statement actually exists.

Behind the Scenes

A client application contains a mix of C code and SQL code comprised of the following elements:

- C preprocessor directives
- C declarations (variables, types, functions, ...)
- C definitions (variables, types, functions, ...)
- SQL preprocessor directives
- SQL statements

For example:

```
 1 #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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
8 void displayCustomer(int custNumber)
9 {
10  EXEC SQL BEGIN DECLARE SECTION;
11    VARCHAR custName[50];
12    float custBalance;
13    int custID = custNumber;
14  EXEC SQL END DECLARE SECTION;
15
16  EXEC SQL SELECT name, balance
17    INTO :custName, :custBalance
18    FROM customer
19    WHERE id = :custID;
20
21  printInt("ID", custID);
22  printStr("Name", custName);
23  printFloat("Balance", custBalance);
24 }
```

In the above code fragment:

- Line 1 specifies a directive to the C preprocessor.

  C preprocessor directives may be interpreted or ignored; the option is controlled by a command line option (`-C 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 will appear in the output file.

- Line 2 specifies a directive to the SQL preprocessor.

  SQL preprocessor directives are interpreted by the ECPGPlus preprocessor, and are not 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 within SQL code are known as `host variables` . If you invoke the ECPGPlus preprocessor in Pro*C mode (`-C PROC`), you may refer to *any* C variable within a SQL statement; otherwise you must declare each host variable within a `BEGIN/END DECLARATION SECTION` pair.

- Lines 16 through 19 contain a SQL statement.

  SQL statements are translated into calls to the ECPGPlus run-time library.

- Lines 21 through 23 contain C code.

  C code is copied to the output file without change.

Any SQL statement must be prefixed with `EXEC SQL` and extends to the next (unquoted) semicolon. For example:

```
printf(“Updating employee salaries\n”);

EXEC SQL UPDATE emp SET sal = sal * 1.25;
EXEC SQL COMMIT;

printf(“Employee salaries updated\n”);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

When the preprocessor encounters the code fragment shown above, 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 would appear similar to the following:

```
printf("Updating employee salaries\n");

{
  ECPGdo( __LINE__, 0, 1, NULL, 0, ECPGst_normal,
          "update emp set sal = sal * 1.25",
          ECPGt_EOIT, ECPGt_EORT);
}

{
  ECPGtrans(__LINE__, NULL, "commit");
}

printf(“Employee salaries updated\n”);
```

Installation and Configuration

On Windows, ECPGPlus is installed by the Advanced Server installation wizard as part of the `Database Server` component. On Linux, install with the `edb-asxx-server-devel` RPM package where `xx` is the Advanced Server version number. By default, the executable is located in:

On Windows:

```
C:\Program Files\edb\as11\bin
```

On Linux:

```
/usr/edb/as11/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\as11\bin
set PATH=%EDB_PATH%;%PATH%
```

On Linux:

```
export EDB_PATH==/usr/edb/as11/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 in this guide, you will need:

- a C compiler (and linker)
- the `make` utility

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- 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 guide. To use the sample code, save it in a file named `makefile` in the directory that contains the source code file.

```
INCLUDES = -I$(shell pg_config --includedir)
LIBPATH = -L $(shell pg_config --libdir)
CFLAGS += $(INCLUDES) -g
LDFLAGS += -g
LDLIBS += $(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:

```
INCLUDES = -I$(shell pg_config --includedir)
LIBPATH = -L $(shell pg_config --libdir)
```

The `pg_config` program is shipped with Advanced Server.

`make` knows that it should 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 run-time library (`-lecpg`) and the libpq library (`-lpq`)

```
CFLAGS += $(INCLUDES) -g
LDFLAGS += -g
LDLIBS += $(LIBPATH) -lecpg -lpq
```

The sample `makefile` instructs 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 will be compiled. The first compile option is:

```
.pgc.c:
     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 pre-compiler with the `-c` flag (instructing the compiler to convert SQL code into C), using the value of the `INCLUDES` variable and the name of the `.pgc` file.

```
.pc.c:
    ecpg -C PROC -c $(INCLUDES) $?
```

The second option tells `make` how to transform a file that ends in `.pg` (an ECPG source file) into a file that ends in `.c` (a C program), using the ECPGPlus extensions. It invokes the ECPG pre-compiler with the `-c` flag (instructing the compiler to convert SQL code into C), as well as the `-C PROC` flag (instructing 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 wish to compile. For example, to compile an ECPG source code file named `customer_list.pgc` , use the command:

```
make customer_list
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `make` utility consults the `makefile` (located in the current directory), discovers that the `makefile` contains a rule that will compile `customer_list.pgc` into a C program (`customer_list.c`), and then uses the rules built into `make` to compile `customer_list.c` into an executable program.

ECPGPlus Command Line Options

In the sample `makefile` shown above, `make` includes the `-C` option when invoking ECPGPlus to specify that ECPGPlus should be invoked in Pro*C compatible mode.

If you include the `-C` `PROC` keywords on the command line, in addition to the ECPG syntax, you may use Pro*C command line syntax; for example:

```
$ ecpg -C PROC INCLUDE=/usr/edb/as11/include acct_update.c
```

To display a complete list of the other ECPGPlus options available, navigate to the ECPGPlus installation directory, and enter:

```
./ecpg --help
```

The command line options are:

Option	Description
-c	Automatically generate C code from embedded SQL code.

-C *mode*	Use the `-C` option to specify a compatibility mode:
	`INFORMIX`
	`INFORMIX_SE`
	`PROC`

-D *symbol*	Define a preprocessor *symbol*. The *-D* keyword is not supported when compiling in *PROC mode*. Instead, use the Oracle-style *'DEFINE='*clause.
-h	Parse a header file, this option includes option `'-c'` .
-i	Parse system, include files as well.
-I directory	Search *directory* for `include` files.
-o *outfile*	Write the result to *outfile*.
-r *option*	Specify run-time behavior; *option* can be: `no_indicator` - Do not use indicators, but instead use special values to represent NULL values. `prepare` - Prepare all statements before using them. `questionmarks` - Allow use of a question mark as a placeholder. `usebulk` - Enable bulk processing for `INSERT` , `UPDATE` , and `DELETE` statements that operate on host variable arrays.
--regression	Run in regression testing mode.
-t	Turn on `autocommit` of transactions.
-l	Disable `#line` directives.
--help	Display the help options.
--version	Output version information.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> **Note**
>
> If you do not specify an output file name when invoking ECPGPlus, the output file name is created by stripping off the `.pgc` filename extension, and appending `.c` to the file name.

17.2 Using Embedded SQL

Each of the following sections leads with a code sample, followed by an explanation of each section within the code sample.

Example - A Simple Query

The first code sample demonstrates how to execute a `SELECT` statement (which 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.

```
/************************************************************
 * 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 '1safepwd';

  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;

  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);
  EXEC SQL DISCONNECT;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
}
/*************************************************************\*
```

The code sample begins by including the prototypes and type definitions for the C `stdio` library, and then declares the `main` function:

```
 #include <stdio.h>

int main(void)
{
```

Next, the application declares a set of host variables used to interact with the database server:

```
 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;
```

Please note that if you plan to pre-compile the code in `PROC` mode, you may omit the `BEGIN DECLARE…END DECLARE` section. For more information about declaring host variables, refer to the Declaring Host Variables.

The data type associated with each variable within 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 instructs the server how to handle an error:

```
 EXEC SQL WHENEVER SQLERROR sqlprint;
```

If the client application encounters an error in the SQL code, the server will print an error message to `stderr` (standard error), using the `sqlprint()` function supplied with `ecpglib`. The next `EXEC SQL` statement establishes a connection with Advanced Server:

```
EXEC SQL CONNECT TO edb
  USER 'alice' IDENTIFIED BY '1safepwd';
```

In our 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:

```
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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `comm` column contains the commission values earned by an employee, and could 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 on-screen results:

```
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`), 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`. Please note that 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 which may be `NULL`.

The final statement in the code sample closes the connection to the server:

```
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 may contain the following values:

Indicator Value	Denotes
If an indicator variable is less than `0`.	The value returned by the server was `NULL`.
If an indicator variable is equal to `0`.	The value returned by the server was not `NULL`, and was not truncated.
If an indicator variable is greater than `0`.	The value returned by the server was truncated when stored in the host variable.

When including an indicator variable in an `INTO` clause, you are not required to include the optional `INDICATOR` keyword.

You may omit an indicator variable if you are certain that a query will never return a `NULL` value into the corresponding host variable. If you omit an indicator variable and a query returns a `NULL` value, `ecpglib` will raise a run-time error.

Declaring Host Variables

You can use a *host variable* in a SQL statement at any point that a value may appear within 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 demonstrate 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

and the `loc` column, respectively.

If you are compiling in `PROC` mode, you may 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:

```
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 are not 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, as shown below:

```
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, :v_dname, :v_loc);
}
```

You can also include the `INTO` clause in a `SELECT` statement to use the host variables to retrieve information:

```
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 demonstrated in the next example.

Example - Using a Cursor to Process a Result Set

The code sample that follows demonstrates using a cursor to process a result set. There are four basic steps 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, our 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.

```
/************************************************************
 * print_emps.pgc
 *
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;

  EXEC SQL WHENEVER SQLERROR sqlprint;

  EXEC SQL CONNECT TO edb USER :username IDENTIFIED BY :password;

  EXEC SQL DECLARE employees CURSOR FOR
    SELECT
      empno, ename, sal, comm
    FROM
      emp;

  EXEC SQL OPEN employees;

  EXEC SQL WHENEVER NOT FOUND DO break;

  for (;;)
  {
    EXEC SQL FETCH NEXT FROM employees
      INTO
       :v_empno, :v_ename, :v_sal, :v_comm INDICATOR :v_comm_ind;

    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);
  }
  EXEC SQL CLOSE employees;
  EXEC SQL DISCONNECT;
}
/*************************************************************
```

The code sample begins by including the prototypes and type definitions for the C `stdio` library, and then declares the `main` function:

```
 #include <stdio.h>

int main(int argc, char *argv[])
{
```

Next, the application declares a set of host variables used to interact with the database server:

```
 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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`argv[]` 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`); please note that we have omitted the error-checking code you would normally include 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 may be thinking that you could refer to `argv[1]` and `argv[2]` in a SQL statement (instead of creating a separate copy of each variable); that will not work. All host variables must be declared within a `BEGIN/END DECLARE SECTION` (unless you are compiling in `PROC` mode). Since `argv` is a function *parameter* (not an automatic variable), it cannot be declared within a `BEGIN/END DECLARE SECTION`. If you are compiling in `PROC` mode, you can refer to *any* C variable within a SQL statement.

The next statement instructs 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 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` will contain 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 do not 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:

```
EXEC SQL OPEN employees;
```

In the subsequent `FETCH` section, the client application will loop 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:

```
EXEC SQL WHENEVER NOT FOUND DO break;
```

The client application then uses a `FETCH` statement to retrieve each row from the cursor `INTO` the previously declared host variables:

```
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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 on-screen results:

```
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'`. Please note that 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`.

The final statements in the code sample close the cursor `(employees)`, and the connection to the server:

```
EXEC SQL CLOSE employees;
EXEC SQL DISCONNECT;
```

17.3 Using Descriptors

Dynamic SQL allows a client application to execute SQL statements that are composed at runtime. This is useful when you don't know the content or form a statement will take when you are writing a client application. ECPGPlus does *not* allow you to use a host variable in place of an identifier (such as a table name, column name or index name); instead, you should 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.

A client application must use a `GET DESCRIPTOR` statement to retrieve information from a descriptor. The following steps describe the basic flow of a client application using dynamic SQL:

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 syntax-check 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.

A descriptor may contain the attributes listed in the table below:

Field	Type	Attribute Description
`CARDINALITY`	`integer`	The number of rows in the result set.
`DATA`	N/A	The data value.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Field	Type	Attribute Description
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).
TYPE	integer	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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Example - Using a Descriptor to Return Data

The following simple application executes an SQL statement entered by an end user. The code sample demonstrates:

- 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.

When invoking the application, an end user must provide the name of the database on which the SQL statement will be performed, 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_stmt.pgc
*
*/

#include <stdio.h>
#include <stdlib.h>
#include <sql3types.h>
#include <sqlca.h>

EXEC SQL WHENEVER SQLERROR SQLPRINT;
static void print_meta_data( char * desc_name );

char *md1 = "col field                data              ret";
char *md2 = "num name                 type              len";
char *md3 = "--- -------------------- ----------------- ---";

int main( int argc, char *argv[] )
{

  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
{
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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 )
          print_meta_data( "row_desc" );

       printf("[RECORD %d]\n", row+1);

       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( "  %-20s : <null>\n", name.arr );
           else if( ind > 0 )
             printf( "  %-20s : <truncated>\n", name.arr );
           else
             printf( "  %-20s : %s\n", 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[] =
{
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  "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++ )
{

   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" );
}

/************************************************************
```

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>
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
#include <stdlib.h>
#include <sql3types.h>
#include <sqlca.h>
```

The sample provides minimal error handling; when the application encounters an SQL error, it prints the error message to screen:

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;
```

The application includes a forward-declaration for a function named `print_meta_data()` that will print 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 will use 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 that will contain the name of the database to which the application will connect, the content of the SQL Statement, and a host variable that will hold 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 an SQL descriptor to hold the metadata for a statement:

```
EXEC SQL ALLOCATE DESCRIPTOR parse_desc;
```

The application uses a `PREPARE` statement to syntax check the string provided by the user:

```
EXEC SQL PREPARE query FROM :stmt;
```

and a `DESCRIBE` statement to move the metadata for the query into the SQL descriptor.

```
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`.

```
EXEC SQL GET DESCRIPTOR parse_desc :col_count = COUNT;
```

If the column count is zero, the end user did not enter a `SELECT` statement; the application uses an `EXECUTE IMMEDIATE` statement to process the contents of the statement:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
if( col_count == 0 )
{
   EXEC SQL EXECUTE IMMEDIATE :stmt;
```

If the statement executes successfully, the application performs a `COMMIT`:

```
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`.

```
int row;
```

Then, the application allocates another descriptor that holds the description and the values of a specific row in the result set:

```
EXEC SQL ALLOCATE DESCRIPTOR row_desc;
```

The application declares and opens a cursor for the prepared statement:

```
EXEC SQL DECLARE my_cursor CURSOR FOR query;
EXEC SQL OPEN my_cursor;
```

Loops through the rows in result set:

```
for( row = 0; ; row++ )
{
  EXEC SQL BEGIN DECLARE SECTION;
    int col;
  EXEC SQL END DECLARE SECTION;
```

Then, uses a `FETCH` to retrieve the next row from the cursor into the descriptor:

```
EXEC SQL FETCH IN my_cursor INTO SQL DESCRIPTOR row_desc;
```

The application confirms that the `FETCH` did not fail; if the `FETCH` fails, the application has reached the end of the result set, and breaks the loop:

```
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.

```
if( row == 0 )
   print_meta_data( "row_desc" );
```

Next, it prints a record header containing the row number:

```
printf("[RECORD %d]\n", row+1);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Then, it loops through each column in the 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;
```

The application interrogates the row descriptor `(row_desc)` to copy the column value `(:val)`, null indicator `(:ind)` and column name `(:name)` into the host variables declared above. Notice that you can retrieve multiple items from a descriptor using a comma-separated list.

```
EXEC SQL GET DESCRIPTOR row_desc
  VALUE :col
  :val = DATA, :ind = INDICATOR, :name = NAME;
```

If the null indicator `(ind)` is negative, the column value is `NULL`; if the null indicator is greater than `0`, the column value is too long to fit into the val host variable (so we print `<truncated>)`; otherwise, the null indicator is `0` (meaning `NOT NULL`) so we print the value. In each case, we prefix the value (or `<null>` or `<truncated>`) with the name of the column.

```
if( ind == -1 )
  printf( " %-20s : <null>\n", name.arr );
else if( ind > 0 )
  printf( " %-20s : <truncated>\n", name.arr );
else
  printf( " %-20s : %s\n", name.arr, val.arr );
}

printf( "\n" );
}
```

When the loop terminates, the application prints the number of rows fetched, and exits:

```
   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:

```
static void print_meta_data( char *desc_name )
{
```

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]`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
static char *types[] =
{
  "unused ",
  "CHARACTER ",
  "NUMERIC ",
  "DECIMAL ",
  "INTEGER ",
  "SMALLINT ",
  "FLOAT ",
  "REAL ",
  "DOUBLE ",
  "DATE_TIME ",
  "INTERVAL ",
  "unused ",
  "CHARACTER_VARYING",
  "ENUMERATED ",
  "BIT ",
  "BIT_VARYING ",
  "BOOLEAN ",
  "abstract "
};
```

The application retrieves the column count from the descriptor. Notice that the program refers to the descriptor using a host variable `(desc)` that contains the name of the descriptor. In most scenarios, you would use an identifier to refer to a descriptor, but in this case, the caller provided the descriptor name, so we can use a host variable to refer to the descriptor.

```
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, 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;
```

It retrieves the name, type code, and length of the current column:

```
EXEC SQL GET DESCRIPTOR :desc
 VALUE :col
 :name = NAME,
 :type = TYPE,
 :ret_len = RETURNED_OCTET_LENGTH;
```

If the numeric type code matches a 'known' type code (that is, a type code found in the `types[]` array), it sets `type_name` to the name of the corresponding type; otherwise, it sets `type_name` to `"unknown"`.

```
if( type > 0 && type < SQL3_abstract )
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  type_name = types[type];
else
  type_name = "unknown";
```

and prints the column number, name, type name, and length:

```
    printf( "%02d: %-20s %-17s %04d\n",
      col, name.arr, type_name, ret_len );
  }
  printf( "\n" );
}
```

If you invoke the sample application with the following command:

```
./exec_stmt test "SELECT * FROM emp WHERE empno IN(7902, 7934)"
```

The application returns:

```
 col field                data              ret
num name                  type              len
--- --------------------- ----------------- ---
01: empno                 NUMERIC           0004
02: ename                 CHARACTER_VARYING 0004
03: job                   CHARACTER_VARYING 0007
04: mgr                   NUMERIC           0004
05: hiredate              DATE_TIME         0018
06: sal                   NUMERIC           0007
07: comm                  NUMERIC           0000
08: deptno                NUMERIC           0002

[RECORD 1]
  empno                : 7902
  ename                : FORD
  job                  : ANALYST
  mgr                  : 7566
  hiredate             : 03-DEC-81 00:00:00
  sal                  : 3000.00
  comm                 : <null>
  deptno               : 20

[RECORD 2]
  empno                : 7934
  ename                : MILLER
  job                  : CLERK
  mgr                  : 7782
  hiredate             : 23-JAN-82 00:00:00
  sal                  : 1300.00
  comm                 : <null>
  deptno               : 10

2 rows
```

17.4 Building and Executing Dynamic SQL Statements

The following examples demonstrate four techniques for building and executing dynamic SQL statements. Each example demonstrates processing a

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

different combination of statement and input types:

- The first example demonstrates processing and executing a SQL statement that does not contain a `SELECT` statement and does not require input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 1.
- The second example demonstrates processing and executing a SQL statement that does not 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 demonstrates processing and executing a SQL statement that may 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 demonstrates processing and executing a SQL statement that may contain a `SELECT` statement, and includes an unknown number of input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 4.

Example - Executing a Non-query Statement Without Parameters

The following example demonstrates how to use the `EXECUTE IMMEDIATE` command to execute a SQL statement where the text of the statement is not known until you run the application. You cannot use `EXECUTE IMMEDIATE` to execute a statement that returns a result set. You cannot 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 the next example.

```
/************************************************************/
#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);
}
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
/************************************************************/
```

The code sample begins by including the prototypes and type definitions for the C `stdio, string`, and `stdlib` libraries, and providing basic infrastructure for the program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void handle_error(void);
int main(int argc, char *argv[])
{
  char *insertStmt;
```

The example then sets up an error handler; ECPGPlus calls the `handle_error()` function whenever a SQL error occurs:

```
EXEC SQL WHENEVER SQLERROR DO handle_error();
```

Then, the example connects to the database using the credentials specified on the command line:

```
EXEC SQL CONNECT :argv[1];
```

Next, the program uses an `EXECUTE IMMEDIATE` statement to execute a SQL statement, adding a row to the `dept` table:

```
insertStmt = "INSERT INTO dept VALUES(50, 'ACCTG', 'SEATTLE')";

EXEC SQL EXECUTE IMMEDIATE :insertStmt;
```

If the `EXECUTE IMMEDIATE` command fails for any reason, ECPGPlus will invoke 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.

```
   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.

```
static void handle_error(void)
{
  fprintf(stderr, "%s\n", sqlca.sqlerrm.sqlerrmc);

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK RELEASE;

  exit(EXIT_FAILURE);
}
```

Example - Executing a Non-query Statement with a Specified Number of Placeholders

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

To execute a non-query 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 may execute many times (providing different *values* each time).

ECPGPlus will convert each parameter value to the type required by the SQL statement, if possible; if not possible, ECPGPlus will report an error.

```
/************************************************************/
#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;

  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);
}

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:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[])
{
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  char *stmtText;
```

The example then sets up an error handler; ECPGPlus calls the `handle_error()` function whenever a SQL error occurs:

```
EXEC SQL WHENEVER SQLERROR DO handle_error();
```

Then, the example connects to the database using the credentials specified on the command line:

```
EXEC SQL CONNECT :argv[1];
```

Next, the program uses a `PREPARE` statement to parse and plan a statement that includes three parameter markers - if the `PREPARE` statement succeeds, it will create 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.

```
stmtText = "INSERT INTO dept VALUES(?, ?, ?)";

EXEC SQL PREPARE stmtHandle FROM :stmtText;
```

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:

```
EXEC SQL EXECUTE stmtHandle USING :argv[2], :argv[3], :argv[4];
```

If the `EXECUTE` command fails for any reason, ECPGPlus will invoke the `handle_error()` function (which terminates the application after displaying an error message to the user). If the `EXECUTE` command succeeds, the application displays a message `(ok)` to the user, commits the changes, disconnects from the server, and terminates the application.

```
   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.

```
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 a Known Number of Placeholders

This example demonstrates 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, before opening a cursor and iterating through the result set.

```
/************************************************************/
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
#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, stdbool`, and `sqlca` libraries, and providing basic infrastructure for the program:

```
 #include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
#include <sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[])
{
  VARCHAR empno[10];
  VARCHAR ename[20];
```

The example then sets up an error handler; ECPGPlus calls the `handle_error()` function whenever a SQL error occurs:

```
EXEC SQL WHENEVER SQLERROR DO handle_error();
```

Then, the example connects to the database using the credentials specified on the command line:

```
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 will create 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.

```
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. Notice that the `OPEN` statement includes a `USING` clause: the `USING` clause must provide a *value* for each placeholder found in the query:

```
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:

```
   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.

```
   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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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

The next example demonstrates executing a query with an unknown number of input parameters and/or columns in the result set. This type of query may 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).

```
/************************************************************/
#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);
}
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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) */
       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;

       sqlnul(&results->T[col],
              &results->T[col],
              &null_permitted);

       switch (results->T[col])
       {
          case 2:        /* NUMERIC */
          {
            int precision, scale;

            sqlprc(&results->L[col], &precision, &scale);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
            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;
     }

     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%s", delimiter, "<null>");
          else
            printf("%s%s", delimiter, results->V[col]);
          delimiter = ", ";
       }


 printf("\n");
      }
  }
}
/************************************************************/
```

The code sample begins by including the prototypes and type definitions for the C `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)` will be used to describe the metadata for any parameter markers found in the dynamic query text. The second SQLDA structure `(results)` will contain both the metadata and the result set obtained

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

by executing the dynamic query.

```
SQLDA *params;
SQLDA *results;
```

The program then declares two helper functions (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 `(stmtTxt)` is a NULL-terminated C string containing the text of the query to execute. Notice that the values for these three host variables are derived from the command-line arguments. When the program begins execution, 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 (among other things):

- a pointer to an array of column names
- a pointer to an array of indicator names
- a pointer to an array of data types
- a pointer to an array of lengths
- a pointer to an array of 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 section for a complete description of the `SQLDA` structure.

```
params = sqlald(20, 64, 64);
results = sqlald(20, 64, 64);
```

After allocating the `SELECT` -list and bind descriptors, the program prepares the dynamic statement and declares a cursor over the result set.

```
EXEC SQL PREPARE stmt FROM :stmtText;

EXEC SQL DECLARE dynCursor CURSOR FOR stmt;
```

Next, the program calls the `bindParams()` function. The `bindParams()` function examines the bind descriptor `(params)` and prompt the user for a value to substitute in place of each parameter marker found in the dynamic query.

```
bindParams();
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Finally, the program opens the cursor (using the parameter values supplied by the user, if any) and calls the `displayResultSet()` function to print the result set produced by the query.

```
   EXEC SQL OPEN dynCursor USING DESCRIPTOR params;

  displayResultSet();
}
```

The `bindParams()` function determines whether the dynamic query contains any parameter markers, and, if so, 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 the statement contains no parameter markers, `params->F` will contain 0. If the statement contains more parameters than will fit into the descriptor, `params->F` will contain a negative number (in this case, the absolute value of `params->F` indicates the number of parameter markers found in the statement). If `params->F` contains a positive number, that number indicates how many parameter markers were found in the statement.

```
if (params->F < 0)
   fprintf(stderr, "Too many parameters required\n");
else
{
   int i;

   params->N = params->F;
```

Next, the program executes a loop that prompts the user for a value, iterating once for each parameter marker found in the statement.

```
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);
```

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 (see `Type Codes` for a list of type codes), `params->L[i]` to the length of the value (we subtract one to trim off the trailing new-line character added by `fgets()`), and `params->V[i]` to point to a copy of the NULL-terminated string provided by the user.

```
     params->T[i] = 1;        /* Data type = Character (1) */
    params->L[i] = strlen(paramValue) + 1;
    params->V[i] = strdup(paramValue);
   }
  }
}
```

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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
static void displayResultSet(void)
{
  EXEC SQL DESCRIBE SELECT LIST FOR stmt INTO results;
```

If the dynamic statement returns no columns (that is, the dynamic statement is not a `SELECT` statement), `results->F` will contain 0. If the statement returns more columns than will fit into the descriptor, `results->F` will contain a negative number (in this case, the absolute value of `results->F` indicates the number of columns returned by the statement). If `results->F` contains a positive number, that number indicates how many columns where returned by the query.

```
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;
```

Next, the program enters a loop, iterating once for each column in the result set:

```
for (col = 0; col < results->F; col++)
{
  int null_permitted, length;
```

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 The 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 necessary because the `DESCRIBE SELECT LIST` statement encodes the type of each column and the nullability of each column into the `T` array.

```
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.

```
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 somewhat arbitrary, hard-coded length of 30. In a real-world application, you may want to more carefully compute the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

amount of space required.

```
case 12:     /* DATE */
{
  length = 30;
  break;
}
```

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 may want to include more careful calculations for other data types.

```
  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]`, and set 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.

```
  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.

```
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.

```
EXEC SQL FETCH dynCursor USING DESCRIPTOR results;
```

The `FETCH` statement will populate the following members of the results descriptor:

- `*results->I[col]` will indicate whether the column contains a NULL value `(-1)` or a non-NULL value `(0)`. If the value non-NULL but too large to fit into the space provided, the value is truncated and `*results->I[col]` will contain a positive value.
- `results->V[col]` will contain the value fetched for the given column `(unless *results->I[col]` indicates that the column value is NULL).
- `results->L[col]` will contain 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 is not aligned - instead, each value is separated from the previous value by a comma.

```
    for (col = 0; col < results->F; col++)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    {
       if (*results->I[col] == -1)
         printf("%s%s", delimiter, "<null>");
       else
         printf("%s%s", delimiter, results->V[col]);
       delimiter = ", ";
    }

    printf("\n");
   }
 }
}
/*************************************************************/
```

17.5 Error Handling

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.
- A client application can include `EXEC SQL WHENEVER` directives to instruct the ECPGPlus compiler to add error-handling code.

Error Handling with sqlca

`sqlca` (SQL communications area) is a global variable used by `ecpglib` to communicate information from the server to the client application. After executing a SQL statement (for example, an `INSERT` or `SELECT` statement) you can inspect the contents of `sqlca` to determine if the statement has completed successfully or if the statement has failed.

`sqlca` has the following structure:

```
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[5];

} sqlca;
```

Use the following directive to implement `sqlca` functionality:

```
EXEC SQL INCLUDE sqlca;
```

If you include the `ecpg` directive, you do not need to `#include` the `sqlca.h` file in the client application's header declaration.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The Advanced Server `sqlca` structure contains the following members:

`sqlcaid`

`sqlcaid` contains the string: `"SQLCA"`.

`sqlabc`

`sqlabc` contains the size of the `sqlca` structure.

`sqlcode`

The `sqlcode` member has been deprecated with SQL 92; Advanced Server supports `sqlcode` for backward compatibility, but you should use the `sqlstate` member when writing new code.

`sqlcode` is an integer value; a positive `sqlcode` value indicates that the client application has encountered a harmless processing condition, while a negative value indicates a warning or error.

If a statement processes without error, `sqlcode` will contain a value of `0`. If the client application encounters an error (or warning) during a statement's execution, `sqlcode` will contain the last code returned.

The SQL standard defines only a positive value of 100, which indicates that he most recent SQL statement processed returned/affected no rows. Since the SQL standard does not define other `sqlcode` values, please be aware that the values assigned to each condition may vary from database to database.

`sqlerrm` is a structure embedded within `sqlca`, composed of two members:

`sqlerrml`

`sqlerrml` contains the length of the error message currently stored in `sqlerrmc`.

`sqlerrmc`

`sqlerrmc` contains the null-terminated message text associated with the code stored in `sqlstate`. If a message exceeds 149 characters in length, `ecpglib` will truncate the error message.

`sqlerrp`

`sqlerrp` contains the string `"NOT SET"`.

`sqlerrd` is an array that contains six elements:

- `sqlerrd[1]` contains the OID of the processed row (if applicable).
- `sqlerrd[2]` contains the number of processed or returned rows.
- `sqlerrd[0], sqlerrd[3], sqlerrd[4]` and `sqlerrd[5]` are unused.

`sqlwarn` is an array that contains 8 characters:

- `sqlwarn[0]` contains a value of `'W'` if any other element within `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 non-fatal warning.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `sqlwarn[3], sqlwarn[4], sqlwarn[5], sqlwarn[6]`, and `sqlwarn[7]` are unused.

`sqlstate`

`sqlstate` is a 5 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` will contain a value of `00000`. Please note that `sqlstate` is *not* 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` will contain the last code returned.

The following table lists the `sqlstate` and `sqlcode` values, as well as the symbolic name and error description for the related condition:

sqlstate	sqlcode (Deprecated)	Symbolic Name	Description
`YE001`	`-12`	`ECPG_OUT_OF_MEMORY`	Virtual memory is exhausted.
`YE002`	`-200`	`ECPG_UNSUPPORTED`	The preprocessor has generated an unrecognized item. Could indicate incompatibility between the preprocessor and the library.
`07001`, or `07002`	`-201`	`ECPG_TOO_MANY_ARGUMENTS`	The program specifies more variables than the command expects.
`07001`, or `07002`	`-202`	`ECPG_TOO_FEW_ARGUMENTS`	The program specified fewer variables than the command expects.
`21000`	`-203`	`ECPG_TOO_MANY_MATCHES`	The SQL command has returned multiple rows, but the statement was prepared to receive a single row.
`42804`	`-204`	`ECPG_INT_FORMAT`	The host variable (defined in the C code) is of type INT, and the selected data is of a type that cannot be converted into an INT. `ecpglib` uses the `strtol()` function to convert string values into numeric form.
`42804`	`-205`	`ECPG_UINT_FORMAT`	The host variable (defined in the C code) is an unsigned INT, and the selected data is of a type that cannot be converted into an unsigned INT. `ecpglib` uses the `strtoul()` function to convert string values into numeric form.
`42804`	`-206`	`ECPG_FLOAT_FORMAT`	The host variable (defined in the C code) is of type FLOAT, and the selected data is of a type that cannot be converted into an FLOAT. `ecpglib` uses the `strtod()` function to convert string values into numeric form.
`42804`	`-211`	`ECPG_CONVERT_BOOL`	The host variable (defined in the C code) is of type BOOL, and the selected data cannot be stored in a BOOL.
`YE002`	`-2-1`	`ECPG_EMPTY`	The statement sent to the server was empty.
`22002`	`-213`	`ECPG_MISSING_INDICATOR`	A NULL indicator variable has not been supplied for the NULL value returned by the server (the client application has received an unexpected NULL value).
`42804`	`-214`	`ECPG_NO_ARRAY`	The server has returned an array, and the corresponding host variable is not capable of storing an array.
`42804`	`-215`	`ECPG_DATA_NOT_ARRAY`	The server has returned a value that is not an array into a host variable that expects an array value.
`08003`	`-220`	`ECPG_NO_CONN`	The client application has attempted to use a non-existent connection.
`YE002`	`-221`	`ECPG_NOT_CONN`	The client application has attempted to use an allocated, but closed connection.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

sqlstate	sqlcode (Deprecated)	Symbolic Name	Description
26000	-230	ECPG_INVALID_STMT	The statement has not been prepared.
33000	-240	ECPG_UNKNOWN_DESCRIPTOR	The specified descriptor is not found.
07009	-241	ECPG_INVALID_DESCRIPTOR_INDEX	The descriptor index is out-of-range.
YE002	-242	ECPG_UNKNOWN_DESCRIPTOR_ITEM	The client application has requested an invalid descriptor item (internal error).
07006	-243	ECPG_VAR_NOT_NUMERIC	A dynamic statement has returned a numeric value for a non-numeric host variable.
07006	-244	ECPG_VAR_NOT_CHAR	A dynamic SQL statement has returned a CHAR value, and the host variable is not a CHAR.
	-400	ECPG_PGSQL	The server has returned an error message; the resulting message contains the error text.
08007	-401	ECPG_TRANS	The server cannot start, commit or rollback the specified transaction.
08001	-402	ECPG_CONNECT	The client application's attempt to connect to the database has failed.
02000	100	ECPG_NOT_FOUND	The last command retrieved or processed no rows, or you have reached the end of a cursor.

EXEC SQL WHENEVER

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 that it should perform a specified action if the client application detects a given condition. The*condition* may 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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` will generate a `CONTINUE` statement in the emitted C code that if it encounters the condition, skips the rest of the code in the loop and continues with the next iteration. You can only use it within 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;
EXEC SQL WHENEVER SQLERROR STOP;
```

> **Note**
>
> The ECPGPlus compiler processes your program from top to bottom, even though the client application may 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 within your program is not top-to-bottom, you should consider adding error-handling directives to any parts of the program that may be inadvertently missed during compilation.

17.6 Reference

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The sections that follow describe ecpgPlus language elements:

- C-Preprocessor Directives
- Supported C Data Types
- Type Codes
- The SQLDA Structure
- ECPGPlus Statements

C-preprocessor Directives

The ECPGPlus C-preprocessor enforces two behaviors that are dependent 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 it is of a data type 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:

```
#include
#if expression
#ifdef symbolName
#ifndef symbolName
#else
#elif expression
#endif
#define symbolName expansion
#define symbolName([macro arguments]) expansion
#undef symbolName
#defined(symbolName)
```

Pre-processor directives are used to effect or direct the code that is received by the compiler. For example, using the following code sample:

```
#if HAVE_LONG_LONG == 1
#define BALANCE_TYPE long long
#else
#define BALANCE_TYPE double
#endif
...
BALANCE_TYPE customerBalance;
```

If you invoke ECPGPlus with the following command-line arguments:

```
ecpg -C PROC -DHAVE_LONG_LONG=1
```

ECPGPlus will copy the entire fragment (without change) to the output file, but will only send the following tokens to the ECPG parser:

```
long long customerBalance;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

On the other hand, if you invoke ECPGPlus with the following command-line arguments:

```
ecpg -C PROC -DHAVE_LONG_LONG=0
```

The ECPG parser will receive the following tokens:

```
double customerBalance;
```

If your code uses preprocessor directives to filter the code that is 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:

```
EXEC ORACLE DEFINE
EXEC ORACLE UNDEF
EXEC ORACLE INCLUDE
EXEC ORACLE IFDEF
EXEC ORACLE IFNDEF
EXEC ORACLE ELIF
EXEC ORACLE ELSE
EXEC ORACLE ENDIF
EXEC ORACLE OPTION
```

For example, if your code includes the following:

```
EXEC ORACLE IFDEF HAVE_LONG_LONG;
#define BALANCE_TYPE long long
EXEC ORACLE ENDIF;
BALANCE_TYPE customerBalance;
```

If you invoke ECPGPlus with the following command-line arguments:

```
ecpg -C PROC DEFINE=HAVE_LONG_LONG=1
```

ECPGPlus will send the following tokens to the output file, and the ECPG parser:

```
long long customerBalance;
```

> **Note**
>
> The `EXEC ORACLE` pre-processor directives only work 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 instruct your program how to handle result sets that contain more rows than the host variable can accommodate. The syntax is:

```
SELECT_ERROR={YES|NO}
```

The default value is `YES`; a `SELECT` statement will return an error message if the result set exceeds the capacity of the host variable. Specify `NO` to instruct the program 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Compiling in non-PROC Mode

If you do not 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 within an `EXEC SQL BEGIN/END DECLARE` section.

When invoked in non-`PROC` mode, ECPG implements the behavior described in the PostgreSQL Core documentation.

Supported C Data Types

An ECPGPlus application must deal with two sets of data types: SQL data types (such as `SMALLINT`, `DOUBLE PRECISION` and `CHARACTER VARYING`) and C data types (like `short`, `double` and `varchar[n]`). When an application fetches data from the server, ECPGPlus will map 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 will try to convert a SQL character value into a C integer value, but the conversion may 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 will try to convert the C data type into the required SQL type. Again, the conversion may fail (at execution time) if the C value cannot be converted into the required SQL type.

ECPGPlus can convert any SQL type into C character values `(char[n]` or `varchar[n])`. Although it is safe to convert any SQL type to/from `char[n]` or `varchar[n]`, it is often convenient to use more natural C types such as `int`, `double`, or `float`.

The supported C data types are:

- `short`
- `int`
- `unsigned int`
- `long long int`
- `float`
- `double`
- `char[n+1]`
- `varchar[n+1]`
- `bool`
- and any equivalent created by a `typedef`

In addition to the numeric and character types supported by C, the `pgtypeslib` run-time library offers custom data types (and functions to operate on those types) for dealing with date/time and exact numeric values:

- `timestamp`
- `interval`
- `date`
- `decimal`
- `numeric`

To use a data type supplied by `pgtypeslib`, you must `#include` the proper header file.

Type Codes

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 `SELECT`-list item, the type code in the corresponding SQLDA descriptor `(descriptor->T[column])` should be set to one of the following values:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Type Code	Host Variable Type (C Data Type)
1, 2, 8, 11, 12, 15, 23, 24, 91, 94, 95, 96, 97	char[]
3	int
4, 7, 21	float
5, 6	null-terminated string (char[length+1])
9	varchar
22	double
68	unsigned int

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.

Internal Type Code	Server Type
1	VARCHAR2
2	NUMBER
8	LONG
11	ROWID
12	DATE
23	RAW
24	LONG RAW
96	CHAR
100	BINARY FLOAT
101	BINARY DOUBLE
104	UROWID
187	TIMESTAMP
188	TIMESTAMP W/TIMEZONE
189	INTERVAL YEAR TO MONTH
190	INTERVAL DAY TO SECOND
232	TIMESTAMP LOCAL_TZ

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. The layout of the SQLDA structure is:

```
struct SQLDA
{
  int     N; /* Number of entries             */
  char  **V; /* Variables                  */
  int    *L; /* Variable lengths           */
  short  *T; /* Variable types             */
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  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 may 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. Refer to displayResultSet () function for an example of how to allocate space for `SELECT` -list values.

`L - length of each data value`

The `L` structure member is a pointer 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]` should 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 may 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 may 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:

```
void sqlnul(unsigned short *valType,
            unsigned short *typeCode,
            int            *isNull)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

For example, to find the type code and nullable flag for the third column of a descriptor named results, you would invoke `sqlnul()` as follows:

```
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 which may be described by the descriptor; `F` indicates the actual number of values). The value of the `F` member is set by ECPGPlus when you execute a `DESCRIBE` statement. `F` may be positive, negative, or zero.

- For a `SELECT` -list descriptor, `F` will contain 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); 0 if the statement is *not* a `SELECT` statement, or a negative value if the query returns more columns than allowed by the `N` structure member.
- For a bind descriptor, `F` will contain 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); 0 if the statement contains no parameters markers, or 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 may 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `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 is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

`Y - maximum indicator name length`

The `Y` structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the `X` array (that is, `Y[0]` specifies the maximum length of the indicator name found at `X[0]`).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

`Z - actual indicator name length`

The `Z` structure member points to an array of lengths. Each member in this array specifies the *actual* length of the corresponding member of the `X` array (that is, `Z[0]` specifies the actual length of the indicator name found at `X[0]`).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

ECPGPlus Statements

An embedded SQL statement allows your client application to interact with the server, while an embedded directive is an instruction to the ECPGPlus compiler.

You can embed any Advanced Server SQL statement in a C program. Each statement should begin with the keywords `EXEC SQL`, and must be terminated with a semi-colon (;). Within the C program, a SQL statement takes the form:

```
EXEC SQL <sql_command_body>;
```

Where `sql_command_body` represents a standard SQL statement. You can use a host variable anywhere that the SQL statement expects a value expression. For more information about substituting host variables for value expressions, refer to Declaring Host Variables.

ECPGPlus extends the PostgreSQL server-side syntax for some statements; for those statements, syntax differences are outlined in the following reference sections. For a complete reference to the supported syntax of other SQL commands, refer to the *PostgreSQL Core Documentation* available at:

https://www.postgresql.org/docs/11/static/sql-commands.html

ALLOCATE DESCRIPTOR

Use the `ALLOCATE DESCRIPTOR` statement to allocate an SQL descriptor area:

```
EXEC SQL [FOR <array_size>] ALLOCATE DESCRIPTOR <descriptor_name>
    [WITH MAX <variable_count>];
```

Where:

`array_size` is a variable that specifies the number of array elements to allocate for the descriptor. `array_size` may 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 may take the form of an identifier, a quoted string literal, or of a host variable.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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 may be processed as an array `(emp_array)`:

```
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 Advanced Server. The `CALL` statement comes in two forms; the first form is used to call a *function*:

```
EXEC SQL CALL <program_name> '('[<actual_arguments>]')'
  INTO [[:<ret_variable>][: <ret_indicator>]];
```

The second form is used to call a *procedure*:

```
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 may be schema-qualified or package-qualified (or both); if you do not specify the schema or package in which the program resides, ECPGPlus will use the value of `search_path` to locate the program.

`actual_arguments` specifies a comma-separated list of arguments required by the program. Note that each `actual_argument` corresponds to a formal argument expected by the program. Each formal argument may be an `IN` parameter, an `OUT` parameter, or an `INOUT` parameter.

`:ret_variable` specifies a host variable that will receive the value returned if the program is a function.

`:ret_indicator` specifies a host variable that will receive 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:

```
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:

```
EXEC SQL CLOSE [<cursor_name>];
```

Where:

`cursor_name` is the name of the cursor closed by the statement. The cursor name may take the form of an identifier or of a host variable.

The `OPEN` statement initializes a cursor. Once initialized, a cursor result set will remain unchanged unless the cursor is re-opened. You do not need to `CLOSE` a cursor before re-opening it.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

To manually close a cursor named `emp_cursor`, use the command:

```
EXEC SQL CLOSE emp_cursor;
```

A cursor is automatically closed when an application terminates.

COMMIT

Use the `COMMIT` statement to complete the current transaction, making all changes permanent and visible to other users. The syntax is:

```
EXEC SQL [AT <database_name>] COMMIT [WORK]
         [COMMENT <'text'>] [COMMENT <'text'> RELEASE];
```

Where:

`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 may take the form of an unquoted string literal, or of a host variable.

For compatibility, ECPGPlus accepts the `COMMENT` clause without error but does *not* store any text included with the `COMMENT` clause.

Include the `RELEASE` clause to close the current connection after performing the commit.

For example, the following command commits all work performed on the `dept` database and closes the current connection:

```
EXEC SQL AT dept COMMIT RELEASE;
```

By default, statements are committed only when a client application performs a `COMMIT` statement. Include the `-t` option when invoking ECPGPlus to specify that a client application should invoke `AUTOCOMMIT` functionality. You can also control `AUTOCOMMIT` functionality in a client application with the following statements:

```
EXEC SQL SET AUTOCOMMIT TO ON
```

and

```
EXEC SQL SET AUTOCOMMIT TO OFF
```

CONNECT

Use the `CONNECT` statement to establish a connection to a database. The `CONNECT` statement is available in two forms - one form is compatible with Oracle databases, the other is not.

The first form is compatible with Oracle databases:

```
EXEC SQL CONNECT
  {{:<user_name> IDENTIFIED BY :<password>}  | :<connection_id>}
  [AT <database_name>]
  [USING :database_string]
  [ALTER AUTHORIZATION :new_password];
```

Where:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`user_name` is a host variable that contains the role that the client application will use to connect to the server.

`password` is a host variable that contains the password associated with that role.

`connection_id` is a host variable that contains a slash-delimited user name and password used to connect to the database.

Include the `AT` clause to specify the database to which the connection is established. `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 `USING` clause to specify a host variable that contains a null-terminated string identifying the database to which the connection will be established.

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 could 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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, do not include a `connection_name` or `credentials`.

`connection_name` is the name of the connection to the database. `connection_name` should take 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 do not 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 do not 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 Advanced Server database. The privileges of the specified role will be 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 may 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 may take the form of an identifier or of a host variable. If you do

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

not 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 cannot 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 is 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 demonstrates declaring an identifier for the `acctg` database:

```
EXEC SQL DECLARE acctg DATABASE;
```

After invoking the command declaring `acctg` as a database identifier, the `acctg` database can be referenced 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. 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>;
```

Copyright © 2009 - 2024 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 might be:

```
 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> ]
```

Where:

Include the `FOR exec_count` clause to specify the number of times the statement will execute; 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 do not 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 which rows should be deleted. If you do not include a `WHERE` clause in the statement, `DELETE` will delete 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 will be deleted.

`cursor_name` is the name of the cursor to use in the `WHERE CURRENT OF` clause; the row to be deleted will be the one most recently fetched from this cursor. The cursor must be a non-grouping query on the `DELETE` statements target table. You cannot 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 is 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `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 available at:

https://www.postgresql.org/docs/11/static/sql-delete.html

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>;
```

or

```
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 DESCRIBE BIND VARIABLES
  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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

When you execute the second form, ECPG populates the given descriptor with a description of each value *returned* by the statement. For example, the following statement returns three values:

```
EXEC SQL DESCRIBE SELECT LIST
  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 alternate 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 DESCRIBE [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 do not 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

For example, the following statement returns three values:

```
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:

```
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 do not specify a connection name, the current connection is closed.

Include the `CURRENT` keyword to specify that ECPGPlus should close the most-recently used connection.

Include the `DEFAULT` keyword to specify that ECPGPlus should close the connection named `DEFAULT`. If you do not specify a name when opening a connection, ECPGPlus assigns the name, `DEFAULT`, to the connection.

Include the `ALL` keyword to instruct ECPGPlus 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:

```
 /* 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:

```
EXEC SQL [FOR <array_size>] EXECUTE <statement_name>
  [USING {DESCRIPTOR <SQLDA_descriptor>
  |: <host_variable> [[INDICATOR] :<indicator_variable>]}];
```

Where:

`array_size` is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. 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 within the prepared statement:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- 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:

```
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 8000, 'DAWSON', 'CLERK', 7788;
EXEC SQL EXECUTE add_emp USING 8001, '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:

```
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 be processed. 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.

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 will write the results returned by the prepared 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.

The following example executes the prepared statement, `give_raise`, using the values contained in the descriptor `stmtText:`

```
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:

```
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 will be executed on the current default database.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`anonymous_block` is an inline sequence of PL/pgSQL or SPL statements and declarations. You may include host variables and optional indicator variables within the block; each such variable is treated as an `IN/OUT` value.

The following example executes an anonymous block:

```
EXEC SQL EXECUTE
  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 Advanced Server.

EXECUTE IMMEDIATE

Use the `EXECUTE IMMEDIATE` statement to execute a string that contains a SQL command. The syntax is:

```
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 will be executed 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 (ie., when writing a client application). For example, a client application may 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 may not contain references to host variables. If the statement may contain parameter markers or returns one or more values, you must use the `PREPARE` and `DESCRIBE` statements.

The following example executes the command contained in the `:command_text` host variable:

```
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:

```
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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 will populate the specified SQLDA descriptor with the values returned by the server.

If you include an `INTO` clause, the `FETCH` statement will populate 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:

```
EXEC SQL DECLARE employees CURSOR FOR
    SELECT empno, ename, esal FROM emp;
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor 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:

```
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. The client must `DECLARE` and `OPEN` the cursor before calling the `FETCH DESCRIPTOR` statement.

Include the `INTO` clause to specify an SQL descriptor into which the `EXECUTE` statement will write the results returned by the prepared 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. Prior to use, the descriptor must be allocated using an `ALLOCATE DESCRIPTOR` statement.

The following example allocates a descriptor named `row_desc` that will hold 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:

```
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';
```

GET DESCRIPTOR

Use the `GET DESCRIPTOR` statement to retrieve information from a descriptor. The `GET DESCRIPTOR` statement comes in two forms. The first form

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

returns the number of values (or columns) in the descriptor.

```
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).

```
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 be processed. 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.

Include the `VALUE` clause to specify the information retrieved from the descriptor.

- `column_number` identifies the position of the variable within the descriptor.
- `host_variable` specifies the name of the host variable that will receive 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 demonstrates using a `GET DESCRIPTOR` statement to obtain the number of columns entered in a user-provided string:

```
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;
```

The example allocates an SQL descriptor (named `parse_desc`), before using a `PREPARE` statement to syntax check 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 any place 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>] [, ...]]
```

Where:

Include the `FOR exec_count` clause to specify the number of times the statement will execute; 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 may 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 will be assigned to the corresponding column. Specify `DEFAULT` to fill the corresponding column with its default value.

`query` specifies a `SELECT` statement that supplies the row(s) to be inserted.

`output_expression` is an expression that will be computed and returned by the `INSERT` command after each row is inserted. The expression can refer to any column within the table. Specify * to return all columns of the inserted row(s).

`output_name` specifies a name to use for a returned column.

The following example adds a row to the `employees` table:

```
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 available at:

https://www.postgresql.org/docs/11/static/sql-insert.html

OPEN

Use the `OPEN` statement to open a cursor. The syntax is:

```
EXEC SQL [FOR <array_size>] OPEN <cursor> [USING <parameters>];
```

Where `parameters` is one of the following:

```
DESCRIPTOR <SQLDA_descriptor>
```

or

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
<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 will remain unchanged unless the cursor is closed and re-opened. A cursor is automatically closed when an application terminates.

The following example declares a cursor named `employees`, that queries the `emp` table, returning the `employee number`, `name`, `salary` and `commission` of an employee whose name matches a user-supplied value (stored in the host variable, `:emp_name`).

```
   EXEC SQL DECLARE employees CURSOR FOR
    SELECT
      empno, ename, sal, comm
    FROM
      emp
    WHERE ename = :emp_name;
  EXEC SQL OPEN employees;
...
```

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:

```
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`:

```
EXEC SQL OPEN emp_cursor USING DESCRIPTOR :employees;
```

PREPARE

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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, saving repetitive processing time.

Use the `PREPARE` statement to prepare an SQL statement or PL/pgSQL block for execution. The statement is available in two forms; the first form is:

```
EXEC SQL [AT <database_name>] PREPARE <statement_name>
  FROM <sql_statement>;
```

The second form is:

```
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 will execute. If you omit the `AT` clause, the statement will execute against the current default database.

`statement_name` is the identifier associated with a prepared SQL statement or PL/SQL block.

`sql_statement` may take the form of a `SELECT` statement, a single-quoted string literal or host variable that contains the text of an SQL statement.

To include variables within a prepared statement, substitute placeholders (`$1, $2, $3` , etc.) for statement values that might change when you `PREPARE` the statement. When you `EXECUTE` the statement, provide a value for each parameter. The values must be provided in the order in which they will replace placeholders.

The following example creates a prepared statement (named `add_emp`) that inserts a record into the `emp` table:

```
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:

```
EXEC SQL EXECUTE add_emp(8003, 'Davis', 'CLERK', 2000.00);
EXEC SQL EXECUTE add_emp(8004, 'Myer', 'CLERK', 2000.00);
```

> **Note**
>
> A client application must issue a `PREPARE` statement within each session in which a statement will be executed; 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:

```
EXEC SQL [AT <database_name>] ROLLBACK [WORK]
  [ { TO [SAVEPOINT] <savepoint> } | RELEASE ]
```

Where:

`database_name` is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the `AT` clause, the statement will execute against the current default database.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Include the `TO` clause to abort any commands that were executed after the specified `savepoint`; use the `SAVEPOINT` statement to define the `savepoint`. If you omit the `TO` clause, the `ROLLBACK` statement will abort 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 rollback a complete transaction:

```
EXEC SQL ROLLBACK;
```

Invoking this statement will abort the transaction, undoing all changes, erasing any savepoints, and releasing all transaction locks. If you include a savepoint (`my_savepoint` in the following example):

```
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` will be 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 within 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:

```
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 will execute against the current default database.

`savepoint_name` is the name of the savepoint. If you re-use a `savepoint_name`, the original savepoint is discarded.

Savepoints can only be established within a transaction block. A transaction block may contain multiple savepoints.

To create a savepoint named `my_savepoint`, include the statement:

```
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 Advanced Server database into a host variable. The syntax for the `SELECT` statement is:

```
 EXEC SQL [AT <database_name>]
SELECT
  [ <hint> ]
  [ ALL | DISTINCT [ ON( <expression>, ...) ]]
  select_list INTO <host_variables>

  [ FROM from_item [, from_item ]...]
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  [ WHERE condition ]
  [ hierarchical_query_clause ]
  [ GROUP BY expression [, ...]]
  [ HAVING condition ]
  [ { UNION [ ALL ] | INTERSECT | MINUS } (subquery) ]
  [ ORDER BY expression [order_by_options]]
  [ LIMIT { count | ALL }]
  [ OFFSET start [ ROW | ROWS ] ]
  [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
  [ FOR { UPDATE | SHARE } [OF table_name [, ...]][NOWAIT ][...]]
```

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 may take the form of an unquoted string literal, or of a host variable.

`host_variables` is a list of host variables that will be 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 available at:

https://www.postgresql.org/docs/11/static/sql-select.html

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`:

```
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 would be permitted. For example, the following query contains a parameter marker in the `WHERE` clause:

```
SELECT * FROM emp WHERE dept_no = ?;
```

This `SELECT` statement allows you to provide a value at run-time for the `dept_no` parameter marker.

SET CONNECTION

There are (at least) three reasons you may need more than one connection in a given client application:

- You may want different privileges for different statements;
- You may need to interact with multiple databases within the same client.
- Multiple threads of execution (within a client application) cannot share a connection concurrently.

The syntax for the `SET CONNECTION` statement is:

```
EXEC SQL SET CONNECTION <connection_name>;
```

Where:

`connection_name` is the name of the connection to the database.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

To use the `SET CONNECTION` statement, you should 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 only used when you execute an `EXEC SQL` statement that does not explicitly specify a connection name. For example, the following statement will use the default connection because it does not include an `AT connection_name` clause. :

```
EXEC SQL DELETE FROM emp;
```

This statement will not 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, a client application that creates and maintains multiple connections (such as):

```
EXEC SQL CONNECT TO edb AS acctg_conn
  USER 'alice' IDENTIFIED BY 'acctpwd';
```

and

```
EXEC SQL CONNECT TO edb AS hr_conn
  USER 'bob' IDENTIFIED BY 'hrpwd';
```

Can change between the connections with the `SET CONNECTION` statement:

```
SET CONNECTION acctg_conn;
```

or

```
SET CONNECTION hr_conn;
```

The server will use the privileges associated with the connection when determining the privileges available to the connecting client. When using the `acctg_conn` connection, the client will have the privileges associated with the role, `alice`; when connected using `hr_conn`, the client will have 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

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.

```
EXEC SQL ALLOCATE DESCRIPTOR query_params;
EXEC SQL PREPARE emp_query FROM :query_text;

EXEC SQL DESCRIBE INPUT emp_query
  USING SQL DESCRIPTOR 'query_params';
```

After describing the query, the `query_params` descriptor contains information about each parameter required by the query.

For this example, we'll assume that the user has entered:

```
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:

```
EXEC SQL GET DESCRIPTOR … :host_variable = COUNT;
```

Then, you can use `EXEC SQL GET DESCRIPTOR` to retrieve the name of each parameter. You can also use `EXEC SQL GET DESCRIPTOR` to retrieve the type of each parameter (along with the number of parameters) from the descriptor, or you can supply each `value` in the form of a character string and ECPG will convert 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

to find the number of parameters in the query.

```
EXEC SQL GET DESCRIPTOR 'query_params'
  :param_count = COUNT;

for(param_number = 1;
    param_number <= param_count;
    param_number++)
{
```

Use `GET DESCRIPTOR` to copy the name of the parameter into the `param_name` host variable:

```
EXEC SQL GET DESCRIPTOR 'query_params'
  VALUE :param_number :param_name = NAME;

reply = promptUser(param_name);
if (reply == NULL)
 reply_ind = 1; /* NULL */
else
 reply_ind = 0; /* NOT NULL */
```

To associate a `value` with each parameter, you use the `EXEC SQL SET DESCRIPTOR` statement. For example:

```
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:

```
EXEC SQL [AT <database_name>][FOR <exec_count>]
    UPDATE [ ONLY ] table [ [ AS ] alias ]
    SET {column = { expression | DEFAULT } |
        (column [, ...]) = ({ expression|DEFAULT } [, ...])} [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [[ AS ] output_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 may 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 will execute; 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 available at:

https://www.postgresql.org/docs/11/static/sql-update.html

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

A host variable can be used in any clause that specifies a value. To use a host variable, simply 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, and increases the employees salary, by multiplying the current salary by the value in the `:increase` host variable:

```
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 would be permitted. For example, we can write the same update statement with a parameter marker in the `WHERE` clause:

```
EXEC SQL UPDATE emp
  SET job = ?, sal = sal * ?
  WHERE ename = :ename;
```

This `UPDATE` statement could allow 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:

```
EXEC SQL WHENEVER <condition> <action>;
```

The following table describes the different conditions that might trigger an `action`:

Condition	Description
`NOT FOUND`	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.
`SQLERROR`	The server returns an `SQLERROR` condition when it encounters a serious error returned by an SQL statement.
`SQLWARNING`	The server returns an `SQLWARNING` condition when it encounters a non-fatal warning returned by an SQL statement.

The following table describes the actions that result from a client encountering a `condition`:

Action	Description
`CALL function [([args])]`	Instructs the client application to call the named `function`.
`CONTINUE`	Instructs the client application to proceed to the next statement.
`DO BREAK`	Instructs the client application to a C break statement. A break statement may appear in a `loop` or a `switch` statement. If executed, the break statement terminate the `loop` or the `switch` statement.
`DO CONTINUE`	Instructs the client application to emit a C `continue` statement. A `continue` statement may only exist within a loop, and if executed, will cause the flow of control to return to the top of the loop.
`DO function ([args])`	Instructs the client application to call the named `function`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Action	Description
`GOTO label` or `GO TO label`	Instructs the client application to proceed to the statement that contains the `label`.
`SQLPRINT`	Instructs the client application to print a message to standard error.
`STOP`	Instructs the client application to stop execution.

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;
EXEC SQL WHENEVER SQLERROR STOP;
```

Include the following code to specify that a client should continue processing after warning a user of a problem:

```
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:

```
EXEC SQL WHENEVER NOT FOUND CALL error_handler(__LINE__);
```

18 EDB Postgres Language Pack Guide

This guide provides information about how to install and configure Language Pack, as well as how to use the procedural languages (PL/Perl, PL/Python, and PL/Tcl).

Language pack installers contain supported languages that may be used with EDB Postgres Advanced Server and EnterpriseDB PostgreSQL database installers. The language pack installer allows you to install Perl, Tcl/TK, and Python without installing supporting software from third-party vendors.

The Language Pack installer includes:

- Tcl with TK version 8.6.8
- Perl version 5.26.2
- Python version 3.6.8

The Perl package contains the `cpan` package manager, and Python contains `pip` and `easy_install` package managers. There is no package manager for Tcl/TK.

In previous Postgres releases, `plpython` was statically linked with ActiveState's python library. The Language Pack Installer dynamically links with our shared object for python. In ActiveState Linux installers for Python, there is no dynamic library. As a result of these changes, `plpython` will no longer work with ActiveState installers.

Convention Used in this Guide

The term *Postgres* refers to either PostgreSQL or EDB Postgres Advanced Server.

18.1 Supported Database Server Versions

Language Pack installers are version specific. For EDB Postgres Advanced Server 11, use Language Pack Version 1.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

18.2 Installing and Configuring Language Pack

This section walks you through installing and configuring Language Pack.

Installing Language Pack

The graphical installer is available from the EDB website, as well as via Stack Builder and StackBuilder Plus. StackBuilder Plus is distributed with EDB Postgres Advanced Server and Stack Builder is distributed with PostgreSQL.

Invoking the Graphical Installer

Assume Administrator privileges, and double-click the installer icon; if prompted, provide the password associated with the Administrator account. When prompted, select an installation language, and click `OK`.

The Language Pack setup wizard welcome window opens.

Fig. 1: The Language Pack Welcome Window

Click `Next` to continue.

The Ready to Install window displays the Language Pack installation directory:

On Windows 64: `C:/edb/languagepack-11\x64`

On OSX: `/Library/edb/languagepack-11`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

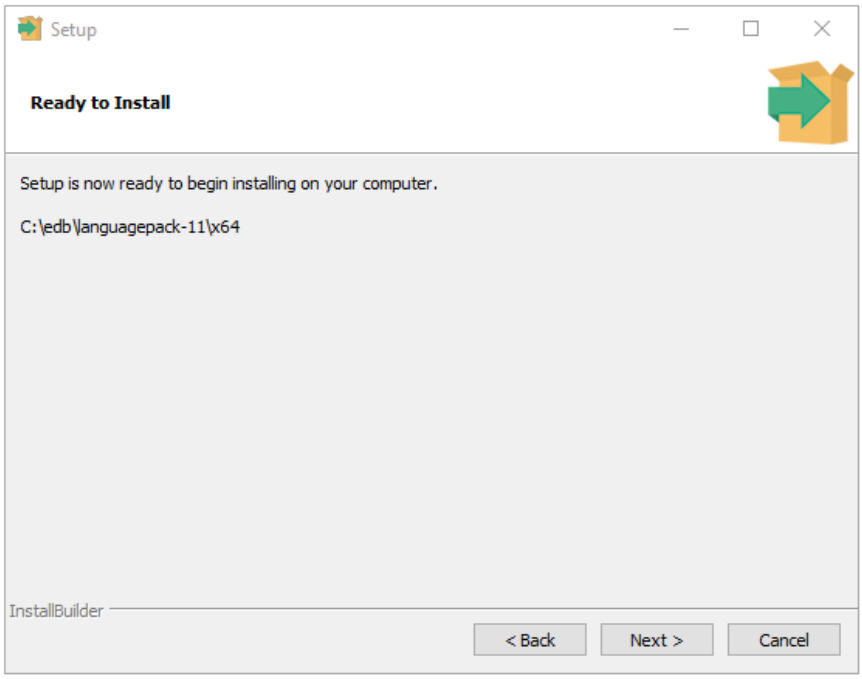


Fig. 2: The Ready to Install dialog

You cannot modify the installation directory. Click `Next` to continue.

A progress bar marks installation progress.

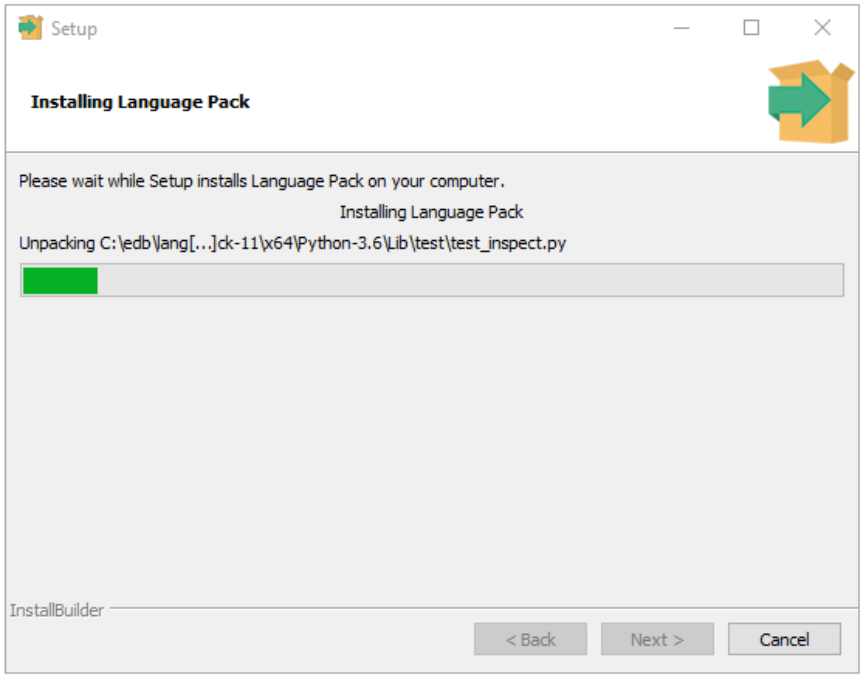


Fig. 3: The Installing dialog

Click `Next` to continue.

The installer will inform you that the Language Pack installation has completed; click `Finish` to exit the installer.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

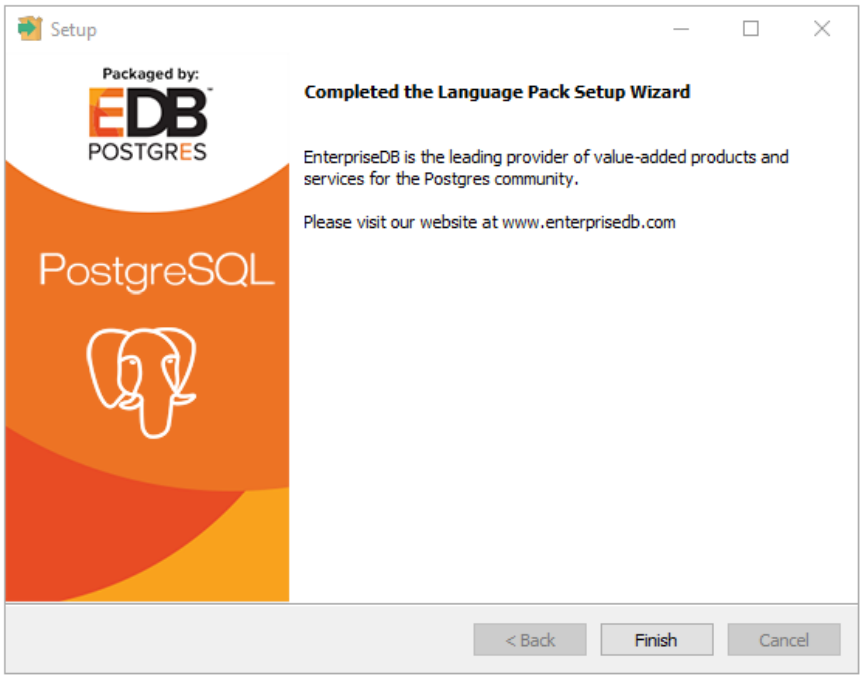


Fig. 4: The Language Pack Setup Complete dialog

Installing Language Pack with Stack Builder

You can use Stack Builder or StackBuilder Plus to download and invoke the Language Pack graphical installer. StackBuilder Plus is distributed with EDB Postgres Advanced Server and Stack Builder is distributed with PostgreSQL.

The following section walks you through installing Language Pack with Stack Builder.

The Stack Builder utility provides a graphical interface that simplifies the process of downloading and installing modules that complement your PostgreSQL installation.

Stack Builder requires Internet access; if your installation of PostgreSQL resides behind a firewall (with restricted Internet access), Stack Builder can download program installers through a proxy server. The module provider deter-mines if the module can be accessed through an HTTP proxy or an FTP proxy; currently, all updates are transferred via an HTTP proxy and the FTP proxy information is not used.

You can invoke Stack Builder at any time after the installation has completed by selecting the `Application Stack Builder` menu option from the `PostgreSQL 11` menu.

Select your server from the drop-down menu on the Stack Builder Welcome window and click `Next` to continue.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

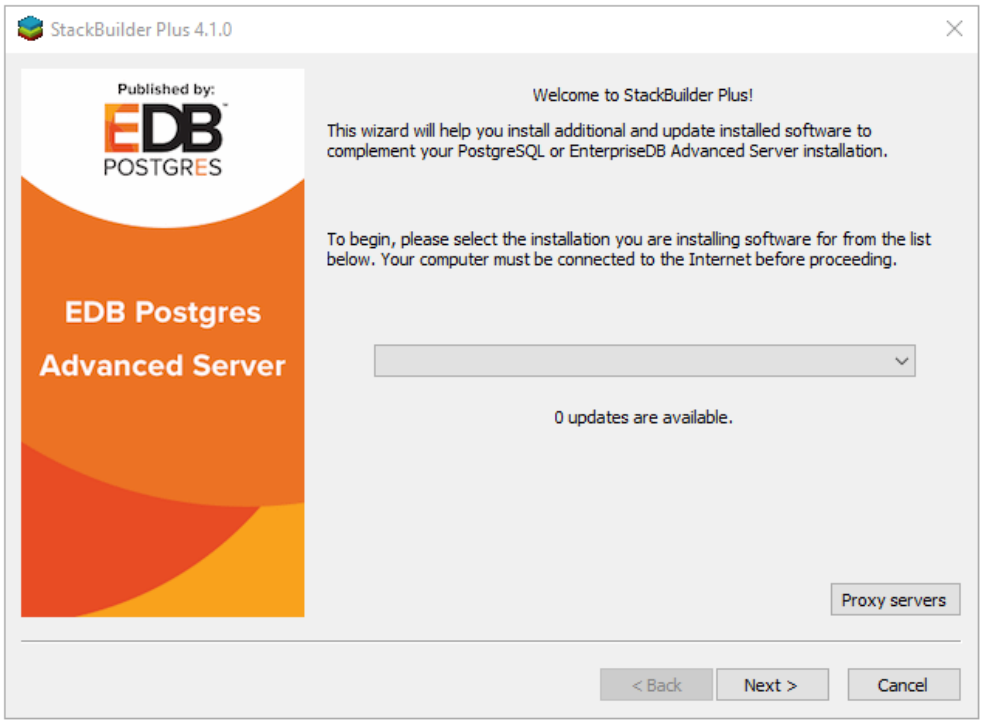


Fig. 5: The Stack Builder Welcome Window

Expand the `Add-ons, tools and utilities` node of the `Categories` tree control, and check the box next to EDB Language Pack version. Click `Next` to continue.

Stack Builder will confirm your package selection before downloading the installer. Click `Next` to continue.

When the download completes, Stack Builder will offer to invoke the installer for you, or will delay the installation until a more convenient time. To invoke the installer, click `Next` and follow the steps provided in the Invoking the Graphical Installer section.

Configuring Language Pack

This section walks you through configuring Language Pack on an Advanced Server and PostgreSQL hosts.

Configuring Language Pack on an Advanced Server Host

After installing Language Pack on an Advanced Server host, you must configure the installation.

Configuring Language Pack on Windows

On Windows, the Language Pack installer places the languages in:

```
C:\edb\languagepack-11\x64
```

After installing Language Pack, you must set the following variables:

```
set PYTHONHOME=C:\edb\languagepack-11\x64\Python-3.6
```

Use the following commands to add Python, Perl and Tcl to your search path:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
set PATH=C:\edb\languagepack-11\x64\Python-3.6;
C:\edb\languagepack-11\x64\Perl-5.26\bin;
C:\edb\languagepack-11\x64\Tcl-8.6\bin;%PATH%
```

After performing the steps required to configure Language Pack on Windows, use the Windows `Services` applet to restart the Advanced Server.

Configuring Language Pack on a PostgreSQL Host

After installing Language Pack on a PostgreSQL host, you must configure the installation.

Configuring Language Pack on Windows

After installing Language Pack, you must set the following variables:

```
set PYTHONHOME=C:\edb\languagepack\v1\Python-3.7
```

Then, use the following commands to add Language Pack to your search path:

```
set PATH=C:\edb\languagepack-11\x64\Python-3.6;
C:\edb\languagepack\-11\x64\Perl-5.26\bin;
C:\edb\languagepack\-11\x64\Tcl-8.6\bin;%PATH%
```

After setting the system-specific steps required to configure Language Pack on Windows, restart the database server.

Configuring Language Pack on OSX

To simplify setting the value of `PATH` or `LD_LIBRARY_PATH`, you can create environment variables that identify the installation location:

```
PERLHOME=/Library/edb/languagepack-11/Perl-5.26
PYTHONHOME=/Library/edb/languagepack-11/Python-3.6
TCLHOME=/Library/edb/languagepack-11/Tcl-8.6
```

Then, execute the following command to instruct the Python interpreter where to find Python:

```
export PYTHONHOME
```

You can use the same environment variables when setting the value of `PATH`:

```
export PATH=$PYTHONHOME/bin:
$PERLHOME/bin:
$TCLHOME/bin:$PATH
```

Lastly, set the following variables to instruct OSX where to find the shared libraries:

```
export DYLD_LIBRARY_PATH=$PYTHONHOME/lib:
$PERLHOME/lib/CORE:$TCLHOME/lib:
$DYLD_LIBRARY_PATH
```

18.3 Using the Procedural Languages

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The Postgres procedural languages (PL/Perl, PL/Python, and PL/Tcl) are installed by the Language Pack installer. You can also use a native package to add procedural language functionality to your EDB Postgres Advanced Server installation.

PL/Perl

The PL/Perl procedural language allows you to use Perl functions in Postgres applications.

You must install PL/Perl in each database (or in a template database) before creating a PL/Perl function. Use the `CREATE LANGUAGE` command at the EDB-PSQL command line to install PL/Perl. Open the EDB-PSQL client, establish a connection to the database in which you wish to install PL/Perl, and enter the command:

```
CREATE EXTENSION plperl;
```

You can now use a Postgres client application to access the features of the PL/Perl language. The following PL/Perl example creates a function named `perl_max` that returns the larger of two integer values:

```
 CREATE OR REPLACE FUNCTION perl_max (integer, integer) RETURNS integer AS
$$
if ($_[0] > $_[1])
{ return $_[0]; }
return $_[1];
$$ LANGUAGE plperl;
```

Pass two values when calling the function:

```
SELECT perl_max(1, 2);
```

The server returns:

```
 perl_max
---------
       2
(1 row)
```

For more information about using the Perl procedural language, consult the official PostgreSQL documentation.

PL/Python

The PL/Python procedural language allows you to create and execute functions written in Python within Postgres applications. The version of PL/Python used by EDB Postgres Advanced Server and PostgreSQL is untrusted (`plpython3u`); it offers no restrictions on users to prevent potential security risks.

Install PL/Python in each database (or in a template database) before creating a PL/Python function. You can use the `CREATE LANGUAGE` command at the EDB-PSQL command line to install PL/Python. Use EDB-PSQL to connect to the database in which you wish to install PL/Python, and enter the command:

```
CREATE EXTENSION plpython3u;
```

After installing PL/Python in your database, you can use the features of the PL/Python language.

> **Note**
>
> The indentation shown in the following example must be included as you enter the sample function in EDB-PSQL.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following PL/Python example creates a function named `pymax` that returns the larger of two integer values:

```
CREATE OR REPLACE FUNCTION pymax (a integer, b integer) RETURNS integer AS
$$
if a > b:
return a
return b
$$ LANGUAGE plpython3u;
```

When calling the `pymax` function, pass two values as shown below:

```
SELECT pymax(12,
3);
```

The server returns:

```
pymax
-------
    12
(1 row)
```

For more information about using the Python procedural language, consult the official PostgreSQL documentation.

PL/Tcl

The PL/Tcl procedural language allows you to use Tcl/Tk functions in applications.

You must install PL/Tcl in each database (or in a template database) before creating a PL/Tcl function. Use the `CREATE LANGUAGE` command at the EDB-PSQL command line to install PL/Tcl. Use the psql client to connect to the database in which you wish to install PL/Tcl, and enter the command:

```
CREATE EXTENSION pltcl;
```

After creating the `pltcl` language, you can use the features of the PL/Tcl language from within your Postgres server.

The following PL/Tcl example creates a function named `tcl_max` that returns the larger of two integer values:

```
CREATE OR REPLACE FUNCTION tcl_max(integer, integer) RETURNS integer AS
$$
if {[argisnull 1]} {
if {[argisnull 2]} { return_null }
return $2
}
if {[argisnull 2]} { return $1 }
if {$1 > $2} {return $1}
return $2
$$ LANGUAGE pltcl;
```

Pass two values when calling the function:

```
SELECT tcl_max(1,
2);
```

The server returns:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 tcl_max
--------
       2
(1 row)
```

For more information about using the Tcl procedural language, consult the official PostgreSQL documentation.

18.4 Uninstalling Language Pack

The following section outlines the process of uninstalling Language Pack.

The Language Pack graphical installer creates an uninstaller that you can use to remove Language Pack. The uninstaller is created in the installation directory.

Perform the following steps to uninstall Language Pack:

1. Navigate into the directory that contains the uninstaller and assume superuser privileges. Open the uninstaller and click `Yes` to begin uninstalling Language Pack.

2. The uninstallation process begins. Click `OK` when the uninstallation completes.

19 EDB Postgres Advanced Server Security Features Guide

This guide describes features that provide added security to EDB Postgres Advanced Server installations. It is not a comprehensive guide to the security functionality provided by PostgreSQL that is built into Advanced Server.

- SQL/Protect provides protection from SQL injection attacks.
- Virtual Private Database provides fine-grained access control for sensitive data.
- sslutils is a Postgres extension that allows you to generate SSL certificates.
- Data redaction functionality allows you to dynamically mask portions of data.

For information about Postgres authentication and security features, consult the PostgreSQL core documentation.

19.1 Protecting Against SQL Injection Attacks

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 is a module that allows a database administrator to protect a database from SQL injection attacks. SQL/Protect provides a layer of security in addition to the normal database security policies by examining incoming queries for common SQL injection profiles.

SQL/Protect gives the control back to the database administrator by alerting the administrator to potentially dangerous queries and by blocking these queries.

19.1.1 SQL/Protect Overview

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

This section contains an introduction to the different types of SQL injection attacks and describes how SQL/Protect guards against them.

Types of SQL Injection Attacks

There are a number of different techniques 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 Advanced Server allows administrators to restrict access to relations (tables, views, etc.), many administrators do not perform this tedious task. SQL/Protect provides a *learn* mode that tracks the relations a user accesses.

This allows administrators to examine the workload of an application, and for SQL/Protect to learn which relations an application should be allowed to access for a given user or group of users in a role.

When SQL/Protect is switched to either *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 has the ability to access other system resources.

SQL/Protect can block the running of all utility commands, which are not 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 will 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 the running of unbounded DML statements. These are `UPDATE` and `DELETE` statements with no `WHERE` clause. For example, an attacker may 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

This section describes how SQL/Protect monitors and reports 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 Advanced Server user or group that the database administrator has chosen to monitor using SQL/Protect. (In Advanced Server, users and groups are collectively referred to as *roles*.)

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Each protected role can be customized for the types of SQL injection attacks for which it is to be monitored, thus providing different levels of protection by role and significantly reducing the user maintenance load for DBAs.

A role with the superuser privilege cannot be made a protected role. If a protected non-superuser role is subsequently altered to become a superuser, certain behaviors are exhibited whenever an attempt is made by that superuser to issue any command:

- A warning message is issued by SQL/Protect on every command issued by the protected superuser.
- The statistic in column superusers of `edb_sql_protect_stats` is incremented with every command issued by the protected superuser. See *Attack Attempt Statistics* for information on the `edb_sql_protect_stats` view.
- When SQL/Protect is in active mode, all commands issued by the protected superuser are prevented from running.

A protected role that has the superuser privilege should either be altered so that it is no longer a superuser, or it should be reverted back to an unprotected role.

Attack Attempt Statistics

Each usage of a command by a protected role that is considered an attack by SQL/Protect is recorded. Statistics are collected by type of SQL injection attack as discussed in *Types of SQL Injection Attacks*.

These statistics are accessible from view `edb_sql_protect_stats` that can be easily monitored 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 on protected superusers.
- **relations.** Number of SQL statements issued referencing relations that were not learned by a protected role. (That is, relations that are not in a role's protected relations list.)
- **commands.** Number of DDL 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 did not contain a `WHERE` clause.

This gives database administrators the opportunity 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 Advanced Server home directory.

Attack Attempt Queries

Each usage of a command by a protected role that is considered an attack by SQL/Protect is recorded in the `edb_sql_protect_queries` view.

The `edb_sql_protect_queries` view 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, if known, from which the attack originated.
- **date_time.** Date and time at which the query was received by the database server. The time is stored to the precision of a minute.
- **query.** The query string sent by the attacker.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 and are viewable only when connected to the respective database.

19.1.2 Configuring SQL/Protect

Ensure the following prerequisites are met before configuring SQL/Protect:

- The library file (`sqlprotect.so` on Linux, `sqlprotect.dll` on Windows) necessary to run `SQL/Protect` should be installed in the `lib` subdirectory of your Advanced Server home directory. For Windows, this should be done by the Advanced Server installer. For Linux, install the `edb-asxx-server-sqlprotect` RPM package where `xx` is the Advanced Server version number.
- You will also need the SQL script file `sqlprotect.sql` located in the `share/contrib` subdirectory of your 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:
    - The database server configuration file, `postgresql.conf` , must be modified by adding and enabling configuration parameters used by `SQL/Protect` .
    - Database objects used by `SQL/Protect` must be installed in each database that you want `SQL/Protect` to monitor.

Step 1: Edit the following configuration parameters in the `postgresql.conf` file located in the `data` subdirectory of your Advanced Server home directory.

- **shared_preload_libraries.** Add `$libdir/sqlprotect` to the list of libraries.
- **edb_sql_protect.enabled.** Controls whether or not `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 are ready to begin monitoring with `SQL/Protect` set this parameter to `on` . If this parameter is omitted, the default is `off` .
- **edb_sql_protect.level.** Sets the action taken by `SQL/Protect` when a SQL statement is issued by a protected role. If this parameter is omitted, the default behavior is `passive` . Initially, set this parameter to `learn` .

  See Setting the Protection Level for more information.
- **edb_sql_protect.max_protected_roles.** Sets the maximum number of roles that can be protected. If this parameter is omitted, the default setting is `64` .
- **edb_sql_protect.max_protected_relations.** Sets the maximum number of relations that can be protected per role. If this parameter is omitted, the default setting is `1024` .

  Please note that the total number of protected relations for the server will be 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. If this parameter is omitted, the default setting is `5000` . If the number of offending queries reaches the limit, additional queries are not saved in the view, but are accessible in the database server log file.

  Please note that the minimum valid value for this parameter is `100` . If a value less than `100` is specified, the database server starts using the default setting of `5000` . A warning message is recorded in the database server log file.

The following example shows the settings of these parameters in the `postgresql.conf` file:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/sqlprotect'
                                        # (change requires restart)
                          .
                          .
                          .
edb_sql_protect.enabled = off
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
```

Step 2: Restart the database server after you have modified the `postgresql.conf` file.

On Linux: Invoke the Advanced Server service script with the `restart` option.

On a Redhat or CentOS 7.x installation, use the command:

```
systemctl restart edb-as-11
```

On Windows: Use the Windows Services applet to restart the service named `edb-as-11`.

Step 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 upon your installation options) and run the script `sqlprotect.sql` located in the `share/contrib` subdirectory of your Advanced Server home directory. The script creates the SQL/Protect database objects in a schema named `sqlprotect`.

The following example shows this process to set up protection for a database named `edb`:

```
$ /usr/edb/as11/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (11.0.2, server 11.0.2)
Type "help" for help.

edb=# \i /usr/edb/as11/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
DO
CREATE FUNCTION
CREATE FUNCTION
DO
CREATE VIEW
GRANT
DO
CREATE VIEW
GRANT
CREATE VIEW
GRANT
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
CREATE FUNCTION
CREATE FUNCTION
SET
```

Selecting Roles to Protect

After the SQL/Protect database objects have been created in a database, you can select the roles for which SQL queries are to be monitored for protection, and the level of protection that will be assigned 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.

Step 1: Connect as a superuser to a database that you wish to protect with either `psql` or Postgres Enterprise Manager Client:

```
$ /usr/edb/as11/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (11.0.2, server 11.0.2)
Type "help" for help.

edb=#
```

Step 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. This eliminates the need to schema-qualify any operation or query involving SQL/Protect database objects:

```
edb=# SET search_path TO sqlprotect;
SET
```

Step 3: Each role that you wish to protect must be added 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')`. The following example protects a role named `appuser`:

```
edb=# SELECT protect_role('appuser');
 protect_role
--------------

(1 row)
```

You can list the roles that have been added to the protected roles list by issuing 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):

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 edb=# \x
Expanded display is on.
edb=# SELECT * FROM list_protected_users;
-[ RECORD 1 ]------+--------
dbname             | edb
username           | appuser
protect_relations  | t
allow_utility_cmds | f
allow_tautology    | f
allow_empty_dml    | f
```

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.

The `edb_sql_protect.level` configuration parameter (in the `postgresql.conf` file) can be set to one of the following values to use either `learn` mode, `passive` mode, or `active` mode:

- **learn.** Tracks the activities of protected roles and records the relations used by the roles. This is used 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 does not stop any SQL statements from executing. This is the next step after SQL/Protect has learned the expected behavior of the protected roles. This essentially behaves in intrusion detection mode and can be run in production when properly monitored.
- **active.** Stops all invalid statements for a protected role. This behaves as a SQL firewall preventing 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, but it tracks the blocked queries allowing administrators to be alerted before the attacker finds an alternate method of penetrating the system.

If the `edb_sql_protect.level` parameter is not set or is omitted from the configuration file, the default behavior of `SQL/Protect` is `passive`.

If you are using `SQL/Protect` for the first time, set `edb_sql_protect.level` to `learn`.

Monitoring Protected Roles

Once you have configured SQL/Protect in a database, added roles to the protected roles list, and set the desired protection level, you can then activate SQL/Protect in either `learn` mode, `passive` mode, 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 should be permitted 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 SQL/Protect is run in passive or active mode. In passive and active modes, the role is permitted to access the relations in its protected relations list as these were determined to be the relations the role should be able to access during typical usage.

However, if a role attempts to access a relation that is not in its protected relations list, a `WARNING` or `ERROR` severity level message is returned by SQL/Protect. The role's attempted action on the relation may or may not be carried out depending upon whether the mode is passive or active.

Learn Mode

Step 1: To activate SQL/Protect in learn mode, set the parameters in the `postgresql.conf` file as shown below:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb_sql_protect.enabled = on
edb_sql_protect.level = learn
```

Step 2: Reload the `postgresql.conf` file.

Choose `Expert Configuration`, then `Reload Configuration` from the Advanced Server application menu.

For an alternative method of reloading the configuration file, use the `pg_reload_conf` function. Be sure you are connected to a database as a superuser and execute `function pg_reload_conf` as shown by the following example:

```
edb=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

Step 3: Allow the protected roles to run their applications.

As an example the following queries are issued in the `psql` application by protected role `appuser`:

```
edb=> SELECT * FROM dept;
NOTICE:  SQLPROTECT: Learned relation: 16384
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)

edb=> SELECT empno, ename, job FROM emp WHERE deptno = 10;
NOTICE:  SQLPROTECT: Learned relation: 16391
 empno | ename  |    job
-------+--------+-----------
  7782 | CLARK  | MANAGER
  7839 | KING   | PRESIDENT
  7934 | MILLER | CLERK
(3 rows)
```

SQL/Protect generates a `NOTICE` severity level message indicating the relation has been added to the role's protected relations list.

In SQL/Protect learn mode, SQL statements that are cause for suspicion are not prevented from executing, but a message is issued to alert the user to potentially dangerous statements as shown by the following example:

```
edb=> CREATE TABLE appuser_tab (f1 INTEGER);
NOTICE: SQLPROTECT: This command type is illegal for this user
CREATE TABLE
edb=> DELETE FROM appuser_tab;
NOTICE: SQLPROTECT: Learned relation: 16672
NOTICE: SQLPROTECT: Illegal Query: empty DML
DELETE 0
```

Step 4: As a protected role runs applications, the SQL/Protect tables can be queried to observe the addition of relations to the role's protected relations list.

Connect as a superuser to the database you are monitoring and set the search path to include the `sqlprotect` schema:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# SET search_path TO sqlprotect;
SET
```

Query the `edb_sql_protect_rel` table to see the relations added to the protected relations list:

```
edb=# SELECT * FROM edb_sql_protect_rel;
 dbid   | roleid | relid
--------+--------+-------
 13917  | 16671  | 16384
 13917  | 16671  | 16391
 13917  | 16671  | 16672
(3 rows)
```

The `list_protected_rels` view provides more comprehensive information along with the object names instead of the OIDs:

```
edb=# SELECT * FROM list_protected_rels;
 Database | Protected User | Schema |    Name     | Type  |    Owner
----------+----------------+--------+-------------+-------+------------
 edb      | appuser        | public | dept        | Table | enterprisedb
 edb      | appuser        | public | emp         | Table | enterprisedb
 edb      | appuser        | public | appuser_tab | Table | appuser
(3 rows)
```

Passive Mode

Once you have determined that a role's applications have accessed all relations they will need, you can now change the protection level so that SQL/Protect can actively monitor the incoming SQL queries and protect against SQL injection attacks.

Passive mode is the less restrictive of the two protection modes, passive and active.

Step 1: To activate `SQL/Protect` in passive mode, set the following parameters in the `postgresql.conf` file as shown below:

```
edb_sql_protect.enabled = on
edb_sql_protect.level = passive
```

Step 2: Reload the configuration file as shown in Step 2 of the Learn Mode section.

Now SQL/Protect is in passive mode. For relations that have been learned such as the `dept` and `emp` tables of the prior examples, SQL statements are permitted with no special notification to the client by `SQL/Protect` as shown by the following queries run by user `appuser`:

```
edb=> SELECT * FROM dept;
 deptno |   dname    |   loc
--------+------------+----------
    10  | ACCOUNTING | NEW YORK
    20  | RESEARCH   | DALLAS
    30  | SALES      | CHICAGO
    40  | OPERATIONS | BOSTON
(4 rows)

edb=> SELECT empno, ename, job FROM emp WHERE deptno = 10;
 empno | ename  |    job
-------+--------+-----------
  7782 | CLARK  | MANAGER
  7839 | KING   | PRESIDENT
  7934 | MILLER | CLERK
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(3 rows)
```

SQL/Protect does not prevent any SQL statement from executing, but issues a message of `WARNING` severity level for SQL statements executed against relations that were not learned, or for SQL statements that contain a prohibited signature as shown in the following example:

```
edb=> CREATE TABLE appuser_tab_2 (f1 INTEGER);
WARNING: SQLPROTECT: This command type is illegal for this user
CREATE TABLE
edb=> INSERT INTO appuser_tab_2 VALUES (1);
WARNING: SQLPROTECT: Illegal Query: relations
INSERT 0 1
edb=> INSERT INTO appuser_tab_2 VALUES (2);
WARNING: SQLPROTECT: Illegal Query: relations
INSERT 0 1
edb=> SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
WARNING: SQLPROTECT: Illegal Query: relations
WARNING: SQLPROTECT: Illegal Query: tautology
f1
----
  1
  2
(2 rows)
```

Step 3: Monitor the statistics for suspicious activity.

By querying the view `edb_sql_protect_stats`, you can see the number of times SQL statements were executed that referenced relations that were not in a role's protected relations list, or contained SQL injection attack signatures. See *Attack Attempt Statistics* for more information on view `edb_sql_protect_stats`.

The following is a query on `edb_sql_protect_stats`:

```
edb=# SET search_path TO sqlprotect;
SET
edb=# SELECT * FROM edb_sql_protect_stats;
 username  | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
 appuser   |     0      |     3     |    1     |     1     | 0
(1 row)
```

Step 4: View information on specific attacks.

By querying the `edb_sql_protect_queries` view, you can see the SQL statements that were executed that referenced relations that were not in a role's protected relations list, or contained SQL injection attack signatures. See *Attack Attempt Queries* for more information on view `edb_sql_protect_queries`.

The following code sample shows a query on `edb_sql_protect_queries`:

```
edb=# SELECT * FROM edb_sql_protect_queries;
-[ RECORD 1 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (1);
-[ RECORD 2 ]+---------------------------------------------
 username     | appuser
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | CREATE TABLE appuser_tab_2 (f1 INTEGER);
-[ RECORD 3 ]+-----------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (2);
-[ RECORD 4 ]+-----------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

> **Note**
>
> The `ip_address` and `port` columns do not 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`).

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`.

Step 1: To activate `SQL/Protect` in active mode, set the following parameters in the `postgresql.conf` file as shown below:

```
 edb_sql_protect.enabled = on
edb_sql_protect.level = active
```

Step 2: Reload the configuration file as shown in Step 2 of the Learn Mode section.

The following example illustrates SQL statements similar to those given in the examples of Step 2 in `Passive Mode`, but executed by user `appuser` when `edb_sql_protect.level` is set to `active`:

```
 edb=> CREATE TABLE appuser_tab_3 (f1 INTEGER);
ERROR: SQLPROTECT: This command type is illegal for this user
edb=> INSERT INTO appuser_tab_2 VALUES (1);
ERROR: SQLPROTECT: Illegal Query: relations
edb=> SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
ERROR: SQLPROTECT: Illegal Query: relations
```

The following shows the resulting statistics:

```
 edb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
 username  | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
 appuser   |      0     |     5     |     2    |     1     | 0
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following is a query on `edb_sql_protect_queries`:

```
edb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
-[ RECORD 1 ]+-------------------------------------------
username     | appuser
ip_address   |
port         |
machine_name |
date_time    | 20-JUN-14 13:21:00 -04:00
query        | CREATE TABLE appuser_tab_2 (f1 INTEGER);
-[ RECORD 2 ]+-------------------------------------------
username     | appuser
ip_address   |
port         |
machine_name |
date_time    | 20-JUN-14 13:22:00 -04:00
query        | INSERT INTO appuser_tab_2 VALUES (2);
-[ RECORD 3 ]+-------------------------------------------
username     | appuser
ip_address   | 192.168.2.6
port         | 50098
machine_name |
date_time    | 20-JUN-14 13:39:00 -04:00
query        | CREATE TABLE appuser_tab_3 (f1 INTEGER);
-[ RECORD 4 ]+-------------------------------------------
username     | appuser
ip_address   | 192.168.2.6
port         | 50098
machine_name |
date_time    | 20-JUN-14 13:39:00 -04:00
query        | INSERT INTO appuser_tab_2 VALUES (1);
-[ RECORD 5 ]+-------------------------------------------
username     | appuser
ip_address   | 192.168.2.6
port         | 50098
machine_name |
date_time    | 20-JUN-14 13:39:00 -04:00
query        | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

19.1.3 Common Maintenance Operations

The following describes how to perform other common operations.

You must be connected as a superuser to perform these operations and have included the `sqlprotect` schema in your search path.

Adding a Role to the Protected Roles List

To add a role to the protected roles list run `protect_role('rolename')` as shown in the following example:

```
edb=# SELECT protect_role('newuser');
 protect_role
--------------
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.

The statistics for a role that has been removed are not deleted until you use the drop_stats function.

The offending queries for a role that has been removed are not deleted until you use the drop_queries function.

The following is an example of the `unprotect_role` function:

```
edb=# SELECT unprotect_role('newuser');
 unprotect_role
----------------
(1 row)
```

Alternatively, the role could be removed by giving its OID of `16693`:

```
edb=# SELECT unprotect_role(16693);
 unprotect_role
----------------
(1 row)
```

Setting the Types of Protection for a Role

You can change whether or not 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 protection of a role is to be disabled or enabled.

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 are making the change
- **roleid.** OID of the role for which you are changing the Boolean settings

For example, to allow a given role to issue utility commands, update the `allow_utility_cmds` column as follows:

```
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`:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# SELECT dbid, roleid, allow_utility_cmds FROM edb_sql_protect;
 dbid   | roleid | allow_utility_cmds
--------+--------+--------------------
 13917  | 16671  | t
(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 has learned that a given relation is accessible for a given role, you can subsequently remove that relation from the role's protected relations list.

Delete its 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` is not 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.

The following example illustrates the removal of the `public.emp` relation from the protected relations list of the role `appuser`:

```
edb=# SELECT unprotect_rel('appuser', 'public', 'emp');
 unprotect_rel
---------------
(1 row)
```

The following query shows there is no longer an entry for the `emp` relation:

```
edb=# SELECT * FROM list_protected_rels;
 Database | Protected User | Schema |    Name     | Type  |    Owner
----------+----------------+--------+-------------+-------+--------------
 edb      | appuser        | public | dept        | Table | enterprisedb
 edb      | appuser        | public | appuser_tab | Table | appuser
(2 rows)
```

SQL/Protect will now issue a warning or completely block access (depending upon the setting of `edb_sql_protect.level`) whenever the role attempts to utilize that relation.

Deleting Statistics

You can delete statistics from view `edb_sql_protect_stats` using either of the two following functions:

```
drop_stats('rolename')

drop_stats(roleoid)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 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`.

The following is an example of 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
-----------+------------+-----------+----------+-----------+-----
(0 rows)
```

The following is an example of using the `drop_stats(roleoid)` form of the function when a role is dropped before deleting its statistics:

```
edb=# SELECT * FROM edb_sql_protect_stats;
      username       | superusers | relations | commands | tautology | dml
---------------------+------------+-----------+----------+-----------+-----
 unknown (OID=16693) |          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 view `edb_sql_protect_queries` using either of the two 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`.

The following is an example of the `drop_queries` function:

```
edb=# SELECT drop_queries('appuser');
 drop_queries
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
--------------
           5
(1 row)

edb=# SELECT * FROM edb_sql_protect_queries;
 username  | ip_address | port | machine_name | date_time | query
-----------+------------+------+--------------+-----------+-------
(0 rows)
```

The following is an example of using the `drop_queries(roleoid)` form of the function when a role is dropped before deleting its queries:

```
edb=# SELECT username, query FROM edb_sql_protect_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';
(5 rows)

edb=# SELECT drop_queries(16454);
 drop_queries
--------------
           5
(1 row)

edb=# SELECT * FROM edb_sql_protect_queries;
 username | ip_address | port | machine_name | date_time | query
----------+------------+------+--------------+-----------+-------
(0 rows)
```

Disabling and Enabling Monitoring

If you wish 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 wish to turn on SQL/Protect monitoring, modify the `postgresql.conf` file, setting the `edb_sql_protect.enabled` parameter to `on`. After saving the file, reload the server configuration to apply the settings.

19.1.4 Backing Up and Restoring a SQL/Protect Database

Backing up a database that is configured with SQL/Protect, and then restoring the backup file to a new database requires additional considerations to what is normally associated with backup and restore procedures. This is primarily due to the use of Object Identification numbers (OIDs) in the SQL/Protect tables as explained in this section.

> **Note**
>
> This section is applicable if your backup and restore procedures result in the re-creation of database objects in the new database with new OIDs such as is the case when using the `pg_dump` backup program.

If you are backing up your Advanced Server database server by simply using the operating system's copy utility to create a binary image of the Advanced Server data files (file system backup method), then this section does not apply.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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, and not the objects' text names. The OID is a numeric data type used by Advanced Server to uniquely identify each database object.

When a database object is created, Advanced Server assigns an OID to the object, which is then used whenever a reference is needed to the object 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 the re-creation of the backed up database objects, the restored objects end up with different OIDs in the new database than 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 simply dumped to a backup file and then restored to a new database.

The following sections describe two functions, `export_sqlprotect` and `import_sqlprotect` , that are used specifically for backing up and restoring SQL/Protect tables in order to ensure the OIDs in the SQL/Protect tables reference the correct database objects after the tables are restored.

Backing Up the Database

The following steps back up a database that has been configured with SQL/Protect.

Step 1: Create a backup file using `pg_dump` .

The following example shows a plain-text backup file named `/tmp/edb.dmp` created from database `edb` using the `pg_dump` utility program:

```
 $ cd /usr/edb/as11/bin
$ ./pg_dump -U enterprisedb -Fp -f /tmp/edb.dmp edb
Password:
$
```

Step 2: Connect to the database as a superuser and export the SQL/Protect data using the `export_sqlprotect('sqlprotect_file')` function (where `sqlprotect_file` is the fully qualified path to a file where the `SQL/Protect` data is to be saved).

The `enterprisedb` operating system account (`postgres` if you installed 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

Step 1: Restore the backup file to the new database.

The following example uses the `psql` utility program to restore the plain-text backup file `/tmp/edb.dmp` to a newly created database named `newdb` :

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
$ /usr/edb/as11/bin/psql -d newdb -U enterprisedb -f /tmp/edb.dmp
Password for user enterprisedb:
SET
SET
SET
SET
SET
COMMENT
CREATE SCHEMA
    .
    .
    .
```

Step 2: Connect to the new database as a superuser and delete all rows from the `edb_sql_protect_rel` table.

This step removes any existing rows in the `edb_sql_protect_rel` table that were backed up from the original database. These rows do not contain the correct OIDs relative to the database where the backup file has been restored:

```
$ /usr/edb/as11/bin/psql -d newdb -U enterprisedb
Password for user enterprisedb:
psql.bin (11.0.2, server 11.0.2)
Type "help" for help.

newdb=# DELETE FROM sqlprotect.edb_sql_protect_rel;
DELETE 2
```

Step 3: Delete all rows from the `edb_sql_protect` table.

This step removes any existing rows in the `edb_sql_protect` table that were backed up from the original database. These rows do not contain the correct OIDs relative to the database where the backup file has been restored:

```
newdb=# DELETE FROM sqlprotect.edb_sql_protect;
DELETE 1
```

Step 4: Delete any statistics that may exist for the database.

This step removes any existing statistics that may exist for the database to which you are restoring the backup. The following query displays any existing statistics:

```
newdb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
 username  | superusers | relations | commands | tautology | dml
-----------+------------+-----------+----------+-----------+-----
(0 rows)
```

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:

```
newdb=# SELECT sqlprotect.drop_stats('appuser');
 drop_stats
------------
(1 row)
```

Step 5: Delete any offending queries that may exist for the database.

This step removes any existing queries that may exist for the database to which you are restoring the backup. The following query displays any existing

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

queries:

```
edb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
 username  | ip_address | port | machine_name | date_time | query
-----------+------------+------+--------------+-----------+-------
(0 rows)
```

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:

```
edb=# SELECT sqlprotect.drop_queries('appuser');
 drop_queries
--------------
(1 row)
```

Step 6: Make sure the role names that were protected by SQL/Protect in the original database exist in the database server where the new database resides.

If the original and new databases reside in the same database server, then nothing needs to be done assuming you have not deleted any of these roles from the database server.

Step 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*.

```
newdb=# SELECT sqlprotect.import_sqlprotect('/tmp/sqlprotect.dmp');
 import_sqlprotect
-------------------
(1 row)
```

Tables `edb_sql_protect` and `edb_sql_protect_rel` are now populated with entries containing the OIDs of the database objects as assigned in the new database. The statistics view `edb_sql_protect_stats` also now displays the statistics imported from the original database.

The SQL/Protect tables and statistics are now properly restored for this database. This is verified by the following queries on the Advanced Server system catalogs:

```
newdb=# SELECT datname, oid FROM pg_database;
  datname  |  oid
-----------+-------
 template1 |     1
 template0 | 13909
 edb       | 13917
 newdb     | 16679
(4 rows)

newdb=# SELECT rolname, oid FROM pg_roles;
   rolname    |  oid
--------------+-------
 enterprisedb |    10
 appuser      | 16671
 newuser      | 16678
(3 rows)

newdb=# SELECT relname, oid FROM pg_class WHERE relname IN
('dept','emp','appuser_tab');
   relname   |  oid
-------------+-------
 appuser_tab | 16803
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 dept        | 16809
 emp         | 16812
(3 rows)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect;
 dbid  | roleid | protect_relations | allow_utility_cmds | allow_tautology |
 allow_empty_dml
-------+--------+-------------------+--------------------+-----------------+--
---------------
 16679 |  16671 | t                 | t                  | f               | f
(1 row)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect_rel;
 dbid  | roleid | relid
-------+--------+-------
 16679 |  16671 | 16809
 16679 |  16671 | 16803
(2 rows)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
 username | superusers | relations | commands | tautology | dml
----------+------------+-----------+----------+-----------+-----
 appuser  |          0 |         5 |        2 |         1 |   0
(1 row)

newedb=# \x
Expanded display is on.
nwedb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
-[ RECORD 1 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | CREATE TABLE appuser_tab_2 (f1 INTEGER);
-[ RECORD 2 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (2);
-[ RECORD 3 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | CREATE TABLE appuser_tab_3 (f1 INTEGER);
-[ RECORD 4 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (1);
-[ RECORD 5 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
date_time    | 20-JUN-14 13:39:00 -04:00
query        | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

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 note that 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`.

Step 8: Verify that the SQL/Protect 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.

19.2 Virtual Private Database

Virtual Private Database is a type of fine-grained access control using security policies. *Fine-grained access control* means that access to data can be controlled 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 value. The *security policy* is the named association of the policy function to a particular database object, typically a table.

In Advanced Server, the policy function can be written in any language supported by Advanced Server such as SQL and PL/pgSQL in addition to SPL.

> **Note**
>
> The database objects currently supported by Advanced Server Virtual Private Database are tables. Policies cannot be applied to views or synonyms.

The advantages of using Virtual Private Database are the following:

- 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 instance.
- A different security policy can be applied depending upon 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 upon factors such as the session user of the application accessing the database object.
- Invocation of the security policy is transparent to all applications that access the database object and thus, individual applications do not have to be modified to apply the security policy.
- Once a security policy is enabled, it is not possible for any application (including new applications) to circumvent the security policy except by the system privilege noted by the following.
- Even superusers cannot circumvent the security policy except by the system privilege noted by the following.

> **Note**
>
> The only way security policies can be circumvented is if the `EXEMPT ACCESS POLICY` system privilege has been granted to a user. The `EXEMPT ACCESS POLICY` privilege should be granted 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.

19.3 sslutils

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`sslutils` is a Postgres extension that provides SSL certificate generation functions to Advanced Server for use by the EDB Postgres Enterprise Manager server. `sslutils` is installed by using the `edb-asxx-server-sslutils` RPM package where `xx` is the Advanced Server version number.

The `sslutils` package provides the functions shown in the following sections.

In these sections, each parameter in the function's parameter list is described by `parameter n` under the **Parameters** subsection where `n` refers to the `nth` ordinal position (for example, first, second, third, etc.) within the function's parameter list.

openssl_rsa_generate_key

The `openssl_rsa_generate_key` function generates an RSA private key. The function signature is:

```
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:

```
openssl_rsa_key_to_csr(<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.

`parameter 2`

The common name (e.g., `agentN`) of the agent that will use the signing request.

`parameter 3`

The name of the country in which the server resides.

`parameter 4`

The name of the state in which the server resides.

`parameter 5`

The location (city) within the state in which the server resides.

`parameter 6`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The name of the organization unit requesting the certificate.

`parameter 7`

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.

19.4 Data Redaction

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Data redaction limits sensitive data exposure by dynamically changing data as it is 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 only see the last four digits `xxx-xx-9567`.

Data redaction is implemented by defining a function for each field to which redaction is to be applied. The function returns the value that should be displayed to the users subject to the data redaction.

So for example, for the SSN field, the redaction function would return `xxx-xx-9567` for an input SSN of `021-23-9567`.

For a salary field, a redaction function would always return `$0.00` regardless of the input salary value.

These functions are then incorporated into a redaction policy by using the `CREATE REDACTION POLICY` command. This command specifies the table on which the policy applies, the table columns to be affected by the specified redaction functions, expressions to determine which session users are to be affected, and other options.

The `edb_data_redaction` parameter in the `postgresql.conf` file then determines whether or not data redaction is to be applied.

By default, the parameter is enabled so the redaction policy is in effect and the following occurs:

- Superusers and the table owner bypass data redaction and see the original data.
- All other users get 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 will get an error.

A redaction policy can be changed by using the `ALTER REDACTION POLICY` command, or it can be eliminated using the `DROP REDACTION POLICY` command.

The redaction policy commands are described in more detail in the subsequent sections.

CREATE REDACTION POLICY

`CREATE REDACTION POLICY` defines a new data redaction policy for a table.

Synopsis

```
CREATE REDACTION POLICY <name> ON <table_name>
  [ FOR ( <expression> ) ]
  [ ADD [ COLUMN ] <column_name> USING <funcname_clause>
    [ WITH OPTIONS ( [ <redaction_option> ]
      [, <redaction_option> ] )
    ]
  ] [, ...]
```

where `redaction_option` is:

```
{ SCOPE <scope_value> |
  EXCEPTION <exception_value> }
```

Description

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `CREATE REDACTION POLICY` command defines a new column-level security policy for a table by redacting column data using redaction function. A newly created data redaction policy will be enabled by default. The policy can be disabled using `ALTER REDACTION POLICY ... DISABLE`.

`FOR ( expression )`

This form adds a redaction policy expression.

`ADD [ COLUMN ]`

This optional form adds a column of the table to the data redaction policy. The `USING` specifies a redaction function expression. Multiple `ADD [ COLUMN ]` form can be used, 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 and/or an exception to the data redaction policy to be applied. If the scope and/or exception are not specified, the default values for scope and exception will be `query` and `none` respectively.

Parameters

`name`

The name of the data redaction policy to be created. This must be distinct from the name of any other existing data redaction policy for the table.

`table_name`

The name (optionally schema-qualified) of the table the data redaction policy applies to.

`expression`

The data redaction policy expression. No redaction will be applied if this expression evaluates to false.

`column_name`

Name of the existing column of the table on which the data redaction policy being created.

`funcname_clause`

The data redaction function which decides how to compute the redacted column value. Return type of the redaction function should be same as the column type on which data redaction policy being added.

`scope_value`

The scope identified the query part where redaction to be applied for the column. Scope value could be `query, top_tlist` or `top_tlist_or_error`. If the scope is `query` then, the redaction applied on the column irrespective of where it appears in the query. If the scope is `top_tlist` then, the redaction 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 will be same as the `top_tlist`, but throws an errors when the column appears anywhere else in the query.

`exception_value`

The exception identified the query part where redaction to be exempted. Exception value could be `none, equal` or `leakproof`. If exception is `none` then there is no exemption. If exception is `equal`, then the column is not redacted when used in an equality test. If exception is `leakproof`, the column will is not 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Examples

Below is an example of how this feature can be used in production environments. Create the components for a data redaction policy on the `employees` table:

```
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::money;
END;
```

Now create a data redaction policy on `employees` to redact column `ssn` which should be accessible in equality condition and `salary` with default scope and exception. The redaction policy will be exempt for the `hr` user.

```
 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 will be:

```
 -- hr can view all columns data
edb=# \c edb hr
edb=> SELECT * FROM employees;
 id | name         | ssn         | phone      | birthday           |
 salary     | email
----+--------------+-------------+------------+--------------------+---
--+--------------------
  1 | Sally Sample | 020-78-9345 | 5081234567 | 02-FEB-61 00:00:00 |
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  $51,234.34 | sally.sample@enterprisedb.com
  2 | Jane Doe     | 123-33-9345 | 6171234567 | 14-FEB-63 00:00:00 |
  $62,500.00 | jane.doe@gmail.com
  3 | Bill Foo     | 123-89-9345 | 9781234567 | 14-FEB-63 00:00:00 |
  $45,350.00 | william.foe@hotmail.com
(3 rows)
```

The visible data for the normal user `alice` will be:

```
 -- Normal user cannot see salary and ssn number.
edb=> \c edb alice
edb=> SELECT * FROM employees;
id  | name         | ssn         | phone      | birthday           | salary |
email
----+--------------+-------------+------------+--------------------+--------+-
------------------------------
  1 | Sally Sample | xxx-xx-9345 | 5081234567 | 02-FEB-61 00:00:00 |  $0.00 |
  sally.sample@enterprisedb.com
  2 | Jane Doe     | xxx-xx-9345 | 6171234567 | 14-FEB-63 00:00:00 |  $0.00 |
  jane.doe@gmail.com
  3 | Bill Foo     | xxx-xx-9345 | 9781234567 | 14-FEB-63 00:00:00 |  $0.00 |
  william.foe@hotmail.com
(3 rows)
```

But `ssn` data is accessible when it used for equality check due to `exception_value` setting.

```
 -- Get ssn number starting from 123
edb=> SELECT * FROM employees WHERE substring(ssn from 0 for 4) = '123';
 id | name     | ssn         | phone      | birthday           | salary |
 email
----+----------+-------------+------------+--------------------+--------+-----
--------------------
  2 | Jane Doe | xxx-xx-9345 | 6171234567 | 14-FEB-63 00:00:00 |  $0.00 |
  jane.doe@gmail.com
  3 | Bill Foo | xxx-xx-9345 | 9781234567 | 14-FEB-63 00:00:00 |  $0.00 |
  william.foe@hotmail.com
(2 rows)
```

Caveats

1. The data redaction policy created on inheritance hierarchies will not be cascaded. For example, if the data redaction policy is created for a parent, it will not be applied to the child table, which inherits it and vice versa. Someone who has access to these child tables can see the non-redacted data. For information about inheritance hierarchies, see *Inheritance* in the PostgreSQL Core Documentation available at:

   https://www.postgresql.org/docs/11/static/ddl-inherit.html

2. If the superuser or the table owner has created any materialized view on the table and has provided the access rights `GRANT SELECT` on the table and the materialized view to any non-superuser, then the non-superuser will be able to access the non-redacted data through the materialized view.

3. The objects accessed in the redaction function body should be schema qualified otherwise `pg_dump` might fail.

Compatibility

`CREATE REDACTION POLICY` is an EnterpriseDB extension.

See Also

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`ALTER REDACTION POLICY, DROP REDACTION POLICY`

ALTER REDACTION POLICY

`ALTER REDACTION POLICY` 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>
```

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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 the details.

`MODIFY [ COLUMN ]`

This form modifies the data redaction policy on the column of the table. You can update the redaction function clause and/or the redaction options for the column. The `USING` clause specifies the redaction function expression to be updated and the `WITH OPTIONS ( ... )` clause specifies the scope and/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 name (optionally schema-qualified) 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 being altered or dropped.

`funcname_clause`

The data redaction function expression for the column. See `CREATE REDACTION POLICY` for details.

`scope_value`

The scope identified the query part where redaction to be applied for the column. See `CREATE REDACTION POLICY` for the details.

`exception_value`

The exception identified the query part where redaction to be exempted. See `CREATE REDACTION POLICY` for the details.

Examples

Update data redaction policy called `redact_policy_personal_info` on the table named `employees`:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
ALTER REDACTION POLICY redact_policy_personal_info ON employees
FOR (session_user != 'hr' AND session_user != 'manager');
```

And to update data redaction function for the column `ssn` in the same policy:

```
ALTER REDACTION POLICY redact_policy_personal_info ON employees
MODIFY COLUMN ssn USING redact_ssn_new(ssn);
```

Compatibility

`ALTER REDACTION POLICY` is an EnterpriseDB extension.

See Also

`CREATE REDACTION POLICY, DROP REDACTION POLICY`

DROP REDACTION POLICY

`DROP REDACTION POLICY` 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`

Do not throw an error if the data redaction policy does not exist. A notice is issued in this case.

`name`

The name of the data redaction policy to drop.

`table_name`

The name (optionally schema-qualified) of the table that the data redaction policy is on.

`CASCADE`

`RESTRICT`

These keywords do not have any effect, since there are no dependencies on the data redaction policies.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

`DROP REDACTION POLICY` is an EnterpriseDB extension.

See Also

`CREATE REDACTION POLICY, ALTER REDACTION POLICY`

System Catalogs

This section describes the system catalogs that 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.

Column	Type	References	Description
`oid`	`oid`		Row identifier (hidden attribute, must be explicitly selected)
`rdpolicyid`	`oid`	`edb_redaction_policy.oid`	The data redaction policy applies to the described column
`rdrelid`	`oid`	`pg_class.oid`	The table that the described column belongs to
`rdattnum`	`int2`	`pg_attribute.attnum`	The number of the described column
`rdscope`	`int2`		The redaction scope: `1` = query, `2` = top_tlist, `4` = top_tlist_or_error
`rdexception`	`int2`		The redaction exception: `8` = none, `16` = equal, `32` = leakproof
`rdfuncexpr`	`pg_node_tree`		Data redaction function expression

> **Note**
>
> The described column will be 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.

Column	Type	References	Description
`oid`	`oid`		Row identifier (hidden attribute, must be explicitly selected)

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	References	Description
`rdname`	`name`		The name of the data redaction policy
`rdrelid`	`oid`	`pg_class.oid`	The table to which the data redaction policy applies
`rdenable`	`boolean`		Is the data redaction policy enabled?
`rdexpr`	`pg_node_tree`		The data redaction policy expression

> **Note**
>
> The data redaction policy applies for the table if it is enabled and the expression ever evaluated true.

20 EDB Postgres Advanced Server Guide

Advanced Server adds extended functionality to the open-source PostgreSQL database. The extended functionality supports database administration, enhanced SQL capabilities, database and application security, performance monitoring and analysis, and application development utilities. This guide documents those features that are exclusive to Advanced Server.

- **Enhanced Compatibility Features.** This section provides an overview of compatibility features supported by Advanced Server.
- **Database Administration.** This section contains information about features and tools that are helpful to the database administrator.
  - `Index Advisor` helps to determine the additional indexes needed on tables to improve application performance.
  - `SQL Profiler` locates and diagnoses poorly running SQL queries in applications.
  - `pgsnmpd` is an SNMP agent that returns hierarchical monitoring information regarding the current state of Advanced Server.

- **EDB Resource Manager.** This section contains information about the EDB Resource Manager feature, which provides the capability to control system resource usage by Advanced Server processes.
  - `Resource Groups` shows how to create and maintain the groups on which resource limits can be defined.
  - `CPU Usage Throttling` provides a method to control CPU usage by Advanced Server processes.
  - `Dirty Buffer Throttling` provides a method to control the dirty rate of shared buffers by Advanced Server processes.

- **The libpq C Library.** The `libpq C library` is the C application programming interface (API) language for Advanced Server.
- **The PL Debugger.** The `PL Debugger` is a graphically oriented debugging tool for PL/pgSQL.
- **Performance Analysis and Tuning.** This section contains the various tools for analyzing and improving application and database server performance.
  - `Dynatune` provides a quick and easy means for configuring Advanced Server depending upon the type of application usage.
  - `EDB wait states` provides a way to capture wait events and other data for performance diagnosis.

- **EDB Clone Schema.** This section contains information about the EDB Clone Schema feature, which provides the capability to copy a schema and its database objects within a single database or from one database to another database.
- **Enhanced SQL and Other Miscellaneous Features.** This section contains information on enhanced SQL functionality and other features that provide additional flexibility and convenience.
- **System Catalog Tables.** This section contains additional *system catalog tables* added for Advanced Server specific database objects.
- **Advanced Server Keywords.** This section contains information about the words that Advanced Server recognizes as keywords.

For information about the features that are shared by EDB Postgres Advanced Server and PostgreSQL, see thePostgreSQL core documentation.

20.1 Introduction

See the release notes for the features added in EDB Postgres Advanced Server 11.

20.1.1 Conventions Used in this Guide

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following is a list of conventions used throughout this document.

- This guide applies 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 the information in this document may apply interchangeably to the PostgreSQL and EDB Postgres Advanced Server database systems. The term *Advanced Server* is used to refer to EDB Postgres Advanced Server. The term *Postgres* is used to generically refer to both PostgreSQL and Advanced Server. When a distinction needs to be made between these two database systems, the specific names, PostgreSQL or Advanced Server are used.
- The installation directory path of the PostgreSQL or Advanced Server products is referred to as *POSTGRES_INSTALL_HOME*.
    - For PostgreSQL Linux installations, this defaults to `/opt/PostgreSQL/x.x` for version 10 and earlier. For later versions, use the PostgreSQL community packages.
    - For Advanced Server Linux installations accomplished using the interactive installer for version 10 and earlier, this defaults to `/opt/edb/asx.x`.
    - For Advanced Server Linux installations accomplished using an RPM package, this defaults to `/usr/edb/asxx`.
    - For Advanced Server Windows installations, this defaults to `C:\Program Files\edb\asxx`. The product version number is represented by `x.x` or by `xx` for version 10 and later.

20.1.2 About the Examples Used in this Guide

The examples in this guide are shown in the type and background illustrated below.

Examples and output from examples are shown in fixed-width, white font on a black background.

The examples use the sample tables, `dept, emp`, and `jobhist`, created and loaded when Advanced Server is installed.

The tables and programs in the sample database can be re-created at any time by executing the following script:

```
/usr/edb/asxx/share/pg-sample.sql
```

where `xx` is the Advanced Server version number.

In addition there is a script in the same directory containing the database objects created using syntax compatible with Oracle databases. This script file is `edb-sample.sql`.

The script:

- 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 will be 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:

```
SHOW SEARCH_PATH;
```

You can use PSQL commands to modify the search path.

Sample Database Description

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

employee-related information is stored in the `emp` table.

The sample company is regionally diverse, so it tracks the locations of its 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, etc. 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 the `pg-sample.sql` script:

```
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(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
    CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
        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';
--
--  Sequence to generate values for function 'new_empno'.
--
CREATE SEQUENCE next_empno START WITH 8000 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;
--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,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'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,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',1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,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',800,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,'New Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,'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',1250,1400,30,'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,'New Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-88','CLERK',1000,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-89','CLERK',1040,NULL,20,'Raise');
INSERT INTO jobhist VALUES (7788,'05-MAY-90',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',1500,0,30,'New Hire');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-83','CLERK',950,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7900,'15-JAN-83',NULL,'CLERK',950,NULL,30,'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,'New Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,'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 $$
DECLARE
    v_empno             NUMERIC(4);
    v_ename             VARCHAR(10);
    emp_cur CURSOR FOR
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    RAISE INFO 'EMPNO ENAME';
    RAISE INFO '----- -------';
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN NOT FOUND;
        RAISE INFO '% %', v_empno, v_ename;
    END LOOP;
    CLOSE emp_cur;
    RETURN;
END;
$$ 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 INTO
        v_ename, v_hiredate, v_sal, v_comm, v_dname
        ename, hiredate, sal, COALESCE(comm, 0), dname
        FROM emp e, dept d
        WHERE empno = p_empno
          AND e.deptno = d.deptno;
    IF NOT FOUND THEN
        RAISE INFO 'Employee % not found', p_empno;
        RETURN;
    END IF;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO 'The following is SQLERRM : %', SQLERRM;
        RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
        RETURN;
END;
$$ LANGUAGE 'plpgsql';
--
--  A RECORD type used to format the return value of
--  function, 'emp_query'.
--
CREATE TYPE emp_query_type AS (
    empno           NUMERIC,
    ename           VARCHAR(10),
    job             VARCHAR(9),
    hiredate        DATE,
    sal             NUMERIC
);
--
--  Function that queries the 'emp' table based on
--  department number and employee number or name. Returns
--  employee number and name as INOUT parameters and job,
--  hire date, and salary as OUT parameters. These are
--  returned in the form of a record defined by
--  RECORD type, 'emp_query_type'.
--
CREATE OR REPLACE FUNCTION emp_query (
    IN p_deptno         NUMERIC,
    INOUT p_empno       NUMERIC,
    INOUT p_ename       VARCHAR,
    OUT p_job           VARCHAR,
    OUT p_hiredate      DATE,
    OUT p_sal           NUMERIC
)
AS $$
BEGIN
    SELECT INTO
        p_empno, p_ename, p_job, p_hiredate, p_sal
        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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO 'The following is SQLERRM : %', SQLERRM;
        RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
        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 v_new_empno;
END;
$$ LANGUAGE 'plpgsql';
--
--  Function that adds a new clerk to table 'emp'.
--
CREATE OR REPLACE FUNCTION hire_clerk (
    p_ename        VARCHAR,
    p_deptno       NUMERIC
) RETURNS NUMERIC
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
AS $$
DECLARE
    v_empno        NUMERIC(4);
    v_ename        VARCHAR(10);
    v_job          VARCHAR(9);
    v_mgr          NUMERIC(4);
    v_hiredate     DATE;
    v_sal          NUMERIC(7,2);
    v_comm         NUMERIC(7,2);
    v_deptno       NUMERIC(2);
BEGIN
    v_empno := new_empno();
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        CURRENT_DATE, 950.00, NULL, p_deptno);
    SELECT INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        empno, ename, job, mgr, hiredate, sal, comm, deptno
        FROM emp WHERE empno = v_empno;
    RAISE INFO 'Department : %', v_deptno;
    RAISE INFO 'Employee No: %', v_empno;
    RAISE INFO 'Name       : %', v_ename;
    RAISE INFO 'Job        : %', v_job;
    RAISE INFO 'Manager    : %', v_mgr;
    RAISE INFO 'Hire Date  : %', v_hiredate;
    RAISE INFO 'Salary     : %', v_sal;
    RAISE INFO 'Commission : %', v_comm;
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO 'The following is SQLERRM : %', SQLERRM;
        RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
        RETURN -1;
END;
$$ 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);
    v_comm         NUMERIC(7,2);
    v_deptno       NUMERIC(2);
BEGIN
    v_empno := new_empno();
    INSERT INTO emp VALUES (v_empno, p_ename, 'SALESMAN', 7698,
        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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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;
$$ LANGUAGE 'plpgsql';
--
--  Rule to INSERT into view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO salesemp
DO INSTEAD
    INSERT INTO emp VALUES (NEW.empno, NEW.ename, 'SALESMAN', 7698,
        NEW.hiredate, NEW.sal, NEW.comm, 30);
--
--  Rule to UPDATE view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO salesemp
DO INSTEAD
    UPDATE emp SET empno    = NEW.empno,
                   ename    = NEW.ename,
                   hiredate = NEW.hiredate,
                   sal      = NEW.sal,
                   comm     = NEW.comm
        WHERE empno = OLD.empno;
--
--  Rule to DELETE from view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO salesemp
DO INSTEAD
    DELETE FROM emp WHERE empno = OLD.empno;
--
--  After statement-level trigger that displays a message after
--  an insert, update, or deletion to the 'emp' table. One message
--  per SQL command is displayed.
--
CREATE OR REPLACE FUNCTION user_audit_trig() RETURNS TRIGGER
AS $$
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;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
$$ LANGUAGE 'plpgsql';
CREATE TRIGGER user_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH STATEMENT EXECUTE PROCEDURE user_audit_trig();
--
--  Before row-level trigger that displays employee number and
--  salary of an employee that is about to be added, updated,
--  or deleted in the 'emp' table.
--
CREATE OR REPLACE FUNCTION emp_sal_trig() RETURNS TRIGGER
AS $$
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;
        RETURN OLD;
    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;
```

20.2 Enhanced Compatibility Features

Advanced Server includes extended functionality that provides compatibility for syntax supported by Oracle applications. Detailed information about the compatibility features supported by Advanced Server is provided in the *Database Compatibility for Oracle Developers Guide*; the version-specific guides are available at:

https://www.enterprisedb.com/docs

The following sections highlight some of the compatibility features supported by Advanced Server.

Enabling Compatibility Features

There are several ways to install Advanced Server that will allow you to take advantage of compatibility features:

- Use the `INITDBOPTS` variable (in the Advanced Server service configuration file) to specify `--redwood-like` before initializing your cluster.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- When invoking `initdb` to initialize your cluster compatible with Oracle mode, include the `--redwood-like` option or `--no-redwood-compat` option to initialize your cluser in Oracle non-compatible mode.

For more information about the installation options supported by the Advanced Server installers, see the *EDB Postgres Advanced Server Installation Guide*, available from the EDB website at:

https://www.enterprisedb.com/docs

Stored Procedural Language

Advanced Server supports a highly productive procedural language that allows you to write custom procedures, functions, triggers and packages. The procedural language:

- complements the SQL language and built-in packages.
- provides a seamless development and testing environment.
- allows you to create reusable code.

For information about using the Stored Procedural Language, see the *Database Compatibility for Stored Procedural Language Guide*, available at:

https://www.enterprisedb.com/docs

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 will (generally) return the result set in the least amount of time. The server's choice of plan is dependent upon 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 have been gathered by the `ANALYZE` command.

As a rule, the query planner will select the least expensive plan. You can use an optimizer hint to influence the server as it selects a query plan.

An optimizer hint is a directive (or multiple 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 employ or avoid a specific plan when producing the result set. For information about using optimizer hints, see the *Database Compatibility for Oracle Developers Guide*, available at:

https://www.enterprisedb.com/docs

Data Dictionary Views

Advanced Server includes a set of views that provide information about database objects in a manner compatible with the Oracle data dictionary views. For detailed information about the views available with Advanced Server, see the *Database Compatibility for Oracle Developers Catalog Views Guide*, available at:

https://www.enterprisedb.com/docs

dblink_ora

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

dblink_ora provides an OCI-based database link that allows you to `SELECT, INSERT, UPDATE` or `DELETE` data stored on an Oracle system from within Advanced Server. For detailed information about using `dblink_ora`, and the supported functions and procedures, see the *Database Compatibility for Oracle Developers Guide*, available at:

https://www.enterprisedb.com/docs

Profile Management

Advanced Server supports compatible SQL syntax for profile management. Profile management commands allow a database superuser to create and manage named *profiles*. 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 re-use

A profile is a named set of attributes that allow you to easily manage a group of roles that share comparable authentication requirements. If password requirements change, you can modify the profile to have the new requirements applied to each user that is 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 is associated with their login role. Profiles are shared by all databases within a cluster, but each cluster may have multiple profiles. A single user with access to multiple databases will use the same profile when connecting to each database within the cluster.

For information about using profile management commands, see the *Database Compatibility for Oracle Developers Guide*, available at:

https://www.enterprisedb.com/docs

Built-In Packages

Advanced Server supports a number of built-in packages that provide compatibility with Oracle procedures and functions.

Package Name	Description
DBMS_ALERT	The DBMS_ALERT package provides the capability to register for, send, and receive alerts.
DBMS_AQ	The DBMS_AQ package provides message queueing and processing for Advanced Server.
DBMS_AQADM	The DBMS_AQADM package provides supporting procedures for Advanced Queueing functionality.
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.
DBMS_JOB	The DBMS_JOB package provides for the creation, scheduling, and managing of jobs.
DBMS_LOB	The DBMS_LOB package provides the capability to operate on large objects.
DBMS_LOCK	Advanced Server provides support for the DBMS_LOCK.SLEEP procedure.
DBMS_MVIEW	Use procedures in the DBMS_MVIEW package to manage and refresh materialized views and their dependencies.
DBMS_OUTPUT	The DBMS_OUTPUT package provides the capability to send messages to a message buffer, or get messages from the message buffer.
DBMS_PIPE	The DBMS_PIPE package provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.
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.
DBMS_RANDOM	The DBMS_RANDOM package provides a number of methods to generate random values.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Package Name	Description
DBMS_REDACT	The DBMS_REDACT package enables the redacting or masking of data that is returned by a query.
DBMS_RLS	The DBMS_RLS package enables the implementation of Virtual Private Database on certain Advanced Server database objects.
DBMS_SCHEDULER	The DBMS_SCHEDULER package provides a way to create and manage jobs, programs and job schedules.
DBMS_SESSION	Advanced Server provides support for the DBMS_SESSION.SET_ROLE procedure.
DBMS_SQL	The DBMS_SQL package provides an application interface to the EnterpriseDB dynamic SQL functionality.
DBMS_UTILITY	The DBMS_UTILITY package provides various utility programs.
UTL_ENCODE	The UTL_ENCODE package provides a way to encode and decode data.
UTL_FILE	The UTL_FILE package provides the capability to read from, and write to files on the operating system's file system.
UTL_HTTP	The UTL_HTTP package provides a way to use the HTTP or HTTPS protocol to retrieve information found at an URL.
UTL_MAIL	The UTL_MAIL package provides the capability to manage e-mail.
UTL_RAW	The UTL_RAW package allows you to manipulate or retrieve the length of raw data types.
UTL_SMTP	The UTL_SMTP package provides the capability to send e-mails over the Simple Mail Transfer Protocol (SMTP).
UTL_URL	The UTL_URL package provides a way to escape illegal and reserved characters within an URL.

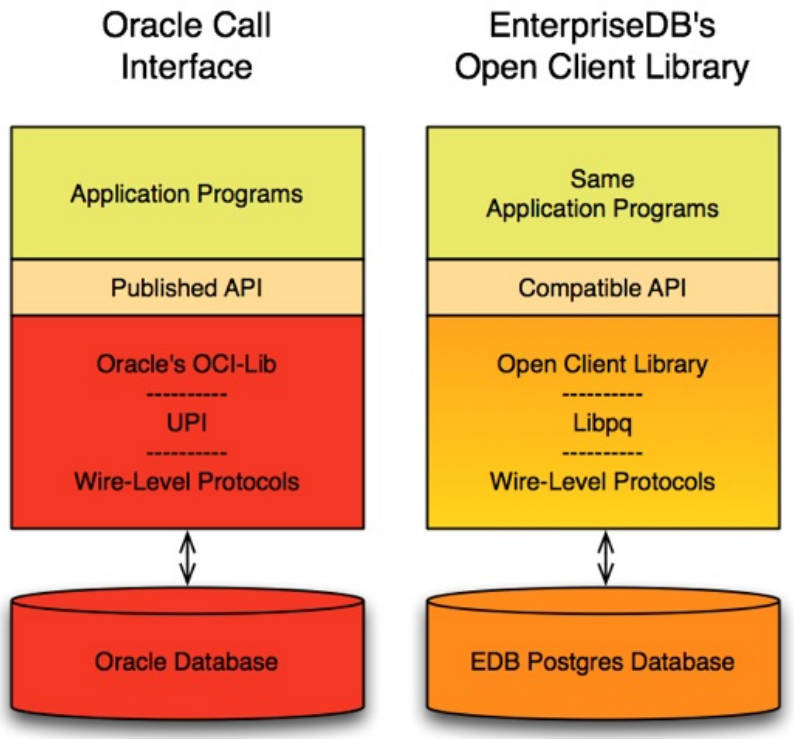
For detailed information about the procedures and functions available within each package, see the *Database Compatibility for Oracle Developers Built-In Package Guide*, available at:

https://www.enterprisedb.com/docs

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 Advanced Server or an Oracle database with minimal to no changes to the application code. The EnterpriseDB implementation of the Open Client Library is written in C.

The following diagram compares the Open Client Library and Oracle Call Interface application stacks.



Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

For detailed information about the functions supported by the Open Client Library, see the *EDB Postgres Advanced Server OCL Connector Guide*, available at:

https://www.enterprisedb.com/docs

Utilities

For detailed information about the compatible syntax supported by the utilities listed below, see the *Database Compatibility for Oracle Developers Tools and Utilities Guide*, available at:

https://www.enterprisedb.com/docs

EDB*Plus

EDB*Plus is a utility program that provides a command line user interface to the Advanced Server that will be 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

For detailed information about EDB*Plus, see the *EDB*Plus User's Guide* available at:

https://www.enterprisedb.com/docs

EDB*Loader

EDB*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for 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 do not meet the selection criteria of any target table
- Log file for recording the EDB*Loader session and any error messages
- Data loading from standard input and remote loading

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 is nearly impossible to read. Once you have the obfuscated form of the code, you can send that code to Advanced Server and it will store 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Dynamic Runtime Instrumentation Tools Architecture (DRITA)

The Dynamic Runtime Instrumentation Tools Architecture (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. Each snapshot is identified by a unique ID number; you can use snapshot ID numbers with DRITA reporting functions to return system performance statistics.

ECPGPlus

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) 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 syntax in C programs when connected to an Advanced Server database. ECPGPlus supports:

- Oracle Dynamic SQL – Method 4 (ODS-M4)
- Pro*C compatible anonymous blocks
- A `CALL` statement compatible with Oracle databases

For information about using ECPGPlus, see the *EDB Postgres Advanced Server ECPG Connector Guide*, available from the EnterpriseDB website at:

https://www.enterprisedb.com/docs

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 fits in memory.
- When a query or update accesses a large percentage of a single partition, performance may improve because the server will perform a sequential scan of the partition instead of using an index and random access reads scattered across the whole table.
- A bulk load (or unload) can be implemented by adding or removing partitions, if you plan that requirement into the partitioning design. `ALTER TABLE` is far faster than a bulk operation. It also entirely avoids the `VACUUM` overhead caused by a bulk `DELETE`.
- Seldom-used data can be migrated 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 will benefit from partitioning depends on the application; a good rule of thumb is that the size of the table should exceed the physical memory of the database server.

For information about database compatibility features supported by Advanced Server see the *Database Compatibility Table Partitioning Guide*, available at:

https://www.enterprisedb.com/docs

20.3 Database Administration

This chapter describes the features that aid in the management and administration of Advanced Server databases.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

20.3.1 Configuration Parameters

This section describes the database server configuration parameters of Advanced Server. These 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, locale and formatting settings, and so on.

Configuration parameters that apply only to Advanced Server are noted in theSummary of Configuration Parameters.

Additional information about configuration parameters can be found in the PostgreSQL Core Documentation available at:

https://www.postgresql.org/docs/11/static/runtime-config.html

20.3.1.1 Setting Configuration Parameters

This section provides an overview of how configuration parameters are specified and set.

Each configuration parameter is set using a `name/value` pair. Parameter names are case-insensitive. The parameter name is typically separated from its value by an optional equals sign `(=)`.

The following is an example of 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
```

Parameter values are specified as one of five types:

- **Boolean.** Acceptable values can be written as `on, off, true, false, yes, no, 1, 0`, or any unambiguous prefix of these.
- **Integer.** Number without a fractional part.
- **Floating Point.** Number with an optional fractional part separated by a decimal point.
- **String.** Text value. Enclose in single quotes if the value is not a simple identifier or number (that is, the value contains special characters such as spaces or other punctuation marks).
- **Enum.** Specific set of string values. The allowed values can be found in the system view `pg_settings.enumvals`. Enum values are case-insensitive.

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. Default units can be found by referencing the system view `pg_settings.unit`. A different unit can be specified explicitly.

Valid memory units are `kB` (kilobytes), `MB` (megabytes), and `GB` (gigabytes). Valid time units are `ms` (milliseconds), `s` (seconds), `min` (minutes), `h` (hours), and `d` (days). The multiplier for memory units is 1024.

The configuration parameter settings can be established in a number of different ways:

- There is a number of parameter settings that are established when the Advanced Server database product is built. These are read-only parameters, and their values cannot be changed. There are also a couple of parameters that are permanently set for each database when the database is created. These parameters are read-only as well and cannot be subsequently changed for the database.

- The initial settings for almost all configurable parameters across the entire database cluster are listed in the configuration file, `postgresql.conf`. These settings are put into effect upon database server start or restart. Some of these initial parameter settings can be overridden as discussed in the following bullet points. All configuration parameters have built-in default settings that are in effect if not explicitly overridden.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Configuration parameters in the `postgresql.conf` file are overridden when the same parameters are included in the `postgresql.auto.conf` file. The `ALTER SYSTEM` command is used to manage the configuration parameters in the `postgresql.auto.conf` file.

- Parameter settings can be modified 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 other of these parameter types, a new session must be started to use the new settings. And yet for other parameter types, modified settings do not take effect until the database server is stopped and restarted. See the following section of the PostgreSQL Core Documentation for information on how to reload the configuration file:

  https://www.postgresql.org/docs/11/config-setting.html

- The SQL commands `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` can be used to modify certain parameter settings. The modified parameter settings take effect for new sessions after the command is executed. `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 methods discussed in the second and third bullet points. Parameter settings established using the `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` commands can only be changed by:

  a) re-issuing these commands with a different parameter value, or

  b) issuing these commands using either of the `SET parameter TO DEFAULT` clause or the `RESET parameter` clause. These clauses change the parameter back to using the setting established by the methods set forth in the prior bullet points. See the “SQL Commands” section of Chapter VI “Reference” in the *PostgreSQL Core Documentation* for the exact syntax of these SQL commands:

  https://www.postgresql.org/docs/11/sql-commands.html

- Changes can be made for certain parameter settings for the duration of individual sessions using the `PGOPTIONS` environment variable or by using the `SET` command within the `EDB-PSQL` or `PSQL` command line terminal programs. Parameter settings made in this manner override settings established using any of the methods described by the second, third, and fourth bullet points, but only for the duration of the session.

Modifying the postgresql.conf File

The configuration parameters in the `postgresql.conf` file specify server behavior with regards to auditing, authentication, encryption, and other behaviors. On Linux and Windows host, the `postgresql.conf` file resides in the `data` directory under your 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 `reload` or `restart` the server for the new parameter value to take effect.

Within 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, execute 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 method or methods that the server will use when authenticating connecting clients. Before connecting to the server, you may be required to modify the authentication properties specified in the `pg_hba.conf` file.

When you invoke the `initdb` utility to create a cluster, `initdb` creates a `pg_hba.conf` file for that cluster that specifies the type of authentication

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

required from connecting clients. To modify the `pg_hba.conf` file, open the file with your choice of editor. 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 at:

https://www.postgresql.org/docs/11/static/auth-pg-hba-conf.html

20.3.1.2 Summary of Configuration Parameters

This section contains a summary table listing all 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.
- **Scope of Effect.** 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 whereby the latter settings in this list override prior ones: a) the entire database cluster, b) specific databases in the database cluster, c) specific roles, d) specific roles when connected to specific databases, e) a specific session.
- **When Takes Effect.** When a changed parameter setting takes effect.
    - `Preset` – Established when the Advanced Server product is built or a particular database is created. This is a read-only parameter and cannot be changed.
    - `Restart` – Database server must be restarted.
    - `Reload` – Configuration file must be reloaded (or the database server can be restarted).
    - `Immediate` – 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.
- **Authorized User.** Type of operating system account or database role that must be used to put the parameter setting into effect.
    - `EPAS service account` – 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 (the database or role to be altered with the `ALTER` command).
    - `n/a` – Parameter setting cannot be changed by any user.
- **Description.** Brief description of the configuration parameter.
- **EPAS Only.** 'X' – Configuration parameter is applicable to EDB Postgres Advanced Server only. No entry in this column indicates the configuration parameter applies to PostgreSQL as well.

> **Note**
>
> There are a number of parameters that should never be altered. These are designated as "**Note: For internal use only**" in the Description column.

The following table shows the summary of configuration parameters:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
allow_system_table_mods	Cluster	Restart	EPAS service account	Allows modifications of the structure of system tables.	
application_name	Session	Immediate	User	Sets the application name to be reported in statistics and logs.	
archive_command	Cluster	Reload	EPAS service account	Sets the shell command that will be called to archive a WAL file.	
archive_mode	Cluster	Restart	EPAS service account	Allows archiving of WAL files using archive_command.	
archive_timeout	Cluster	Reload	EPAS service account	Forces a switch to the next xlog file if a new file has not been started within N seconds.	
array_nulls	Session	Immediate	User	Enable input of NULL elements in arrays.	
authentication_timeout	Cluster	Reload	EPAS service account	Sets the maximum allowed time to complete client authentication.	
autovacuum	Cluster	Reload	EPAS service account	Starts the autovacuum subprocess.	
autovacuum_analyze_scale_factor	Cluster	Reload	EPAS service account	Number of tuple inserts, updates, or deletes prior to analyze as a fraction of reltuples.	
autovacuum_analyze_threshold	Cluster	Reload	EPAS service account	Minimum number of tuple inserts, updates, or deletes prior to analyze.	
autovacuum_freeze_max_age	Cluster	Restart	EPAS service account	Age at which to autovacuum a table to prevent transaction ID wraparound.	
autovacuum_max_workers	Cluster	Restart	EPAS service account	Sets the maximum number of simultaneously running autovacuum worker processes.	
autovacuum_multixact_freeze_max_age	Cluster	Restart	EPAS service account	Multixact age at which to autovacuum a table to prevent multixact wraparound.	
autovacuum_naptime	Cluster	Reload	EPAS service account	Time to sleep between autovacuum runs.	
autovacuum_vacuum_cost_delay	Cluster	Reload	EPAS service account	Vacuum cost delay in milliseconds, for autovacuum.	
autovacuum_vacuum_cost_limit	Cluster	Reload	EPAS service account	Vacuum cost amount available before napping, for autovacuum.	
autovacuum_vacuum_scale_factor	Cluster	Reload	EPAS service account	Number of tuple updates or deletes prior to vacuum as a fraction of reltuples.	
autovacuum_vacuum_threshold	Cluster	Reload	EPAS service account	Minimum number of tuple updates or deletes prior to vacuum.	
autovacuum_work_mem	Cluster	Reload	EPAS service account	Sets the maximum memory to be used by each autovacuum worker process.	
backslash_quote	Session	Immediate	User	Sets whether "\'" is allowed in string literals.	
bgwriter_delay	Cluster	Reload	EPAS service account	Background writer sleep time between rounds.	
bgwriter_lru_maxpages	Cluster	Reload	EPAS service account	Background writer maximum number of LRU pages to flush per round.	
bgwriter_lru_multiplier	Cluster	Reload	EPAS service account	Multiple of the average buffer usage to free per round.	
block_size	Cluster	Preset	n/a	Shows the size of a disk block.	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
bonjour	Cluster	Restart	EPAS service account	Enables advertising the server via Bonjour.	
bonjour_name	Cluster	Restart	EPAS service account	Sets the Bonjour service name.	
bytea_output	Session	Immediate	User	Sets the output format for bytea.	
check_function_bodies	Session	Immediate	User	Check function bodies during CREATE FUNCTION.	
checkpoint_completion_target	Cluster	Reload	EPAS service account	Time spent flushing dirty buffers during checkpoint, as fraction of checkpoint interval.	
checkpoint_segments	Deprecated	Deprecated	Deprecated	Specifying a value for the parameter will prevent the server from starting.	
checkpoint_timeout	Cluster	Reload	EPAS service account	Sets the maximum time between automatic WAL checkpoints.	
checkpoint_warning	Cluster	Reload	EPAS service account	Enables warnings if checkpoint segments are filled more frequently than this.	
client_encoding	Session	Immediate	User	Sets the client's character set encoding.	
client_min_messages	Session	Immediate	User	Sets the message levels that are sent to the client.	
commit_delay	Session	Immediate	Superuser	Sets the delay in microseconds between transaction commit and flushing WAL to disk.	
commit_siblings	Session	Immediate	User	Sets the minimum concurrent open transactions before performing commit_delay.	
config_file	Cluster	Restart	EPAS service account	Sets the server's main configuration file.	
constraint_exclusion	Session	Immediate	User	Enables the planner to use constraints to optimize queries.	
cpu_index_tuple_cost	Session	Immediate	User	Sets the planner's estimate of the cost of processing each index entry during an index scan.	
cpu_operator_cost	Session	Immediate	User	Sets the planner's estimate of the cost of processing each operator or function call.	
cpu_tuple_cost	Session	Immediate	User	Sets the planner's estimate of the cost of processing each tuple (row).	
cursor_tuple_fraction	Session	Immediate	User	Sets the planner's estimate of the fraction of a cursor's rows that will be retrieved.	
custom_variable_classes	Cluster	Reload	EPAS service account	Deprecated in Advanced Server 9.2.	X
data_checksums	Cluster	Preset	n/a	Shows whether data checksums are turned on for this cluster.	
data_directory	Cluster	Restart	EPAS service account	Sets the server's data directory.	
datestyle	Session	Immediate	User	Sets the display format for date and time values.	
db_dialect	Session	Immediate	User	Sets the precedence of built-in namespaces.	X
dbms_alert.max_alerts	Cluster	Restart	EPAS service account	Sets maximum number of alerts.	X
dbms_pipe.total_message_buffer	Cluster	Restart	EPAS service account	Specifies the total size of the buffer used for the DBMS_PIPE package.	X

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
db_user_namespace	Cluster	Reload	EPAS service account	Enables per-database user names.	
deadlock_timeout	Session	Immediate	Superuser	Sets the time to wait on a lock before checking for deadlock.	
debug_assertions	Cluster	Preset	n/a	Turns on various assertion checks. (Not supported in EPAS builds.)	
debug_pretty_print	Session	Immediate	User	Indents parse and plan tree displays.	
debug_print_parse	Session	Immediate	User	Logs each query's parse tree.	
debug_print_plan	Session	Immediate	User	Logs each query's execution plan.	
debug_print_rewritten	Session	Immediate	User	Logs each query's rewritten parse tree.	
default_heap_fillfactor	Session	Immediate	User	Create new tables with this heap fillfactor by default.	X
default_statistics_target	Session	Immediate	User	Sets the default statistics target.	
default_tablespace	Session	Immediate	User	Sets the default tablespace to create tables and indexes in.	
default_text_search_config	Session	Immediate	User	Sets default text search configuration.	
default_transaction_deferrable	Session	Immediate	User	Sets the default deferrable status of new transactions.	
default_transaction_isolation	Session	Immediate	User	Sets the transaction isolation level of each new transaction.	
default_transaction_read_only	Session	Immediate	User	Sets the default read-only status of new transactions.	
default_with_oids	Session	Immediate	User	Create new tables with OIDs by default.	
default_with_rowids	Session	Immediate	User	Create new tables with ROWID support (ROWIDs with indexes) by default.	X
dynamic_library_path	Session	Immediate	Superuser	Sets the path for dynamically loadable modules.	
dynamic_shared_memory_type	Cluster	Restart	EPAS service account	Selects the dynamic shared memory implementation used.	
edb_audit	Cluster	Reload	EPAS service account	Enable EDB Auditing to create audit reports in XML or CSV format.	X
edb_audit_connect	Cluster	Reload	EPAS service account	Audits each successful connection.	X
edb_audit_destination	Cluster	Reload	EPAS service account	Sets edb_audit_directory or syslog as the destination directory for audit files. The syslog setting is only valid for a Linux system.	X
edb_audit_directory	Cluster	Reload	EPAS service account	Sets the destination directory for audit files.	X
edb_audit_disconnect	Cluster	Reload	EPAS service account	Audits end of a session.	X
edb_audit_filename	Cluster	Reload	EPAS service account	Sets the file name pattern for audit files.	X

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
edb_audit_rotation_day	Cluster	Reload	EPAS service account	Automatic rotation of log files based on day of week.	X
edb_audit_rotation_seconds	Cluster	Reload	EPAS service account	Automatic log file rotation will occur after N seconds.	X
edb_audit_rotation_size	Cluster	Reload	EPAS service account	Automatic log file rotation will occur after N Megabytes.	X
edb_audit_statement	Cluster	Reload	EPAS service account	Sets the type of statements to audit.	X
edb_audit_tag	Session	Immediate	User	Specify a tag to be included in the audit log.	X
edb_connectby_order	Session	Immediate	User	Sort results of `CONNECT BY` queries with no `ORDER BY` to depth-first order. Note: For internal use only.	X
edb_custom_plan_tries	Session	Immediate	User	Specifies the number of custom execution plans considered by the planner before the planner selects a generic execution plan.	X
edb_data_redaction	Session	Immediate	User	Enable data redaction.	X
edb_dynatune	Cluster	Restart	EPAS service account	Sets the edb utilization percentage.	X
edb_dynatune_profile	Cluster	Restart	EPAS service account	Sets the workload profile for dynatune.	X
edb_enable_pruning	Session	Immediate	User	Enables the planner to early-prune partitioned tables.	X
edb_log_every_bulk_value	Session	Immediate	Superuser	Sets the statements logged for bulk processing.	X
edb_max_resource_groups	Cluster	Restart	EPAS service account	Specifies the maximum number of resource groups for simultaneous use.	X
edb_max_spins_per_delay	Cluster	Restart	EPAS service account	Specifies the number of times a session will spin while waiting for a lock.	X
edb_redwood_date	Session	Immediate	User	Determines whether `DATE` should behave like a `TIMESTAMP` or not.	X
edb_redwood_greatest_least	Session	Immediate	User	Determines how `GREATEST` and `LEAST` functions should handle NULL parameters.	X
edb_redwood_raw_names	Session	Immediate	User	Return the unmodified name stored in the PostgreSQL system catalogs from Redwood interfaces.	X
edb_redwood_strings	Session	Immediate	User	Treat `NULL` as an empty string when concatenated with a text value.	X
edb_resource_group	Session	Immediate	User	Specifies the resource group to be used by the current process.	X
edb_sql_protect.enabled	Cluster	Reload	EPAS service account	Defines whether SQL/Protect should track queries or not.	X
edb_sql_protect.level	Cluster	Reload	EPAS service account	Defines the behavior of SQL/Protect when an event is found.	X
edb_sql_protect.max_protected_relations	Cluster	Restart	EPAS service account	Sets the maximum number of relations protected by SQL/Protect per role.	X
edb_sql_protect.max_protected_roles	Cluster	Restart	EPAS service account	Sets the maximum number of roles protected by SQL/Protect.	X

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
edb_sql_protect.max_queries_to_save	Cluster	Restart	EPAS service account	Sets the maximum number of offending queries to save by SQL/Protect.	X
edb_stmt_level_tx	Session	Immediate	User	Allows continuing on errors instead of requiring a transaction abort.	X
edb_wait_states.directory	Cluster	Restart	EPAS service account	The EDB wait states logs are stored in this directory.	X
edb_wait_states.retention_period	Cluster	Reload	EPAS service account	EDB wait state log files will be automatically deleted after retention period.	X
edb_wait_states.sampling_interval	Cluster	Reload	EPAS service account	The interval between two EDB wait state sampling cycles.	X
edbldr.empty_csv_field	Session	Immediate	Superuser	Specifies how EDB*Loader handles empty strings.	X
effective_cache_size	Session	Immediate	User	Sets the planner's assumption about the size of the disk cache.	
effective_io_concurrency	Session	Immediate	User	Number of simultaneous requests that can be handled efficiently by the disk subsystem.	
enable_bitmapscan	Session	Immediate	User	Enables the planner's use of bitmap-scan plans.	
enable_hashagg	Session	Immediate	User	Enables the planner's use of hashed aggregation plans.	
enable_hashjoin	Session	Immediate	User	Enables the planner's use of hash join plans.	
enable_hints	Session	Immediate	User	Enable optimizer hints in SQL statements.	X
enable_indexonlyscan	Session	Immediate	User	Enables the planner’s use of index-only-scan plans.	
enable_indexscan	Session	Immediate	User	Enables the planner's use of index-scan plans.	
enable_material	Session	Immediate	User	Enables the planner's use of materialization.	
enable_mergejoin	Session	Immediate	User	Enables the planner's use of merge join plans.	
enable_nestloop	Session	Immediate	User	Enables the planner's use of nested-loop join plans.	
enable_seqscan	Session	Immediate	User	Enables the planner's use of sequential-scan plans.	
enable_sort	Session	Immediate	User	Enables the planner's use of explicit sort steps.	
enable_tidscan	Session	Immediate	User	Enables the planner's use of TID scan plans.	
escape_string_warning	Session	Immediate	User	Warn about backslash escapes in ordinary string literals.	
event_source	Cluster	Restart	EPAS service account	Sets the application name used to identify PostgreSQL messages in the event log.	
exit_on_error	Session	Immediate	User	Terminate session on any error.	
external_pid_file	Cluster	Restart	EPAS service account	Writes the postmaster PID to the specified file.	
extra_float_digits	Session	Immediate	User	Sets the number of digits displayed for floating-point values.	
from_collapse_limit	Session	Immediate	User	Sets the FROM -list size beyond which subqueries are not collapsed.	
fsync	Cluster	Reload	EPAS service account	Forces synchronization of updates to disk.	
full_page_writes	Cluster	Reload	EPAS service account	Writes full pages to WAL when first modified after a checkpoint.	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
geqo	Session	Immediate	User	Enables genetic query optimization.	
geqo_effort	Session	Immediate	User	GEQO: effort is used to set the default for other GEQO parameters.	
geqo_generations	Session	Immediate	User	GEQO: number of iterations of the algorithm.	
geqo_pool_size	Session	Immediate	User	GEQO: number of individuals in the population.	
geqo_seed	Session	Immediate	User	GEQO: seed for random path selection.	
geqo_selection_bias	Session	Immediate	User	GEQO: selective pressure within the population.	
geqo_threshold	Session	Immediate	User	Sets the threshold of FROM items beyond which GEQO is used.	
gin_fuzzy_search_limit	Session	Immediate	User	Sets the maximum allowed result for exact search by GIN.	
hba_file	Cluster	Restart	EPAS service account	Sets the server's "hba" configuration file.	
hot_standby	Cluster	Restart	EPAS service account	Allows connections and queries during recovery.	
hot_standby_feedback	Cluster	Reload	EPAS service account	Allows feedback from a hot standby to the primary that will avoid query conflicts.	
huge_pages	Cluster	Restart	EPAS service account	Use of huge pages on Linux.	
icu_short_form	Database	Preset	n/a	Shows the ICU collation order configuration.	X
ident_file	Cluster	Restart	EPAS service account	Sets the server's "ident" configuration file.	
ignore_checksum_failure	Session	Immediate	Superuser	Continues processing after a checksum failure.	
ignore_system_indexes	Cluster/Session	Reload/Immediate	EPAS service account/User	Disables reading from system indexes. (Can also be set with PGOPTIONS at session start.)	
index_advisor.enabled	Session	Immediate	User	Enable Index Advisor plugin.	X
integer_datetimes	Cluster	Preset	n/a	Datetimes are integer based.	
intervalStyle	Session	Immediate	User	Sets the display format for interval values.	
join_collapse_limit	Session	Immediate	User	Sets the FROM -list size beyond which JOIN constructs are not flattened.	
krb_caseins_users	Cluster	Reload	EPAS service account	Sets whether Kerberos and GSSAPI user names should be treated as case-insensitive.	
krb_server_keyfile	Cluster	Reload	EPAS service account	Sets the location of the Kerberos server key file.	
lc_collate	Database	Preset	n/a	Shows the collation order locale.	
lc_ctype	Database	Preset	n/a	Shows the character classification and case conversion locale.	
lc_messages	Session	Immediate	Superuser	Sets the language in which messages are displayed.	
lc_monetary	Session	Immediate	User	Sets the locale for formatting monetary amounts.	
lc_numeric	Session	Immediate	User	Sets the locale for formatting numbers.	
lc_time	Session	Immediate	User	Sets the locale for formatting date and time values.	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
listen_addresses	Cluster	Restart	EPAS service account	Sets the host name or IP address(es) to listen to.	
local_preload_libraries	Cluster/Session	Reload/Immediate	EPAS service account/User	Lists shared libraries to preload into each backend. (Can also be set with PGOPTIONS at session start.)	
lock_timeout	Session	Immediate	User	Sets the maximum time allowed that a statement may wait for a lock.	
lo_compat_privileges	Session	Immediate	Superuser	Enables backward compatibility mode for privilege checks on large objects.	
log_autovacuum_min_duration	Cluster	Reload	EPAS service account	Sets the minimum execution time above which autovacuum actions will be logged.	
log_checkpoints	Cluster	Reload	EPAS service account	Logs each checkpoint.	
log_connections	Cluster/Session	Reload/Immediate	EPAS service account/User	Logs each successful connection. (Can also be set with PGOPTIONS at session start.)	
log_destination	Cluster	Reload	EPAS service account	Sets the destination for server log output.	
log_directory	Cluster	Reload	EPAS service account	Sets the destination directory for log files.	
log_disconnections	Cluster/Session	Reload/Immediate	EPAS service account/User	Logs end of a session, including duration. (Can also be set with PGOPTIONS at session start.)	
log_duration	Session	Immediate	Superuser	Logs the duration of each completed SQL statement.	
log_error_verbosity	Session	Immediate	Superuser	Sets the verbosity of logged messages.	
log_executor_stats	Session	Immediate	Superuser	Writes executor performance statistics to the server log.	
log_file_mode	Cluster	Reload	EPAS service account	Sets the file permissions for log files.	
log_filename	Cluster	Reload	EPAS service account	Sets the file name pattern for log files.	
log_hostname	Cluster	Reload	EPAS service account	Logs the host name in the connection logs.	
log_line_prefix	Cluster	Reload	EPAS service account	Controls information prefixed to each log line.	
log_lock_waits	Session	Immediate	Superuser	Logs long lock waits.	
log_min_duration_statement	Session	Immediate	Superuser	Sets the minimum execution time above which statements will be logged.	
log_min_error_statement	Session	Immediate	Superuser	Causes all statements generating error at or above this level to be logged.	
log_min_messages	Session	Immediate	Superuser	Sets the message levels that are logged.	
log_parser_stats	Session	Immediate	Superuser	Writes parser performance statistics to the server log.	
log_planner_stats	Session	Immediate	Superuser	Writes planner performance statistics to the server log.	
log_rotation_age	Cluster	Reload	EPAS service account	Automatic log file rotation will occur after N minutes.	
log_rotation_size	Cluster	Reload	EPAS service account	Automatic log file rotation will occur after N kilobytes.	
log_statement	Session	Immediate	Superuser	Sets the type of statements logged.	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
log_statement_stats	Session	Immediate	Superuser	Writes cumulative performance statistics to the server log.	
log_temp_files	Session	Immediate	Superuser	Log the use of temporary files larger than this number of kilobytes.	
log_timezone	Cluster	Reload	EPAS service account	Sets the time zone to use in log messages.	
log_truncate_on_rotation	Cluster	Reload	EPAS service account	Truncate existing log files of same name during log rotation.	
logging_collector	Cluster	Restart	EPAS service account	Start a subprocess to capture stderr output and/or csv logs into log files.	
maintenance_work_mem	Session	Immediate	User	Sets the maximum memory to be used for maintenance operations.	
max_connections	Cluster	Restart	EPAS service account	Sets the maximum number of concurrent connections.	
max_files_per_process	Cluster	Restart	EPAS service account	Sets the maximum number of simultaneously open files for each server process.	
max_function_args	Cluster	Preset	n/a	Shows the maximum number of function arguments.	
max_identifier_length	Cluster	Preset	n/a	Shows the maximum identifier length.	
max_index_keys	Cluster	Preset	n/a	Shows the maximum number of index keys.	
max_locks_per_transaction	Cluster	Restart	EPAS service account	Sets the maximum number of locks per transaction.	
max_pred_locks_per_transaction	Cluster	Restart	EPAS service account	Sets the maximum number of predicate locks per transaction.	
max_prepared_transactions	Cluster	Restart	EPAS service account	Sets the maximum number of simultaneously prepared transactions.	
max_replication_slots	Cluster	Restart	EPAS service account	Sets the maximum number of simultaneously defined replication slots.	
max_stack_depth	Session	Immediate	Superuser	Sets the maximum stack depth, in kilobytes.	
max_standby_archive_delay	Cluster	Reload	EPAS service account	Sets the maximum delay before canceling queries when a hot standby server is processing archived WAL data.	
max_standby_streaming_delay	Cluster	Reload	EPAS service account	Sets the maximum delay before canceling queries when a hot standby server is processing streamed WAL data.	
max_wal_senders	Cluster	Restart	EPAS service account	Sets the maximum number of simultaneously running WAL sender processes.	
max_wal_size	Cluster	Reload	EPAS service account	Sets the maximum size to which the WAL will grow between automatic WAL checkpoints. The default is 1GB.	
max_worker_processes	Cluster	Restart	EPAS service account	Maximum number of concurrent worker processes.	
min_wal_size	Cluster	Reload	EPAS service account	Sets the threshold at which WAL logs will be recycled rather than removed. The default is 80 MB.	
nls_length_semantics	Session	Immediate	Superuser	Sets the semantics to use for char, varchar, varchar2 and long columns.	X
odbc_lib_path	Cluster	Restart	EPAS service account	Sets the path for ODBC library.	X

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
optimizer_mode	Session	Immediate	User	Default optimizer mode.	X
oracle_home	Cluster	Restart	EPAS service account	Sets the path for the Oracle home directory.	X
password_encryption	Session	Immediate	User	Encrypt passwords.	
pg_prewarm.autoprewarm	Cluster	Restart	EPAS service account	Enables the autoprewarm background worker.	X
pg_prewarm.autoprewarm_interval	Cluster	Reload	EPAS service account	Sets the minimum number of seconds after which autoprewarm dumps shared buffers.	X
port	Cluster	Restart	EPAS service account	Sets the TCP port on which the server listens.	
post_auth_delay	Cluster/Session	Reload/Immediate	EPAS service account/User	Waits N seconds on connection startup after authentication. (Can also be set with PGOPTIONS at session start.)	
pre_auth_delay	Cluster	Reload	EPAS service account	Waits N seconds on connection startup before authentication.	
qreplace_function	Session	Immediate	Superuser	The function to be used by Query Replace feature. Note: For internal use only.	X
query_rewrite_enabled	Session	Immediate	User	Child table scans will be skipped if their constraints guarantee that no rows match the query.	X
query_rewrite_integrity	Session	Immediate	Superuser	Sets the degree to which query rewriting must be enforced.	X
quote_all_identifiers	Session	Immediate	User	When generating SQL fragments, quote all identifiers.	
random_page_cost	Session	Immediate	User	Sets the planner's estimate of the cost of a nonsequentially fetched disk page.	
restart_after_crash	Cluster	Reload	EPAS service account	Reinitialize server after backend crash.	
search_path	Session	Immediate	User	Sets the schema search order for names that are not schema-qualified.	
segment_size	Cluster	Preset	n/a	Shows the number of pages per disk file.	
seq_page_cost	Session	Immediate	User	Sets the planner's estimate of the cost of a sequentially fetched disk page.	
server_encoding	Database	Preset	n/a	Sets the server (database) character set encoding.	
server_version	Cluster	Preset	n/a	Shows the server version.	
server_version_num	Cluster	Preset	n/a	Shows the server version as an integer.	
session_preload_libraries	Session	Immediate but only at connection start	Superuser	Lists shared libraries to preload into each backend.	
session_replication_role	Session	Immediate	Superuser	Sets the session's behavior for triggers and rewrite rules.	
shared_buffers	Cluster	Restart	EPAS service account	Sets the number of shared memory buffers used by the server.	
shared_preload_libraries	Cluster	Restart	EPAS service account	Lists shared libraries to preload into server.	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
sql_inheritance	Session	Immediate	User	Causes subtables to be included by default in various commands.	
ssl	Cluster	Reload	EPAS service account	Enables SSL connections.	
ssl_ca_file	Cluster	Reload	EPAS service account	Location of the SSL certificate authority file.	
ssl_cert_file	Cluster	Reload	EPAS service account	Location of the SSL server certificate file.	
ssl_ciphers	Cluster	Reload	EPAS service account	Sets the list of allowed SSL ciphers.	
ssl_crl_file	Cluster	Reload	EPAS service account	Location of the SSL certificate revocation list file.	
ssl_ecdh_curve	Cluster	Reload	EPAS service account	Sets the curve to use for ECDH.	
ssl_key_file	Cluster	Reload	EPAS service account	Location of the SSL server private key file.	
ssl_prefer_server_ciphers	Cluster	Reload	EPAS service account	Give priority to server ciphersuite order.	
ssl_renegotiation_limit	Session	Immediate	User	Set the amount of traffic to send and receive before renegotiating the encryption keys.	
standard_conforming_strings	Session	Immediate	User	Causes '...' strings to treat backslashes literally.	
statement_timeout	Session	Immediate	User	Sets the maximum allowed duration of any statement.	
stats_temp_directory	Cluster	Reload	EPAS service account	Writes temporary statistics files to the specified directory.	
superuser_reserved_connections	Cluster	Restart	EPAS service account	Sets the number of connection slots reserved for superusers.	
synchronize_seqscans	Session	Immediate	User	Enable synchronized sequential scans.	
synchronous_commit	Session	Immediate	User	Sets immediate fsync at commit.	
synchronous_standby_names	Cluster	Reload	EPAS service account	List of names of potential synchronous standbys.	
syslog_facility	Cluster	Reload	EPAS service account	Sets the syslog "facility" to be used when syslog enabled.	
syslog_ident	Cluster	Reload	EPAS service account	Sets the program name used to identify PostgreSQL messages in syslog.	
tcp_keepalives_count	Session	Immediate	User	Maximum number of TCP keepalive retransmits.	
tcp_keepalives_idle	Session	Immediate	User	Time between issuing TCP keepalives.	
tcp_keepalives_interval	Session	Immediate	User	Time between TCP keepalive retransmits.	
temp_buffers	Session	Immediate	User	Sets the maximum number of temporary buffers used by each session.	
temp_file_limit	Session	Immediate	Superuser	Limits the total size of all temporary files used by each session.	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
temp_tablespaces	Session	Immediate	User	Sets the tablespace(s) to use for temporary tables and sort files.	
timed_statistics	Session	Immediate	User	Enables the recording of timed statistics.	X
timezone	Session	Immediate	User	Sets the time zone for displaying and interpreting time stamps.	
timezone_abbreviations	Session	Immediate	User	Selects a file of time zone abbreviations.	
trace_hints	Session	Immediate	User	Emit debug info about hints being honored.	X
trace_notify	Session	Immediate	User	Generates debugging output for LISTEN and NOTIFY.	
trace_recovery_messages	Cluster	Reload	EPAS service account	Enables logging of recovery-related debugging information.	
trace_sort	Session	Immediate	User	Emit information about resource usage in sorting.	
track_activities	Session	Immediate	Superuser	Collects information about executing commnands.	
track_activity_query_size	Cluster	Restart	EPAS service account	Sets the size reserved for pg_stat_activity.current_query, in bytes.	
track_counts	Session	Immediate	Superuser	Collects statistics on database activity.	
track_functions	Session	Immediate	Superuser	Collects function-level statistics on database activity.	
track_io_timing	Session	Immediate	Superuser	Collects timing statistics for database I/O activity.	
transaction_deferrable	Session	Immediate	User	Whether to defer a read-only serializable transaction until it can be executed with no possible serialization failures.	
transaction_isolation	Session	Immediate	User	Sets the current transaction's isolation level.	
transaction_read_only	Session	Immediate	User	Sets the current transaction's read-only status.	
transform_null_equals	Session	Immediate	User	Treats "expr=NULL" as "expr IS NULL".	
unix_socket_directories	Cluster	Restart	EPAS service account	Sets the directory where the Unix-domain socket will be created.	
unix_socket_group	Cluster	Restart	EPAS service account	Sets the owning group of the Unix-domain socket.	
unix_socket_permissions	Cluster	Restart	EPAS service account	Sets the access permissions of the Unix-domain socket.	
update_process_title	Session	Immediate	Superuser	Updates the process title to show the active SQL command.	
utl_encode.uudecode_redwood	Session	Immediate	User	Allows decoding of Oracle-created uuencoded data.	X
utl_file.umask	Session	Immediate	User	Umask used for files created through the UTL_FILE package.	X
vacuum_cost_delay	Session	Immediate	User	Vacuum cost delay in milliseconds.	
vacuum_cost_limit	Session	Immediate	User	Vacuum cost amount available before napping.	
vacuum_cost_page_dirty	Session	Immediate	User	Vacuum cost for a page dirtied by vacuum.	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
vacuum_cost_page_hit	Session	Immediate	User	Vacuum cost for a page found in the buffer cache.	
vacuum_cost_page_miss	Session	Immediate	User	Vacuum cost for a page not found in the buffer cache.	
vacuum_defer_cleanup_age	Cluster	Reload	EPAS service account	Number of transactions by which VACUUM and HOT cleanup should be deferred, if any.	
vacuum_freeze_min_age	Session	Immediate	User	Minimum age at which VACUUM should freeze a table row.	
vacuum_freeze_table_age	Session	Immediate	User	Age at which VACUUM should scan whole table to freeze tuples.	
vacuum_multixact_freeze_min_age	Session	Immediate	User	Minimum age at which VACUUM should freeze a MultiXactId in a table row.	
vacuum_multixact_freeze_table_age	Session	Immediate	User	Multixact age at which VACUUM should scan whole table to freeze tuples.	
wal_block_size	Cluster	Preset	n/a	Shows the block size in the write ahead log.	
wal_buffers	Cluster	Restart	EPAS service account	Sets the number of disk-page buffers in shared memory for WAL.	
wal_keep_segments	Cluster	Reload	EPAS service account	Sets the number of WAL files held for standby servers.	
wal_level	Cluster	Restart	EPAS service account	Set the level of information written to the WAL.	
wal_log_hints	Cluster	Restart	EPAS service account	Writes full pages to WAL when first modified after a checkpoint, even for non-critical modifications.	
wal_receiver_status_interval	Cluster	Reload	EPAS service account	Sets the maximum interval between WAL receiver status reports to the primary.	
wal_receiver_timeout	Cluster	Reload	EPAS service account	Sets the maximum wait time to receive data from the primary.	
wal_segment_size	Cluster	Preset	n/a	Shows the number of pages per write ahead log segment.	
wal_sender_timeout	Cluster	Reload	EPAS service account	Sets the maximum time to wait for WAL replication.	
wal_sync_method	Cluster	Reload	EPAS service account	Selects the method used for forcing WAL updates to disk.	
wal_writer_delay	Cluster	Reload	EPAS service account	WAL writer sleep time between WAL flushes.	
work_mem	Session	Immediate	User	Sets the maximum memory to be used for query workspaces.	
xloginsert_locks	Cluster	Restart	EPAS service account	Sets the number of locks used for concurrent xlog insertions.	
xmlbinary	Session	Immediate	User	Sets how binary values are to be encoded in XML.	
xmloption	Session	Immediate	User	Sets whether XML data in implicit parsing and serialization operations is to be considered as documents or content fragments.	
zero_damaged_pages	Session	Immediate	Superuser	Continues processing past damaged page headers.	

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

20.3.1.3 Configuration Parameters by Functionality

This section provides more detail for certain groups of configuration parameters.

The section heading for each parameter is followed by a list of attributes:

- `Parameter Type.` Type of values the parameter can accept. See Setting Configuration Parameters section for a discussion of parameter type values.
- `Default Value.` Default setting if a value is not explicitly set in the configuration file.
- `Range.` Permitted range of values.
- `Minimum Scope of Effect.` Smallest scope for which a distinct setting can be made. Generally, the minimal scope of a distinct setting is either the entire `cluster` (the setting is the same for all databases and sessions thereof, in the cluster), or `per session` (the setting may vary by role, database, or individual session). (This attribute has the same meaning as the `Scope of Effect` column in the table of Summary of Configuration Parameters).
- `When Value Changes Take Effect.` Least invasive action required to activate a change to a parameter's value. All parameter setting changes made in the configuration file can be put into effect with a restart of the database server; however certain parameters require a database server `restart`. Some parameter setting changes can be put into effect with a `reload` of the configuration file without stopping the database server. Finally, other parameter setting changes can be put into effect with some client side action whose result is `immediate`. (This attribute has the same meaning as the `When Takes Effect` column in the table of Summary of Configuration Parameters).
- `Required Authorization to Activate.` The type of user authorization to activate a change to a parameter's setting. If a database server restart or a configuration file reload is required, then the user must be a EPAS service account (`enterprisedb` or `postgres` depending upon the Advanced Server compatibility installation mode). (This attribute has the same meaning as the `Authorized User` column in the table of Summary of Configuration Parameters).

20.3.1.3.1 Top Performance Related Parameters

This section discusses the configuration parameters that have the most immediate impact on performance.

20.3.1.3.1.1 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: EPAS service account

Sets the amount of memory the database server uses for shared memory buffers. The default is typically 32 megabytes (`32MB`), but might be less if your kernel settings will not support it (as determined during `initdb`). This setting must be at least 128 kilobytes. (Non-default values of `BLCKSZ` change the minimum.) However, settings significantly higher than the minimum are usually needed 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. There are some workloads where even large settings for `shared_buffers` are effective, but because Advanced Server also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to `shared_buffers` will work better than a smaller amount.

On systems with less than 1GB of RAM, a smaller percentage of RAM is appropriate, so as to leave adequate space for the operating system (15% of

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

memory is more typical in these situations). Also, on Windows, large values for `shared_buffers` aren't as effective. You may find 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 Advanced Server to request more System V shared memory than your operating system's default configuration allows. See the section `Shared Memory and Semaphores` in the *PostgreSQL Core Documentation* for information on how to adjust those parameters, if necessary.

20.3.1.3.1.2 work_mem

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 to be used by internal sort operations and hash tables before writing to temporary disk files. The value defaults to one megabyte (`1MB`). Note that for a complex query, several sort or hash operations might be running in parallel; each operation will be allowed to use as much memory as this value specifies before it starts to write data into temporary files. Also, several running sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of `work_mem` ; it is necessary to keep this fact 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.

20.3.1.3.1.3 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 to be used by maintenance operations, such as `VACUUM` , `CREATE INDEX` , and `ALTER TABLE ADD FOREIGN KEY` . It defaults to 16 megabytes (`16MB`). Since only one of these operations can be executed at a time by a database session, 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Note that when autovacuum runs, up to `autovacuum_max_workers` times this memory may be allocated, 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 won't necessarily improve performance.

20.3.1.3.1.4 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: EPAS service account

The amount of memory used in shared memory for WAL data. The default is 64 kilobytes (`64kB`). The setting need only be large enough to hold the amount of WAL data generated by one typical transaction, since the data is written out to disk at every transaction commit.

Increasing this parameter might cause Advanced Server to request more System V shared memory than your operating system's default configuration allows. See the section “Shared Memory and Semaphores” in the *PostgreSQL Core Documentation* for information on how to adjust those parameters, if necessary.

Although even this very small setting does not always cause a problem, there are situations where it can result in extra `fsync` calls, and degrade overall system throughput. Increasing this value to 1MB or so can alleviate this problem. On very busy systems, an even higher value may be needed, up to a maximum of about 16MB. Like `shared_buffers` , this parameter increases Advanced Server’s initial shared memory allocation, so if increasing it causes an Advanced Server start failure, you will need to increase the operating system limit.

20.3.1.3.1.5 checkpoint_segments

Now deprecated; this parameter is not supported by Advanced Server.

20.3.1.3.1.6 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: EPAS service account

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Specifies the target of checkpoint completion as a fraction of total time between checkpoints. This spreads out the checkpoint writes while the system starts working towards the next checkpoint.

The default of 0.5 means aim to finish the checkpoint writes when 50% of the next checkpoint is ready. A value of 0.9 means aim to finish the checkpoint writes when 90% of the next checkpoint is done, thus throttling the checkpoint writes over a larger amount of time and avoiding spikes of performance bottlenecking.

20.3.1.3.1.7 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: EPAS service account

Maximum time between automatic WAL checkpoints, in seconds. The default is five minutes (`5min`). 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 the aforementioned adjustments to the checkpoint parameters is that your system will use a modest amount of additional disk space, and will take longer to recover in the event of a crash. However, for most users, this is a small price to pay for a significant performance improvement.

20.3.1.3.1.8 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: EPAS service account

`max_wal_size` specifies the maximum size that the WAL will reach between automatic WAL checkpoints. This is a soft limit; WAL size can exceed `max_wal_size` under special circumstances (when under a heavy load, a failing archive_command, or a high `wal_keep_segments` setting).

Increasing this parameter can increase the amount of time needed for crash recovery. This parameter can only be set in the `postgresql.conf` file or on the server command line.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

20.3.1.3.1.9 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: EPAS 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 ensures that enough WAL space is reserved to handle spikes in WAL usage (like when running large batch jobs). This parameter can only be set in the `postgresql.conf` file or on the server command line.

20.3.1.3.1.10 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: EPAS 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 (controllable by the `bgwriter_lru_maxpages` and `bgwriter_lru_multiplier` parameters). It then sleeps for `bgwriter_delay` milliseconds, and repeats.

The default value is 200 milliseconds (200ms). Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting `bgwriter_delay` to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10.

Typically, when tuning `bgwriter_delay`, it should be reduced from its default value. This parameter is rarely increased, except perhaps to save on power consumption on a system with very low utilization.

20.3.1.3.1.11 seq_page_cost

Parameter Type: Floating point

Default Value: 1

Range: 0 to 1.79769e+308

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 is part of a series of sequential fetches. The default is 1.0. This value can be overridden for a particular tablespace by setting the tablespace parameter of the same name. (Refer to the `ALTER TABLESPACE` command in the *PostgreSQL Core Documentation*.)

The default value assumes very little caching, so it's frequently 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 less than 1 (rather than its default value of 1) to see whether you get better query plans that way. If your database fits entirely within memory, you can lower this value much more, perhaps to 0.1.

20.3.1.3.1.12 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 non-sequentially-fetched disk page. The default is 4.0. This value can be overridden for a particular tablespace by setting the tablespace parameter of the same name. (Refer to the `ALTER TABLESPACE` command in the *PostgreSQL Core Documentation*.)

Reducing this value relative to `seq_page_cost` will cause the system to prefer index scans; raising it will make 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 `cpu_tuple_cost` and `cpu_index_tuple_cost` parameters.

The default value assumes very little caching, so it's frequently 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 (rather than its default of 4) to see whether you get better query plans that way. If your database fits entirely within memory, you can lower this value much more, perhaps to 0.1.

Although the system will let you do so, never set `random_page_cost` less than `seq_page_cost`. However, setting them equal (or very close to equal) makes sense if the database fits mostly or entirely within memory, since in that case there is no penalty for touching pages out of sequence. Also, in a heavily-cached database you should lower both values relative to the CPU parameters, since the cost of fetching a page already in RAM is much smaller than it would normally be.

20.3.1.3.1.13 effective_cache_size

Parameter Type: Integer

Default Value: 128MB

Range: 8kB to 17179869176kB

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 is available to a single query. This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. When setting this parameter you should consider both Advanced Server's shared buffers and the portion of the kernel's disk cache that will be used for Advanced Server data files. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by Advanced Server, nor does it reserve kernel disk cache; it is used only for estimation purposes. The default is 128 megabytes (`128MB`).

If this parameter is set too low, the planner may decide not to use an index even when it would be beneficial to do so. Setting `effective_cache_size` to 50% of physical memory is a normal, conservative setting. A more aggressive setting would be approximately 75% of physical memory.

20.3.1.3.1.14 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 transaction commit will wait for WAL records to be written to disk before the command returns a `success` indication to the client. The default, and safe, setting is on. 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, but the database state will be just the same as if those transactions had been 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 the section `Asynchronous Commit` in the *PostgreSQL Core Documentation* for information.

This parameter can be changed at any time; the behavior for any one transaction is determined by the setting in effect when it commits. It is 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` within the transaction.

20.3.1.3.1.15 edb_max_spins_per_delay

Parameter Type: Integer

Default Value: 1000

Range: 10 to 1000

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Minimum Scope of Effect: Per cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: EPAS service account

Use `edb_max_spins_per_delay` to specify the maximum number of times that a session will spin while waiting for a spin-lock. If a lock is not acquired, the session will sleep. If you do not specify an alternative value for `edb_max_spins_per_delay`, the server will enforce the default value of `1000`.

This may be useful for systems that use `NUMA` (non-uniform memory access) architecture.

20.3.1.3.1.16 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: EPAS service account

This parameter controls whether or not the database server should run `autoprewarm`, which is a background worker process that automatically 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 is that it shortens the warm up times after the server has been restarted by loading the data that has been dumped to disk before shutdown.

If `pg_prewarm.autoprewarm` is set to on, the `autoprewarm` worker is enabled. If the parameter is set to off, `autoprewarm` is disabled. The parameter is on by default.

Before `autoprewarm` can be used, you must add `$libdir/pg_prewarm` to the libraries listed in the `shared_preload_libraries` configuration parameter of the `postgresql.conf` file as shown by the following example:

```
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 which the `autoprewarm` background worker is launched immediately after the server has reached a consistent state.

The `autoprewarm` process will start loading blocks that were previously recorded in `$PGDATA/autoprewarm.blocks` until there is no free buffer space left in the buffer pool. In this manner, any new blocks that were loaded either by the recovery process or by the querying clients, are not replaced.

Once the `autoprewarm` process has finished loading buffers from disk, it will periodically dump shared buffers to disk at the interval specified by the `pg_prewarm.autoprewarm_interval` parameter. See the pg_prewarm.autoprewarm_interval for information on the `autoprewarm` background worker. Upon the next server restart, the `autoprewarm` process will prewarm shared buffers with the blocks that were last dumped to disk.

20.3.1.3.1.17 pg_prewarm.autoprewarm_interval

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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: EPAS service account

This is the minimum number of seconds after which the `autoprewarm` background worker dumps shared buffers to disk. The default is 300 seconds. If set to 0, shared buffers are not dumped at regular intervals, but only when the server is shut down.

See the pg_prewarm.autoprewarm for information on the `autoprewarm` background worker.

20.3.1.3.2 Resource Usage / Memory

The configuration parameters in this section control resource usage 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: EPAS service account

Determines how much of the host system's resources are to be used by the database server based upon the host machine's total available resources and the intended usage of the host machine.

When Advanced Server is initially installed, the `edb_dynatune` parameter is set in accordance with the selected usage of the host machine on which it was installed (i.e., development machine, mixed use machine, or dedicated server). For most purposes, there is no need for the database administrator to adjust the various configuration parameters in the `postgresql.conf` file in order to improve performance.

The `edb_dynatune` parameter can be set to any integer value between 0 and 100, inclusive. A value of 0, turns off the dynamic tuning feature thereby leaving the database server resource usage totally under the control of the other configuration parameters in the `postgresql.conf` file.

A low non-zero, value (e.g., 1 - 33) dedicates the least amount of the host machine's resources to the database server. This setting would be used 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. This setting might be used for a dedicated application server that may have a fixed number of other applications running on the same machine as Advanced Server.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The highest values (e.g., 67 - 100) dedicate most of the server's resources to the database server. This setting would be used for a host machine that is totally dedicated to running Advanced Server.

Once a value of `edb_dynatune` is selected, database server performance can be further fine-tuned by adjusting the other configuration parameters in the `postgresql.conf` file. Any adjusted setting overrides the corresponding value chosen by `edb_dynatune`. You can change the value of a parameter by un-commenting the configuration parameter, specifying the desired value, and restarting the database server.

edb_dynatune_profile

Parameter Type: Enum

Default Value: `oltp`

Range: `{oltp | reporting | mixed}`

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: EPAS service account

This parameter is used to control tuning aspects based upon the expected workload profile on the database server.

The following are the possible values:

- `oltp.` Recommended when the database server is processing heavy online transaction processing workloads.
- `reporting.` Recommended for database servers used for heavy data reporting.
- `mixed.` Recommended for servers that provide a mix of transaction processing and data reporting.

20.3.1.3.3 Resource Usage / EDB Resource Manager

The configuration parameters in this section control resource usage 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: EPAS service account

This parameter controls the maximum number of resource groups that can be used simultaneously by EDB Resource Manager. More resource groups can

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

be created than the value specified by `edb_max_resource_groups`, however, the number of resource groups in active use by processes in these groups cannot exceed this value.

Parameter `edb_max_resource_groups` should be set comfortably larger than the number of groups you expect to maintain so as not to run out.

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

Set the `edb_resource_group` parameter to the name of the resource group to which the current session is to be controlled by EDB Resource Manager according to the group's resource type settings.

If the parameter is not set, then the current session does not utilize EDB Resource Manager.

20.3.1.3.4 Query Tuning

This section describes the configuration parameters 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

Optimizer hints embedded in SQL commands are utilized when `enable_hints` is on. Optimizer hints are ignored when this parameter is off.

20.3.1.3.5 Query Tuning / Planner Method Configuration

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

This section describes the configuration parameters used for planner method configuration.

edb_custom_plan_tries

Parameter Type: Integer

Default Value: 5

Range: -1 to 100

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

This configuration parameter controls the number of custom execution plans considered by the planner before the planner settles on a generic execution plan.

When a client application repeatedly executes a prepared statement, the server may decide to evaluate several execution plans before deciding to choose a *custom* plan or a *generic* plan.

- A custom plan is a plan built for a specific set of parameter values.
- A generic plan is a plan that will work with any set of parameter values supplied by the client application.

By default, the optimizer will generate five custom plans before evaluating a generic plan. That means that if you execute a prepared statement six times, the optimizer will generate five custom plans, then one generic plan, and then decide whether to stick with the generic plan.

In certain workloads, this extra planning can have a negative impact on performance. You can adjust the `edb_custom_plan_tries` configuration parameter to decrease the number of custom plans considered before evaluating a generic plan.

Setting `edb_custom_plan_tries` to `0` will effectively disable custom plan generation.

Setting `edb_custom_plan_tries` to `-1` allows custom plan generation for an infinite number of tries.

Consider the following query:

```
PREPARE custQuery AS SELECT * FROM customer WHERE salesman >= $1
```

The `$1` token in this query is a parameter marker - the client application must provide a value for each parameter marker each time the statement executes.

If an index has been defined on `customer.salesman`, the optimizer may choose to execute this query using a sequential scan, or using an index scan. In some cases, an index is faster than a sequential scan; in other cases, the sequential scan will win. The optimal plan will depend on the distribution of salesman values in the table and on the search value (the value provided for the $1 parameter).

When the client application repeatedly executes the `custQuery` prepared statement, the optimizer will generate some number of parameter-value-specific execution plans (custom plans), followed by a generic plan or to generate custom plans for each execution. The decision process takes into account not only the cost of executing the plans, but the cost of generating custom plans as well.

edb_enable_pruning

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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

When set to `TRUE` , `edb_enable_pruning` allows the query planner to early-prune partitioned tables. `Early-pruning` means that the query planner can `prune` (i.e., ignore) partitions that would not be searched in a query `before` generating query plans. This helps improve performance time as it eliminates the generation of query plans of partitions that would not be 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 upon the nature of the query in the `WHERE` clause. Early-pruning can be utilized in only simple queries with constraints of the type `WHERE column = literal` (e.g., `WHERE deptno = 10)` .

Early-pruning is not used for more complex queries such as `WHERE column = expression` (e.g., `WHERE deptno = 10 + 5)` .

20.3.1.3.6 Reporting and Logging / What to Log

The configuration parameters in this section control reporting and logging.

trace_hints

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

Use with the optimizer hints feature to provide more detailed information regarding whether or not a hint was used by the planner. Set the `client_min_messages` and `trace_hints` configuration parameters as follows:

```
SET client_min_messages TO info;
SET trace_hints TO true;
```

The `SELECT` command with the `NO_INDEX` hint shown below illustrates the additional information produced when the aforementioned configuration parameters are set.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 EXPLAIN SELECT /*+ NO_INDEX(accounts accounts_pkey) */ * FROM accounts
WHERE aid = 100;

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 = 100)
(2 rows)
```

edb_log_every_bulk_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 Advanced Server log file and the EDB Audit log file. However, logging each and every statement in bulk processing is costly. This can be controlled by the `edb_log_every_bulk_value` configuration parameter. When set to `true`, each and every statement in bulk processing is logged. When set to `false`, a log message is recorded once per bulk processing. In addition, the duration is emitted once per bulk processing. Default is set to `false`.

20.3.1.3.7 Auditing Settings

This section describes configuration parameters used by the Advanced Server database auditing feature.

20.3.1.3.7.1 edb_audit

Parameter Type: Enum

Default Value: `none`

Range: `{none | csv | xml}`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

Enables or disables database auditing. The values `xml` or `csv` will enable database auditing. These values represent the file format in which auditing information will be captured. `none` will disable database auditing and is also the default.

20.3.1.3.7.2 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: EPAS service account

Specifies the directory where the audit log files will be created. The path of the directory can be absolute or relative to the Advanced Server `data` directory.

20.3.1.3.7.3 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: EPAS service account

Specifies the file name of the audit file where the auditing information will be stored. The default file name will be `audit-%Y%m%d_%H%M%S`. The escape sequences, `%Y, %m` etc., will be replaced by the appropriate current values according to the system date and time.

20.3.1.3.7.4 edb_audit_rotation_day

Parameter Type: String

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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: EPAS 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.

20.3.1.3.7.5 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: EPAS service account

Specifies a file size threshold in megabytes when file rotation will be forced to occur. The default value is 0MB. If the parameter is commented out or set to 0, rotation of the file on a size basis will not occur.

20.3.1.3.7.6 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: EPAS service account

Specifies the rotation time in seconds when a new log file should be created. To disable this feature, set this parameter to 0.

20.3.1.3.7.7 edb_audit_connect

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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: EPAS 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`.

20.3.1.3.7.8 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: EPAS 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`.

20.3.1.3.7.9 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 | all} ...`

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

This configuration parameter is used to specify auditing of different categories of SQL statements as well as those statements related to specific SQL commands. To log errors, set the parameter value to `error`. To audit all DDL statements such as `CREATE TABLE`, `ALTER TABLE`, etc., 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`alter`). In addition, the object type may be specified following the command such as `create table`, `create view`, `drop role`, etc. All modification statements such as `INSERT`, `UPDATE`, `DELETE` or `TRUNCATE` can be audited by setting `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. Setting the value to `all` will audit every statement while `none` disables this feature.

20.3.1.3.7.10 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

Use `edb_audit_tag` to specify a string value that will be included in the audit log when the `edb_audit` parameter is set to `csv` or `xml`.

20.3.1.3.7.11 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: EPAS service account

Specifies whether the audit log information is to be recorded 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` (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 only for Advanced Server running on a Linux host, and is not supported on Windows systems.

> **Note**
>
> In recent Linux versions, `syslog` has been replaced by `rsyslog` and the configuration file is in `/etc/rsyslog.conf`.

20.3.1.3.7.12 edb_log_every_bulk_value

For information on `edb_log_every_bulk_value`, see the edb_log_every_bulk_value.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

20.3.1.3.8 Client Connection Defaults / Locale and Formatting

This section describes configuration parameters affecting 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

The configuration parameter `icu_short_form` is a parameter containing the default ICU short form name assigned to a database or to the Advanced Server instance. See Unicode Collation Algorithm for general information about the ICU short form and the Unicode Collation Algorithm.

This configuration parameter is set either when the `CREATE DATABASE` command is used with the `ICU_SHORT_FORM` parameter in which case the specified short form name is set and appears in the `icu_short_form` configuration parameter when connected to this database, or when an Advanced Server instance is created with the `initdb` command used with the `--icu_short_form` option in which case the specified short form name is set and appears in the `icu_short_form` configuration parameter when connected to a database in that Advanced Server instance, and the database does not override it with its own `ICU_SHORT_FORM` parameter with a different short form.

Once established in the manner described, the `icu_short_form` configuration parameter cannot be changed.

20.3.1.3.9 Client Connection Defaults / Statement Behavior

This section describes configuration parameters affecting 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Sets the fillfactor for a table when the `FILLFACTOR` storage parameter is omitted from a `CREATE TABLE` command.

The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller `fillfactor` is specified, `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 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, but in heavily updated tables smaller fillfactors are appropriate.

edb_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.

The default setting is `TRUE` so 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.

If the parameter is disabled by setting it to `FALSE`, then the following occurs:

- Superusers and table owner still bypass data redaction.
- All other users will get an error.

For information on data redaction, see the *EDB Postgres Advanced Server Security Features Guide*.

20.3.1.3.10 Client Connection Defaults / Other Defaults

The parameters in this section set other miscellaneous client connection defaults.

oracle_home

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Cluster

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

When Value Changes Take Effect: Restart

Required Authorization to Activate: EPAS service account

Before creating a link to an Oracle server, you must direct 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 will override the `LD_LIBRARY_PATH` environment variable in Linux and `PATH` environment variable in Windows.

> **Note**
>
> The `oracle_home` configuration parameter should provide the correct path to the Oracle client i.e. `OCI` library.

To set the `oracle_home` configuration parameter in the `postgresql.conf` file, edit the file, adding the following line:

```
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`.

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: EPAS service account

If you will be using an ODBC driver manager, and if it is installed in a non-standard location, you 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 (typically `libodbc.so`).

20.3.1.3.11 Compatibility Options

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The configuration parameters described in this section 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

When `DATE` appears as the data type of a column in the commands, it is translated to `TIMESTAMP` at the time the table definition is stored in the database if the configuration parameter `edb_redwood_date` is set to `TRUE`. Thus, a time component will also be 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 such as 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, it is always internally translated to a `TIMESTAMP` and thus, can handle a time component if present.

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);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
greatest
----------

(1 row)
```

When `edb_redwood_greatest_least` is set to `FALSE`, null parameters are ignored except when all parameters are null in which case null is returned by the functions.

```
 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 its default value of `FALSE`, database object names such as table names, column names, trigger names, program names, user names, etc. 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 exactly as they are stored in the PostgreSQL system catalogs when viewed from the Redwood catalogs. Thus, names created without enclosing quotation marks appear in lowercase as expected in PostgreSQL. Names created with enclosing quotation marks appear exactly 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;
edb=# \c - reduser
Password for user reduser:
You are now connected to database "edb" as user "reduser".
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 uppercase except for the `Mixed_Case` name, which appears as created and also with enclosing quotation marks.

```
edb=> SELECT * FROM USER_TABLES;
 schema_name  | table_name   | tablespace_name  | status  | temporary
--------------+--------------+------------------+---------+-----------
 REDUSER      | ALL_LOWER    |                  | VALID   | N
 REDUSER      | ALL_UPPER    |                  | VALID   | N
 REDUSER      | "Mixed_Case" |                  | VALID   | N
(3 rows)
```

When viewed with `edb_redwood_raw_names` set to `TRUE`, the names appear in lowercase except for the `Mixed_Case` name, which appears as created, but now without the enclosing quotation marks.

```
 edb=> SET edb_redwood_raw_names TO true;
SET
edb=> SELECT  * FROM USER_TABLES;
 schema_name | table_name | tablespace_name  | status | temporary
-------------+------------+------------------+--------+-----------
reduser      | all_lower  |                  | VALID  | N
reduser      | all_upper  |                  | VALID  | N
reduser      | Mixed_Case |                  | VALID  | N
(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';
 schemaname | tablename  | tableowner
------------+------------+------------
reduser     | all_lower  | reduser
reduser     | all_upper  | reduser
reduser     | Mixed_Case | reduser
(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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Required Authorization to Activate: Session user

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 gives a null result.

The following example illustrates the difference.

The sample application contains a table of employees. This table has a column named `comm` that is 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.

```
 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 3,000.00
 KING 5,000.00
 TURNER 1,500.00 .00
 ADAMS 1,100.00
 JAMES 950.00
 FORD 3,000.00
 MILLER 1,300.00
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(14 rows)
```

edb_stmt_level_tx

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

The term `statement level transaction isolation` describes the behavior whereby when a runtime error occurs in a SQL command, 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 have not yet been committed or rolled back are pending until a `COMMIT` or `ROLLBACK` command is executed.

In 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 be started.

If `edb_stmt_level_tx` is set to `TRUE`, then an exception will not automatically roll back prior uncommitted database updates. If `edb_stmt_level_tx` is set to `FALSE`, then an exception will roll back uncommitted database updates.

> **Note**
>
> Use `edb_stmt_level_tx` set to `TRUE` only when absolutely necessary, as this may cause a negative performance impact.

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. Note that in PSQL, the command `\set AUTOCOMMIT off` must be issued, otherwise every statement commits automatically defeating the purpose of this demonstration of the effect of `edb_stmt_level_tx`.

```
 \set AUTOCOMMIT off
SET edb_stmt_level_tx TO off;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
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 > 9000;

empno  | ename | deptno
-------+-------+--------
(0 rows)
```

In the following example, with `edb_stmt_level_tx` set to `TRUE`, the first `INSERT` command has not been rolled back after the error on the second `INSERT` command. At this point, the first INSERT command can either be committed or rolled back.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 \set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
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 > 9000;

empno  | ename | deptno
-------+-------+--------
  9001 | JONES |     40
(1 row)

COMMIT;
```

A `ROLLBACK` command could have been issued instead of the `COMMIT` command in which case the insert of employee number `9001` would have been rolled back as well.

db_dialect

Parameter Type: Enum

Default Value: `postgres`

Range: `{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`, 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` then `sys`, giving the PostgreSQL catalog the highest precedence. When set to `redwood`, the namespace precedence is `sys` 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 can then be referenced in SQL commands.

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:

Hint	Description
`ALL_ROWS`	Optimizes for retrieval of all rows of the result set.
`CHOOSE`	Does no default optimization based on assumed number of rows to be retrieved from the result set. This is the default.
`FIRST_ROWS`	Optimizes for retrieval of only the first row of the result set.
`FIRST_ROWS_10`	Optimizes for retrieval of the first 10 rows of the results set.
`FIRST_ROWS_100`	Optimizes for retrieval of the first 100 rows of the result set.
`FIRST_ROWS_1000`	Optimizes for retrieval of the first 1000 rows of the result set.

These optimization modes are based upon the assumption that the client submitting the SQL command is interested in viewing only the first `n` rows of the result set and will then abandon the remainder of the result set. Resources allocated to the query are adjusted as such.

20.3.1.3.12 Customized Options

In previous releases of Advanced Server, the `custom_variable_classes` was required by those parameters not normally known to be added by add-on modules (such as procedural languages).

custom_variable_classes

The `custom_variable_classes` parameter is deprecated in Advanced Server 9.2; parameters that previously depended on this parameter no longer require its support.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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: EPAS 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: EPAS 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

Provides the capability to temporarily suspend Index Advisor in an EDB-PSQL or PSQL session. The Index Advisor plugin, `index_advisor`, must be loaded in the EDB-PSQL or PSQL session in order to use the `index_advisor.enabled` configuration parameter.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The Index Advisor plugin can be loaded as shown by the following example:

```
 $ psql -d edb -U enterprisedb
Password for user enterprisedb:
psql (11.0.0)
Type "help" for help.

edb=# LOAD 'index_advisor';
LOAD
```

Use the `SET` command to change the parameter setting to control whether or not Index Advisor generates an alternative query plan as shown by the following example:

```
 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)
```

edb_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: EPAS service account

Controls whether or not 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 are ready to begin monitoring with SQL/Protect set this parameter to on.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

edb_sql_protect.level

Parameter Type: Enum

Default Value: `passive`

Range: `{learn | passive | active}`

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

Sets the action taken by SQL/Protect when a SQL statement is issued by a protected role.

The `edb_sql_protect.level` configuration parameter can be set to one of the following values to use either learn mode, passive mode, or active mode:

- **learn.** Tracks the activities of protected roles and records the relations used by the roles. This is used 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 does not stop any SQL statements from executing. This is the next step after SQL/Protect has learned the expected behavior of the protected roles. This essentially behaves in intrusion detection mode and can be run in production when properly monitored.
- **active.** Stops all invalid statements for a protected role. This behaves as a SQL firewall preventing 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, but it tracks the blocked queries allowing administrators to be alerted before the attacker finds an alternate method of penetrating the system.

If you are 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: EPAS service account

Sets the maximum number of relations that can be protected per role. Please note the total number of protected relations for the server will be 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"
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
2014-07-18 16:04:12 EDT HINT: Value exceeds integer range.
```

The database server starts successfully, but with `edb_sql_protect.max_protected_relations` set to the default value of 1024.

Though 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 will be acceptable. If the value is such that the space in shared memory cannot 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 such cases, reduce the parameter value until the database server can be started successfully.

edb_sql_protect.max_protected_roles

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: EPAS service account

Sets the maximum number of roles that can be protected.

Every protected role consumes space in shared memory. Please note that the server will reserve 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"
2014-07-18 16:04:12 EDT HINT: Value exceeds integer range.
```

The database server starts successfully, but with `edb_sql_protect.max_protected_roles` set to the default value of 64.

Though 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 will be acceptable. If the value is such that the space in shared memory cannot

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 such cases, reduce the parameter value until the database server can be started 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: EPAS service account

Sets the maximum number of offending queries to save in view `edb_sql_protect_queries`.

Every query that is saved 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 for
parameter "edb_sql_protect.max_queries_to_save" (100 .. 2147483647)
```

The database server starts successfully, but with `edb_sql_protect.max_queries_to_save` set to the default value of 5000.

Though 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 will be acceptable. If the value is such that the space in shared memory cannot 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.
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

In such cases, reduce the parameter value until the database server can be started 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: EPAS service account

Sets the directory path where the EDB wait states log files are stored. The specified path should be a full, absolute path and not a relative path. However, the default setting is `edb_wait_states`, which makes `$PGDATA/edb_wait_states` the default directory location. For information on EDB wait states see `EDB Wait States` under Performance Analysis and Tuning.

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: EPAS service account

Sets the period of time after which the log files for EDB wait states will be deleted. The default is 604800 seconds, which is 7 days. For information on EDB wait states see `EDB Wait States` under Performance Analysis and Tuning.

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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Required Authorization to Activate: EPAS service account

Sets the timing interval between two sampling cycles for EDB wait states. The default setting is 1 second. For information on EDB wait states see `EDB Wait States` under Performance Analysis and Tuning.

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 will treat an empty string. The valid values for the `edbldr.empty_csv_field` parameter are:

Parameter Setting	EDB*Loader Behavior
`NULL`	An empty field is treated as `NULL`.
`empty_string`	An empty field is treated as a string of length zero.
`pgsql`	An empty field is treated as a `NULL` if it does not contain quotes and as an empty string if it contains quotes.

For more information about the `edbldr.empty_csv_field` parameter in EDB*Loader, see the *Database Compatibility for Oracle Developers Tools and Utilities Guide* at the EnterpriseDB website:

https://www.enterprisedb.com/docs

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`, Advanced Server's `UTL_ENCODE.UUDECODE` function can decode uuencoded data that was created by the Oracle implementation

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

of the `UTL_ENCODE.UUENCODE` function.

When set to the default setting of `FALSE` , Advanced Server's `UTL_ENCODE.UUDECODE` function can decode uuencoded data created by 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

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

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 is for usage only within the Advanced Server `UTL_FILE` package.

> **Note**
>
> The `utl_file.umask` parameter is not supported on Windows systems.

The value specified for `utl_file.umask` is a 3 or 4-character octal string that would be 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 usage of the `umask` command.)

The following shows the results of the default `utl_file.umask` setting of 0077. Note that all permissions are denied on users belonging to the `enterprisedb` group as well as all other users. Only user `enterprisedb` has read and write permissions on the file.

```
-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile
```

20.3.1.3.13 Ungrouped

Configuration parameters in this section apply to Advanced Server only and are for a specific, limited purpose.

nls_length_semantics

Parameter Type: Enum

Default Value: `byte`

Range: `{byte | char}`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Superuser

This parameter has no effect in Advanced Server.

For example, the form of the `ALTER SESSION` command is accepted in Advanced Server without throwing a syntax error, but does not alter the session environment:

```
ALTER SESSION SET nls_length_semantics = char;
```

> **Note**
>
> Since the setting of this parameter has no effect on the server environment, it does not 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 Advanced Server.

For example, the following form of the `ALTER SESSION` command is accepted in Advanced Server without throwing a syntax error, but does not alter the session environment:

```
ALTER SESSION SET query_rewrite_enabled = force;
```

> **Note**
>
> Since the setting of this parameter has no effect on the server environment, it does not appear in the system view `pg_settings`.

query_rewrite_integrity

Parameter Type: Enum

Default Value: `enforced`

Range: `{enforced | trusted | stale_tolerated}`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Superuser

This parameter has no effect in Advanced Server.

For example, the following form of the `ALTER SESSION` command is accepted in Advanced Server without throwing a syntax error, but does not alter the session environment:

```
ALTER SESSION SET query_rewrite_integrity = stale_tolerated;
```

> **Note**
>
> Since the setting of this parameter has no effect on the server environment, it does not 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 the collection of timing data for the Dynamic Runtime Instrumentation Tools Architecture (DRITA) feature. When set to on, timing data is collected.

> **Note**
>
> When Advanced Server is installed, the `postgresql.conf` file contains an explicit entry setting `timed_statistics` to off. If this entry is commented out letting `timed_statistics` to default, and the configuration file is reloaded, timed statistics collection would be turned on.

20.3.2 Index Advisor

The Index Advisor utility helps determine which columns you should index to improve performance in a given workload. Index Advisor considers B-tree (single-column or composite) index types, and does not identify other index types (GIN, GiST, Hash) that may improve performance. Index Advisor is installed with EDB Postgres Advanced Server.

Index Advisor works with 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.

There are three ways to use Index Advisor to analyze SQL queries:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Invoke the Index Advisor utility program, supplying a text file containing the SQL queries that you wish to analyze; Index Advisor will generate 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 client. When accessed via 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 the Using the Index Advisor section in the Postgres Enterprise Manager documentation.

Index Advisor will attempt to make indexing recommendations on `INSERT`, `UPDATE`, `DELETE` and `SELECT` statements. When invoking Index Advisor, you supply the workload in the form of a set of queries (if you are providing the command in an SQL file) or an `EXPLAIN` statement (if you are specifying the SQL statement at the psql command line). Index Advisor displays the query plan and estimated execution cost for the supplied query, but does not actually 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, and 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 actually 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 forego running the script, Index Advisor will log recommendations in a temporary table that is available only for the duration of the Index Advisor session.

20.3.2.1 Index Advisor Components

The Index Advisor shared library interacts with the query planner to make indexing recommendations. On Windows, the Advanced Server installer creates the following shared library in the `libdir` subdirectory of your Advanced Server home directory. For Linux, install the `edb-asxx-server-indexadvisor` RPM package where `xx` is the Advanced Server version number.

On Linux:

```
index_advisor.so
```

On Windows:

```
index_advisor.dll
```

Please note that libraries in the `libdir` directory can only be loaded by a superuser. 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 Advanced Server home directory). Only a trusted non-superuser should be allowed 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`pg_advise_index` is a utility program that reads a user-supplied input file containing SQL queries and produces a text file containing `CREATE INDEX` statements that can be used to create the indexes recommended by the Index Advisor. The `pg_advise_index` program is located in the `bin` subdirectory of the Advanced Server home directory.

> **Note**
>
> `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 does not find the `index_advisor_log` table in the user's search path, Index Advisor will store any indexing recommendations in a temporary table of the same name. The temporary table exists only for the duration of the current session.

`show_index_recommendations()`

`show_index_recommendations()` is 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, while the result set returned by the `show_index_recommendations()` function are limited to a specified session.

20.3.2.2 Index Advisor Configuration

Index Advisor does not require configuration to generate recommendations that are available only for the duration of the current session; to store the results of multiple sessions, you must create the `index_advisor_log` table (where Advanced Server will store Index Advisor recommendations). To create the `index_advisor_log` table , you must run the `index_advisor.sql` script.

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 the Index Advisor.

1. Place the selected schema at the start of your `search_path` parameter. For example, if your search path is currently:

   ```
   search_path=public, accounting
   ```

   and you want the Index Advisor objects to be created 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:

   ```
   # running as the database superuser using psql
   create extension index_advisor
   ;
   ```

   > **Note**
   >
   > If you're using Index Advisor in an earlier version of EDB Postgres Advanced Server and upgrading to the latest version, then use this

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> command to create the index_advisor extension:
>
> ```
> # running as the database superuser using psql
> create extension index_advisor version 1.1;
> ```

This command creates the extension and links any of the old database objects to it.

3. Grant privileges on the `index_advisor_log` table to all Index Advisor users; this step is not 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.

The following example demonstrates the creation of the Index Advisor database objects in a schema named `ia` , which will then be accessible to an Index Advisor user with user name `ia_user` :

```
# running as enterprisedb 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 `ia_user` 's `search_path` parameter.

20.3.2.3 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 (executed by the `pg_advise_index()` function). After analyzing the workload, Index Advisor will either store the result set in 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**
>
> You should not run Index Advisor in read-only transactions.

The following examples assume that superuser `enterprisedb` is the Index Advisor user, and the Index Advisor database objects have been created in a schema in the `search_path` of superuser `enterprisedb` .

The examples in the following sections use the table created with the statement shown below:

```
CREATE TABLE t( a INT, b INT );
INSERT INTO t SELECT s, 99999 - s FROM generate_series(0,99999) AS s;
ANALYZE t;
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The resulting table contains the following rows:

```
 a     | b
-------+-------
  0    | 99999
  1    | 99998
  2    | 99997
  3    | 99996
       .
       .
       .
99997 | 2
99998 | 1
99999 | 0
```

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 that will be executed by `pg_advise_index`; the queries may be on the same line, or on separate lines, but each query must be terminated by a semicolon. Queries within the file should not begin with the `EXPLAIN` keyword.

The following example shows the contents of a sample `workload.sql` file:

```
 SELECT * FROM t WHERE a = 500;
SELECT * FROM t WHERE b < 1000;
```

Run the `pg_advise_index` program as shown in the code sample below:

```
 $ 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`

`-s` is an optional parameter that limits the maximum size of the indexes recommended by Index Advisor. If Index Advisor does not return a result set, `-s` may 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
create index idx_t_1 on t (a);
create index idx_t_2 on t (b);
```

You can create the recommended indexes at the psql command line with the `CREATE INDEX` statements in the file, or 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
create index idx_t_2 on t (b);
CREATE INDEX
```

> **Note**
>
> `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`.

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; the following steps detail loading the Index Advisor plugin and using 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 enterprisedb
...
edb=# LOAD 'index_advisor';
LOAD
```

2. Use the `edb-psql` command line to invoke each SQL command that you would like Index Advisor to analyze. Index Advisor stores any recommendations for the queries in the `index_advisor_log` table. If the `index_advisor_log` table does not exist in the user's `search_path`, a temporary table is created with the same name. This temporary table exists only for the duration of the user's session.

After loading the Index Advisor plugin, Index Advisor will analyze all SQL statements and log any indexing recommendations for the duration of the session.

If you would like Index Advisor to analyze a query (and make indexing recommendations) without actually executing the query, preface the SQL statement with the `EXPLAIN` keyword.

If you do not preface the statement with the `EXPLAIN` keyword, Index Advisor will analyze the statement while the statement executes, writing the indexing recommendations to the `index_advisor_log` table for later review.

In the example that follows, the `EXPLAIN` statement displays the normal query plan, followed by the query plan of the same query, if the query were using the recommended hypothetical index:

```
edb=# 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)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(6 rows)

edb=# EXPLAIN SELECT * FROM t WHERE a = 100;
                        QUERY PLAN
-----------------------------------------------------------------------------
Seq Scan on t (cost=0.00..1693.00 rows=1 width=8)
  Filter: (a = 100)
Result (cost=0.00..8.28 rows=1 width=8)
  One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text
  -> Index Scan using "<hypothetical-index>:3" on t
     (cost=0.00..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:

```
edb=# SET index_advisor.enabled TO off;
SET
```

To enable Index Advisor, set the parameter to `on`:

```
edb=# SET index_advisor.enabled TO on;
SET
```

20.3.2.4 Reviewing the Index Advisor Recommendations

There are several ways to review the index recommendations generated by Index Advisor. You can:

- Query the `index_advisor_log` table.
- Run the `show_index_recommendations` function.
- 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 do not know the process ID of your current session, passing a value of `NULL` will also return a result set for the current session.

The following code fragment shows an example of a row in a result set:

```
edb=# SELECT show_index_recommendations(null);
                      show_index_recommendations
-----------------------------------------------------------------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
create index idx_t_a on t(a);/* size: 2184 KB, benefit: 3040.62,
gain: 1.39222666981456 */
(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 will result 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.

Column	Type	Description
`reloid`	`oid`	OID of the base table for the index
`relname`	`name`	Name of the base table for the index
`attrs`	`integer[]`	Recommended index columns (identified by column number)
`benefit`	`real`	Calculated benefit of the index for this query
`index_size`	`integer`	Estimated index size in disk-pages
`backend_pid`	`integer`	Process ID of the process generating this recommendation
`timestamp`	`timestamp`	Date/Time when the recommendation was generated

You can query the `index_advisor_log` table at the psql command line. The following 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 below) by a different `backend_pid` value. For each session, Index Advisor generated two index recommendations.

```
edb=# SELECT * FROM index_advisor_log;
 reloid  | relname | attrs | benefit | index_size | backend_pid | timestamp
---------+---------+-------+---------+------------+-------------+------------
----------------------
 16651   |    t    |  {1}  | 1684.72 |    2184    |     3442    | 22-MAR-11
16:44:32.712638 -04:00
 16651   |    t    |  {2}  | 1655.52 |    2184    |     3442    | 22-MAR-11
16:44:32.759436 -04:00
 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
(4 rows)
```

Index Advisor added the first two rows to the table after analyzing the following two queries executed by the `pg_advise_index` utility:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
SELECT * FROM t WHERE a = 500;
SELECT * FROM t WHERE b < 1000;
```

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):

```
edb=# 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)

edb=# EXPLAIN SELECT * FROM t WHERE a = 100;
                                  QUERY PLAN
------------------------------------------------------------------------------
----------
Seq Scan on t (cost=0.00..1693.00 rows=1 width=8)
  Filter: (a = 100)
Result (cost=0.00..8.28 rows=1 width=8)
  One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text
  -> Index Scan using "<hypothetical-index>:3" on t (cost=0.00..8.28
rows=1 width=8)
           Index Cond: (a = 100)
(6 rows)
```

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:

```
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>)
```

and the benefit is added to the table:

```
benefit = 1693.00 - 8.28

benefit = 1684.72
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

You can delete rows from the `index_advisor_log` table when you no longer have the 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 as shown below:

```
SELECT * FROM index_recommendations;
```

Using the example shown in the previous section (`Querying the index_advisor_log Table`), the `index_recommendations` view displays the following:

```
edb=# SELECT * FROM index_recommendations;
 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 */
(3 rows)
```

Within 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 as follows:

```
 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)
```

So 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 as follows:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
benefit = (benefit from 1st query) + (benefit from 2nd query)
benefit = 1355.9 + 1684.72
benefit = 3040.62
```

and

```
gain = ((benefit from 1st query) + (benefit from 2nd query)) / MAX(index
size of all queries)
gain = (1355.9 + 1684.72) / MAX(2184, 2184)
gain = 3040.62 / 2184
gain = 1.39223
```

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.

20.3.2.5 Index Advisor Limitations

Index Advisor does not 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 will not be any kind of expression; the field will be a simple column name.

Index Advisor does not consider inheritance when recommending hypothetical indexes. If a query references a parent table, Index Advisor does not make any index recommendations on child tables.

Restoration of 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, may result in "broken links" between the `index_advisor_log` table and the restored tables referenced by rows in the `index_advisor_log` table because of changes in object identifiers (OIDs).

If it is necessary 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:

```
UPDATE index_advisor_log SET reloid = new_oid WHERE reloid = old_oid;
```

20.3.3 SQL Profiler

Inefficient SQL code is one of, if not the 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 the following:

- **On-Demand Traces.** You can capture SQL traces at any time by manually setting up your parameters and starting the trace.
- **Scheduled Traces.** For inconvenient times, you can also specify your trace parameters and schedule them to run at some later time.
- **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, and a graphical or text based `EXPLAIN` plan lays out your query paths and joins.
- **Index Advisor Integration.** Once you have found your slow queries and optimized them, you can also let the Index Advisor recommend the creation of underlying table indices to further improve performance.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following describes the installation process.

Step 1: Install SQL Profiler

SQL Profiler is installed by the EDB Postgres Advanced Server installer on Windows or from the `edb-asxx-server-sqlprofiler` package along with the `edb-asxx-server` package installation, where `xx` is the EDB Postgres Advanced Server version number.

Step 2: Add the SQL Profiler library

Modify the `postgresql.conf` parameter file for the instance to include the SQL Profiler library in the `shared_preload_libraries` configuration parameter.

For Linux installations, the parameter value should include:

```
$libdir/sql-profiler
```

On Windows, the parameter value should include:

```
$libdir\sql-profiler.dll
```

Step 3: Create the SQL Profiler extension

Create the SQL Profiler extension in your database with the following command:

```
CREATE EXTENSION sql_profiler;
```

Step 4: Stop and restart the server for the changes to take effect

After configuring SQL Profiler, it's ready to use with all databases that reside on the server. You can take advantage of SQL Profiler functionality with EDB Postgres Enterprise Manager. For more information about Postgres Enterprise Manager, see the PEM documentation.

20.3.4 pgsnmpd

`pgsnmpd` is an SNMP agent that can return hierarchical information about the current state of Advanced Server on a Linux system. `pgsnmpd` is distributed and installed using the `edb-asxx-pgsnmpd` RPM package where `xx` is the Advanced Server version number. The `pgsnmpd` agent can operate as a stand-alone SNMP agent, as a pass-through sub-agent, or as an AgentX sub-agent.

After installing Advanced Server, you will need to update the `LD_LIBRARY_PATH` variable. Use the command:

```
$ export LD_LIBRARY_PATH=/usr/edb/as11/lib:$LD_LIBRARY_PATH
```

This command does not persistently alter the value of `LD_LIBRARY_PATH`. Consult the documentation for your distribution of Linux for information about persistently setting the value of `LD_LIBRARY_PATH`.

The examples that follow demonstrate the simplest usage of `pgsnmpd`, implementing read only access. `pgsnmpd` is based on the net-snmp library; for more information about net-snmp, visit:

http://net-snmp.sourceforge.net/

Configuring pgsnmpd

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `pgsnmpd` configuration file is named `snmpd.conf`. For information about the directives that you can specify in the configuration file, please review the `snmpd.conf` man page (`man snmpd.conf`).

You can create the configuration file by hand, or you can use the `snmpconf` perl script to create the configuration file. The perl script is distributed with net-snmp package.

net-snmp is an open-source package available from:

http://www.net-snmp.org/

To use the snmpconf configuration file wizard, download and install net-snmp. When the installation completes, open a command line and enter:

```
snmpconf
```

When the configuration file wizard opens, it may prompt you to read in an existing configuration file. Enter `none` to generate a new configuration file (not based on a previously existing configuration file).

`snmpconf` is a menu-driven wizard. Select menu item `1: snmpd.conf` to start the configuration wizard. As you select each top-level menu option displayed by `snmpconf`, the wizard walks through a series of questions, prompting you for information required to build the configuration file. When you have provided information in each of the category relevant to your system, enter `Finished` to generate a configuration file named `snmpd.conf`. Copy the file to:

```
/usr/edb/as11/share/
```

Setting the Listener Address

By default, `pgsnmpd` listens on port `161`. If the listener port is already being used by another service, you may receive the following error:

```
Error opening specified endpoint "udp:161".
```

You can specify an alternate listener port by adding the following line to your `snmpd.conf` file:

```
agentaddress $host_address:2000
```

The example instructs `pgsnmpd` to listen on UDP port `2000`, where `$host_address` is the IP address of the server (e.g., `127.0.0.1`).

Invoking pgsnmpd

Ensure that an instance of Advanced Server is up and running (`pgsnmpd` will connect to this server). Open a command line and assume superuser privileges, before invoking pgsnmpd with a command that takes the following form:

```
POSTGRES_INSTALL_HOME/bin/pgsnmpd -b
  -c POSTGRES_INSTALL_HOME/share/snmpd.conf
  -C "user=enterprisedb dbname=edb password=safe_password
      port=5444"
```

Where `POSTGRES_INSTALL_HOME` specifies the Advanced Server installation directory.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Include the `-b` option to specify that `pgsnmpd` should run in the background.

Include the `-c` option, specifying the path and name of the `pgsnmpd` configuration file.

Include connection information for your installation of Advanced Server (in the form of a `libpq` connection string) after the `-C` option.

Viewing pgsnmpd Help

Include the `--help` option when invoking the `pgsnmpd` utility to view other `pgsnmpd` command line options:

```
pgsnmpd --help
  Version PGSQL-SNMP-Ver1.0
  usage: pgsnmpd [-s] [-b] [-c FILE ] [-x address ] [-g] [-C "Connect String"]
     -s : run as AgentX sub-agent of an existing snmpd process
     -b : run in the background
     -c : configuration file name
     -g : use syslog
     -C : libpq connection string
     -x : address:port of a network interface
     -V : display version strings
```

Requesting Information from pgsnmpd

You can use `net-snmp` commands to query the `pgsnmpd` service. For example:

```
snmpgetnext -v 2c -c public localhost .1.3.6.1.4.1.5432.1.4.2.1.1.0
```

In the above example:

`-v 2c` option instructs the `snmpgetnext` client to send the request in SNMP version 2c format.

`-c public` specifies the community name.

`localhost` indicates the host machine running the `pgsnmpd` server.

`.1.3.6.1.4.1.5432.1.4.2.1.1.0` is the identity of the requested object. To see a list of all databases, increment the last digit by one (e.g. .1.1, .1.2, .1.3 etc.).

The encodings required to query any given object are defined in the MIB (Management Information Base). An SNMP client can monitor a variety of servers; the server type determines the information exposed by a given server. Each SNMP server describes the exposed data in the form of a MIB (Management information base). By default, pgsnmpd searches for MIB's in the following locations:

```
/usr/share/snmp/mibs

$HOME/.snmp/mibs
```

20.3.5 EDB Audit Logging

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Advanced Server allows database and security administrators, auditors, and operators to track and analyze database activities using the *EDB Audit Logging* functionality. EDB Audit Logging generates audit log files, which contains all of the relevant information. The audit logs can be configured to record information such as:

- When a role establishes a connection to an Advanced Server database
- What database objects a role creates, modifies, or deletes when connected to 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.

20.3.5.1 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, or if the configuration needs to be reloaded, or if the database server needs to be restarted.

`edb_audit`

Enables or disables database auditing. The values `xml` or `csv` will enable database auditing. These values represent the file format in which auditing information will be captured. `none` will disable database auditing and is also the default.

> **Note**
>
> The `logging_collector` parameter must be set to `on` to enable the `edb_audit` parameter.

`edb_audit_directory`

Specifies the directory where the log files will be created. The path of the directory can be relative or absolute to the data folder. The default is the `PGDATA/edb_audit` directory.

`edb_audit_filename`

Specifies the file name of the audit file where the auditing information will be stored. The default file name will be `audit-%Y%m%d_%H%M%S`. The escape sequences, `%Y`, `%m` etc., will be 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 will be 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 will not occur.

`edb_audit_rotation_seconds`

Specifies the rotation time in seconds when a new log file should be created. 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

audit all failed connection attempts, set the value to `failed`, which is the default. To audit all connection attempts, set the value to `all`.

`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`

This configuration parameter is used to specify auditing of different categories of SQL statements. Various combinations of the following values may be specified: `none`, `dml`, `insert`, `update`, `delete`, `truncate`, `select`, `error`, `rollback`, `ddl`, `create`, `drop`, `alter`, `grant`, `revoke`, and `all`. The default is `ddl` and `error`. See Selecting SQL Statements to Audit for information regarding the setting of this parameter.

`edb_audit_tag`

Use this configuration parameter to specify a string value that will be included 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 Advanced Server log file and the EDB Audit log file. However, logging each and every statement in bulk processing is costly. This can be controlled by the `edb_log_every_bulk_value` configuration parameter. When set to `true`, each and every statement in bulk processing is logged. When set to `false`, a log message is recorded once per bulk processing. In addition, the duration is emitted once per bulk processing. Default is set to `false`.

`edb_audit_destination`

Specifies whether the audit log information is to be recorded 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 Advanced Server running on a Linux host and is not supported on Windows systems. **Note:** In recent Linux versions, syslog has been replaced by `rsyslog` and the configuration file is in `/etc/rsyslog.conf`.

> **Note**
>
> 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.

The following section describes selection of specific SQL statements for auditing using the `edb_audit_statement` parameter.

20.3.5.2 Selecting SQL Statements to Audit

The `edb_audit_statement` permits inclusion of one or more, comma-separated values to control which SQL statements are to be audited. The following is the general format:

```
edb_audit_statement = 'value_1[, value_2]...'
```

The comma-separated values may include or omit space characters following the comma. The values can be specified in any combination of lowercase or uppercase letters.

The basic parameter values are the following:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `all` – Results in the auditing and logging of every statement including any error messages on statements.
- `none` – Disables all auditing and logging. A value of `none` overrides any other value included in the list.
- `ddl` – Results in the auditing of all data definition language (DDL) statements (`CREATE` , `ALTER` , and `DROP`) as well as `GRANT` and `REVOKE` data control language (DCL) statements.
- `dml` – Results in the auditing of all data manipulation language (DML) statements (`INSERT` , `UPDATE` , `DELETE` , and `TRUNCATE`).
- `select` – Results in the auditing of `SELECT` statements.
- `rollback` – Results in the auditing of `ROLLBACK` statements.
- `error` – Results in the logging of all error messages that occur. Unless the `error` value is included, no error messages are logged regarding any errors that occur on SQL statements related to any of the other preceding parameter values except when `all` is used.

`Data Definition Language and Data Control Language Statements` describes additional parameter values for selecting particular DDL or DCL statements for auditing.

`Data Manipulation Language 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 where `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/as11/data/
postgresql.conf" contains errors
```

The following sections describe the values for the SQL language types DDL, DCL, and DML.

Data Definition Language and Data Control Language Statements

This section describes values that can be included in the `edb_audit_statement` parameter to audit DDL and DCL statements.

The following general rules apply:

- If the `edb_audit_statement` parameter includes either `ddl` or `all` , then all DDL statements are audited. In addition, the DCL statements `GRANT` and `REVOKE` are audited as well.
- If the `edb_audit_statement` is set to `none` , then no DDL nor DCL statements are audited.
- Specific types of DDL and DCL statements can be chosen for auditing by including a combination of values within 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`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `DATABASE`
- `EVENT TRIGGER`
- `EXTENSION`
- `FOREIGN TABLE`
- `FUNCTION`
- `INDEX`
- `LANGUAGE`
- `LARGE OBJECT`
- `MATERIALIZED VIEW`
- `OPERATOR`
- `OPERATOR CLASS`
- `OPERATOR FAMILY`
- `POLICY`
- `PUBLICATION`
- `ROLE`
- `RULE`
- `SCHEMA`
- `SEQUENCE`
- `SERVER`
- `SUBSCRIPTION`
- `TABLE`
- `TABLESPACE`
- `TEXT SEARCH CONFIGURATION`
- `TEXT SEARCH DICTIONARY`
- `TEXT SEARCH PARSER`
- `TEXT SEARCH TEMPLATE`
- `TRANSFORM`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- `TRIGGER`
- `TYPE`
- `USER MAPPING`
- `VIEW`

Descriptions of object types as used in SQL commands can be found in the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/sql-commands.html

If `object_type` is omitted from the parameter value, then all of the specified command statements (either `create`, `alter`, or `drop`) are audited.

Use the following syntax to specify an `edb_audit_statement` parameter value for DCL statements:

```
{ grant | revoke }
```

The following are some DDL and DCL examples.

Example 1

The following is an example where `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
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 database session that occurs is the following:

```
$ psql edb enterprisedb
Password for user enterprisedb:
psql.bin (11.0.0)
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
(1 row)

edb=# CREATE ROLE adminuser;
CREATE ROLE
edb=# ALTER ROLE adminuser WITH LOGIN, SUPERUSER, PASSWORD 'password';
ERROR:  syntax error at or near ","
LINE 1: ALTER ROLE adminuser WITH LOGIN, SUPERUSER, PASSWORD 'passwo...
                                       ^
edb=# ALTER ROLE adminuser WITH LOGIN SUPERUSER PASSWORD 'password';
ALTER ROLE
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# CREATE DATABASE auditdb;
CREATE DATABASE
edb=# ALTER DATABASE auditdb OWNER TO adminuser;
ALTER DATABASE
edb=# \c auditdb adminuser
Password for user adminuser:
You are now connected to database "auditdb" as user "adminuser".
auditdb=# CREATE SCHEMA edb;
CREATE SCHEMA
auditdb=# SET search_path TO edb;
SET
auditdb=# CREATE TABLE department (
auditdb(#  deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
auditdb(#  dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
auditdb(#  loc             VARCHAR2(13)
auditdb(# );
CREATE TABLE
auditdb=# DROP TABLE department;
DROP TABLE
auditdb=# CREATE TABLE dept (
auditdb(#  deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
auditdb(#  dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
auditdb(#  loc             VARCHAR2(13)
auditdb(# );
CREATE TABLE
```

The resulting audit log file contains the following.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
 2017-07-16 12:59:42.125 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,1,"authentication",2017-07-16 12:59:42 EDT,6/2,0,AUDIT,00000,
"connection authorized: user=enterprisedb database=edb",,,,,,,,,"","",""

2017-07-16 12:59:42.125 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,2,"idle",2017-07-16 12:59:42 EDT,6/6,0,AUDIT,00000,
"statement: CREATE ROLE adminuser;",,,,,,,,,"psql.bin","CREATE ROLE",""

2017-07-16 13:00:28.469 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,3,"idle",2017-07-16 12:59:42 EDT,6/7,0,ERROR,42601,
"syntax error at or near ""","""",,,,,,
"ALTER ROLE adminuser WITH LOGIN, SUPERUSER, PASSWORD 'password';",32,,"psql.bin","",""

2017-07-16 13:00:28.469 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,4,"idle",2017-07-16 12:59:42 EDT,6/8,0,AUDIT,00000,
"statement: ALTER ROLE adminuser WITH LOGIN SUPERUSER PASSWORD 'password';",,,,,,,,,
"psql.bin","ALTER ROLE",""

2017-07-16 13:00:28.469 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,5,"idle",2017-07-16 12:59:42 EDT,6/9,0,AUDIT,00000,
"statement: CREATE DATABASE auditdb;",,,,,,,,,"psql.bin","CREATE DATABASE",""

2017-07-16 13:00:28.469 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,6,"idle",2017-07-16 12:59:42 EDT,6/10,0,AUDIT,00000,
"statement: ALTER DATABASE auditdb OWNER TO adminuser;",,,,,,,,,"psql.bin","ALTER DATABASE",""

2017-07-16 13:01:13.735 EDT,"adminuser","auditdb",3377,"[local]",
596b9bd9.d31,1,"authentication",2017-07-16 13:01:13 EDT,4/15,0,AUDIT,00000,
"connection authorized: user=adminuser database=auditdb",,,,,,,,,"","",""
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
2017-07-16 13:01:13.735 EDT,"adminuser","auditdb",3377,"[local]",
596b9bd9.d31,2,"idle",2017-07-16 13:01:13 EDT,4/17,0,AUDIT,00000,
"statement: CREATE SCHEMA edb;",,,,,,,,,"psql.bin","CREATE SCHEMA",""

2017-07-16 13:01:13.735 EDT,"adminuser","auditdb",3377,"[local]",
596b9bd9.d31,3,"idle",2017-07-16 13:01:13 EDT,4/19,0,AUDIT,00000,
"statement: CREATE TABLE department (
 deptno            NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
 dname             VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
 loc               VARCHAR2(13)
);",,,,,,,,,"psql.bin","CREATE TABLE",""

2017-07-16 13:01:13.735 EDT,"adminuser","auditdb",3377,"[local]",
596b9bd9.d31,4,"idle",2017-07-16 13:01:13 EDT,4/21,0,AUDIT,00000,
"statement: CREATE TABLE dept (
 deptno            NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
 dname             VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
 loc               VARCHAR2(13)
);",,,,,,,,,"psql.bin","CREATE TABLE",""
```

The `CREATE` and `ALTER` statements for the `adminuser` role and `auditdb` database are audited. The error for the `ALTER ROLE adminuser` statement is also logged since error is included in the `edb_audit_statement` parameter.

Similarly, the `CREATE` statements for schema `edb` and tables `department` and `dept` are audited.

Note that the `DROP TABLE department` statement is not in the audit log since there is no `edb_audit_statement` setting that would result in the auditing of successfully processed `DROP` statements such as `ddl`, `all`, or `drop`.

Example 2

The following is an example where `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
 edb_audit_connect = 'all'
edb_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 database session that occurs is the following:

```
 $ psql auditdb adminuser
Password for user adminuser:
psql.bin (11.0.0)
Type "help" for help.

auditdb=# SHOW edb_audit_connect;
 edb_audit_connect
-------------------
 all
(1 row)

auditdb=# SHOW edb_audit_statement;
                 edb_audit_statement
------------------------------------------------------------
 create view,create materialized view,create sequence,grant
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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),
auditdb(#   job             VARCHAR2(9),
auditdb(#   mgr             NUMBER(4),
auditdb(#   hiredate        DATE,
auditdb(#   sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
auditdb(#   comm            NUMBER(7,2),
auditdb(#   deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
auditdb(#                       REFERENCES dept(deptno)
auditdb(# );
CREATE TABLE
auditdb=# CREATE VIEW salesemp AS
auditdb-#     SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job =
'SALESMAN';
CREATE VIEW
auditdb=# CREATE MATERIALIZED VIEW managers AS
auditdb-#     SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job =
'MANAGER';
SELECT 0
auditdb=# CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
CREATE SEQUENCE
auditdb=# GRANT ALL ON dept TO PUBLIC;
GRANT
auditdb=# GRANT ALL ON emp TO PUBLIC;
GRANT
```

The resulting audit log file contains the following.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
 2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,1,"authentication",2017-07-16 13:20:09 EDT,4/10,0,AUDIT,00000,
"connection authorized: user=adminuser database=auditdb",,,,,,,,,"","",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,2,"idle",2017-07-16 13:20:09 EDT,4/16,0,AUDIT,00000,
"statement: CREATE VIEW salesemp AS
   SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job =
'SALESMAN';",,,,,,,,,"psql.bin","CREATE VIEW",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,3,"idle",2017-07-16 13:20:09 EDT,4/17,0,AUDIT,00000,
"statement: CREATE MATERIALIZED VIEW managers AS
   SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job =
'MANAGER';",,,,,,,,,"psql.bin","CREATE MATERIALIZED VIEW",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,4,"idle",2017-07-16 13:20:09 EDT,4/18,0,AUDIT,00000,
"statement: CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY
1;",,,,,,,,,"psql.bin","CREATE SEQUENCE",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,5,"idle",2017-07-16 13:20:09 EDT,4/19,0,AUDIT,00000,
"statement: GRANT ALL ON dept TO PUBLIC;",,,,,,,,,"psql.bin","GRANT",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
596ba049.102f,6,"idle",2017-07-16 13:20:09 EDT,4/20,0,AUDIT,00000,
"statement: GRANT ALL ON emp TO PUBLIC;",,,,,,,,,"psql.bin","GRANT",""
```

The `CREATE VIEW` and `CREATE MATERIALIZED VIEW` statements are audited. Note that the prior `CREATE TABLE emp` statement is not audited since 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 since those values are included in the `edb_audit_statement` parameter.

Example 3

The following is an example where `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 that `ADMINUSER` is to be audited.

```
ALTER USER adminuser SET edb_audit_statement = "all";
```

Setting the `edb_audit_statement` parameter to `all` allows auditing all of the SQL statements for an admin user.

The database session that occurs is the following:

```
 $ psql auditdb adminuser
Password for user adminuser:
psql.bin (11.0.0)
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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
auditdb=> SELECT * FROM dept;
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     30 | SALES      | CHICAGO
     20 | RESEARCH   | BEDFORD
(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;
 deptno |  dname   |   loc
--------+----------+---------
     30 | SALES    | CHICAGO
     20 | RESEARCH | BEDFORD
(2 rows)
```

The resulting audit log file contains the following.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
 2021-06-23 06:06:59.027 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,1,"authentication",2021-06-23 06:06:59 IST,4/19,0,AUDIT,00000,"connection authorized:
user=adminuser database=auditdb",,,,,,,,,"","client backend",,"","","connect"

2021-06-23 06:07:33.192 IST,"adminuser","auditdb",66316,"[local]",60daab0c.
1030c,2,"idle",2021-06-23 06:07:33 IST,4/16,0,AUDIT,00000,"statement: SHOW edb_audit_connect;
",,,,,,,,,"psql","client backend",,"SHOW","","sql statement"

2021-06-23 06:08:12.474 IST,"adminuser","auditdb",66316,"[local]",60daab0c.
1030c,3,"idle",2021-06-23 06:08:12 IST,4/17,0,AUDIT,00000,"statement: SHOW edb_audit_statement;
",,,,,,,,,"psql","client backend",,"SHOW","","sql statement"

2021-06-23 06:08:20.519 IST,"adminuser","auditdb",66922,"[local]",60dab036.
1056a,4,"idle",2021-06-23 06:08:20 IST,4/15,0,AUDIT,00000,"statement: SET search_path TO edb;
",,,,,,,,,"psql","client backend",,"SET","","set"

2021-06-23 06:09:27.613 IST,"adminuser","auditdb",60218,"[local]",60dab117.
10602,5,"idle",2021-06-23 06:09:59 IST,4/21,0,AUDIT,00000,"statement: CREATE TABLE dept
       (deptno NUMBER(2),
        dname VARCHAR2(14),
        loc VARCHAR2(13) );",,,,,,,,,"psql","client backend",,"CREATE TABLE","","create"

2021-06-23 06:09:39.238 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,6,"idle",2021-06-23 06:09:29 IST,4/22,0,AUDIT,00000,"statement: INSERT INTO
dept VALUES (10, 'ACCOUNTING', 'NEW YORK');",,,,,,,,,"psql","client backend",,"INSERT","","insert"

2021-06-23 06:09:39.242 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,7,"idle",2021-06-23 06:09:29 IST,4/23,0,AUDIT,00000,"statement: INSERT INTO
dept VALUES (20, 'RESEARCH', 'DALLAS');",,,,,,,,,"psql","client backend",,"INSERT","","insert"

2021-06-23 06:09:39.247 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,8,"idle",2021-06-23 06:08:35 IST,4/24,0,AUDIT,00000,"statement: INSERT INTO
dept VALUES (30, 'SALES', 'CHICAGO');",,,,,,,,,"psql","client backend",,"INSERT","","insert"
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
2021-06-23 06:10:04.849 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,9,"idle",2021-06-23 06:08:59 IST,4/25,0,AUDIT,00000,"statement: UPDATE dept SET loc = 'BEDFORD'
WHERE deptno = 20;",,,,,,,,,"psql","client backend",,"UPDATE","","update"

2021-06-23 06:10:16.045 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,10,"idle",2021-06-23 06:08:59 IST,4/26,0,AUDIT,00000,"statement: SELECT * FROM dept;",,,,,,,,,
"psql","client backend",,"SELECT","","select"

2021-06-23 06:10:40.593 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,11,"idle",2021-06-23 06:08:59 IST,4/27,0,AUDIT,00000,"statement: DELETE FROM emp WHERE deptno = 40;
",,,,,,,,,"psql","client backend",,"DELETE","","delete"

2021-06-23 06:10:40.594 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,12,"DELETE",2021-06-23 06:08:59 IST,4/27,0,ERROR,42P01,"relation ""emp"" does not exist",,,,,,
"DELETE FROM emp WHERE deptno = 40;",13,,"psql","client backend",,"DELETE","","error"

2021-06-23 06:11:02.563 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,13,"idle",2021-06-23 06:08:59 IST,4/28,0,AUDIT,00000,"statement: DELETE FROM dept WHERE deptno = 10;
",,,,,,,,,"psql","client backend",,"DELETE","","delete"

2021-06-23 06:11:14.585 IST,"adminuser","auditdb",60218,"[local]",60d3083b.
eb3a,14,"idle",2021-06-23 06:08:59 IST,4/29,0,AUDIT,00000,"statement: SELECT * FROM dept;",,,,,,,,,
"psql","client backend",,"SELECT","","select"
```

Example 4

The following is an example where `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 = 'all'
```

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";
```

> **Note**
>
> 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 desired value.

The database session that occurs is the following:

```
 $ psql auditdb carol
Password for user carol:
psql.bin (11.0.0)
Type "help" for help.

auditdb=# SHOW edb_audit_connect;
 edb_audit_connect
-------------------
 all
(1 row)

auditdb=# SHOW edb_audit_statement;
 edb_audit_statement
---------------------
 none
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(1 row)

auditdb=# SET search_path TO edb;
SET
auditdb=> CREATE TABLE salgrade
auditdb->         (grade NUMBER,
auditdb(>          losal NUMBER,
auditdb(>          hisal NUMBER);
CREATE TABLE
INSERT INTO salgrade VALUES (1,  700, 1200);
INSERT INTO salgrade VALUES (2, 1201, 1400);
INSERT INTO salgrade VALUES (3, 1401, 2000);
```

The resulting audit log file contains only the connection authentication information. Setting the `edb_audit_statement parameter` to `none` does not allow auditing any of the SQL statements for `carol`, and thereby no audit logs are generated.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
 2021-06-29 02:27:26.240 IST,"carol","auditdb",68072,"[local]",60dabd4e.
109e8,1,"authentication",2021-06-29 02:27:26 IST,4/13,0,AUDIT,00000,"connection authorized: user=carol
database=auditdb",,,,,,,,,"","client backend",,"","","connect"
```

Data Manipulation Language Statements

This section describes the values that can be included in the `edb_audit_statement` parameter to audit DML statements.

The following general rules apply:

- If the `edb_audit_statement` parameter includes either `dml` or `all`, then all DML statements are audited.
- If the `edb_audit_statement` is set to `none`, then no DML statements are audited.
- Specific types of DML statements can be chosen for auditing by including a combination of values within 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

The following is an example where `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
 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 (even errors not related to the `UPDATE` and `DELETE` commands).

The database session that occurs is the following:

```
 $ psql auditdb adminuser
Password for user adminuser:
psql.bin (11.0.0)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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)

auditdb=# SET search_path TO edb;
SET
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=# INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
INSERT 0 1
auditdb=# INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT 0 1
auditdb=# INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
INSERT 0 1
auditdb=# INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT 0 1
                .
                .
                .
auditdb=# INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
INSERT 0 1
auditdb=# UPDATE dept SET loc = 'BEDFORD' WHERE deptno = 40;
UPDATE 1
auditdb=# SELECT * FROM dept;
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BEDFORD
(4 rows)

auditdb=# DELETE FROM emp WHERE deptno = 10;
DELETE 3
auditdb=# TRUNCATE employee;
ERROR:  relation "employee" does not exist
auditdb=# TRUNCATE emp;
TRUNCATE TABLE
auditdb=# \q
```

The resulting audit log file contains the following.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
2017-07-16 13:43:26.638 EDT,"adminuser","auditdb",4574,"[local]",
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
596ba5be.11de,1,"authentication",2017-07-16 13:43:26 EDT,4/11,0,AUDIT,00000,
"connection authorized: user=adminuser database=auditdb",,,,,,,,,"","",""

2017-07-16 13:43:26.638 EDT,"adminuser","auditdb",4574,"[local]",
596ba5be.11de,2,"idle",2017-07-16 13:43:26 EDT,4/34,0,AUDIT,00000,
"statement: UPDATE dept SET loc = 'BEDFORD' WHERE deptno = 40;",,,,,,,,,"psql.bin","UPDATE",""

2017-07-16 13:43:26.638 EDT,"adminuser","auditdb",4574,"[local]",
596ba5be.11de,3,"idle",2017-07-16 13:43:26 EDT,4/36,0,AUDIT,00000,
"statement: DELETE FROM emp WHERE deptno = 10;",,,,,,,,,"psql.bin","DELETE",""

2017-07-16 13:45:46.999 EDT,"adminuser","auditdb",4574,"[local]",
596ba5be.11de,4,"TRUNCATE TABLE",2017-07-16 13:43:26 EDT,4/37,0,ERROR,42P01,
"relation ""employee"" does not exist",,,,,,"TRUNCATE employee;",,,"psql.bin","",""

2017-07-16 13:46:26.362 EDT,,,4491,,596ba59c.118b,1,,2017-07-16 13:42:52 EDT,,0,LOG,00000,
"database system is shut down",,,,,,,,,"","",""
```

The `UPDATE dept` and `DELETE FROM emp` statements are audited. Note that all of the prior `INSERT` statements are not audited since none of the values `insert`, `dml`, nor `all` are included in the `edb_audit_statement` parameter.

The `SELECT * FROM dept` statement is not audited as well since neither `select` nor `all` is included in the `edb_audit_statement` parameter.

Since `error` is specified in the `edb_audit_statement` parameter, but not the `truncate` value, the error on the `TRUNCATE employee` statement is logged in the audit file, but not the successful `TRUNCATE emp` statement.

20.3.5.3 Enabling Audit Logging

The following steps describe how to configure Advanced Server to log all connections, disconnections, DDL statements, DCL statements, DML statements, and any statements resulting in an error.

1. Enable auditing by the setting the `edb_audit` parameter to `xml` or `csv`.
2. Set the file rotation day when the new file will be created by setting the parameter `edb_audit_rotation_day` to the desired value.
3. To audit all connections, set the parameter, `edb_audit_connect`, to `all`.
4. To audit all disconnections, set the parameter, `edb_audit_disconnect`, to `all`.
5. To audit DDL, DCL, DML and other statements, set the parameter, `edb_audit_statement` according to the instructions in Selecting SQL Statements to Audit.

The setting of the `edb_audit_statement` parameter in the configuration file affects the entire database cluster.

The type of statements that are audited as controlled by the `edb_audit_statement` parameter can be further refined according to the database in use as well as the database role running the session:

- The `edb_audit_statement` parameter can be set 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`.
- The `edb_audit_statement` parameter can be set 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 previously described `ALTER DATABASE` command when the specified role is running the current session.
- The `edb_audit_statement` parameter can be set 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 as well as any setting assigned to the database by the previously described `ALTER DATABASE` command as well as any setting assigned to the role with the `ALTER ROLE` command without the `IN DATABASE` clause as previously described.

The following are examples of this technique.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The database cluster is established with `edb_audit_statement` set to `all` as shown in its `postgresql.conf` file:

```
edb_audit_statement = 'all'     # Statement type to be audited:
                                # none, dml, insert, update, delete, truncate,
                                # select, error, rollback, ddl, create, drop,
                                # alter, grant, revoke, all
```

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` and `truncate`
- Role `admin` in database `auditdb` with `create table`, `insert`, and `update`

Creation and alteration of the database and role are shown by the following:

```
$ psql edb enterprisedb
Password for user enterprisedb:
psql.bin (11.0.0)
Type "help" for help.

edb=# SHOW edb_audit_statement;
 edb_audit_statement
---------------------
 all
(1 row)

edb=# CREATE DATABASE auditdb;
CREATE DATABASE
edb=# ALTER DATABASE auditdb SET edb_audit_statement TO 'ddl, insert, update, delete';
ALTER DATABASE
edb=# CREATE ROLE admin WITH LOGIN SUPERUSER PASSWORD 'password';
CREATE ROLE
edb=# ALTER ROLE admin SET edb_audit_statement TO 'select, truncate';
ALTER ROLE
edb=# ALTER ROLE admin IN DATABASE auditdb SET edb_audit_statement TO 'create table, insert, update';
ALTER ROLE
```

The following demonstrates 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:

```
$ psql auditdb enterprisedb
Password for user enterprisedb:
psql.bin (11.0.0)
Type "help" for help.

auditdb=# SHOW edb_audit_statement;
     edb_audit_statement
-----------------------------
 ddl, insert, update, delete
(1 row)

auditdb=# CREATE TABLE audit_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);
CREATE TABLE
auditdb=# INSERT INTO audit_tbl VALUES (1, 'Row 1');
INSERT 0 1
auditdb=# UPDATE audit_tbl SET f2 = 'Row A' WHERE f1 = 1;
UPDATE 1
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
auditdb=# SELECT * FROM audit_tbl;                 <== Should not be audited
 f1 |  f2
----+-------
  1 | Row A
(1 row)

auditdb=# TRUNCATE audit_tbl;                      <== Should not be audited
TRUNCATE TABLE
```

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 were not audited.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,1,"idle",2017-07-13 15:25:59 EDT,7/4,0,AUDIT,00000,
"statement: CREATE TABLE audit_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);",,,,,,,,,
"psql.bin","CREATE TABLE",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,2,"idle",2017-07-13 15:25:59 EDT,7/5,0,AUDIT,00000,
"statement: INSERT INTO audit_tbl VALUES (1, 'Row 1');",,,,,,,,,"psql.bin","INSERT",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,3,"idle",2017-07-13 15:25:59 EDT,7/6,0,AUDIT,00000,
"statement: UPDATE audit_tbl SET f2 = 'Row A' WHERE f1 = 1;",,,,,,,,,"psql.bin","UPDATE",""
```

Case 2: Changes made in database `edb` by role `admin`. Only `select` and `truncate` statements are audited:

```
$ psql edb admin
Password for user admin:
psql.bin (11.0.0)
Type "help" for help.

edb=# SHOW edb_audit_statement;
 edb_audit_statement
----------------------
 select, truncate
(1 row)

edb=# CREATE TABLE edb_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT); <== Should not be audited
CREATE TABLE
edb=# INSERT INTO edb_tbl VALUES (1, 'Row 1');               <== Should not be audited
INSERT 0 1
edb=# SELECT * FROM edb_tbl;
 f1 |  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 `SELECT * FROM edb_tbl` and `TRUNCATE edb_tbl` statements. The `CREATE TABLE edb_tbl` and `INSERT INTO edb_tbl` statements were not audited.

```
2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
5967c947.fb8,1,"idle",2017-07-13 15:25:59 EDT,7/4,0,AUDIT,00000,
"statement: CREATE TABLE audit_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);",,,,,,,,,
"psql.bin","CREATE TABLE",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,2,"idle",2017-07-13 15:25:59 EDT,7/5,0,AUDIT,00000,
"statement: INSERT INTO audit_tbl VALUES (1, 'Row 1');",,,,,,,,,"psql.bin","INSERT",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,3,"idle",2017-07-13 15:25:59 EDT,7/6,0,AUDIT,00000,
"statement: UPDATE audit_tbl SET f2 = 'Row A' WHERE f1 = 1;",,,,,,,,,"psql.bin","UPDATE",""

2017-07-13 15:29:45.616 EDT,"admin","edb",4047,"[local]",
5967ca05.fcf,1,"idle",2017-07-13 15:29:09 EDT,4/33,0,AUDIT,00000,
"statement: SELECT * FROM edb_tbl;",,,,,,,,,"psql.bin","SELECT",""

2017-07-13 15:29:45.616 EDT,"admin","edb",4047,"[local]",
5967ca05.fcf,2,"idle",2017-07-13 15:29:09 EDT,4/34,0,AUDIT,00000,
"statement: TRUNCATE edb_tbl;",,,,,,,,,"psql.bin","TRUNCATE TABLE",""
```

Case 3: Changes made in database `auditdb` by role `admin`. Only `create table, insert`, and `update` statements are audited:

```
$ psql auditdb admin
Password for user admin:
psql.bin (11.0.0)
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);
CREATE TABLE
auditdb=# INSERT INTO audit_tbl_2 VALUES (1, 'Row 1');
INSERT 0 1
auditdb=# SELECT * FROM audit_tbl_2;               <== Should not be audited
 f1 |  f2
----+-------
  1 | Row 1
(1 row)

auditdb=# TRUNCATE audit_tbl_2;                    <== Should not be audited
TRUNCATE TABLE
```

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 were not audited.

```
2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,1,"idle",2017-07-13 15:25:59 EDT,7/4,0,AUDIT,00000,
"statement: CREATE TABLE audit_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);",,,,,,,,,
"psql.bin","CREATE TABLE",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,2,"idle",2017-07-13 15:25:59 EDT,7/5,0,AUDIT,00000,
"statement: INSERT INTO audit_tbl VALUES (1, 'Row 1');",,,,,,,,,"psql.bin","INSERT",""
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,3,"idle",2017-07-13 15:25:59 EDT,7/6,0,AUDIT,00000,
"statement: UPDATE audit_tbl SET f2 = 'Row A' WHERE f1 = 1;",,,,,,,,,"psql.bin","UPDATE",""

2017-07-13 15:29:45.616 EDT,"admin","edb",4047,"[local]",
5967ca05.fcf,1,"idle",2017-07-13 15:29:09 EDT,4/33,0,AUDIT,00000,
"statement: SELECT * FROM edb_tbl;",,,,,,,,,"psql.bin","SELECT",""

2017-07-13 15:29:45.616 EDT,"admin","edb",4047,"[local]",
5967ca05.fcf,2,"idle",2017-07-13 15:29:09 EDT,4/34,0,AUDIT,00000,
"statement: TRUNCATE edb_tbl;",,,,,,,,,"psql.bin","TRUNCATE TABLE",""

2017-07-13 15:35:45.309 EDT,"admin","auditdb",4085,"[local]",
5967cb81.ff5,1,"idle",2017-07-13 15:35:29 EDT,4/72,0,AUDIT,00000,
"statement: CREATE TABLE audit_tbl_2 (f1 INTEGER PRIMARY KEY, f2 TEXT);",,,,,,,,,
"psql.bin","CREATE TABLE",""

2017-07-13 15:35:45.309 EDT,"admin","auditdb",4085,"[local]",
5967cb81.ff5,2,"idle",2017-07-13 15:35:29 EDT,4/73,0,AUDIT,00000,
"statement: INSERT INTO audit_tbl_2 VALUES (1, 'Row 1');",,,,,,,,,"psql.bin","INSERT",""

2017-07-13 15:38:42.028 EDT,,,3942,,5967c934.f66,1,,2017-07-13 15:25:40
EDT,,0,LOG,00000,"database system is shut down",,,,,,,,,"","",""
```

20.3.5.4 Audit Log File

The audit log file can be generated in either CSV or XML format depending upon the setting of the `edb_audit` configuration parameter. The XML format contains less information than the CSV format.

The information in the audit log is based on the logging performed by PostgreSQL as described in the section “Using CSV-Format Log Output” within Section “Error Reporting and Logging” in the PostgreSQL core documentation, available at:

https://www.postgresql.org/docs/11/static/runtime-config-logging.html

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 as previously referenced.
- **XML Element/Attribute.** For the XML format, name of the XML element and its attribute (if used), referencing the value.**Note:** `n/a` indicates that there is no XML representation for this field.
- **Data Type.** Data type of the field as given by the PostgreSQL sample table definition.
- **Description.** Description of the field. For certain fields, no output is generated in the audit log as those fields are not supported by auditing. Those fields are indicated by “Not supported”.

The fields with the `Description` of “Not supported” appear as consecutive commas (,,) in the CSV format.

Field	XML Element/Attribute	Data Type	Description
`log_time`	`event/time`	`timestamp with time zone`	Log date/time of the statement.
`user_name`	`event/user`	`text`	Database user who executed the statement.
`database_name`	`event/database`	`text`	Database in which the statement was executed.
`process_id`	`event/process_id`	`integer`	Operating system process ID in which the statement was executed.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Field	XML Element/Attribute	Data Type	Description
connection_from	event/remote_host	text	Host and port location from where the statement was executed.
session_id	event/session_id	text	Session ID in which the statement was executed.
session_line_num	n/a	bigint	Order of the statement within the session.
process_status	n/a	text	Processing status.
session_start_time	n/a	timestamp with time zone	Date/time when the session was started.
virtual_transaction_id	n/a	text	Virtual transaction ID of the statement.
transaction_id	event/transaction_id	bigint	Regular transaction ID of the statement.
error_severity	message	text	Statement severity. Values are AUDIT for audited statements and ERROR for any resulting error messages.
sql_state_code	n/a	text	SQL state code returned for the statement.
message	message	text	The SQL statement that was attempted for execution.
detail	n/a	text	Error message detail. (Not supported)
hint	n/a	text	Hint (Not supported)
internal_query	n/a	text	Internal query that led to the error, if any. (Not supported)
internal_query_pos	n/a	integer	Character count of the error position therein. (Not supported)
context	n/a	text	Error context. (Not supported)
query	n/a	text	User query that led to the error. (For errors only)
query_pos	n/a	integer	Character count of the error position therein. (For errors only)
location	n/a	text	Location of the error in the PostgreSQL source code. (Not supported)
application_name	n/a	text	Name of the application from which the statement was executed. (for example, psql.bin).
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.

The following examples are generated in the CSV and XML formats.

The non-default audit settings in the postgresql.conf file are as follows:

```
 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 audited session is the following:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 $ psql edb enterprisedb
Password for user enterprisedb:
psql.bin (11.0.0)
Type "help" for help.

edb=# CREATE SCHEMA edb;
CREATE SCHEMA
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 has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
 2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,1,"authentication",2017-07-17 13:28:44 EDT,6/2,0,AUDIT,00000,
"connection authorized: user=enterprisedb database=edb",,,,,,,,,"","","edbaudit"

2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,2,"idle",2017-07-17 13:28:44 EDT,6/4,0,AUDIT,00000,
"statement: CREATE SCHEMA edb;",,,,,,,,,"psql.bin","CREATE SCHEMA","edbaudit"

2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,3,"idle",2017-07-17 13:28:44 EDT,6/6,0,AUDIT,00000,
"statement: CREATE TABLE dept (
    deptno        NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname         VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc           VARCHAR2(13)
);",,,,,,,,,"psql.bin","CREATE TABLE","edbaudit"

2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,4,"idle",2017-07-17 13:28:44 EDT,6/7,0,AUDIT,00000,
"statement: INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');",,,,,,,,,
"psql.bin","INSERT","edbaudit"

2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
596cf3cc.fe4,5,"idle",2017-07-17 13:28:44 EDT,6/8,0,AUDIT,00000,
"statement: UPDATE department SET loc = 'BOSTON' WHERE deptno = 10;",,,,,,,,,
"psql.bin","UPDATE","edbaudit"

2017-07-17 13:29:59.833 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,6,"UPDATE",2017-07-17 13:28:44 EDT,6/8,0,ERROR,42P01,
"relation ""department"" does not exist",,,,,,
"UPDATE department SET loc = 'BOSTON' WHERE deptno = 10;",8,,"psql.bin","","edbaudit"

2017-07-17 13:29:59.833 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,7,"idle",2017-07-17 13:28:44 EDT,6/9,0,AUDIT,00000,
"statement: UPDATE dept SET loc = 'BOSTON' WHERE deptno = 10;",,,,,,,,,
"psql.bin","UPDATE","edbaudit"

2017-07-17 13:29:59.833 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,8,"idle",2017-07-17 13:28:44 EDT,6/10,0,AUDIT,00000,
"statement: SELECT * FROM dept;",,,,,,,,,"psql.bin","SELECT","edbaudit"

2017-07-17 13:29:59.833 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,9,"idle",2017-07-17 13:28:44 EDT,,0,AUDIT,00000,
"disconnection: session time: 0:02:01.511 user=enterprisedb database=edb
host=[local]",,,,,,,,,"psql.bin","SELECT","edbaudit"

2017-07-17 13:30:53.617 EDT,,,3987,,596cf3b3.f93,1,,2017-07-17 13:28:19 EDT,,0,LOG,00000,
"database system is shut down",,,,,,,,,"","","edbaudit"
```

XML Audit Log File

The following is the XML format of the audit log file. The output has been formatted for more clarity in the appearance in the example.

```
 <event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:36:55 EDT"
       transaction_id="0" type="connect" audit_tag="edbaudit">
  <message>AUDIT: connection authorized: user=enterprisedb database=edb</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:02 EDT"
       transaction_id="0" type="create" command_tag="CREATE SCHEMA" audit_tag="edbaudit">
  <message>AUDIT: statement: CREATE SCHEMA edb;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:19 EDT"
       transaction_id="0" type="create" command_tag="CREATE TABLE" audit_tag="edbaudit">
  <message>AUDIT: statement: CREATE TABLE dept (
              deptno         NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
              dname          VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
              loc            VARCHAR2(13));
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:29 EDT"
       transaction_id="0" type="insert" command_tag="INSERT" audit_tag="edbaudit">
  <message>AUDIT: statement: INSERT INTO dept VALUES
                            (10,&amp;apos;ACCOUNTING&amp;apos;,&amp;apos;NEW YORK&amp;apos;);
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:40 EDT"
       transaction_id="0" type="update" command_tag="UPDATE" audit_tag="edbaudit">
  <message>AUDIT: statement: UPDATE department SET
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
                            loc = &apos;BOSTON&apos; WHERE deptno = 10;
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:40 EDT"
       transaction_id="0" type="error" audit_tag="edbaudit">
  <message>ERROR: relation &quot;department&quot; does not exist at character 8
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:51 EDT"
       transaction_id="0" type="update" command_tag="UPDATE" audit_tag="edbaudit">
  <message>AUDIT: statement: UPDATE dept SET loc = &apos;BOSTON&apos; WHERE deptno = 10;
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:59 EDT"
       transaction_id="0" type="select" command_tag="SELECT" audit_tag="edbaudit">
  <message>AUDIT: statement: SELECT * FROM dept;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:38:01 EDT"
       transaction_id="0" type="disconnect" command_tag="SELECT" audit_tag="edbaudit">
  <message>AUDIT: disconnection: session time: 0:01:05.814
                 user=enterprisedb database=edb host=[local]
  </message>
</event>
<event process_id="4696" time="2017-07-17 13:38:08 EDT"
       transaction_id="0" type="shutdown" audit_tag="edbaudit">
  <message>LOG: database system is shut down</message>
</event>
```

20.3.5.5 Using Error Codes to Filter Audit Logs

Advanced Server includes an extension that you can use to exclude log file entries that include a user-specified error code from the Advanced Server log files. To filter audit log entries, you must first enable the extension by modifying the `postgresql.conf` file, adding the following value to the values specified in the `shared_preload_libraries` parameter:

```
$libdir/edb_filter_log
```

Then, use the `edb_filter_log.errcodes` parameter to specify any error codes you wish to omit from the log files:

```
edb_filter_log.errcode = 'error_code'
```

Where `error_code` specifies one or more error codes that you wish 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:

`23505` (for violating a unique constraint)

`23502` (for violating a not-null constraint)

`22012` (for dividing by zero)

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

will be omitted from the log file.

For a complete list of the error codes supported by Advanced Server audit log filtering, see the core documentation at:

https://www.postgresql.org/docs/11/static/errcodes-appendix.html

20.3.5.6 Using Command Tags to Filter Audit Logs

Each entry in the log file except for those displaying an error message contains a *command tag*. A command tag is the SQL command executed for that particular log entry. The command tag makes it possible to use subsequent tools to scan the log file to find entries related to a particular SQL command.

The following is an example in XML form. The example has been 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` in the example.

```
 <event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="595e8537.10f1" process_id="4337" time="2017-07-06 14:45:18 EDT"
       transaction_id="0" type="create"
       command_tag="CREATE ROLE">
  <message>AUDIT: statement: CREATE ROLE newuser WITH LOGIN;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="595e8537.10f1" process_id="4337" time="2017-07-06 14:45:31 EDT"
       transaction_id="0" type="error">
  <message>ERROR: unrecognized role option &quot;super&quot; at character 25
           STATEMENT: ALTER ROLE newuser WITH SUPER USER;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="595e8537.10f1" process_id="4337" time="2017-07-06 14:45:38 EDT"
       transaction_id="0" type="alter" command_tag="ALTER ROLE">
  <message>AUDIT: statement: ALTER ROLE newuser WITH SUPERUSER;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="595e8537.10f1" process_id="4337" time="2017-07-06 14:45:46 EDT"
       transaction_id="0" type="drop" command_tag="DROP ROLE">
  <message>AUDIT: statement: DROP ROLE newuser;</message>
</event>
```

The following is the same example in CSV form. The command tag is the next to last column of each entry. (The last column appears empty as "", which would be the value provided by the `edb_audit_tag` parameter.)

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
 2017-07-06 14:47:22.294 EDT,"enterprisedb","edb",4720,"[local]",
595e85b2.1270,1,"idle",2017-07-06 14:47:14 EDT,6/4,0,AUDIT,00000,
"statement: CREATE ROLE newuser WITH LOGIN;",,,,,,,,,"psql.bin","CREATE ROLE",""

2017-07-06 14:47:29.694 EDT,"enterprisedb","edb",4720,"[local]",
595e85b2.1270,2,"idle",2017-07-06 14:47:14 EDT,6/5,0,ERROR,42601,
"unrecognized role option ""super""",,,,,,"ALTER ROLE newuser WITH SUPER USER;",25,,
"psql.bin","",""

2017-07-06 14:47:29.694 EDT,"enterprisedb","edb",4720,"[local]",
595e85b2.1270,3,"idle",2017-07-06 14:47:14 EDT,6/6,0,AUDIT,00000,
"statement: ALTER ROLE newuser WITH SUPERUSER;",,,,,,,,,"psql.bin","ALTER ROLE",""
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
2017-07-06 14:47:29.694 EDT,"enterprisedb","edb",4720,"[local]",
595e85b2.1270,4,"idle",2017-07-06 14:47:14 EDT,6/7,0,AUDIT,00000,
"statement: DROP ROLE newuser;",,,,,,,,,"psql.bin","DROP ROLE",""
```

20.3.5.7 Redacting Passwords from Audit Logs

You can use the `edb_filter_log.redact_password_commands` extension to instruct the server to redact stored passwords from the log file. Note that the module only recognizes the following syntax:

```
{CREATE|ALTER} {USER|ROLE|GROUP} identifier { [WITH] [ENCRYPTED] PASSWORD
'nonempty_string_literal' | IDENTIFIED BY { 'nonempty_string_literal' |
bareword } } [ REPLACE { 'nonempty_string_literal' | bareword } ]
```

When such a statement is logged by `log_statement`, the server will redact the old and new passwords to 'x'. For example, the command:

```
ALTER USER carol PASSWORD '1safepwd' REPLACE 'old_pwd';
```

Will be added to log files as:

```
statement: ALTER USER carol PASSWORD 'x' REPLACE 'x';
```

When a statement that includes a redacted password is logged, the server redacts the statement text. When the statement is logged as context for some other message, the server omits the statement from the context.

To enable password redaction, you must first enable the extension by modifying the `postgresql.conf` file, adding the following value to the values 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.

20.3.6 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.

Customizability is an important factor for various reasons such as the following.

- Unicode supports a vast number of languages. Letters that may be common to several languages may be expected to collate in different orders depending upon the language.
- Characters that appear with letters in certain languages such as accents or umlauts have an impact on the expected collation depending upon the language.
- In some languages, combinations of several consecutive characters should be treated as a single character with regards to its collation sequence.
- There may be certain preferences as to the collation of letters according to case. For example, should the lowercase form of a letter collate before

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

the uppercase form of the same letter or vice versa.
- There may be preferences as to whether punctuation marks such as underscore characters, hyphens, or space characters should be considered in the collating sequence or should they simply be ignored as if they did not exist in the string.

Given all of these variations with the vast number of languages supported by Unicode, there is a necessity for a method to select the specific criteria for determining a collating sequence. This is what the Unicode Collation Algorithm defines.

> **Note**
>
> In addition, 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.

The following sections provide a brief, simplified explanation of the Unified Collation Algorithm concepts. As the algorithm and its usage are quite complex with numerous variations, refer to the official documents cited in these sections for complete details.

Basic Unicode Collation Algorithm Concepts

The official information for the Unicode Collation Algorithm is specified in *Unicode Technical Report #10*, which can be found on The Unicode Consortium website at:

http://www.unicode.org/reports/tr10/

The ICU – International Components for Unicode also provides much useful information. An explanation of the collation concepts can be found on their website located at:

http://userguide.icu-project.org/collation/concepts

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 cannot 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 cannot 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 determine the order such as `a < 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 < A`.
- **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 Advanced Server is used to create a collation that invokes the ICU components to produce the collation, the result is referred to as an *ICU collation*.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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. `Collation Attributes` provides additional information on collation attributes.

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. This 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`.

```
edb=# SELECT icu_short_form, valid_locale FROM pg_icu_collate_names ORDER BY
valid_locale;
 icu_short_form | valid_locale
----------------+--------------
 LAF            | af
 LAR            | ar
 LAS            | as
 LAZ            | az
 LBE            | be
 LBG            | bg
 LBN            | bn
 LBS            | bs
 LBS_ZCYRL      | bs_Cyrl
 LROOT          | ca
 LROOT          | chr
 LCS            | cs
 LCY            | cy
 LDA            | da
 LROOT          | de
 LROOT          | dz
 LEE            | ee
 LEL            | el
 LROOT          | en
 LROOT          | en_US
 LEN_RUS_VPOSIX | en_US_POSIX
 LEO            | eo
 LES            | es
 LET            | et
 LFA            | fa
 LFA_RAF        | fa_AF
    .
    .
    .
```

If needed, the default characteristics of an ICU short form for a given locale can be overridden by specifying the collation attributes to override that property. This is discussed in the next section.

Collation Attributes

When creating an ICU collation, the desired characteristics of the collation must be specified. As discussed in `Locale Collations`, this can typically be done with an ICU short form for the desired locale. However, if more specific information is required, the specification of the collation properties can be done by using *collation attributes*.

Collation attributes define the rules of how characters are to be compared for determining the collation sequence of text strings. As Unicode covers a vast

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

set of languages in numerous variations according to country, territory and culture, these collation attributes are quite complex.

For the complete, precise meaning and usage of collation attributes, see Section “Collator Naming Scheme” on the ICU – International Components for Unicode website at:

http://userguide.icu-project.org/collation/concepts

The following is a brief summary of the collation attributes and how they are specified using the ICU short form method

Each collation attribute is represented by an uppercase letter, which are listed in the following bullet points. The possible valid values for each attribute are given by codes shown within the parentheses. Some codes have general meanings for all attributes. **X** means to set the attribute off. **O** means to set the attribute on. **D** means to set the attribute to its default value.

- **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). Off (X) is typically specified when lowercase first (L) is desired.
- **E – Case Level (X, O, D).** Set in combination with the Strength attribute, the Case Level attribute is used when accents are to be ignored, 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 or not 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, which then present issues when sorting or comparing such characters. Languages such as Arabic, ancient Greek, Hebrew, Hindi, Thai, or Vietnamese should be used with Normalization Checking set 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 I represents identical strength (that is, level 5).
- **T – Variable Top (hexadecimal digits).** Applicable only when the Alternate attribute is not set to non-ignorable (N). The hexadecimal digits specify the highest character sequence that is to be considered ignorable. For example, if white space is to be ignorable, but visible variable characters are not to be ignorable, then Variable Top set to 0020 would be specified along with the Alternate attribute set to S and the Strength attribute set to 3. (The space character is hexadecimal 0020. Other non-visible variable characters such as backspace, tab, line feed, carriage return, etc. 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 the next pair with an underscore character as shown by the following example.

```
AN_CX_EX_FX_HX_NO_S3
```

Collation attributes can be specified along with a locale’s ICU short form name to override those default attribute settings of the locale.

The following is an example where the ICU short form named `LROOT` is modified with a number of other collation attribute/value pairs.

```
AN_CX_EX_LROOT_NO_S3
```

In the preceding example, 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 `3`. `LROOT` is the ICU short form to which these other attributes are applying modifications.

Creating an ICU Collation

Creating an ICU collation can be done a number of different ways.

- When creating a new database cluster with the `initdb` command, the `--icu-short-form` option can be specified to define the ICU collation to be used by default by all databases in the cluster.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- When creating a new database with the `CREATE DATABASE` command, the `ICU_SHORT_FORM` parameter can be specified to define the ICU collation to be used by default in that database.

- In an existing database, the `CREATE COLLATION` command can be used with the `ICU_SHORT_FORM` parameter to define an ICU collation to be used under specific circumstances such as when assigned with the `COLLATE` clause onto selected columns of certain tables or when appended with the `COLLATE` clause onto an expression such as `ORDER BY expr COLLATE "collation_name"`.

The following describes the various ways of creating an ICU collation.

CREATE COLLATION

Use the `ICU_SHORT_FORM` parameter with the `CREATE COLLATION` command to create an ICU collation:

```
CREATE COLLATION <collation_name> (
[ LOCALE = <locale>, ]
[ LC_COLLATE = <lc_collate>, ]
[ LC_CTYPE = <lc_ctype>, ]
[ ICU_SHORT_FORM = <icu_short_form> ]
);
```

To be able to create a collation, you must have `CREATE` privilege on the destination schema where the collation is to reside.

For information about the general usage of the `CREATE COLLATION` command, please refer to the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/sql-createcollation.html

UTF-8 character encoding of the database is required. Any `LOCALE`, or `LC_COLLATE` and `LC_CTYPE` settings that are accepted with UTF-8 encoding can be used.

Parameters

`collation_name`

The name of the collation to be added. The collation_name may be schema-qualified.

`locale`

The locale to be used. Short cut for setting `LC_COLLATE` and `LC_TYPE`. If `LOCALE` is specified, then `LC_COLLATE` and `LC_TYPE` must be omitted.

`lc_collate`

The collation to be used. If `LC_CTYPE` is specified, then `LC_COLLATE` must also be specified and `LOCALE` must be omitted.

`lc_ctype`

The character classification to be used. If `LC_COLLATE` is specified, then `LC_CTYPE` must also be specified and `LOCALE` must be omitted.

`icu_short_form`

The text string specifying the collation attributes and their settings. This typically consists of an ICU short form name, possibly appended with additional collation attribute/value pairs. A list of ICU short form names is available from column `icu_short_form` in system catalog `pg_catalog.pg_icu_collate_names`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Example

The following creates a collation using the `LROOT` ICU short form.

```
edb=# CREATE COLLATION icu_collate_a (LOCALE = 'en_US.UTF8', ICU_SHORT_FORM = 'LROOT');
CREATE COLLATION
```

The definition of the new collation can be seen with the following `psql` command.

```
edb=# \dO List of collations
  Schema      | Name          | Collate    | Ctype      | ICU
--------------+---------------+------------+------------+-------
enterprisedb  | icu_collate_a | en_US.UTF8 | en_US.UTF8 | LROOT
(1 row)
```

CREATE DATABASE

The following is the syntax for creating a database with an ICU collation:

```
CREATE DATABASE database_name
  [ [ WITH ] [ OWNER [=] user_name ]
    [ TEMPLATE [=] template ]
    [ ENCODING [=] encoding ]
    [ LC_COLLATE [=] lc_collate ]
    [ LC_CTYPE [=] lc_ctype ]
    [ TABLESPACE [=] tablespace_name ]
    [ CONNECTION LIMIT [=] connlimit ]
    [ ICU_SHORT_FORM [=] icu_short_form ]];
```

For complete information about the general usage, syntax, and parameters of the `CREATE DATABASE` command, please refer to the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/sql-createdatabase.html

When using the `CREATE DATABASE` command to create a database using an ICU collation, the `TEMPLATE template0` clause must be specified and the database encoding must be UTF-8.

The following is an example of creating a database using the `LROOT` ICU short form collation, but sorts an uppercase form of a letter before its lowercase counterpart (`CU`) and treats variable characters as non-ignorable (`AN`).

```
CREATE DATABASE collation_db
  TEMPLATE template0
  ENCODING 'UTF8'
  ICU_SHORT_FORM = 'AN_CU_EX_NX_LROOT';
```

The following `psql` command shows the newly created database.

```
edb=# \l collation_db
                                    List of databases
Name          | Owner        | Encoding | Collate     | Ctype       | ICU               | Access privileges
--------------+--------------+----------+-------------+-------------+-------------------+-------------------
 collation_db | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT |
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following table is created and populated with rows in the database.

```
CREATE TABLE collate_tbl (
    id        INTEGER,
    c2        VARCHAR(2)
);

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');
```

The following query shows that the uppercase form of a letter sorts before the lowercase form of the same base letter, and in addition, variable characters are taken into account when sorted as they appear at the beginning of the sort list. (The default behavior for `en_US.UTF-8` is to sort the lowercase form of a letter before the uppercase form of the same base letter, and to ignore variable characters.)

```
collation_db=# SELECT id, c2 FROM collate_tbl ORDER BY c2;
 id  | c2
-----+----
 11 |  B
 10 | -B
  9 | .B
  7 | 1
  8 | 2
  1 | A
  4 | a
  2 | B
  5 | b
  3 | C
  6 | c
(11 rows)
```

initdb

A database cluster can be created with a default ICU collation for all databases in the cluster by using the `--icu-short-form` option with the `initdb` command.

For complete information about the general usage, syntax, and parameters of the `initdb` command, please refer to the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/app-initdb.html

The following illustrates this process.

```
 $ su enterprisedb
Password:
$ /usr/edb/as11/bin/initdb -U enterprisedb -D /tmp/collation_data --encoding UTF8 --icu-short-form
'AN_CU_EX_NX_LROOT'
The files belonging to this database system will be owned by user "enterprisedb".
This user must also own the server process.
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
The database cluster will be initialized with locale "en_US.UTF-8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

creating directory /tmp/collation_data ... ok
creating subdirectories ... ok
    .
    .
    .
Success. You can now start the database server using:

    /usr/edb/as11/bin/edb-postgres -D /tmp/collation_data
or
    /usr/edb/as11/bin/pg_ctl -D /tmp/collation_data -l logfile start
```

The following shows the databases created in the cluster which all have an ICU collation of `AN_CU_EX_NX_LROOT`.

```
edb=# \l
                                      List of databases
 Name      | Owner        | Encoding | Collate     | Ctype       | ICU               | Access privileges
-----------+--------------+----------+-------------+-------------+-------------------+-----------------------
-----
 edb       | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT |
 postgres  | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT |
 template0 | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT | =c/enterprisedb
           |              |    +     |             |             |                   |
enterprisedb=CTc/enterprisedb
 template1 | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT | =c/enterprisedb
           |              |    +     |             |             |                   |
enterprisedb=CTc/enterprisedb
(4 rows)
```

Using a Collation

A newly defined ICU collation can be used anywhere the `COLLATION "collation_name"` clause can be used in a SQL command such as 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 the creation and usage of 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_lowercase` forces the lowercase form of a letter to sort before its uppercase counterpart (`CL`).

Collation `icu_collate_uppercase` 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_collate_ignore_white_sp` causes white space and other non-visible variable characters to be ignored during sorting, but visible variable characters (punctuation marks) are not ignored (`AS`, `T0020`).

```
CREATE COLLATION icu_collate_lowercase (
    LOCALE = 'en_US.UTF8',
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    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 may generate notice and warning messages when attributes are given to modify the `LROOT` collation.

The following `psql` command lists the collations.

```
edb=# \dO
                                    List of collations
    Schema    |            Name              |  Collate   |   Ctype    |            ICU
--------------+------------------------------+------------+------------+----------------------
-------
 enterprisedb | icu_collate_ignore_punct     | en_US.UTF8 | en_US.UTF8 | AS_CX_EX_NX_LROOT_L3
 enterprisedb | icu_collate_ignore_white_sp | en_US.UTF8 | en_US.UTF8 |
AS_CX_EX_NX_LROOT_L3_T0020
 enterprisedb | icu_collate_lowercase       | en_US.UTF8 | en_US.UTF8 | AN_CL_EX_NX_LROOT
 enterprisedb | icu_collate_uppercase       | en_US.UTF8 | en_US.UTF8 | AN_CU_EX_NX_LROOT
(4 rows)
```

The following table is created and populated.

```
CREATE TABLE collate_tbl (
    id              INTEGER,
    c2              VARCHAR(2)
);

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');
```

The following query sorts on column `c2` using the default collation. Note that variable characters (white space and punctuation marks) with `id` column values of `9`, `10`, and `11` are ignored and sort with the letter `B`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# SELECT * FROM collate_tbl ORDER BY c2;
 id  | c2
-----+----
 7   | 1
 8   | 2
 4   | a
 1   | A
 5   | b
 2   | B
 11  | B
 10  | -B
 9   | .B
 6   | c
 3   | C
(11 rows)
```

The following query sorts on column `c2` using collation `icu_collate_lowercase`, which forces the lowercase form of a letter to sort before the uppercase form of the same base letter. Also note that the `AN` attribute forces variable characters to be included in the sort order at the same level when comparing base characters so rows with `id` values of `9`, `10`, and `11` appear at the beginning of the sort list before all letters and numbers.

```
edb=# SELECT * FROM collate_tbl ORDER BY c2 COLLATE "icu_collate_lowercase";
 id | c2
----+----
 11 |  B
 10 | -B
 9  | .B
 7  | 1
 8  | 2
 4  | a
 1  | A
 5  | b
 2  | B
 6  | c
 3  | C
(11 rows)
```

The following query sorts on column `c2` using collation `icu_collate_uppercase`, which forces the uppercase form of a letter to sort before the lowercase form of the same base letter.

```
edb=# SELECT * FROM collate_tbl ORDER BY c2 COLLATE "icu_collate_uppercase";
 id | c2
----+----
 11 |  B
 10 | -B
 9  | .B
 7  | 1
 8  | 2
 1  | A
 4  | a
 2  | B
 5  | b
 3  | C
 6  | c
(11 rows)
```

The following query sorts on column `c2` using collation `icu_collate_ignore_punct`, which causes variable characters to be ignored so rows with id values of `9`, `10`, and `11` sort with the letter `B` as that is the character immediately following the ignored variable character.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 edb=# SELECT * FROM collate_tbl ORDER BY c2 COLLATE
"icu_collate_ignore_punct";
 id  | c2
-----+----
  7  | 1
  8  | 2
  4  | a
  1  | A
  5  | b
  11 |  B
  2  | B
  9  | .B
  10 | -B
  6  | c
  3  | C
(11 rows)
```

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 `10`, which start with visible punctuation marks greater than hexadecimal `0020`, appear at the beginning of the sort list as these particular variable characters are included in the sort order at the same level when comparing base characters.

```
 edb=# SELECT * FROM collate_tbl ORDER BY c2 COLLATE
"icu_collate_ignore_white_sp";
 id | c2
----+----
 10 | -B
 9  | .B
 7  | 1
 8  | 2
 4  | a
 1  | A
 5  | b
 11 |  B
 2  | B
 6  | c
 3  | C
(11 rows)
```

20.4 EDB Resource Manager

EDB Resource Manager is an Advanced Server feature that provides the capability to control the usage of operating system resources used by Advanced Server processes.

This capability allows you to protect the system from processes that may uncontrollably overuse and monopolize certain system resources.

The following are some key points about using EDB Resource Manager.

- The basic component of EDB Resource Manager is a resource group. A *resource group* is a named, global group, available to all databases in an Advanced Server instance, on which various resource usage limits can be defined. Advanced Server processes that are assigned as members of a given resource group are then controlled by EDB Resource Manager so that the aggregate resource usage of all processes in the group is kept near the limits defined on the group.
- Data definition language commands are used to create, alter, and drop resource groups. These commands can only be used by a database user with

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

superuser privileges.

- The desired, aggregate consumption level of all processes belonging to a resource group is defined by *resource type parameters*. There are different resource type parameters for the different types of system resources currently supported by EDB Resource Manager.
- Multiple resource groups can be created, each with different settings for its resource type parameters, thus defining 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 there are multiple resource type parameters with defined settings in a resource group, the actual resource consumption may be significantly lower for certain resource types than their defined resource type parameter settings. This is because EDB Resource Manager throttles processes attempting to keep *all resources with defined resource type settings within their defined limits*.
- The definition of available resource groups and their resource type settings are stored in a shared global system catalog. Thus, resource groups can be utilized by all databases in a given Advanced Server instance.
- The `edb_max_resource_groups` configuration parameter sets the maximum number of resource groups that can be active simultaneously with running processes. The default setting is 16 resource groups. Changes to this parameter take effect on database server restart.
- 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.
- A default resource group can be assigned to a role using the `ALTER ROLE ... SET` command, or to a database by the `ALTER DATABASE ... SET` command. The entire database server instance can be assigned a default resource group by setting the parameter in the `postgresql.conf` file.
- In order to include resource groups in a backup file of the database server instance, use the `pg_dumpall` backup utility with default settings (That is, do not specify any of the `--globals-only`, `--roles-only`, or `--tablespaces-only` options.)

Creating and Managing Resource Groups

The data definition language commands described in this section provide for the creation and management of resource groups.

CREATE RESOURCE GROUP

Use the `CREATE RESOURCE GROUP` command to create a new resource group.

```
CREATE RESOURCE GROUP <group_name>;
```

Description

The `CREATE RESOURCE GROUP` command creates a resource group with the specified name. Resource limits can then be defined on the group with the `ALTER RESOURCE GROUP` command. The resource group is accessible from all databases in the 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

The following example results in the creation of three resource groups named `resgrp_a`, `resgrp_b`, and `resgrp_c`.

```
 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
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following query shows the entries for the resource groups in the `edb_resource_group` catalog.

```
edb=# SELECT * FROM edb_resource_group;
 rgrpname  | rgrpcpuratelimit | rgrpdirtyratelimit
-----------+------------------+--------------------
 resgrp_a  |                0 | 0
 resgrp_b  |                0 | 0
 resgrp_c  |                0 | 0
(3 rows)
```

ALTER RESOURCE GROUP

Use the `ALTER RESOURCE GROUP` command to change the attributes of an existing resource group. The command syntax comes in three forms.

The first form renames the resource group:

```
ALTER RESOURCE GROUP <group_name> RENAME TO <new_name>;
```

The second form assigns a resource type to the resource group:

```
ALTER RESOURCE GROUP <group_name> SET
  <resource_type> { TO | = } { <value> | DEFAULT };
```

The third form resets the assignment of a resource type to its default within the group:

```
ALTER RESOURCE GROUP <group_name> RESET <resource_type>;
```

Description

The `ALTER RESOURCE GROUP` command changes certain attributes of an existing resource group.

The first form with the `RENAME TO` clause assigns a new name to an existing resource group.

The second form with the `SET resource_type TO` clause either assigns the specified literal value to a resource type, or resets the resource type when `DEFAULT` is specified. Resetting or setting a resource type to `DEFAULT` means that the resource group has no defined limit on that resource type.

The third form with the `RESET resource_type` clause resets the resource type for the group as described previously.

To use the `ALTER RESOURCE GROUP` command, you must have superuser privileges.

Parameters

`group_name`

The name of the resource group to be altered.

`new_name`

The new name to be assigned to the resource group.

`resource_type`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The resource type parameter specifying the type of resource to which a usage value is to be set.

`value | DEFAULT`

When `value` is specified, the literal value to be assigned to `resource_type`. When `DEFAULT` is specified, the assignment of `resource_type` is reset for the resource group.

Example

The following are examples of the `ALTER RESOURCE GROUP` command.

```
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
```

The following 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)
```

DROP 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 be removed.

`IF EXISTS`

Do not throw an error if the resource group does not exist. A notice is issued in this case.

Example

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example removes 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 as shown by the following.

```
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 immediately take effect on the current process. If the command is used to change the resource group assigned to the current process, the resource type settings of the newly assigned group immediately take effect.

Processes can be included by default in a resource group by assigning a default resource group to roles, databases, or an entire database server instance.

A default resource group can be assigned to a role using the `ALTER ROLE ... SET` command. For more information about the `ALTER ROLE` command, refer to the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/sql-alterrole.html

A default resource group can be assigned to a database by the `ALTER DATABASE ... SET` command. For more information about the `ALTER DATABASE` command, refer to the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/11/static/sql-alterdatabase.html

The entire database server instance can be assigned a default resource group by setting the `edb_resource_group` configuration parameter in the `postgresql.conf` file as shown by the following.

```
# - EDB Resource Manager -
#edb_max_resource_groups = 16           # 0-65536 (change requires restart)
edb_resource_group = 'resgrp_b'
```

A change to `edb_resource_group` in the `postgresql.conf` file requires a configuration file reload before it takes 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 as shown by the following.

```
edb=# SET edb_resource_group TO DEFAULT;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
--------------------
(1 row)
```

For removing a default resource group from a role, use the `ALTER ROLE ... RESET` form of the `ALTER ROLE` command.

For removing a default resource group from a database, use the `ALTER DATABASE ... RESET` form of the `ALTER DATABASE` command.

For removing 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 and reload the configuration file.

Monitoring Processes in Resource Groups

After resource groups have been created, the number of processes actively using these resource groups can be obtained from the view `edb_all_resource_groups`.

The columns in `edb_all_resource_groups` are the following:

- **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 applicable 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 applicable to an individual, active process in the resource group.

> **Note**
>
> Columns `per_process_cpu_rate_limit` and `per_process_dirty_rate_limit` do not show the *actual* resource consumption used by the processes, but indicate how `EDB Resource Manager` sets the resource limit for an individual process based upon the number of active processes in the resource group.

The following 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.

```
edb=# SELECT * FROM edb_all_resource_groups ORDER BY group_name;
-[ RECORD 1 ]-----------------+------------------
 group_name                  | resgrp_a
 active_processes            | 0
 cpu_rate_limit              | 0.5
 per_process_cpu_rate_limit  |
 dirty_rate_limit            | 12288
 per_process_dirty_rate_limit |
-[ RECORD 2 ]-----------------+------------------
 group_name                  | resgrp_b
 active_processes            | 2
 cpu_rate_limit              | 0.4
 per_process_cpu_rate_limit  | 0.195694289022895
 dirty_rate_limit            | 6144
 per_process_dirty_rate_limit | 3785.92924684337
-[ RECORD 3 ]-----------------+------------------
 group_name                  | resgrp_c
 active_processes            | 1
 cpu_rate_limit              | 0.3
 per_process_cpu_rate_limit  | 0.292342129631091
 dirty_rate_limit            | 3072
 per_process_dirty_rate_limit | 3072
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The CPU rate limit and dirty rate limit settings that are assigned to these resource groups are as follows.

```
edb=# SELECT * FROM edb_resource_group;
 rgrpname  | rgrpcpuratelimit | rgrpdirtyratelimit
-----------+------------------+--------------------
 resgrp_a  | 0.5              | 12288
 resgrp_b  | 0.4              | 6144
 resgrp_c  | 0.3              | 3072
(3 rows)
```

In the `edb_all_resource_groups` view, note that 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.

CPU Usage Throttling

CPU usage of a resource group is controlled by setting 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 should not exceed. Thus, the value assigned to `cpu_rate_limit` should typically be 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 has been set for the resource group.

When multiplied by `100`, the `cpu_rate_limit` can also be interpreted as the CPU usage percentage for a resource group.

EDB Resource Manager utilizes *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 may be interrupted and put into sleep mode for a short interval of time to maintain the defined limit. When and how such interruptions occur is defined by a proprietary algorithm used by EDB Resource Manager.

Setting the CPU Rate Limit for a Resource Group

The `ALTER RESOURCE GROUP` command with the `SET cpu_rate_limit` clause is used to set the CPU rate limit for a resource group.

In the following example the CPU usage limit is set to 50% for `resgrp_a`, 40% for `resgrp_b` and 30% for `resgrp_c`. This means that the combined CPU usage of all processes assigned to `resgrp_a` is maintained at approximately 50%. Similarly, for all processes in `resgrp_b`, the combined CPU usage is kept to approximately 40%, etc.

```
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
```

The following query shows the settings of `cpu_rate_limit` in the catalog.

```
edb=# SELECT rgrpname, rgrpcpuratelimit FROM edb_resource_group;
 rgrpname | rgrpcpuratelimit
----------+------------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 resgrp_a |              0.5
 resgrp_b |              0.4
 resgrp_c |              0.3
(3 rows)
```

Changing the `cpu_rate_limit` of a resource group not only affects new processes that are assigned to the group, but any currently running processes that are members of the group are immediately affected by the change. That is, if the `cpu_rate_limit` is changed from .5 to .3, currently running processes in the group would be throttled downward so that the aggregate group CPU usage would be near 30% instead of 50%.

To illustrate the effect of setting the CPU rate limit for resource groups, the following examples use a CPU-intensive calculation of 20000 factorial (multiplication of 20000 * 19999 * 19998, etc.) performed by the query `SELECT 20000!` ; run in the `psql` command line utility.

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

The following shows that the current process is set to use resource group `resgrp_b` . The factorial calculation is then started.

```
 edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_b
(1 row)
edb=# SELECT 20000!;
```

In a second session, the Linux `top` command is used to display the CPU usage as shown under the `%CPU` column. The following is a snapshot at an arbitrary point in time as the `top` command output periodically changes.

```
 $ top
top - 16:37:03 up  4:15,  7 users,  load average: 0.49, 0.20, 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
Mem:   1025624k total,   791160k used,   234464k free,    23400k buffers
Swap:   103420k total,    13404k used,    90016k free,   373504k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
28915 enterpri  20   0  195m 5900 4212 S 39.9  0.6   3:36.98 edb-postgres
 1033 root      20   0  171m  77m 2960 S  1.0  7.8   3:43.96 Xorg
 3040 user      20   0  278m  22m  14m S  1.0  2.2   3:41.72 knotify4
    .
    .
    .
```

The `psql` session performing the factorial calculation is shown by the row where `edb-postgres` appears under the `COMMAND` column. The CPU usage of the session shown under the `%CPU` column shows 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=# SHOW edb_resource_group;
 edb_resource_group
--------------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(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 - 16:43:03 up 4:21, 7 users, load average: 0.66, 0.33, 0.37
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

  PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
28915 enterpri 20 0 195m 5900 4212 R 93.6 0.6 5:01.56 edb-postgres
 1033 root 20 0 171m 77m 2960 S 1.0 7.8 3:48.15 Xorg
 2907 user 20 0 98.7m 11m 9100 S 0.3 1.2 0:46.51 vmware-user-lo
    .
    .
    .
```

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 illustrated in the following example.

The factorial calculation is performed simultaneously in two separate `psql` sessions, each of which has been 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=# SHOW 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=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_b
(1 row)

edb=# SELECT 20000!;
```

A third session monitors the CPU usage.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 $ 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 4708 3312 S 19.9 0.5 0:57.35 edb-postgres
28915 enterpri 20 0 195m 5900 4212 S 19.6 0.6 5:35.49 edb-postgres
 3040 user 20 0 278m 22m 14m S 1.0 2.2 3:54.99 knotify4
 1033 root 20 0 171m 78m 2960 S 0.3 7.8 3:55.71 Xorg
    .
    .
    .
```

There are now two processes named `edb-postgres` with `%CPU` values of 19.9 and 19.6, whose sum is close to the 40% CPU usage set for resource group `resgrp_b`.

The following command sequence displays the sum of all `edb-postgres` processes sampled over half second time intervals. This 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 within 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)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# SELECT 20000!;
```

Session 4:

```
 edb=# SET edb_resource_group TO resgrp_c;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_c
(1 row)
edb=# SELECT 20000!;
```

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): 70.2%us, 0.0%sy, 0.0%ni, 29.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0
Mem: 1025624k total, 806140k used, 219484k free, 25296k buffers
Swap: 103420k total, 13404k used, 90016k free, 374092k 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`) and 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
72.6
69.55
64.55
79.95
68.55
71.25
74.85
62
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 does not belong to any resource group, the `top` command shows the following output.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 $ 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%hi, 0.0%si, 0.0
Mem: 1025624k total, 797692k used, 227932k free, 24724k buffers
Swap: 103420k total,  13404k used,  90016k free, 374068k 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   1.0  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 is not within a resource group, and basically utilizes 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
90.7
95.7
95.45
93.65
87.95
96.75
94.25
95.45
97.35
92.9
96.05
96.25
94.95
    .
    .
    .
```

Dirty Buffer Throttling

Writing to shared buffers is controlled by setting 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 should write to or "dirty" the shared buffers. An example setting would be 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 has been set for the resource group.

EDB Resource Manager utilizes *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 may be interrupted and put into sleep mode for a short interval of time to maintain the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

defined limit. When and how such interruptions occur is defined by a proprietary algorithm used by EDB Resource Manager.

Setting the Dirty Rate Limit for a Resource Group

The `ALTER RESOURCE GROUP` command with the `SET dirty_rate_limit` clause is used to set the dirty rate limit for a resource group.

In the following 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, etc.

```
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
```

The following query shows the settings of `dirty_rate_limit` in the catalog.

```
edb=# SELECT rgrpname, rgrpdirtyratelimit FROM edb_resource_group;
 rgrpname  | rgrpdirtyratelimit
-----------+--------------------
 resgrp_a  |         12288
 resgrp_b  |         6144
 resgrp_c  |         3072
(3 rows)
```

Changing the `dirty_rate_limit` of a resource group not only affects new processes that are assigned to the group, but any currently running processes that are members of the group are immediately affected by the change. That is, if the `dirty_rate_limit` is changed from 12288 to 3072, currently running processes in the group would be throttled downward so that the aggregate group dirty rate would be near 3072 kilobytes per second instead of 12288 kilobytes per second.

To illustrate the effect of setting the dirty rate limit for resource groups, the following examples use the following table for intensive I/O operations.

```
CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
```

The `FILLFACTOR = 10` clause results in `INSERT` commands packing rows up to only 10% per page. This results in a larger sampling of dirty shared blocks for the purpose of these examples.

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 provides the information 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.

In order to use the `pg_stat_statements` module, perform the following steps.

Step 1: In the `postgresql.conf` file, add `$libdir/pg_stat_statements` to the `shared_preload_libraries` configuration parameter as shown by the following.

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/pg_stat_statements'
```

Step 2: Restart the database server.

Step 3: Use the `CREATE EXTENSION` command to complete the creation of the `pg_stat_statements` module.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# CREATE EXTENSION pg_stat_statements SCHEMA public;
CREATE EXTENSION
```

The `pg_stat_statements_reset()` function is used to clear 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

The following sequence of commands shows the creation of 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.

Finally, the `INSERT` command generates a series of integers from 1 to 10,000 to populate the table, and dirty approximately 10,000 blocks.

```
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
--------------------
 resgrp_b
(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
```

The following shows the results from the `INSERT` command.

```
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          | 13496.184
 shared_blks_dirtied | 10003
```

The actual dirty rate is calculated as follows.

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 13496.184 ms, 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.*

Note that 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 the steps are repeated again 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);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
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
```

The following shows the results from the `INSERT` command without the usage 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
```

First, note the total time was only 2432.165 milliseconds as compared to 13496.184 milliseconds when a resource group with a dirty rate limit set to 6144 kilobytes per second was used.

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*.

Note that the actual dirty rate of 33692 kilobytes per second is significantly 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 has been 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
--------------------
 resgrp_b
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
(1 row)

edb=# INSERT INTO t1 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

Session 2:

```
 edb=# CREATE TABLE t2 (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
--------------------
 resgrp_b
(1 row)

edb=# SELECT pg_stat_statements_reset();
 pg_stat_statements_reset
--------------------------
(1 row)

edb=# INSERT INTO t2 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

> **Note**
>
> The `INSERT` commands in session 1 and session 2 were started after the `SELECT pg_stat_statements_reset()` command in session 2 was run.

The following 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.

```
 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          | 0.43
 shared_blks_dirtied | 0
-[ RECORD 2 ]--------+--------------------------------------------------
 query               | INSERT INTO t2 VALUES (generate_series (?,?), ?);
 rows                | 10000
 total_time          | 30591.551
 shared_blks_dirtied | 10003
-[ RECORD 3 ]--------+--------------------------------------------------
 query               | INSERT INTO t1 VALUES (generate_series (?,?), ?);
 rows                | 10000
 total_time          | 33215.334
 shared_blks_dirtied | 10003
```

First, note 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 as shown in the first example, 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. This is shown in the following calculations.

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.*

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- 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 seconds).

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.

Sessions 1 and 2 are repeated as illustrated in the prior example using resource group `resgrp_b` with a `dirty_rate_limit` of 6144 kilobytes per second.

Session 3:

```
 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:

```
 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)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# INSERT INTO t4 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

> **Note**
>
> The `INSERT` commands in all four sessions were started after the `SELECT pg_stat_statements_reset()` command in session 4 was run.

The following 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.

```
 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          | 0.467
 shared_blks_dirtied | 0
-[ RECORD 2 ]--------+-------------------------------------------------
 query               | INSERT INTO t2 VALUES (generate_series (?,?), ?);
 rows                | 10000
 total_time          | 31343.458
 shared_blks_dirtied | 10003
-[ RECORD 3 ]--------+-------------------------------------------------
 query               | INSERT INTO t1 VALUES (generate_series (?,?), ?);
 rows                | 10000
 total_time          | 28407.435
 shared_blks_dirtied | 10003
-[ RECORD 4 ]--------+-------------------------------------------------
 query               | INSERT INTO t3 VALUES (generate_series (?,?), ?);
 rows                | 10000
 total_time          | 52727.846
 shared_blks_dirtied | 10003
-[ RECORD 5 ]--------+-------------------------------------------------
 query               | INSERT INTO t4 VALUES (generate_series (?,?), ?);
 rows                | 10000
 total_time          | 56063.697
 shared_blks_dirtied | 10003
```

First note that 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, as compared to 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.*

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- 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).

Thus, this demonstrates 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.

System Catalogs

This section describes the system catalogs that 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:

Column	Type	Description
`group_name`	`name`	The name of the resource group.
`active_processes`	`integer`	Number of currently active processes in the resource group.
`cpu_rate_limit`	`float8`	Maximum CPU rate limit for the resource group. `0` means no limit.
`per_process_cpu_rate_limit`	`float8`	Maximum CPU rate limit per currently active process in the resource group.
`dirty_rate_limit`	`float8`	Maximum dirty rate limit for a resource group. `0` means no limit.
`per_process_dirty_rate_limit`	`float8`	Maximum dirty rate limit per currently active process in the resource group.

edb_resource_group

The following table lists the information available in the `edb_resource_group` catalog:

Column	Type	Description

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	Description
`rgrpname`	`name`	The name of the resource group.
`rgrpcpuratelimit`	`float8`	Maximum CPU rate limit for a resource group. `0` means no limit.
`rgrpdirtyratelimit`	`float8`	Maximum dirty rate limit for a resource group. `0` means no limit.

20.5 libpq C Library

libpq is the C application programmer's interface to Advanced Server. libpq is a set of library functions that allow client programs to pass queries to the Advanced Server and to receive the results of these queries.

libpq is also the underlying engine for several other EnterpriseDB application interfaces including those written for C++, Perl, Python, Tcl and ECPG. So some aspects of libpq's behavior will be important to the user if one of those packages is used.

Client programs that use libpq must include the header file `libpq-fe.h` and must link with the `libpq` library.

Using libpq with EnterpriseDB SPL

The EnterpriseDB SPL language can be used 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, Advanced Server provided support for REFCURSORs through the following libpq functions; these functions should now be considered deprecated:

- `PQCursorResult()`
- `PQgetCursorResult()`
- `PQnCursor()`

You may 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.

Please note that the samples that follow do not include error-handling code that would be required in a real-world client application.

Returning a Single REFCURSOR

The following example shows an SPL function that returns a value of type `REFCURSOR`:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 CREATE OR REPLACE FUNCTION getEmployees(p_deptno NUMERIC)
RETURN REFCURSOR AS
  result REFCURSOR;
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 `REFCURSOR` 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()`:

```
 #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(10)");

  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)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
{
  size_t  commandLength = strlen("FETCH ALL FROM ") +
                          strlen(cursorName) + 3;

  char   *commandText = malloc(commandLength);
  PGresult *result;
  int       row;

  sprintf(commandText, "FETCH ALL FROM \"%s\"", cursorName);

  result = PQexec(conn, commandText);

  if (PQresultStatus(result) != PGRES_TUPLES_OK)
    fail(conn, PQerrorMessage(conn));

  printf("-- %s --\n", description);

  for (row = 0; row < PQntuples(result); row++)
  {
    const char *delimiter = "\t";
    int         col;

    for (col = 0; col < PQnfields(result); col++)
    {
      printf("%s%s", delimiter, PQgetvalue(result, row, col));
      delimiter = ",";
    }

    printf("\n");
  }

  PQclear(result);
  free(commandText);
}

static void
fail(PGconn *conn, const char *msg)
{
  fprintf(stderr, "%s\n", msg);

  if (conn != NULL)
    PQfinish(conn);

  exit(-1);
}
```

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`:

```
result = PQexec(conn, "SELECT * FROM getEmployees(10)");
```

The `PQexec()` function returns a result set handle to the C program. The result set will contain exactly one value; that value is the name of the cursor as 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 shown below:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
-- employees --
    7782,CLARK,MANAGER,7839,09-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 00:00:00,1300.00,,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 within 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 should not expect a single `REFCURSOR` in the result set, but should expect two rows, each of which will contain a single value (the first row contains the name of the `employees` cursor, and the second row contains the name of the `departments` cursor).

```
 CREATE OR REPLACE FUNCTION getEmpsAndDepts(p_min NUMERIC,
                                            p_max NUMERIC)
RETURN SETOF REFCURSOR AS
  employees   REFCURSOR;
  departments 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:

```
 #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));
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  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 ") +
                             strlen(cursorName) + 3;
  char        *commandText     = malloc(commandLength);
  PGresult  *result;
  int        row;

  sprintf(commandText, "FETCH ALL FROM \"%s\"", cursorName);

  result = PQexec(conn, commandText);

  if (PQresultStatus(result) != PGRES_TUPLES_OK)
  fail(conn, PQerrorMessage(conn));

  printf("-- %s --\n", description);

  for (row = 0; row < PQntuples(result); row++)
  {
    const char *delimiter = "\t";
    int         col;

    for (col = 0; col < PQnfields(result); col++)
    {
      printf("%s%s", delimiter, PQgetvalue(result, row, col));
      delimiter = ",";
    }

    printf("\n");
  }

  PQclear(result);
  free(commandText);
}

static void
fail(PGconn *conn, const char *msg)
{
  fprintf(stderr, "%s\n", msg);
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  if (conn != NULL)
    PQfinish(conn);

  exit(-1);
}
```

If you call `getEmpsAndDepts(20, 30)`, the server will return 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 00:00:00,800.00,,20
  7499,ALLEN,SALESMAN,7698,20-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,7839,02-APR-81 00:00:00,2975.00,,20
  7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
  7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
  7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
  7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
  7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
  7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
  7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
-- departments --
  20,RESEARCH,DALLAS
  30,SALES,CHICAGO
```

Array Binding

Advanced Server's array binding functionality allows you to send an array of data across the network in a single round-trip. When the back end receives the bulk data, it can use the data to perform insert or update operations.

Perform bulk operations with a prepared statement; use the following function to prepare the statement:

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

Details of `PQprepare()` can be found in the prepared statement section.

The following functions can be used 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`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

API Definition

```
PGresult *PQBulkStart(PGconn *conn,
                      const char *Stmt_Name,
                      unsigned int nCol,
                      const int *paramFmts);
```

PQexecBulk

`PQexecBulk()` is used to supply data `(paramValues)` for a statement that was previously initialized for bulk operation using `PQBulkStart()`.

This function can be used more than once after `PQBulkStart()` to send multiple blocks of data. See the example for more details.

API Definition

```
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 re-preparing it.

API Definition

```
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 are not required 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 will be used repeatedly will be parsed and planned just once, rather than each time they are executed.

API Definition

```
PGresult *PQexecBulkPrepared(PGconn *conn,
                             const char *stmtName,
                             unsigned int nCols,
                             unsigned int nRows,
                             const char *const *paramValues,
                             const int *paramLengths,
                             const int *paramFormats);
```

Example Code (Using PQBulkStart, PQexecBulk, PQBulkFinish)

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following example uses `PGBulkStart`, `PQexecBulk`, and `PQBulkFinish`.

```
 void InsertDataUsingBulkStyle( PGconn *conn )
{
    PGresult            *res;
    Oid                 paramTypes[2];
    char                *paramVals[5][2];
    int                 paramLens[5][2];
    int                 paramFmts[2];
    int                 i;

    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_1", "INSERT INTO testtable1 values( $1, $2
)", 2, paramTypes );
    PQclear( res );

    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 );
       printf( "< -- PQexecBulk -- >\n" );

       res = PQBulkFinish(conn);
       PQclear( res );
       printf( "< -- PQBulkFinish -- >\n" );
}
```

Example Code (Using PQexecBulkPrepared)

The following example uses `PQexecBulkPrepared`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 void InsertDataUsingBulkStyleCombinedVersion( PGconn *conn )
{
       PGresult            *res;
       Oid                 paramTypes[2];
       char                *paramVals[5][2];
       int                 paramLens[5][2];
       int                 paramFmts[2];
       int                 i;

       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 );

       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 = PQexecBulkPrepared(conn, "stmt_2", 2, 5, (const char *const
*)paramVals,(const int *)paramLens, (const int *)paramFmts);
       PQclear( res );
}
```

20.6 Debugger

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 can be debugged are SPL stored procedures, functions, triggers, and packages as well as PL/pgSQL functions and triggers.

The Debugger is integrated with *pgAdmin 4* and *EDB Postgres Enterprise Manager*. If you have installed Advanced Server on a Windows host, pgAdmin 4 is automatically installed; you will find the pgAdmin 4 icon in the `Windows Start` menu. If your 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*
```

On Linux, the `edb-asxx-server-pldebugger` RPM package where `xx` is the Advanced Server version number, must be installed as well. Information about pgAdmin 4 is available at:

https://www.pgadmin.org/

The RPM installation will add the pgAdmin4 icon to your Applications menu.

There are two basic ways the Debugger can be used to test programs:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- **Standalone Debugging.** The Debugger is used to start the program to be tested. You supply any input parameter values required by the program and 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.** The program to be tested is initiated by an application other than the Debugger. You first set a *global breakpoint* on the program to be tested. The application that makes the first call to the program encounters the global breakpoint. The application suspends execution at which 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 within 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 execution. In-context debugging is useful if it is difficult to reproduce a problem using standalone debugging due to complex interaction with the calling application.

The debugging tools and operations are the same whether using standalone or in-context debugging. The difference is in how the program to be debugged is invoked.

The following sections discuss the features and functionality of the Debugger using the standalone debugging method. The directions for starting the Debugger for in-context debugging are discussed in the Setting Global Breakpoint for In-Context Debugging.

Configuring the Debugger

Before using the Debugger, edit the `postgresql.conf` file (located in the `data` subdirectory of your Advanced Server home directory), adding `$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'
```

After modifying the `shared_preload_libraries` parameter, you must restart the database server.

Starting the Debugger

Use pgAdmin 4 to access the Debugger for standalone debugging. To open the Debugger, highlight the name of the stored procedure or function you wish to debug in the pgAdmin 4 `Browser` panel. Then, navigate through the `Object` menu to the `Debugging` menu and select `Debug` from the submenu.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

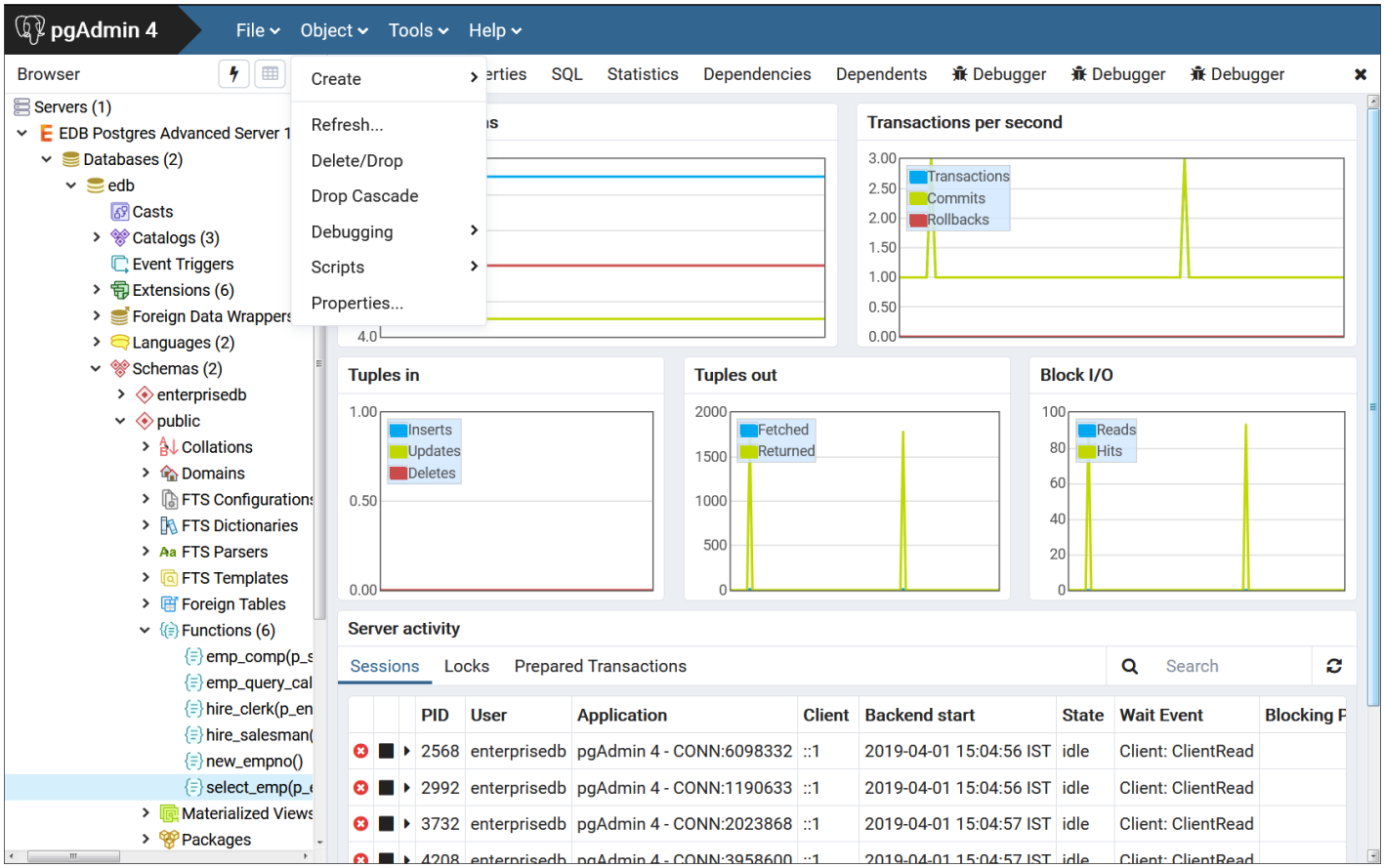
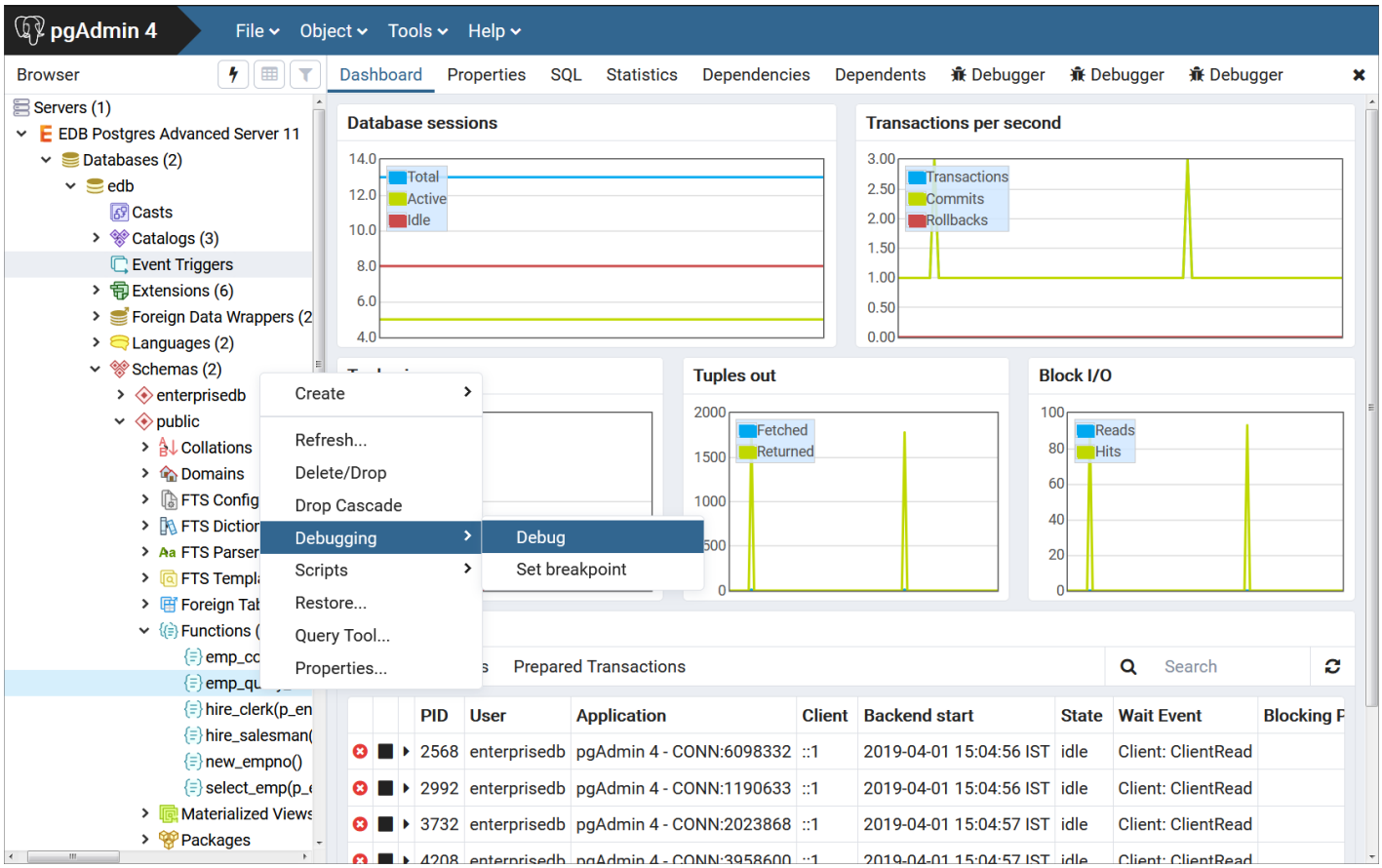


Fig. 1: Starting the Debugger from the Object menu

You can also right-click on the name of the stored procedure or function in the pgAdmin 4 `Browser`, and select `Debugging`, and the `Debug` from the context menu.



Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Fig. 2: Starting the Debugger from the Object's context menu

Note that triggers cannot be debugged using standalone debugging. Triggers must be debugged using in-context debugging. See the Setting Global Breakpoint for In-Context Debugging for information on setting a global breakpoint for in-context debugging.

To debug a package, highlight the specific procedure or function under the package node of the package you wish to debug and follow the same directions as for stored procedures and functions.

The Debugger Window

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 automatically to display any `IN` or `IN OUT` parameters expected by the program. If the program declares no `IN` or `IN OUT` parameters, the `Debugger` window does not open.

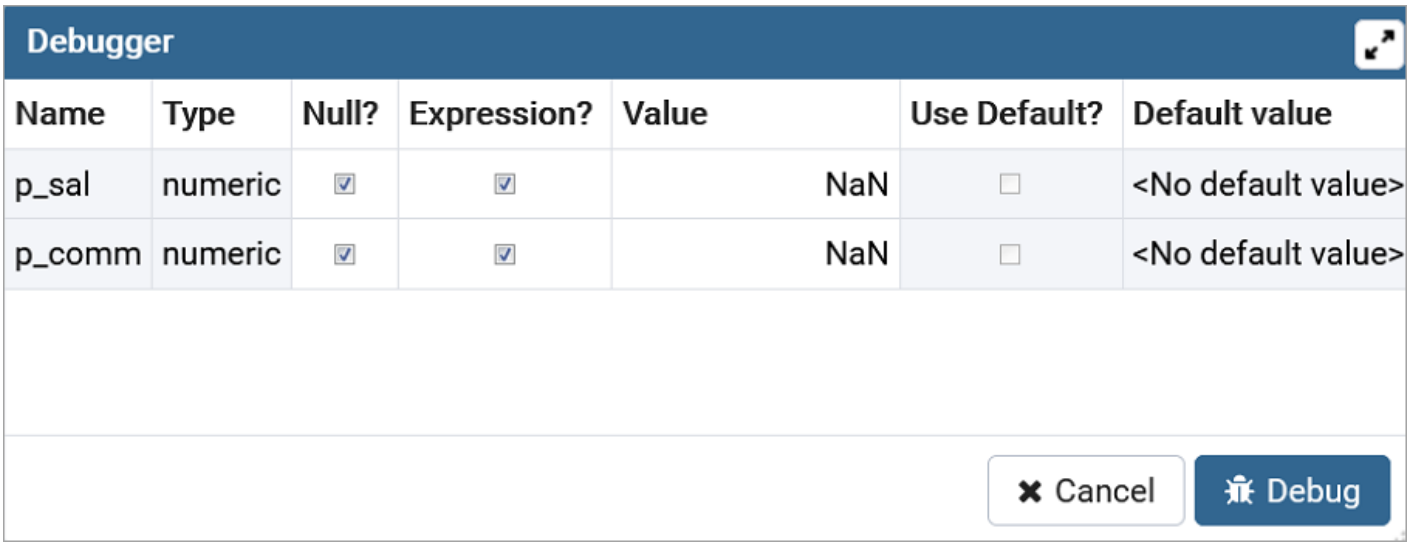


Fig. 3: The Debugger window

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.
- Check the `Null?` checkbox to indicate that the parameter is a `NULL` value.
- Check the `Expression?` checkbox if the `Value` field contains an expression.
- The `Value` field contains the parameter value that will be passed to the program.
- Check the `Use Default?` checkbox to indicate that the program should use the value in the `Default Value` field.
- The `Default Value` field contains the default value of the parameter.

Press the `Tab` key to select the next parameter in the list for data entry, or click on a `Value` field to select the parameter for data entry.

If you are debugging a procedure or function that is a member of a package that has an initialization section, check the `Debug Package Initializer` check box to instruct the Debugger to step into the package initialization section, allowing you to debug the initialization section code before debugging the procedure or function. If you do not 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 are executed.

After entering the desired parameter values, click the `Debug` button to start the debugging process. Click the `Cancel` button to terminate the Debugger.

> **Note**
>
> The `Debugger` window does not open during in-context debugging. Instead, the application calling the program to be debugged must supply

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> any required input parameter values.

When you have completed a full debugging cycle by stepping through the program code, the `Debugger` window re-opens, allowing you to enter new parameter values and repeat the debugging cycle, or end the debugging session.

Main Debugger Window

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.

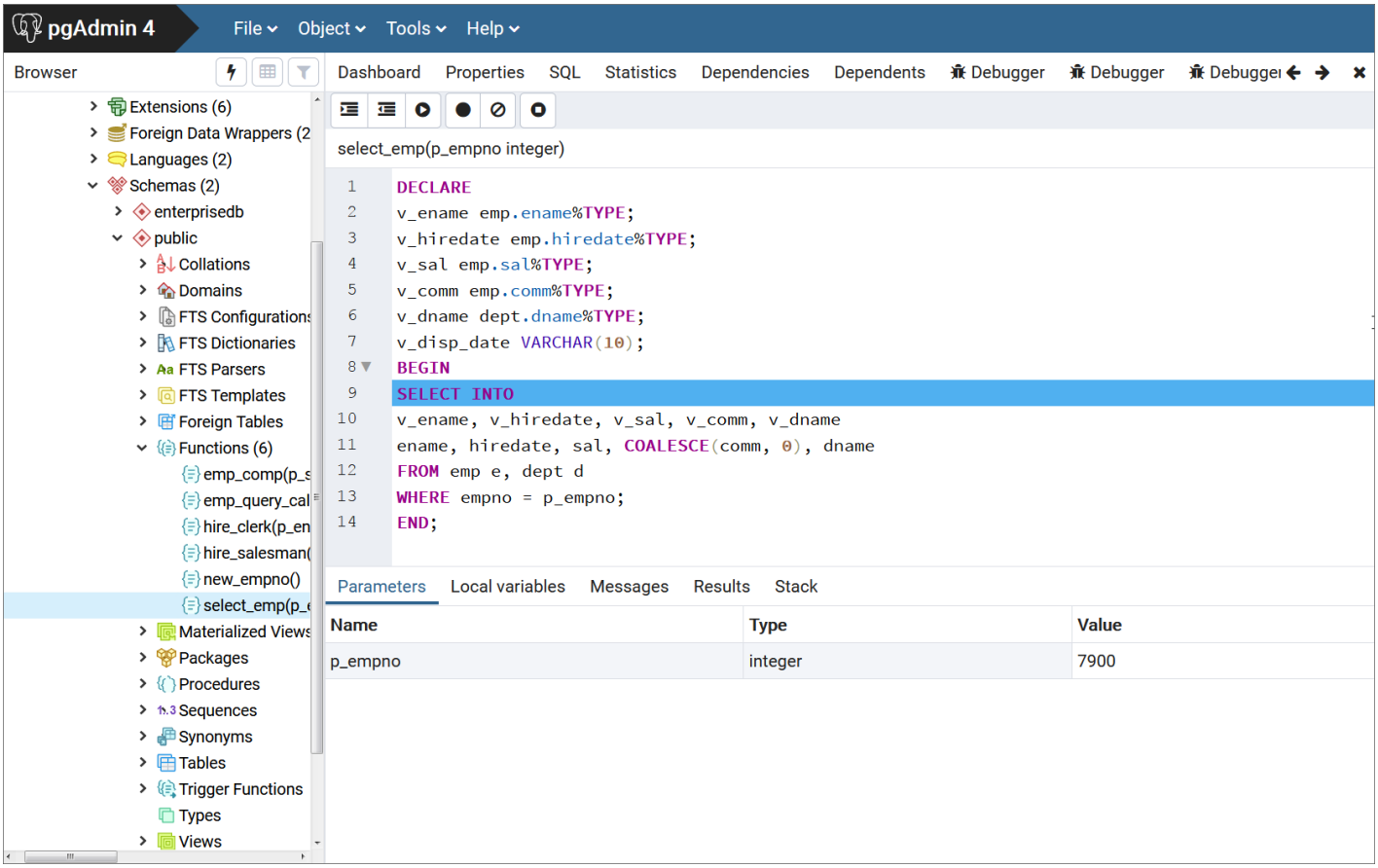


Fig. 4: The Main Debugger window

The two panels are described in the following sections.

The Program Body Panel

The `Program Body` panel displays the source code of the program that is being debugged.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

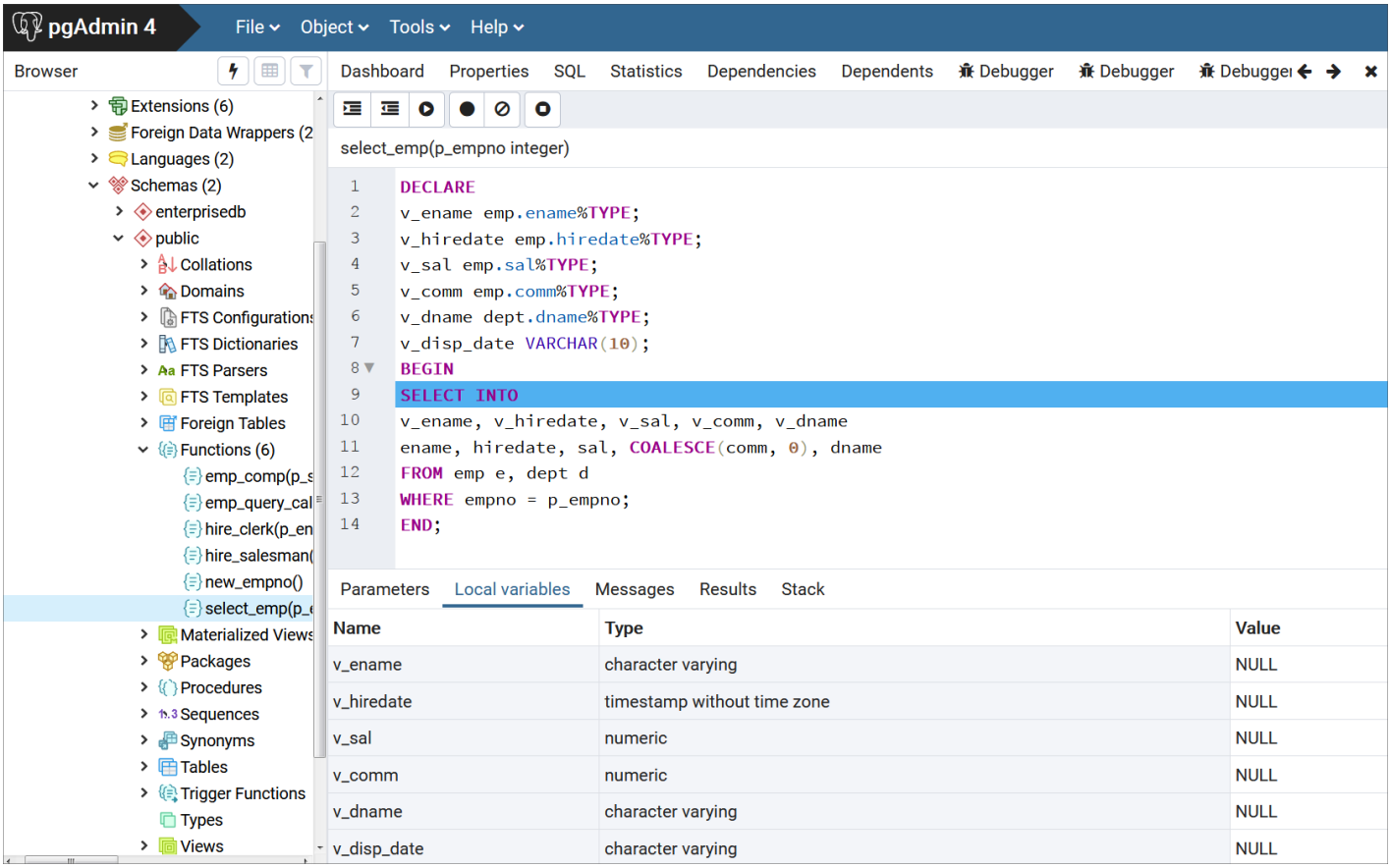


Fig. 5: The Program Body

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 `RAISE INFO` and function results.

The following is the information displayed by the tabs in the panel:

- The `Parameters` tab displays the current parameter values.
- The `Local variables` tab displays the value of any variables declared within the program.
- The `Messages` tab displays any results returned by the program as it executes.
- The `Results` tab displays program results (if applicable) such as the value from the `RETURN` statement of a function.
- The `Stack tab` displays the call stack.

The following figures show the results from the various tabs.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

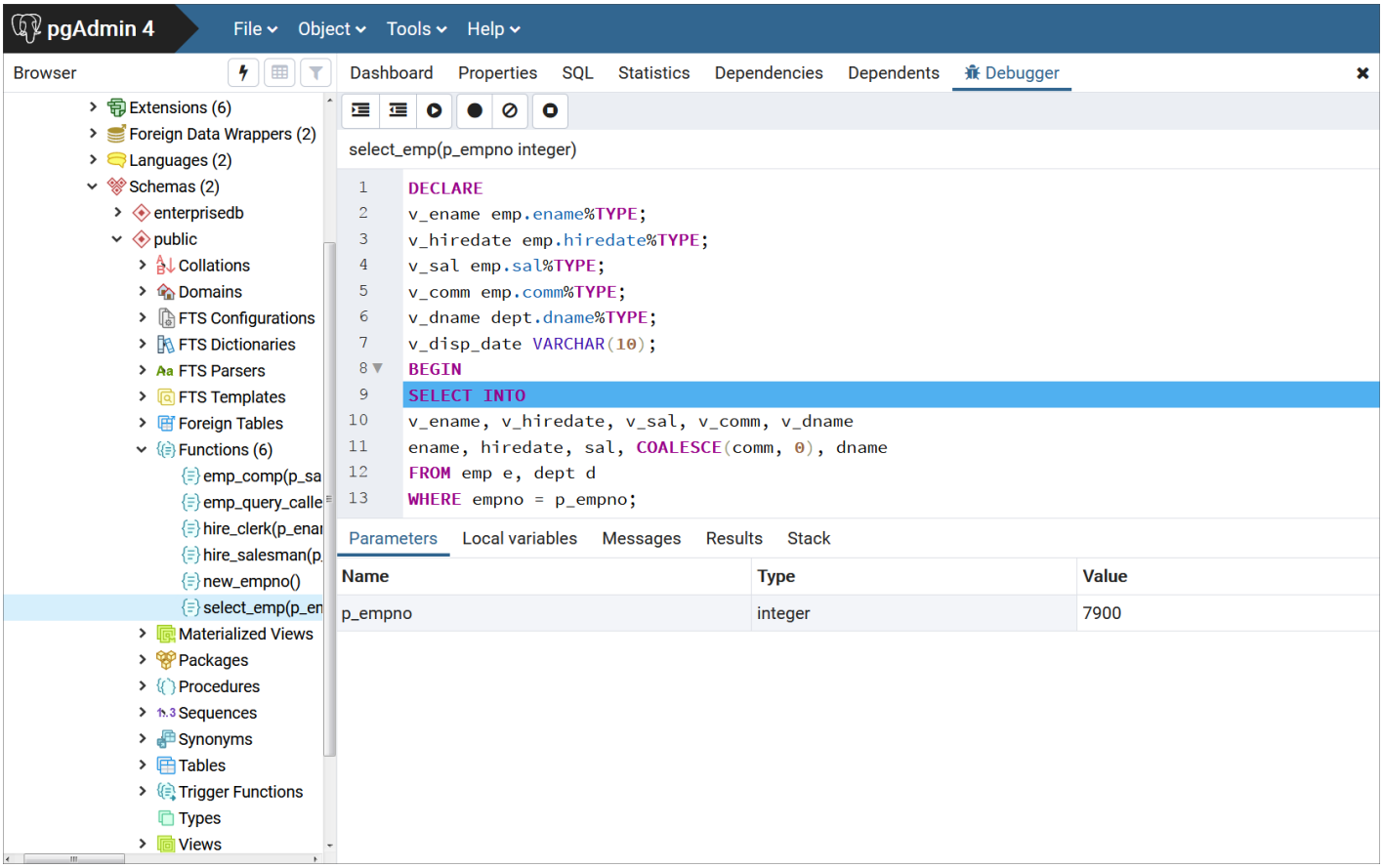


Fig. 6: The Parameters tab

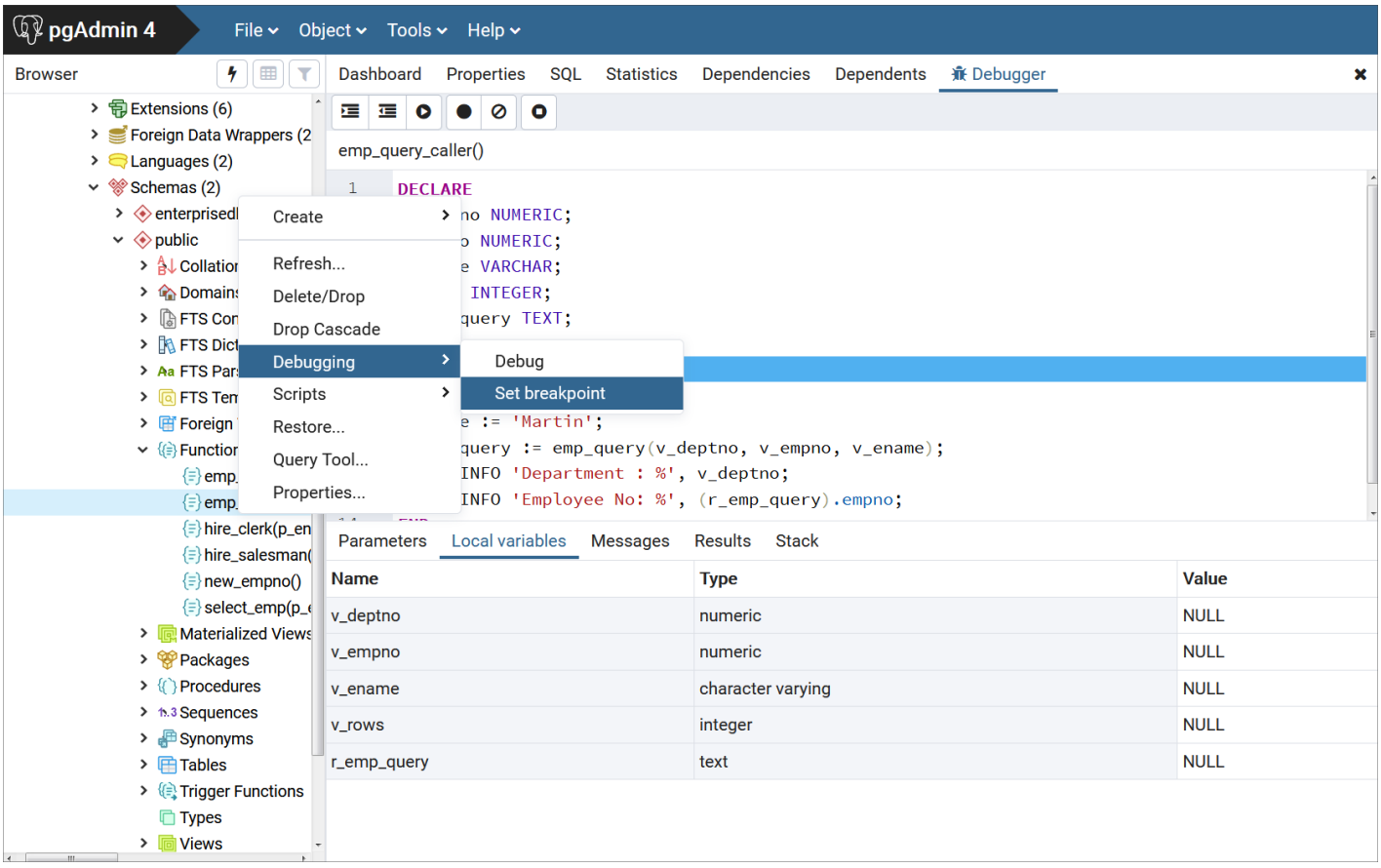


Fig. 7: The Local variables tab

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

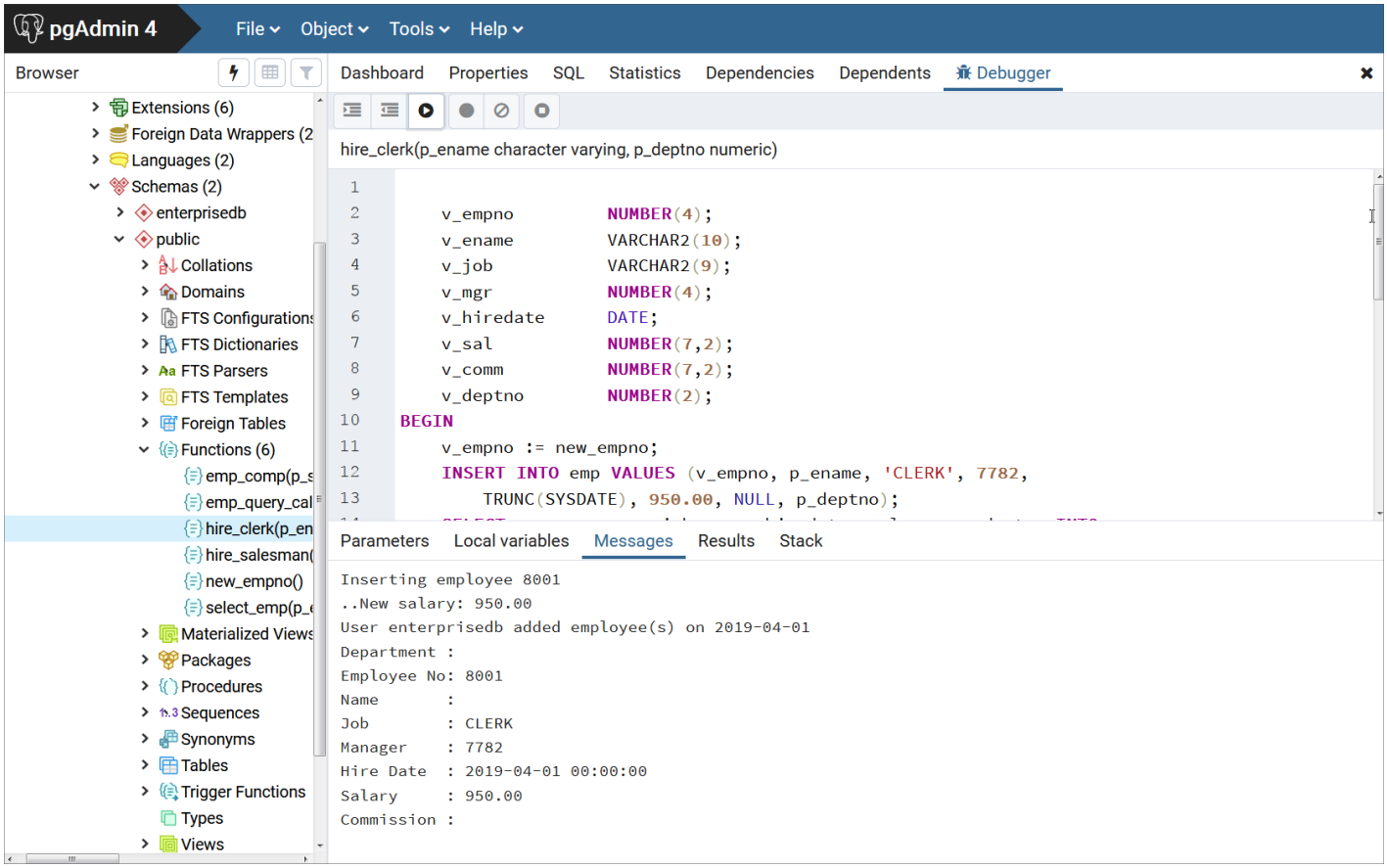


Fig. 8: The Messages tab

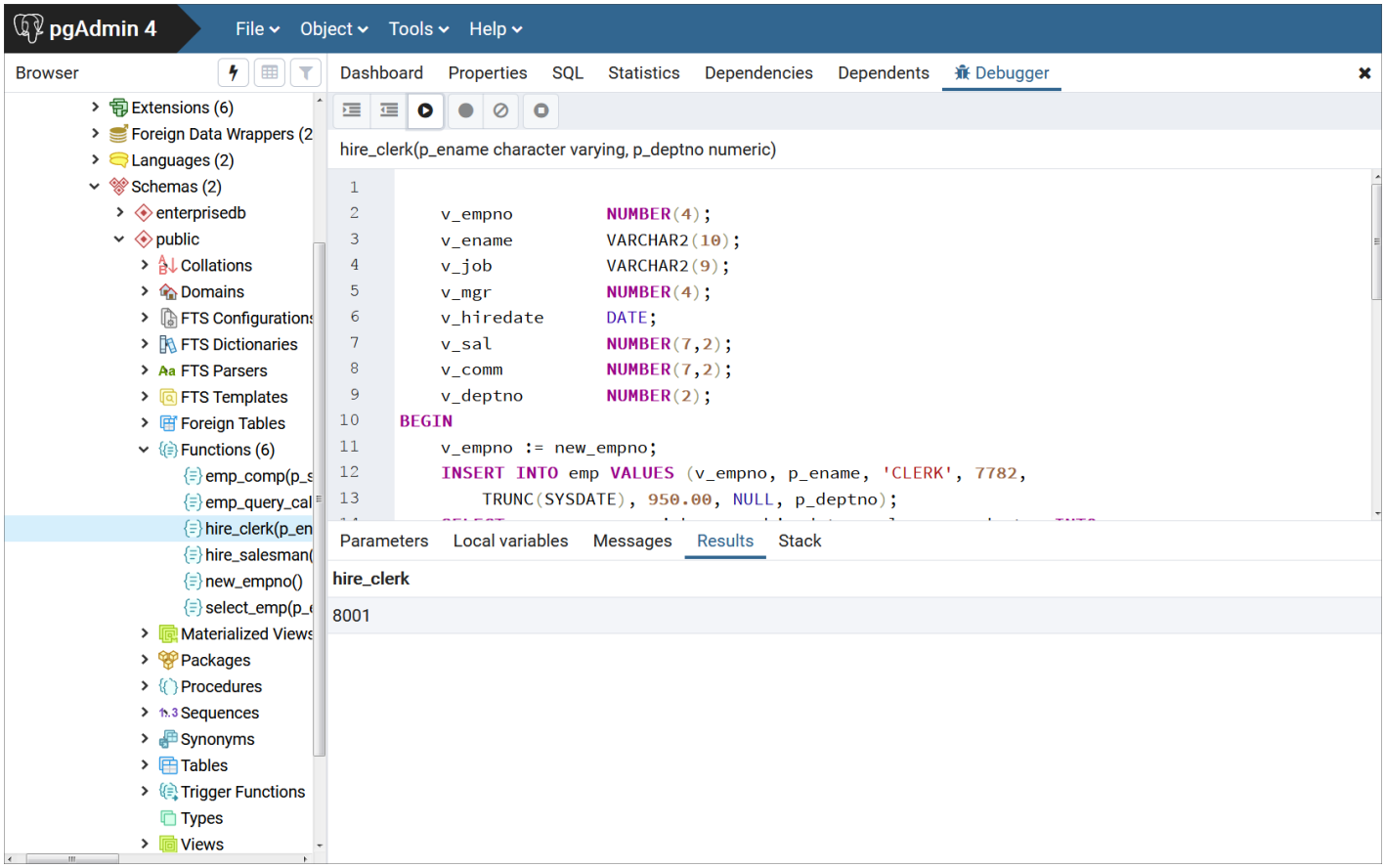


Fig. 9: The Results tab

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The Stack Tab

The `Stack` tab displays a list of programs that are currently on the call stack (programs that have been invoked, but which have not yet completed). When a program is called, the name of the program is added to the top of the list displayed in the `Stack` tab. When the program ends, its name is removed from the list.

The `Stack` tab also displays information about program calls. The information includes:

- The location of the call within 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.

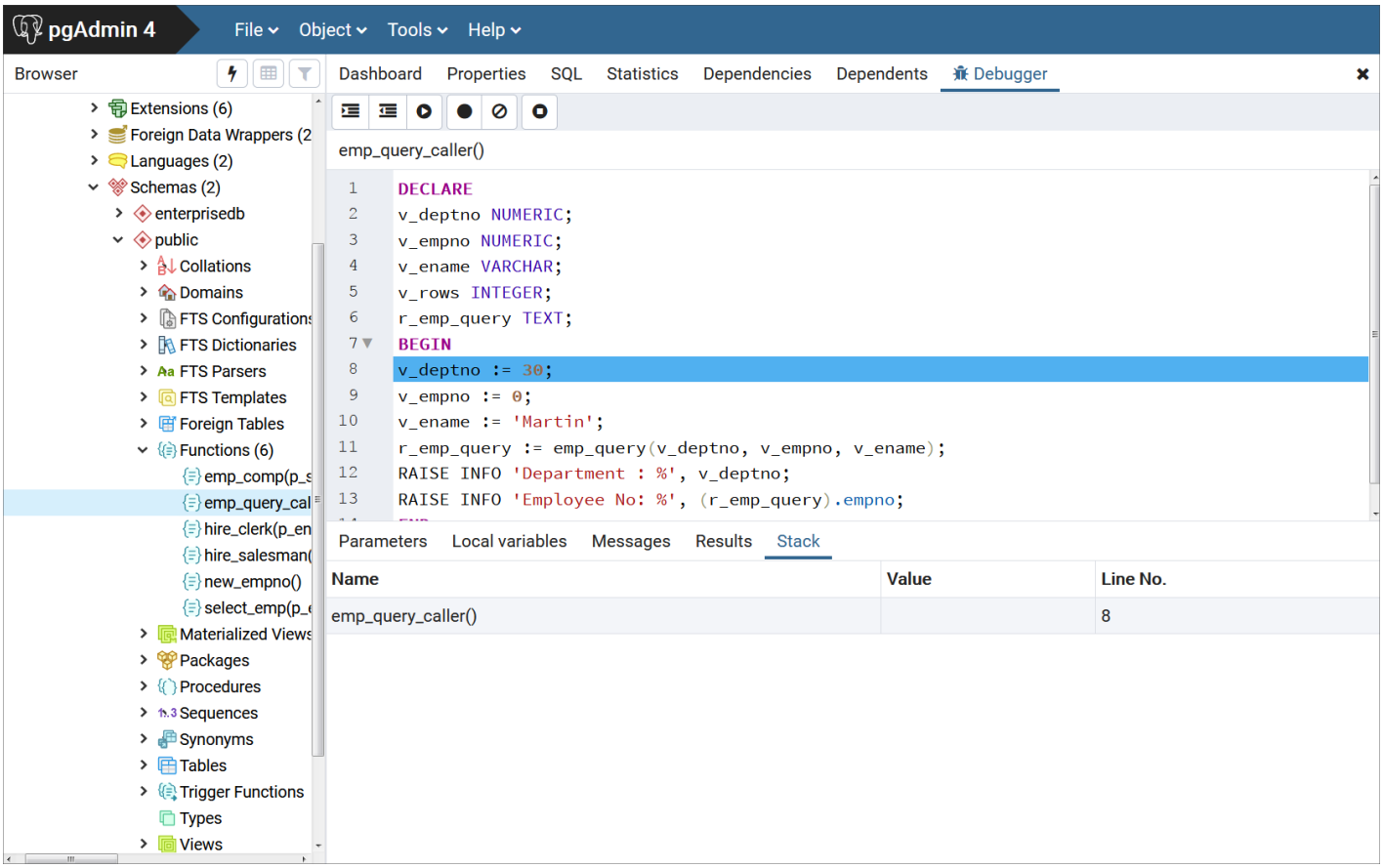


Fig. 10: A debugged program calling a subprogram

The above figure shows that `emp_query_caller` is about to call a subprogram named `emp_query`. `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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

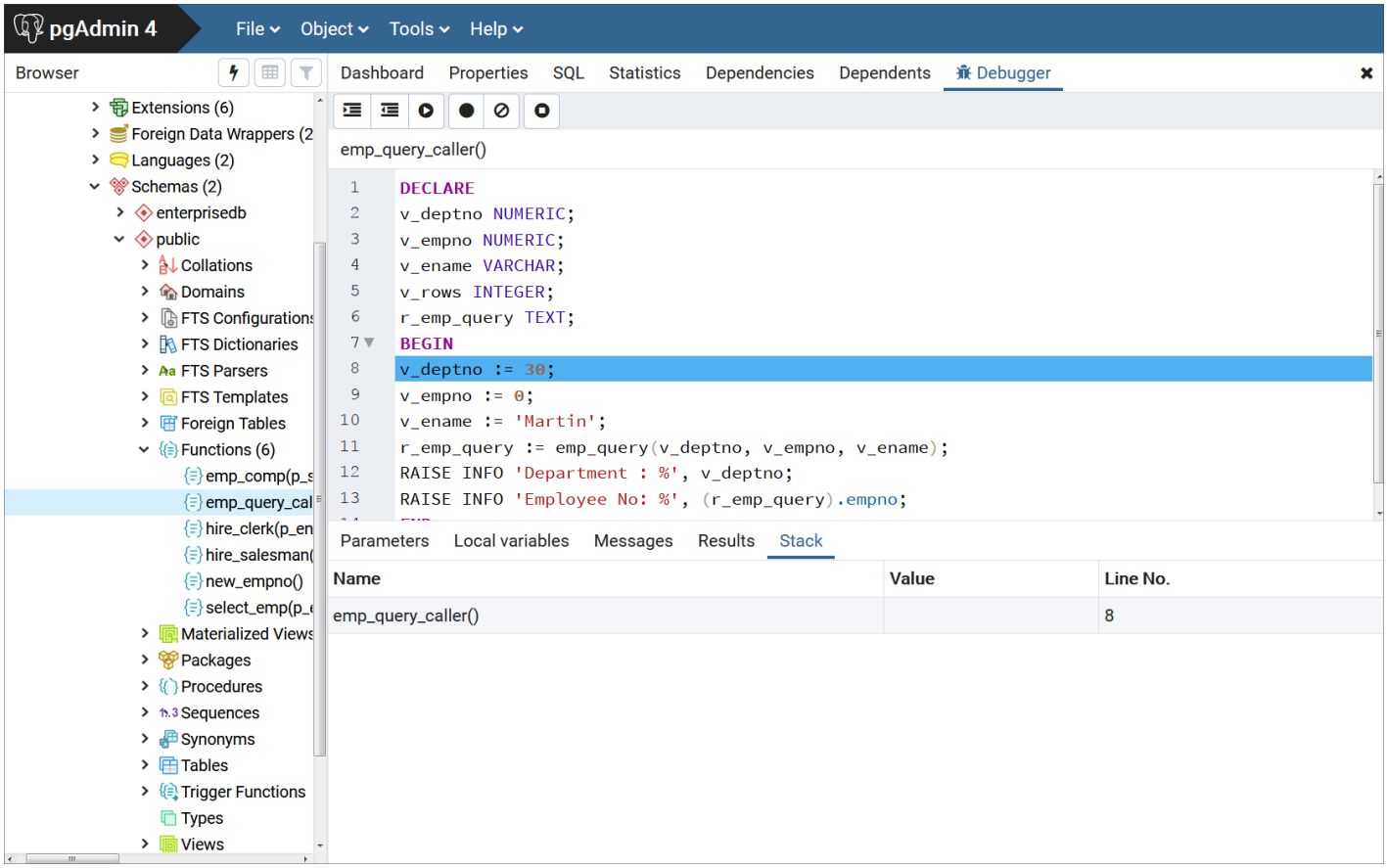
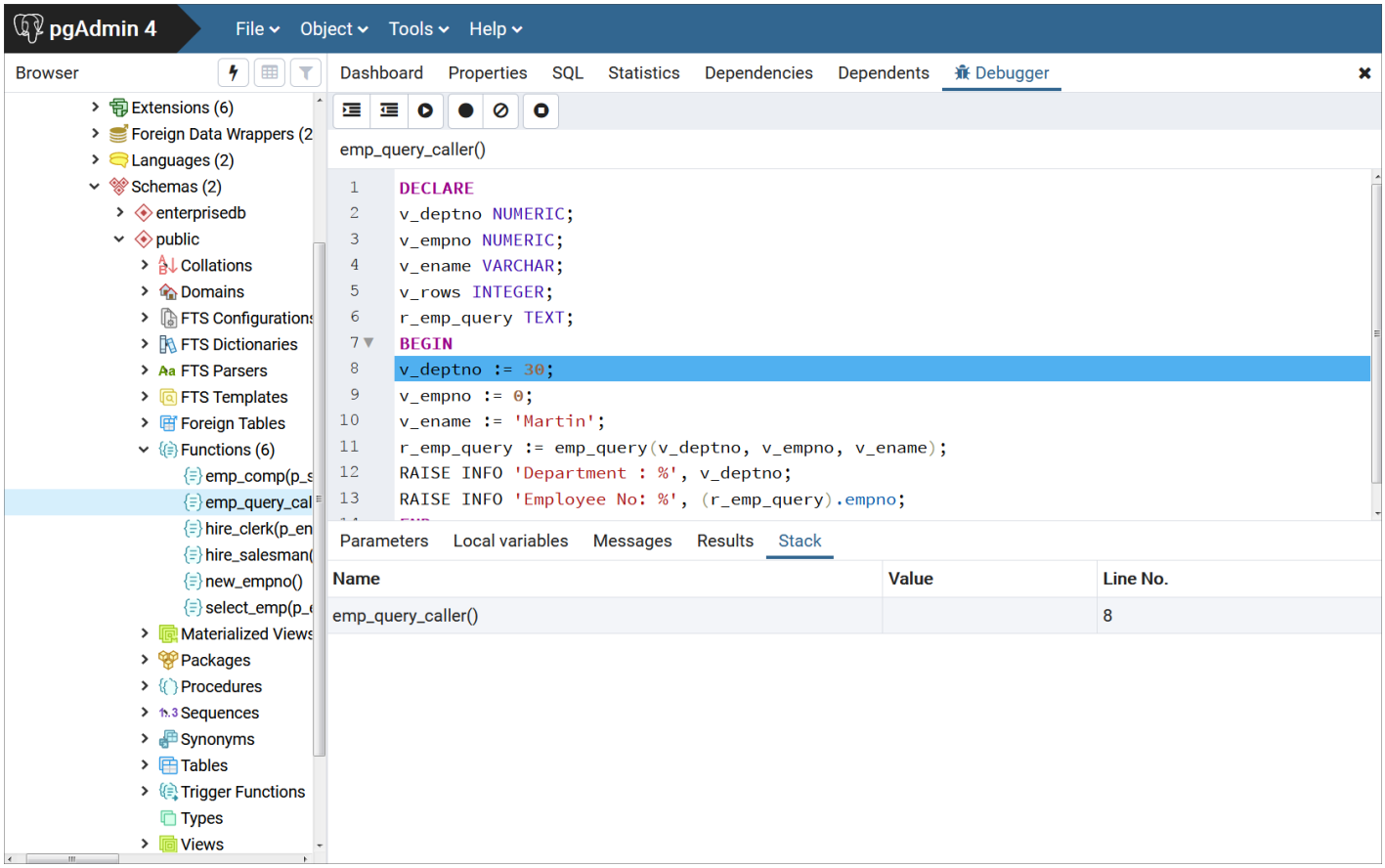


Fig. 11: Debugging the called subprogram

Upon completion of execution of the subprogram, control returns to the calling program `(emp_query_caller)`, now displayed at the top of the `Stack` tab.



Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Fig. 12: Control returns from debugged subprogram

Debugging a Program

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

Stepping Through the Code

Use the tool bar icons to step through a program with the Debugger:

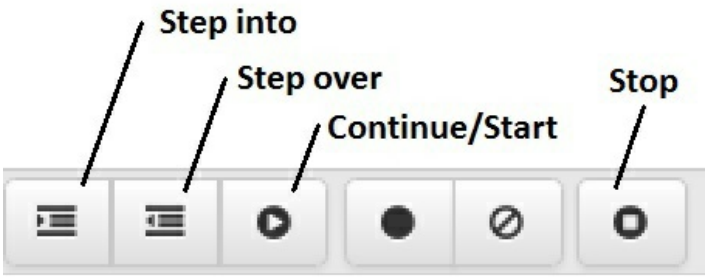


Fig. 13: The Tool bar icons

The icons serve the following purposes:

- **Step into.** Click the `Step into` icon to execute the currently highlighted line of code.
- **Step over.** Click the `Step over` icon to execute a line of code, stepping over any sub-functions invoked by the code. The sub-function executes, but is not debugged unless it contains a breakpoint.
- **Continue/Start.** Click the `Continue/Start` icon to execute the highlighted code, and continue until the program encounters a breakpoint or completes.
- **Stop.** Click the `Stop` icon to halt the execution of a program.

Using Breakpoints

As the Debugger executes a program, it pauses whenever 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 execution with the next line of code following the breakpoint. There are two types of breakpoints:

Local Breakpoint - A local breakpoint can be set at any executable line of code within a program. The Debugger pauses execution when it reaches a line where a local breakpoint has been set.

Global Breakpoint - A global breakpoint will trigger when *any* session reaches that breakpoint. Set a global breakpoint if you want to perform in-context debugging of a program. When a global breakpoint is set on a program, the debugging session that set the global breakpoint waits until that program is invoked in another session. A global breakpoint can only be set by a superuser.

To create a local breakpoint, left-click within the grey shaded margin to the left of the line of code where you want the local breakpoint set. Where you click in the grey shaded margin should 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 created, the Debugger displays a dark dot in the margin, indicating a breakpoint has been set at the selected line of code.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

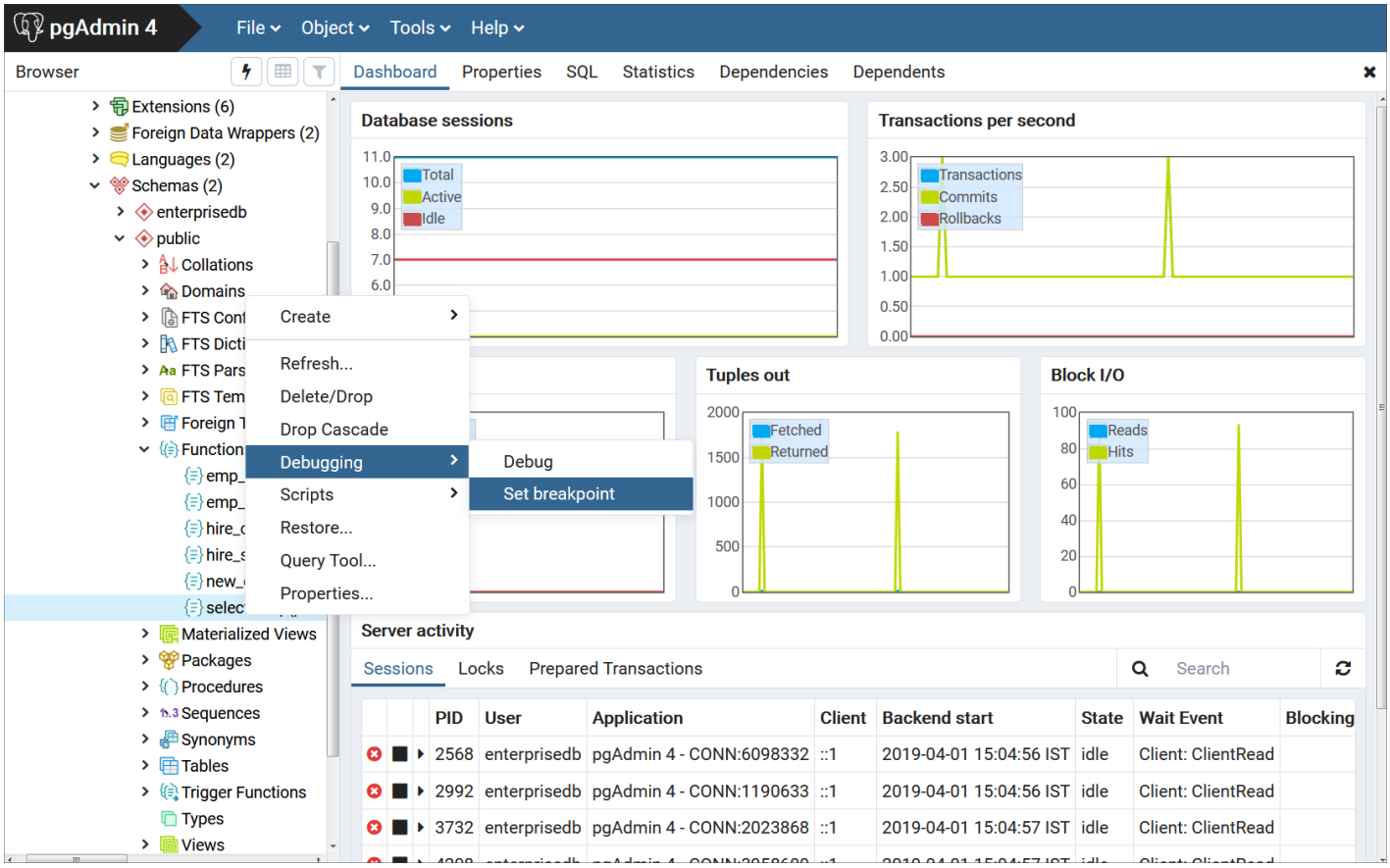


Fig. 14: Set a breakpoint by clicking in left-hand margin

You can set as many local breakpoints as desired. Local breakpoints remain in effect for the duration of a debugging session until they are removed.

Removing a Local Breakpoint

To remove a local breakpoint, left-click the mouse on the breakpoint dot in the grey shaded margin of the `Program Body` panel. The dot disappears, indicating that the breakpoint has been removed.

You can remove all of the breakpoints from the program that currently appears in the `Program Body` frame by clicking the `Clear all breakpoints` icon.

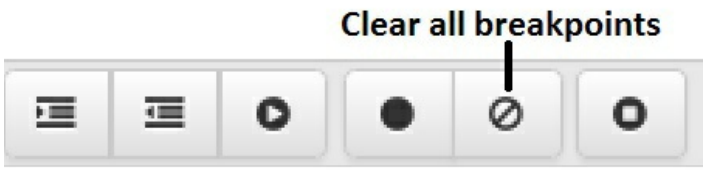


Fig. 15: Clear all breakpoints icon

> **Note**
>
> When you perform any of the preceding 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 are not removed.

Setting a Global Breakpoint for In-Context Debugging

To set a global breakpoint for in-context debugging, highlight the stored procedure, function, or trigger on which you wish to set the breakpoint in the

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`Browser` panel. Navigate through the `Object` menu to select `Debugging`, and then `Set Breakpoint`.

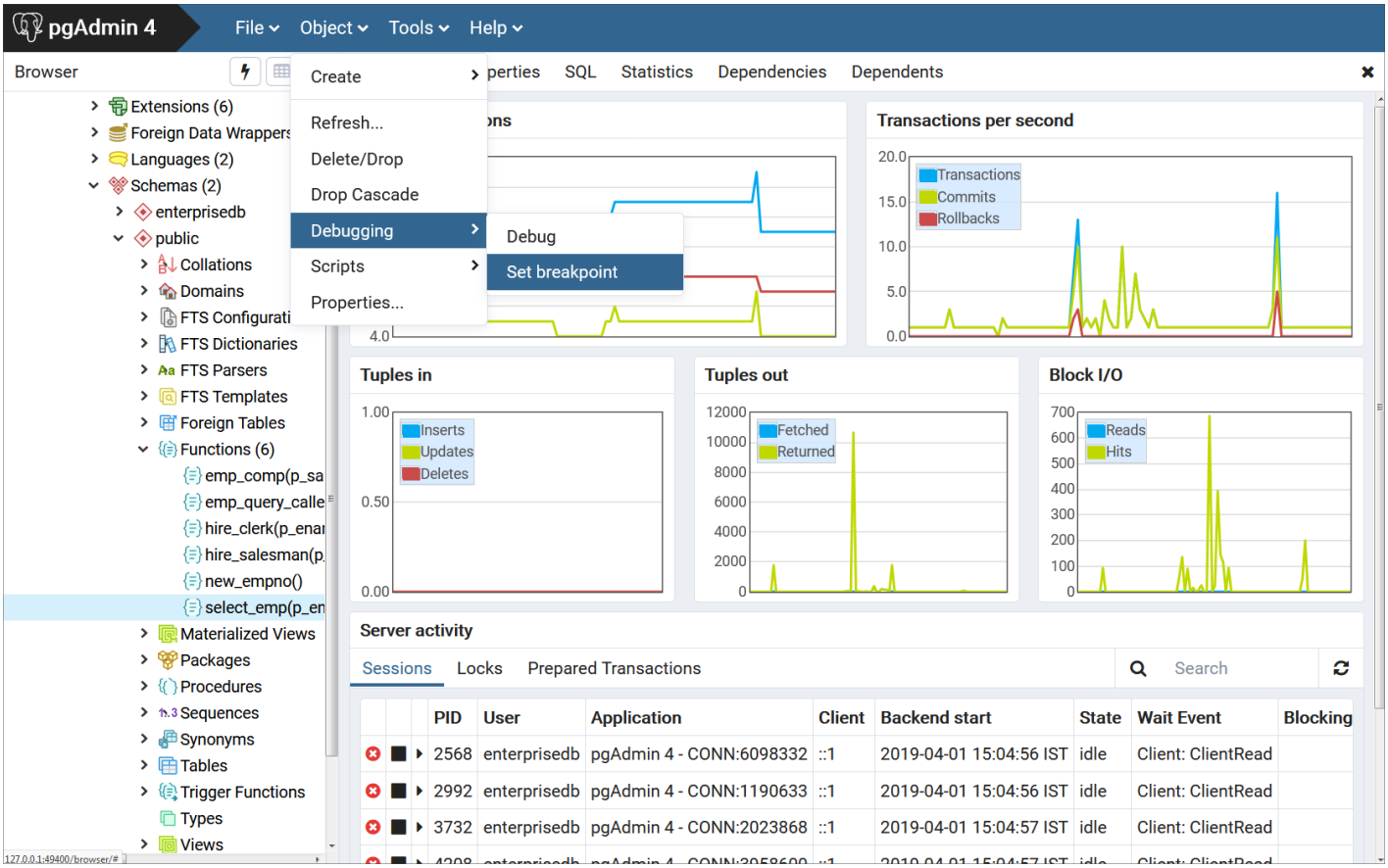


Fig. 16: Setting a global breakpoint from the Object menu

Alternatively, you can right-click on the name of the stored procedure, function, or trigger on which you wish to set a global breakpoint and select `Debugging`, then `Set Breakpoint` from the context menu as shown by the following.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

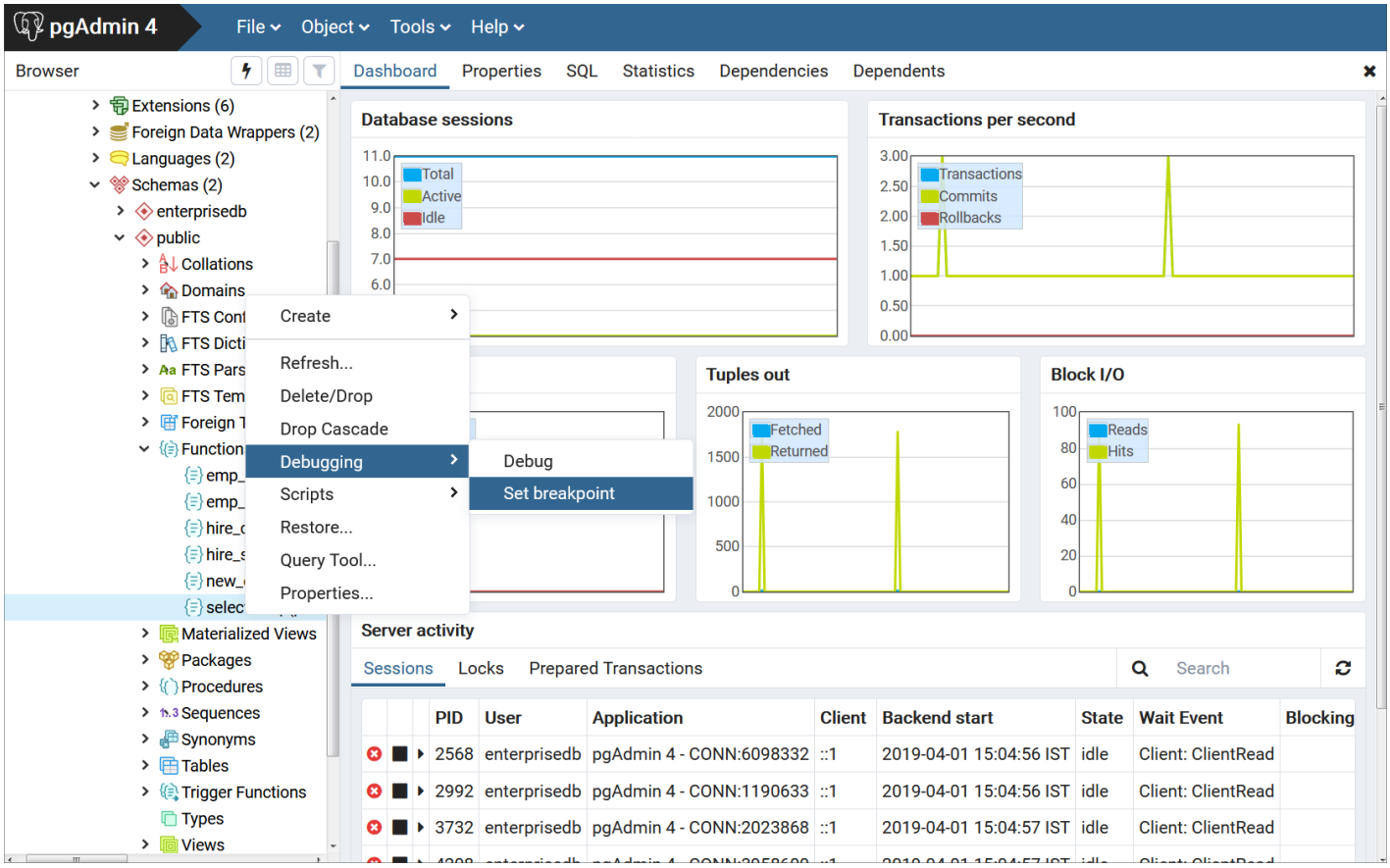


Fig. 17: Setting a global breakpoint from the Object's context menu

To set a global breakpoint on a trigger, expand the table node that contains the trigger, highlight the specific trigger you wish to debug, and follow the same directions as for stored procedures and functions.

To set a global breakpoint in a package, highlight the specific procedure or function under the package node of the package you wish to debug and follow the same directions as for stored procedures and functions.

After you choose `Set Breakpoint`, the Debugger window opens and waits for an application to call the program to be debugged.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

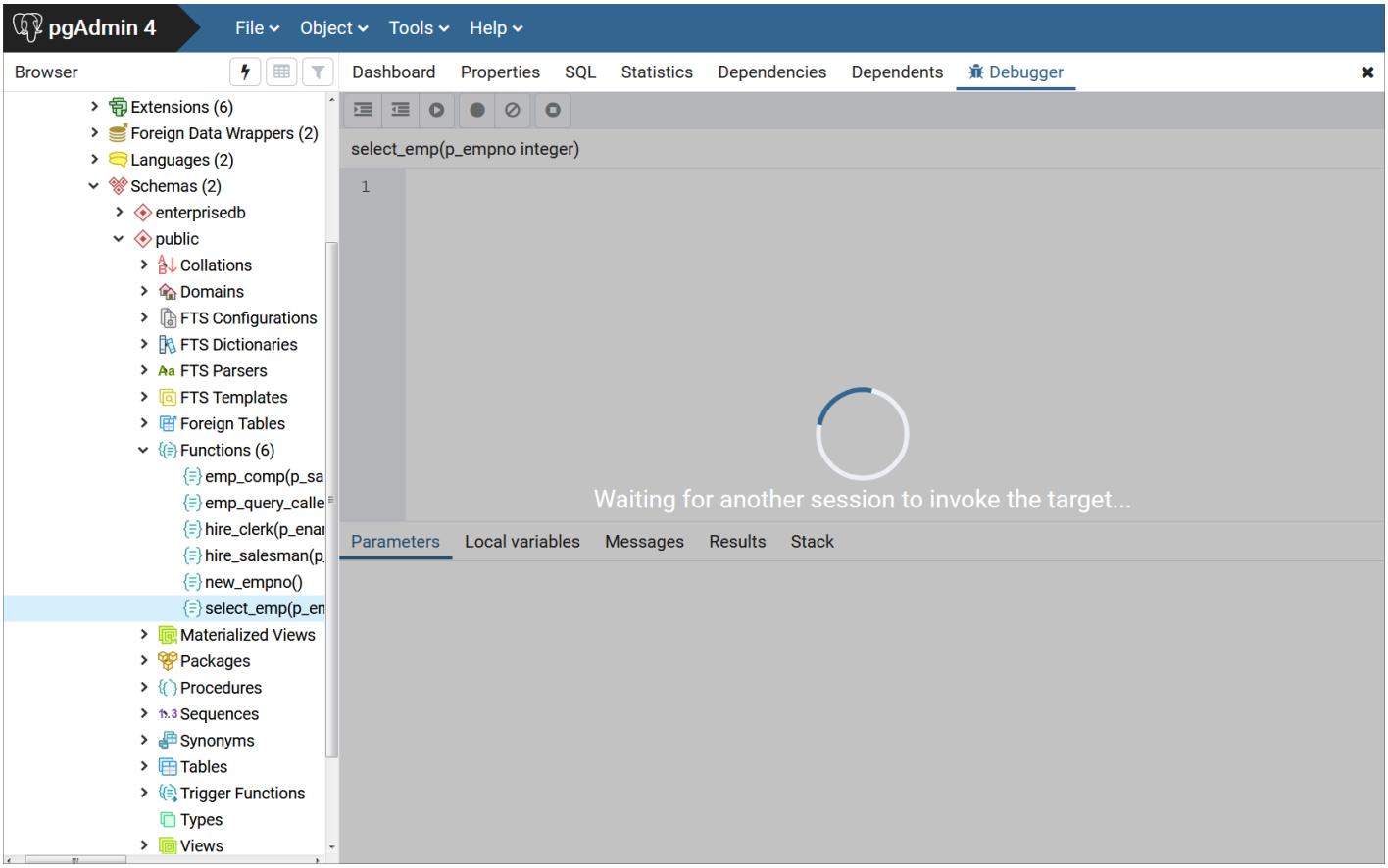


Fig. 18: Waiting for invocation of program to be debugged

The PSQL client invokes the `select_emp` function (on which a global breakpoint has been set).

```
$ psql edb enterprisedb
psql.bin (11.0.2, server 11.0.2)
Type "help" for help.

edb=# SELECT select_emp(7900);
```

The `select_emp` function does not complete until you step through the program in the Debugger.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

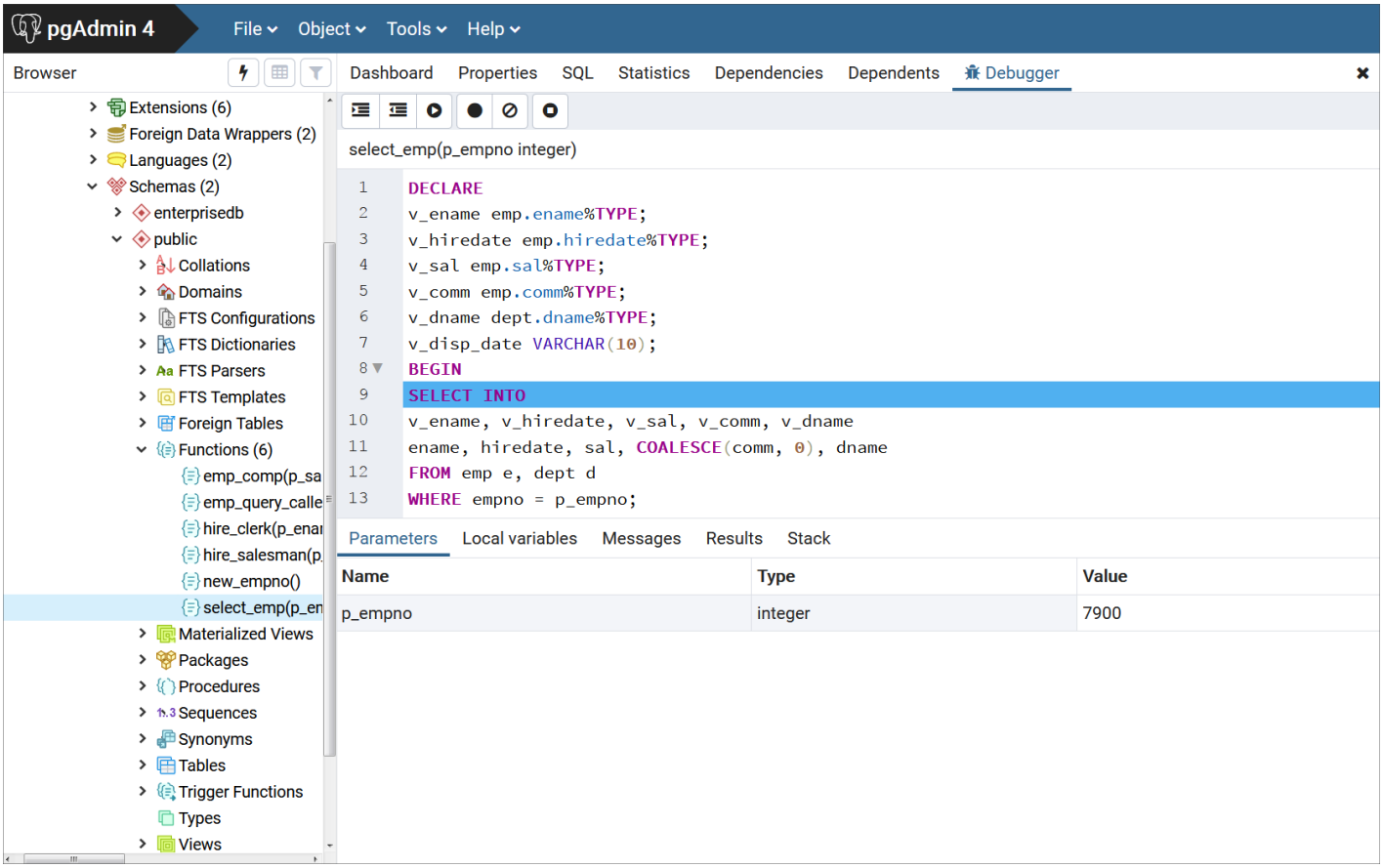


Fig. 19: Program on which a global breakpoint has been set

You can now debug the program using any of the previously discussed operations such as step into, step over, and continue, or set local breakpoints. When you have stepped through execution of the program, the calling application (PSQL) regains control and the `select_emp` function completes execution and its output is displayed.

```
 $ psql edb enterprisedb
psql.bin (11.0.2, server 11.0.2)
Type "help" for help.

edb=# SELECT select_emp(7900);
INFO: Number    : 7900
INFO: Name      : JAMES
INFO: Hire Date : 12/03/1981
INFO: Salary    : 950.00
INFO: Commission: 0.00
INFO: Department: SALES
 select_emp
------------
(1 row)
```

At this point, you can end the Debugger session. If you do not end the Debugger session, the next application that invokes the program will encounter the global breakpoint and the debugging cycle will begin again.

Exiting the Debugger

To end a Debugger session and exit the Debugger, click on the close icon `(x)` located in the upper-right corner to close the tab.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

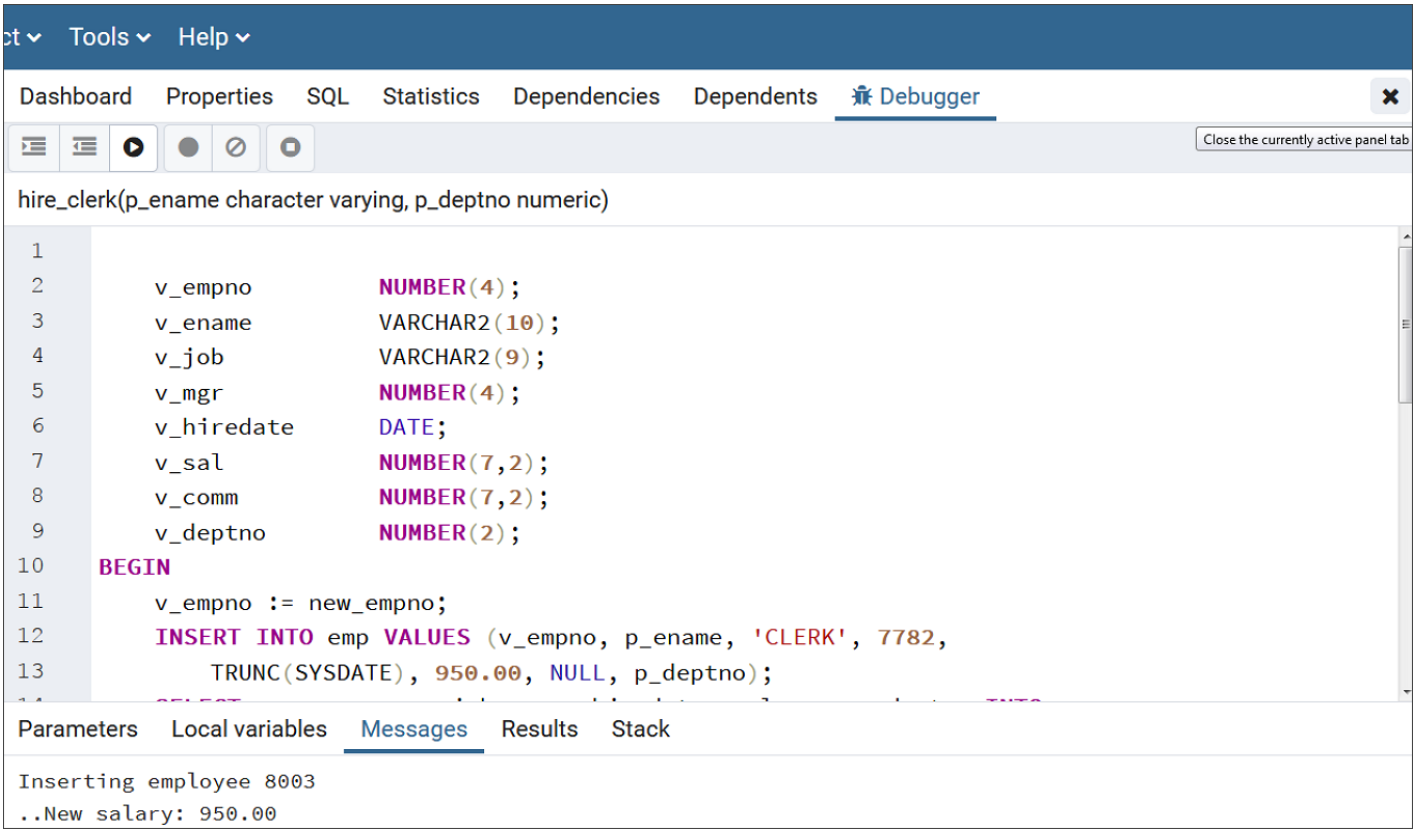


Fig. 20: Exiting from the Debugger

20.7 Performance Analysis and Tuning

Advanced Server provides various tools for performance analysis and tuning. These features are described in this section.

Dynatune

Advanced Server supports dynamic tuning of the database server to make the optimal usage of the system resources available on the host machine on which it is installed. The two parameters that control this functionality are located in the `postgresql.conf` file. These parameters are:

- `edb_dynatune`
- `edb_dynatune_profile`

edb_dynatune

`edb_dynatune` determines how much of the host system's resources are to be used by the database server based upon the host machine's total available resources and the intended usage of the host machine.

When Advanced Server is initially installed, the `edb_dynatune` parameter is set in accordance with the selected usage of the host machine on which it was installed - i.e., development machine, mixed use machine, or dedicated server. For most purposes, there is no need for the database administrator to adjust the various configuration parameters in the `postgresql.conf` file in order to improve performance.

You can change the value of the `edb_dynatune` parameter after the initial installation of Advanced Server by editing the `postgresql.conf` file. The postmaster must be restarted in order for the new configuration to take effect.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `edb_dynatune` parameter can be set to any integer value between `0` and `100`, inclusive. A value of `0`, turns off the dynamic tuning feature thereby leaving the database server resource usage totally under the control of the other configuration parameters in the `postgresql.conf` file.

A low non-zero, value (e.g., 1 - 33) dedicates the least amount of the host machine's resources to the database server. This setting would be used 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. This setting might be used for a dedicated application server that may have a fixed number of other applications running on the same machine as Advanced Server.

The highest values (e.g., 67 - 100) dedicate most of the server's resources to the database server. This setting would be used for a host machine that is totally dedicated to running Advanced Server.

Once a value of `edb_dynatune` is selected, database server performance can be further fine-tuned by adjusting the other configuration parameters in the `postgresql.conf` file. Any adjusted setting overrides the corresponding value chosen by `edb_dynatune`. You can change the value of a parameter by un-commenting the configuration parameter, specifying the desired value, and restarting the database server.

edb_dynatune_profile

The `edb_dynatune_profile` parameter is used to control tuning aspects based upon the expected workload profile on the database server. This parameter takes effect upon startup of the database server.

The possible values for `edb_dynatune_profile` are:

Value	Usage
`oltp`	Recommended when the database server is processing heavy online transaction processing workloads.
`reporting`	Recommended for database servers used for heavy data reporting.
`mixed`	Recommended for servers that provide a mix of transaction processing and data reporting.

EDB Wait States

The *EDB wait states* contrib module contains two main components.

EDB Wait States Background Worker (EWSBW)

When the wait states background worker is registered as one of the shared preload libraries, EWSBW probes each of the running sessions at regular intervals.

For every session it collects information such as the database to which it is connected, the logged in user of the session, the query running in that session, and the wait events on which it is waiting.

This information is saved in a set of files in a user-configurable path and directory folder given by the `edb_wait_states.directory` parameter to be added to the `postgresql.conf` file. The specified path must be a full, absolute path and not a relative path.

The following describes the installation process on a Linux system.

Step 1: EDB wait states is installed with the `edb-asxx-server-edb_wait_states` RPM package where `xx` is the Advanced Server version number.

Step 2: To launch the worker, it must be registered in the `postgresql.conf` file using the `shared_preload_libraries` parameter, for example:

```
shared_preload_libraries = '$libdir/edb_wait_states'
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Step 3: Restart the database server. After a successful restart, the background worker begins collecting data.

Step 4: To review the data, create the following extension:

```
CREATE EXTENSION edb_wait_states;
```

Step 5: To terminate the EDB wait states worker, remove `$libdir/edb_wait_states` from the `shared_preload_libraries` parameter and restart the database server.

The following describes the installation process on a Windows system.

Step 1: EDB wait states module is installed with the `EDB Modules` installer by invoking StackBuilder Plus utility. Follow the onscreen instructions to complete the installation of the `EDB Modules`.

Step 2: To register the worker, modify the `postgresql.conf` file to include the wait states library in the `shared_preload_libraries` configuration parameter. The parameter value must include:

```
shared_preload_libraries = '$libdir/edb_wait_states.dll'
```

The EDB wait states installation places the `edb_wait_states.dll` library file in the following path:

```
C:\Program Files\edb\as11\lib\
```

Step 3: Restart the database server for the changes to take effect. After a successful restart, the background worker gets started and starts collecting the data.

Step 4: To view the data, create the following extension:

```
CREATE EXTENSION edb_wait_states;
```

The installer places the `edb_wait_states.control` file in the following path:

```
C:\Program Files\edb\as11\share\extension
```

Terminating the Wait States Worker

To terminate the EDB wait states worker, use the `DROP EXTENSION` command to drop the `edb_wait_states` extension; then modify the `postgresql.conf` file, removing `$libdir/edb_wait_states.dll` from the `shared_preload_libraries` parameter. Restart the database server after modifying the `postgresql.conf` file to apply your changes.

The Wait States Interface

The interface includes the functions listed in the following sections. Each of these functions has common input and output parameters. Those parameters are as follows:

- **start_ts and end_ts (IN).** Together these specify the time interval and the data within which is to be read. If only `start_ts` is specified, the data starting from `start_ts` is output. If only `end_ts` is provided, data up to `end_ts` is output. If none of those are provided, all the data is output. Every function outputs different data. The output of each function will be explained below.
- **query_id (OUT).** Identifies a normalized query. It is internal hash code computed from the query.
- **session_id (OUT).** Identifies a session.
- **ref_start_ts and ref_end_ts (OUT).** Provide the timestamps of a file containing a particular data point. A data point may be a wait event sample record or a query record or a session record.

The syntax of each function is given in the following sections.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> **Note**
>
> The examples shown in the following sections are based on the following three queries executed on four different sessions connected to different databases using different users, simultaneously:

```
SELECT schemaname FROM pg_tables, pg_sleep(15) WHERE schemaname <>
'pg_catalog'; /* ran on 2 sessions */
SELECT tablename FROM pg_tables, pg_sleep(10) WHERE schemaname <>
'pg_catalog';
SELECT tablename, schemaname FROM pg_tables, pg_sleep(10) WHERE schemaname
<> 'pg_catalog';
```

edb_wait_states_data

This function is used to read the data collected by EWSBW.

```
edb_wait_states_data(
  IN start_ts timestamptz default '-infinity'::timestamptz,
  IN end_ts timestamptz default 'infinity'::timestamptz,
  OUT session_id int4,
  OUT <dbname> text,
  OUT <username> text,
  OUT <query> text,
  OUT <query_start_time> timestamptz,
  OUT <sample_time> timestamptz,
  OUT <wait_event_type> text,
  OUT <wait_event> text
)
```

This function can be used to find out the following:

The queries running in the given duration (defined by `start_ts` and `end_ts`) in all the sessions, and the wait events, if any, they were waiting on. For example:

```
SELECT query, session_id, wait_event_type, wait_event
  FROM edb_wait_states_data(start_ts, end_ts);
```

The progress of a session within a given duration (that is, the queries run in a session (session_id = 100000) and the wait events the queries waited on). For example:

```
SELECT query, wait_event_type, wait_event
  FROM edb_wait_states_data(start_ts, end_ts)
  WHERE session_id = 100000;
```

The duration for which the samples are available. For example:

```
SELECT min(sample_time), max(sample_time)
  FROM edb_wait_states_data();
```

Parameters

In addition to the common parameters described previously, each row of the output gives the following:

`dbname`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The session's database

`username`

The session's logged in user

`query`

The query running in the session

`query_start_time`

The time when .the query started

`sample_time`

The time when wait event data was collected

`wait_event_type`

The type of wait event the session (backend) is waiting on

`wait_event`

The wait event the session (backend) is waiting on

Example

The following is a sample output from the `edb_wait_states_data()` function.

```
edb=# SELECT * FROM edb_wait_states_data();
-[ RECORD 1 ]----+-------------------------------------------------------------------------
session_id       | 4398
dbname           | edb
username         | enterprisedb
query            | SELECT schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
query_start_time | 17-AUG-18 11:56:05.271962 -04:00
sample_time      | 17-AUG-18 11:56:19.700236 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
-[ RECORD 2 ]----+-------------------------------------------------------------------------
session_id       | 4398
dbname           | edb
username         | enterprisedb
query            | SELECT schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
query_start_time | 17-AUG-18 11:56:05.271962 -04:00
sample_time      | 17-AUG-18 11:56:18.699938 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
-[ RECORD 3 ]----+-------------------------------------------------------------------------
session_id       | 4398
dbname           | edb
username         | enterprisedb
query            | SELECT schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
query_start_time | 17-AUG-18 11:56:05.271962 -04:00
sample_time      | 17-AUG-18 11:56:17.700253 -04:00
wait_event_type  | Timeout
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
wait_event       | PgSleep
                    .
                    .
                    .
```

edb_wait_states_queries

This function gives information about the queries sampled by EWSBW.

```
edb_wait_states_queries(
  IN start_ts timestamptz default '-infinity'::timestamptz,
  IN end_ts timestamptz default 'infinity'::timestamptz,
  OUT query_id int8,
  OUT <query> text,
  OUT ref_start_ts timestamptz
  OUT ref_end_ts timestamptz
)
```

A new queries file is created periodically and thus, there can be multiple query files generated corresponding to specific intervals.

This function returns all the queries in query files that overlap with the given time interval. A query as shown below, gives all the queries in query files that contained queries sampled between `start_ts` and `end_ts`.

In other words, the function may output queries that did not run in the given interval. To exactly know that the user should use `edb_wait_states_data()`.

```
SELECT query FROM edb_wait_states_queries(start_ts, end_ts);
```

Parameters

In addition to the common parameters described previously, each row of the output gives the following:

`query`

Normalized query text

Example

The following is a sample output from the `edb_wait_states_queries()` function.

```
edb=# SELECT * FROM edb_wait_states_queries();
-[ RECORD 1 ]+-------------------------------------------------------------------------
query_id     | 4292540138852956818
query        | SELECT schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 2 ]+-------------------------------------------------------------------------
query_id     | 3754591102365859187
query        | SELECT tablename FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 3 ]+-------------------------------------------------------------------------
query_id     | 349089096300352897
query        | SELECT tablename, schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
```

edb_wait_states_sessions

This function gives information about the sessions sampled by EWSBW.

```
edb_wait_states_sessions(
  IN start_ts timestamptz default '-infinity'::timestamptz,
  IN end_ts timestamptz default 'infinity'::timestamptz,
  OUT session_id int4,
  OUT <dbname> text,
  OUT <username> text,
  OUT ref_start_ts timestamptz
  OUT ref_end_ts timestamptz
)
```

This function can be used to identify the databases that were connected and/or which users started those sessions. For example:

```
SELECT dbname, username, session_id
  FROM edb_wait_states_sessions();
```

Similar to `edb_wait_states_queries()`, this function outputs all the sessions logged in session files that contain sessions sampled within the given interval and not necessarily only the sessions sampled within the given interval. To identify that one should use `edb_wait_states_data()`.

Parameters

In addition to the common parameters described previously, each row of the output gives the following:

`dbname`

The database to which the session is connected

`username`

Login user of the session

Example

The following is a sample output from the `edb_wait_states_sessions()` function.

```
edb=# SELECT * FROM edb_wait_states_sessions();
-[ RECORD 1 ]+--------------------------------
session_id   | 4340
dbname       | edb
username     | enterprisedb
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 2 ]+--------------------------------
session_id   | 4398
dbname       | edb
username     | enterprisedb
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 3 ]+--------------------------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
session_id   | 4410
dbname       | db1
username     | user1
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 4 ]+--------------------------------
session_id   | 4422
dbname       | db2
username     | user2
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
```

edb_wait_states_samples

This function gives information about wait events sampled by EWSBW.

```
edb_wait_states_samples(
  IN start_ts timestamptz default '-infinity'::timestamptz,
  IN end_ts timestamptz default 'infinity'::timestamptz,
  OUT query_id int8,
  OUT session_id int4,
  OUT <query_start_time> timestamptz,
  OUT <sample_time> timestamptz,
  OUT <wait_event_type> text,
  OUT <wait_event> text
)
```

Usually, a user would not be required to call this function directly.

Parameters

In addition to the common parameters described previously, each row of the output gives the following:

`query_start_time`

The time when the query started in this session

`sample_time`

The time when wait event data was collected

`wait_event_type`

The type of wait event on which the session is waiting

`wait_event`

The wait event on which the session (backend) is waiting

Example

The following is a sample output from the `edb_wait_states_samples()` function.

```
edb=# SELECT * FROM edb_wait_states_samples();
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
-[ RECORD 1 ]----+---------------------------------
query_id         | 4292540138852956818
session_id       | 4340
query_start_time | 17-AUG-18 11:56:00.39421 -04:00
sample_time      | 17-AUG-18 11:56:00.699934 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
-[ RECORD 2 ]----+---------------------------------
query_id         | 4292540138852956818
session_id       | 4340
query_start_time | 17-AUG-18 11:56:00.39421 -04:00
sample_time      | 17-AUG-18 11:56:01.699003 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
-[ RECORD 3 ]----+---------------------------------
query_id         | 4292540138852956818
session_id       | 4340
query_start_time | 17-AUG-18 11:56:00.39421 -04:00
sample_time      | 17-AUG-18 11:56:02.70001 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
-[ RECORD 4 ]----+---------------------------------
query_id         | 4292540138852956818
session_id       | 4340
query_start_time | 17-AUG-18 11:56:00.39421 -04:00
sample_time      | 17-AUG-18 11:56:03.700081 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
                   .
                   .
                   .
```

edb_wait_states_purge

The function deletes all the sampled data files (queries, sessions and wait event samples) that were created after `start_ts` and aged (rotated) before `end_ts`.

```
edb_wait_states_purge(
  IN start_ts timestamptz default '-infinity'::timestamptz,
  IN end_ts timestamptz default 'infinity'::timestamptz
)
```

Usually a user does not need to run this function. The backend should purge those according to the retention age, but in case, that doesn't happen for some reason, this function may be used.

In order to know the duration for which the samples have been retained, use `edb_wait_states_data()` as explained in the previous examples of that function.

Example

The `$PGDATA/edb_wait_states` directory before running `edb_wait_states_purge()`:

```
[root@localhost data]# pwd
/var/lib/edb/as11/data
[root@localhost data]# ls -l edb_wait_states
total 12
-rw------- 1 enterprisedb ...  253 Aug 17 11:56 edb_ws_queries_587836358698793_587922758698793
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
-rw------- 1 enterprisedb ... 1600 Aug 17 11:56 edb_ws_samples_587836358698793_587839958698793
-rw------- 1 enterprisedb ...   94 Aug 17 11:56 edb_ws_sessions_587836358698793_587922758698793
```

The `$PGDATA/edb_wait_states` directory after running `edb_wait_states_purge()`:

```
edb=# SELECT * FROM edb_wait_states_purge();
 edb_wait_states_purge
-----------------------

(1 row)

[root@localhost data]# pwd
/var/lib/edb/as11/data
[root@localhost data]# ls -l edb_wait_states
total 0
```

20.8 EDB Clone Schema

EDB Clone Schema is an extension module for 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 the same physical database, or different databases within the same database cluster, or separate databases running under different database clusters on separate database server hosts.

Use the following functions with EDB Clone Schema:

- **localcopyschema.** This function makes a copy of a schema and its database objects from a source database back into the same database (the target), but with a different schema name than the original. Use this function when the original source schema and the resulting copy are to reside within the same database. See localcopyschema for information on the `localcopyschema` function.
- **localcopyschema_nb.** This function performs the same purpose as `localcopyschema`, but as a background job, thus freeing up the terminal from which the function was initiated. This is referred to as a *non-blocking* function. See localcopyschema_nb for information on the `localcopyschema_nb` function.
- **remotecopyschema.** This function makes a copy of a schema and its database objects from a source database to a different target database. Use this function when the original source schema and the resulting copy are to reside in two, separate databases. The separate databases can reside in the same, or in different Advanced Server database clusters. See remotecopyschema for information on the `remotecopyschema` function.
- **remotecopyschema_nb.** This function performs the same purpose as `remotecopyschema`, but as a background job, thus freeing up the terminal from which the function was initiated. This is referred to as a *non-blocking* function. See remotecopyschema_nb for information on the `remotecopyschema_nb` function.
- **process_status_from_log.** This function displays the status of the cloning functions. The information is obtained from a log file that must be specified when a cloning function is invoked. See process_status_from_log for information on the `process_status_from_log` function.
- **remove_log_file_and_job.** This function deletes the log file created by a cloning function. This function can also be used to delete a job created by the non-blocking form of the function. See remove_log_file_and_job for information on the `remove_log_file_and_job` function.

The database objects that can be cloned from one schema to another are the following:

- 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

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- Comments for all supported object types
- Access control lists (ACLs) for all supported object types

The following database objects cannot be cloned:

- 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 third bullet point in the following limitations list.)
- Row level security
- Policies
- Operator class

In addition, the following limitations apply:

- EDB Clone Schema is supported on Advanced Server only when a dialect of `Compatible with Oracle` is specified on the Advanced Server `Dialect` dialog during installation, or when the `--redwood-like` keywords are included during a text mode installation or cluster initialization.
- The source code within functions, procedures, triggers, packages, etc., are not modified after being copied to the target schema. If such programs contain coded references to objects with schema names, the programs may fail upon invocation in the target schema if such schema names are no longer consistent within the target schema.
- Cross schema object dependencies are not resolved. If an object in the target schema depends upon an object in another schema, this dependency is not resolved by the cloning functions.
- For remote cloning, if an object in the source schema is dependent upon an extension, then this extension must be created 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, whereas each job can have at most 16 worker processes to copy table data in parallel.
- Queries being run by background workers cannot be cancelled.

The following section describes how to set up EDB Clone Schema on the databases.

Setup Process

Several extensions along with the PL/Perl language must be installed on any database to be used as the source or target database by an EDB Clone Schema function.

In addition, some configuration parameters in the `postgresql.conf` file of the database servers may benefit from some modification.

The following is the setup instructions for these requirements.

Installing Extensions and PL/Perl

The following describes the steps to install the required extensions and the PL/Perl language.

These steps must be performed on any database to be used as the source or target database by an EDB Clone Schema function.

Step 1: The following extensions must be installed on the database:

- `postgres_fdw`
- `dblink`
- `adminpack`
- `pgagent`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Ensure that pgAgent is installed before creating the `pgagent` extension. On Linux, you can use the `edb-asxx-pgagent` RPM package where `xx` is the Advanced Server version number to install pgAgent. On Windows, use StackBuilder Plus to download and install pgAgent.

The previously listed extensions can be installed by the following commands if they do not already exist:

```
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 at:

https://www.postgresql.org/docs/11/static/sql-createextension.html

Step 2: Modify the `postgresql.conf` file.

Modify the `postgresql.conf` file by adding `$libdir/parallel_clone` to the `shared_preload_libraries` configuration parameter as shown by the following example:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/dbms_aq,$libdir/parallel_clone'
```

Step 3: The Perl Procedural Language (PL/Perl) must be installed on the database and the `CREATE TRUSTED LANGUAGE plperl` command must be run. For Linux, install PL/Perl using the `edb-asxx-server-plperl` RPM package where `xx` is the Advanced Server version number. For Windows, use the EDB Postgres Language Pack. For information on EDB Language Pack, see the *EDB Postgres Language Pack Guide* available at:

https://www.enterprisedb.com/docs

Step 4: 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 at:

https://www.postgresql.org/docs/11/static/sql-createlanguage.html

Setting Configuration Parameters

The following sections describe configuration parameters that may need to be altered in the `postgresql.conf` file.

Performance Configuration Parameters

You may 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.

The configuration parameters in the `postgresql.conf` file that may need to be tuned include the following:

- **work_mem.** Specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files.
- **maintenance_work_mem.** Specifies the maximum amount of memory to be used by maintenance operations, such as `VACUUM`, `CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY`.
- **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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

- **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* have been 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.** This parameter 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 at:

https://www.postgresql.org/docs/11/static/runtime-config.html

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 are connected when invoking the cloning function.

The default location is `PGDATA/log` as shown by the following:

```
#log_directory = 'log'         # directory where log files are written,
                               # can be absolute or relative to PGDATA
```

This directory must exist prior to 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 simply navigate to the log directory and delete it manually.

Installing EDB Clone Schema

The following are the directions for installing EDB Clone Schema.

These steps must be performed on any database to be used as the source or target database by an EDB Clone Schema function.

Step 1: If you had previously installed an older version of the `edb_cloneschema` extension, then you must run the following command:

```
DROP EXTENSION parallel_clone CASCADE;
```

This command also drops the `edb_cloneschema` extension.

Step 2: Install the extensions using the following commands:

```
CREATE EXTENSION parallel_clone SCHEMA public;

CREATE EXTENSION edb_cloneschema;
```

Make sure you create the `parallel_clone` extension before creating the `edb_cloneschema` extension.

Creating the Foreign Servers and User Mappings

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

When using one of the local cloning functions, `localcopyschema` or `localcopyschema_nb`, one of the required parameters includes a single, foreign server for identifying the database server along with its database that is the source and the 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 along with its database that is the provider of the cloned schema. The foreign server specified as the second parameter identifies the target database server along with its database that is the receiver of the cloned schema.

For each foreign server, a user mapping must be created. When a selected database superuser invokes a cloning function, that database superuser who invokes the function must have been mapped to a database user name and password that has access to the foreign server that is specified as a parameter in the cloning function.

For general information about foreign data, foreign servers, and user mappings, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/11/static/ddl-foreign-data.html

The following two sections describe how these foreign servers and user mappings are defined.

Foreign Server and User Mapping for Local Cloning Functions

For the `localcopyschema` and `localcopyschema_nb` functions, the source and target schemas are both within the same database of the same database server. Thus, only one foreign server must be defined and specified for these functions. This foreign server is also referred to as the *local server*.

This server is 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.

This foreign server and user mapping must be created within the database of the local server in which the cloning is to occur.

The database user for whom the user mapping is defined must be a superuser and the user connected to the local server when invoking an EDB Clone Schema function.

The following example creates the foreign server for the database containing the schema to be cloned, and to receive the cloned schema as well.

```
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 at:

https://www.postgresql.org/docs/11/static/sql-createserver.html

The user mapping for this server is the following:

```
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 at:

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

https://www.postgresql.org/docs/11/sql-createusermapping.html

The following `psql` commands show the foreign server and user mapping:

```
edb=# \des+
List of foreign servers
-[ RECORD 1 ]--------+---------------------------------------------
Name                 | local_server
Owner                | enterprisedb
Foreign-data wrapper | postgres_fdw
Access privileges    |
Type                 |
Version              |
FDW options          | (host 'localhost', port '5444', dbname 'edb')
Description          |

edb=# \deu+
                        List of user mappings
    Server    |  User name   |                 FDW options
--------------+--------------+---------------------------------------------
 local_server | enterprisedb | ("user" 'enterprisedb', password 'password')
(1 row)
```

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 happen to be the same, identical database user, but that is not an absolute requirement.

For specific usage of these foreign server and user mapping examples, see the example given in `localcopyschema`.

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. Thus, two foreign servers must be defined and specified for these functions.

The foreign server defining the originating database server and its database containing the source schema to be cloned 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 be cloned 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.

All of these foreign servers and user mappings must be created within 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.

The following example creates the foreign server for the local, target database that is to receive the cloned schema.

```
CREATE SERVER tgt_server FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS(
    host 'localhost',
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
    port '5444',
    dbname 'tgtdb'
);
```

The user mapping for this server is the following:

```
CREATE USER MAPPING FOR enterprisedb SERVER tgt_server
  OPTIONS (
    user 'tgtuser',
    password 'tgtpassword'
);
```

The following example creates the foreign server for the remote, source database that is to be the source for the cloned schema.

```
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 the following:

```
CREATE USER MAPPING FOR enterprisedb SERVER src_server
  OPTIONS (
    user 'srcuser',
    password 'srcpassword'
);
```

The following `psql` commands show the foreign servers and user mappings:

```
 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          |

tgtdb=# \deu+
                      List of user mappings
   Server   |  User name   |               FDW options
------------+--------------+-------------------------------------------
 src_server | enterprisedb | ("user" 'srcuser', password 'srcpassword')
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 tgt_server | enterprisedb | ("user" 'tgtuser', password 'tgtpassword')
(2 rows)
```

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`.

> **Note**
>
> Be sure the `pg_hba.conf` file of the database server running the source database `srcdb` has an appropriate entry permitting connection from the target server location (address `192.168.2.27` in the following example) connecting 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 usage of these foreign server and user mapping examples, see the example given `remotecopyschema`.

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.

Verify the following conditions before using an EDB Clone Schema function:

- You are 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 the cloning function is to be invoked with the `edb_util` prefix.
- The target schema does not exist in the target database.
- When using the remote copy functions, if the `on_tblspace` parameter is to be set to `true`, then the target database cluster contains all tablespaces that are referenced by objects in the source schema, otherwise creation of the DDL statements for those database objects will fail in the target schema. This causes a failure of the cloning process.
- When using the remote copy functions, if the `copy_acls` parameter is to be set to `true`, then all roles that have `GRANT` privileges on objects in the source schema exist in the target database cluster, otherwise granting of privileges to those roles will fail in the target schema. This causes a failure of the cloning process.
- pgAgent is running against the target database if the non-blocking form of the function is to be used.

For information about pgAgent, see the pgAdmin documentation available at:

https://www.pgadmin.org/docs/pgadmin4/dev/pgagent.html

Note that pgAgent is provided as a component with Advanced Server.

localcopyschema

The `localcopyschema` function copies a schema and its database objects within a local database specified within the `source_fdw` foreign server from the source schema to the specified target schema within the same database.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
localcopyschema(
  <source_fdw> TEXT,
  <source_schema> TEXT,
  <target_schema> TEXT,
  <log_filename> TEXT
  [, <on_tblspace> BOOLEAN
  [, <verbose_on> BOOLEAN
  [, <copy_acls> BOOLEAN
  [, <worker_count> INTEGER ]]]]
)
```

A `BOOLEAN` value is returned by the function. 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 database objects are to be cloned.

`source_schema`

Name of the schema from which database objects are to be cloned.

`target_schema`

Name of the schema into which database objects are to be cloned 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 or not database objects are to be created within their tablespaces. If `false` is specified, then the `TABLESPACE` clause is not included in the applicable `CREATE` DDL statement when added to the target schema. If true is specified, then the `TABLESPACE` clause is included in the `CREATE` DDL statement when added to the target schema. If the `on_tblspace` parameter is omitted, the default value is `false` .

`verbose_on`

`BOOLEAN` value to specify whether or not the DDLs are to be printed in `log_filename` when creating objects in the target schema. If `false` is specified, then DDLs are not printed. If `true` is specified, then DDLs are printed. If omitted, the default value is `false` .

`copy_acls`

`BOOLEAN` value to specify whether or not the access control list (ACL) is to be included while creating objects in the target schema. The access control list is the set of `GRANT` privilege statements. If `false` is specified, then the access control list is not included for the target schema. If `true` is specified, then the access control list is included for the target schema. If the `copy_acls` parameter is omitted, the default value is `false` .

`worker_count`

Number of background workers to perform the clone in parallel. If omitted, the default value is `1` .

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Example

The following example shows the cloning of schema `edb` containing a set of database objects to target schema `edbcopy`, both within 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, the connection is made by database user `enterprisedb` to database `edb`.

```
 edb=# SET search_path TO "$user",public,edb_util;
SET
edb=# SHOW search_path;
        search_path
---------------------------
 "$user", public, edb_util
(1 row)

edb=# SELECT localcopyschema ('local_server','edb','edbcopy','clone_edb_edbcopy');
 localcopyschema
-----------------
 t
(1 row)
```

The following displays the logging status using the `process_status_from_log` function:

```
 edb=# SELECT process_status_from_log('clone_edb_edbcopy');
                                  process_status_from_log
-----------------------------------------------------------------------------------------------------
 (FINISH,"2017-06-29 11:07:36.830783-04",3855,INFO,"STAGE: FINAL","successfully cloned schema")
(1 row)
```

After the clone has completed, the following shows some of the database objects copied to the `edbcopy` schema:

```
 edb=# SET search_path TO edbcopy;
SET
edb=# \dt+
                      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)

edb=# \dv
         List of relations
 Schema  |   Name   | Type |    Owner
---------+----------+------+--------------
 edbcopy | salesemp | view | enterprisedb
(1 row)
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
edb=# \di
                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)

edb=# \ds
            List of relations
 Schema  |    Name    |   Type   |    Owner
---------+------------+----------+--------------
 edbcopy | next_empno | sequence | enterprisedb
(1 row)

edb=# SELECT DISTINCT schema_name, name, type FROM user_source WHERE
schema_name = 'EDBCOPY' ORDER BY type, name;
 schema_name |              name               |     type
-------------+---------------------------------+--------------
 EDBCOPY     | EMP_COMP                        | FUNCTION
 EDBCOPY     | HIRE_CLERK                      | FUNCTION
 EDBCOPY     | HIRE_SALESMAN                   | FUNCTION
 EDBCOPY     | NEW_EMPNO                       | FUNCTION
 EDBCOPY     | EMP_ADMIN                       | PACKAGE
 EDBCOPY     | EMP_ADMIN                       | PACKAGE BODY
 EDBCOPY     | EMP_QUERY                       | PROCEDURE
 EDBCOPY     | EMP_QUERY_CALLER                | PROCEDURE
 EDBCOPY     | LIST_EMP                        | PROCEDURE
 EDBCOPY     | SELECT_EMP                      | PROCEDURE
 EDBCOPY     | EMP_SAL_TRIG                    | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_19991" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_19992" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_19999" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_20000" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_20004" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_20005" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_19993" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_19994" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_20001" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_20002" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_20006" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_20007" | TRIGGER
 EDBCOPY     | USER_AUDIT_TRIG                 | TRIGGER
(24 rows)
```

localcopyschema_nb

The `localcopyschema_nb` function copies a schema and its database objects within a local database specified within the `source_fdw` foreign server from the source schema to the specified target schema within the same database, but in a non-blocking manner as a job submitted to pgAgent.

```
localcopyschema_nb(
  <source_fdw> TEXT,
  <source> TEXT,
  <target> TEXT,
  <log_filename> TEXT
  [, <on_tblspace> BOOLEAN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  [, <verbose_on> BOOLEAN
  [, <copy_acls> BOOLEAN
  [, <worker_count> INTEGER ]]]]
)
```

An `INTEGER` value job ID is returned by the function for the job submitted to pgAgent. If the function fails, then null is returned.

The `source_fdw, source, target`, and `log_filename` are required parameters while all other parameters are optional.

After completion of the pgAgent job, remove the job 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 database objects are to be cloned.

`source`

Name of the schema from which database objects are to be cloned.

`target`

Name of the schema into which database objects are to be cloned 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 or not database objects are to be created within their tablespaces. If `false` is specified, then the `TABLESPACE` clause is not included in the applicable `CREATE` DDL statement when added to the target schema. If `true` is specified, then the `TABLESPACE` clause is included in the `CREATE` DDL statement when added to the target schema. If the `on_tblspace` parameter is omitted, the default value is `false`.

`verbose_on`

`BOOLEAN` value to specify whether or not the DDLs are to be printed in `log_filename` when creating objects in the target schema. If `false` is specified, then DDLs are not printed. If `true` is specified, then DDLs are printed. If omitted, the default value is `false`.

`copy_acls`

`BOOLEAN` value to specify whether or not the access control list (ACL) is to be included while creating objects in the target schema. The access control list is the set of `GRANT` privilege statements. If `false` is specified, then the access control list is not included for the target schema. If `true` is specified, then the access control list is included for the target schema. If the `copy_acls` parameter is omitted, the default value is `false`.

`worker_count`

Number of background workers to perform the clone in parallel. If omitted, 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`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following command can be used to observe if 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 is not running, it can be started as shown by the following. The `pgagent` program file is located in the `bin` subdirectory of the 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)
```

remotecopyschema

The `remotecopyschema` function copies a schema and its database objects from a source schema in the remote source database specified within the `source_fdw` foreign server to a target schema in the local target database specified within the `target_fdw` foreign server.

```
remotecopyschema(
  <source_fdw> TEXT,
  <target_fdw> TEXT,
  <source_schema> TEXT,
  <target_schema> TEXT,
  <log_filename> TEXT
  [, <on_tblspace> BOOLEAN
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  [, <verbose_on> BOOLEAN
  [, <copy_acls> BOOLEAN
  [, <worker_count> INTEGER ]]]]
)
```

A `BOOLEAN` value is returned by the function. 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 while all other parameters are optional.

Parameters

`source_fdw`

Name of the foreign server managed by the `postgres_fdw` foreign data wrapper from which database objects are to be cloned.

`target_fdw`

Name of the foreign server managed by the `postgres_fdw` foreign data wrapper to which database objects are to be cloned.

`source_schema`

Name of the schema from which database objects are to be cloned.

`target_schema`

Name of the schema into which database objects are to be cloned 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 or not database objects are to be created within their tablespaces. If `false` is specified, then the `TABLESPACE` clause is not included in the applicable `CREATE` DDL statement when added to the target schema. If `true` is specified, then the `TABLESPACE` clause is included in the `CREATE` DDL statement when added to the target schema. If the `on_tblspace` parameter is omitted, the default value is `false`.

> **Note**
>
> If `true` is specified and a database object has a `TABLESPACE` clause, but that tablespace does not exist in the target database cluster, then the cloning function fails.

`verbose_on`

`BOOLEAN` value to specify whether or not the DDLs are to be printed in `log_filename` when creating objects in the target schema. If `false` is specified, then DDLs are not printed. If `true` is specified, then DDLs are printed. If omitted, the default value is `false`.

`copy_acls`

`BOOLEAN` value to specify whether or not the access control list (ACL) is to be included while creating objects in the target schema. The access control list is the set of `GRANT` privilege statements. If `false` is specified, then the access control list is not included for the target schema. If `true` is specified, then the access control list is included for the target schema. If the `copy_acls` parameter is omitted, the default value is `false`.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

> **Note**
>
> If `true` is specified and a role with `GRANT` privilege does not exist in the target database cluster, then the cloning function fails.

`worker_count`

Number of background workers to perform the clone in parallel. If omitted, the default value is `1`.

Example

The following example shows the cloning of schema `srcschema` within database `srcdb` as defined by `src_server` to target schema `tgtschema` within database `tgtdb` as defined by `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: `srcdb`
- Foreign server (`src_server`) and user mapping with the information of the preceding bullet points
- Source schema: `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: `tgtdb`
- Foreign server (`tgt_server`) and user mapping with the information of the preceding bullet points
- Target schema: `tgtschema`
- Database superuser to invoke `remotecopyschema: enterprisedb`

Before invoking the function, the connection is made by database user `enterprisedb` to database `tgtdb`. A `worker_count` of `4` is specified for this function.

```
tgtdb=# SELECT edb_util.remotecopyschema
('src_server','tgt_server','srcschema','tgtschema','clone_rmt_src_tgt',worker_count => 4);
 remotecopyschema
------------------
 t
(1 row)
```

The following displays the status from the log file during various points in the cloning process:

```
tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                                process_status_from_log

------------------------------------------------------------------------------------------------------------------------
-------------------------------
 (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)

tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                                process_status_from_log

------------------------------------------------------------------------------------------------------------------------
-----------------------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 (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)

tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                                    process_status_from_log

------------------------------------------------------------------------------------------------------------------
----------------------------
 (RUNNING,"28-JUN-17 13:18:10.550393 -04:00",4039,INFO,"STAGE: POST-DATA","CREATE PRIMARY KEY
 CONSTRAINT pgbench_tellers_pkey successful"
)
 (1 row)

tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                                    process_status_from_log
------------------------------------------------------------------------------------------------------------------
-
 (FINISH,"28-JUN-17 13:18:12.019627 -04:00",4039,INFO,"STAGE:FINAL","successfully clone
 schema into tgtschema")
 (1 row)
```

The following shows the cloned tables:

```
tgtdb=# \dt+
                            List of relations
  Schema   |       Name       | Type  |    Owner     |    Size    | Description
-----------+------------------+-------+--------------+------------+-------------
 tgtschema | pgbench_accounts | table | enterprisedb | 256 MB     |
 tgtschema | pgbench_branches | table | enterprisedb | 8192 bytes |
 tgtschema | pgbench_history  | table | enterprisedb | 25 MB      |
 tgtschema | pgbench_tellers  | table | enterprisedb | 16 kB      |
(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:

```
 Wed Jun 28 13:18:05.232949 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0] table count [4]
Wed Jun 28 13:18:05.233321 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0][0] worker started to
copy data
Wed Jun 28 13:18:05.233640 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0][1] worker started to
copy data
Wed Jun 28 13:18:05.233919 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0][2] worker started to
copy data
Wed Jun 28 13:18:05.234231 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0][3] worker started to
copy data
Wed Jun 28 13:18:05.298174 2017 EDT: [4024] INFO: [STAGE: DATA-COPY] [0][3] successfully
copied data in [tgtschema.pgbench_branches]
Wed Jun 28 13:18:05.299913 2017 EDT: [4021] INFO: [STAGE: DATA-COPY] [0][0] successfully
copied data in [tgtschema.pgbench_tellers]
Wed Jun 28 13:18:06.634310 2017 EDT: [4022] INFO: [STAGE: DATA-COPY] [0][1] successfully
copied data in [tgtschema.pgbench_history]
Wed Jun 28 13:18:10.477333 2017 EDT: [4023] INFO: [STAGE: DATA-COPY] [0][2] successfully
copied data in [tgtschema.pgbench_accounts]
Wed Jun 28 13:18:10.477609 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0] all workers finished
[4]
Wed Jun 28 13:18:10.477654 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0] copy done [4] tables
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
Wed Jun 28 13:18:10.493938 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] successfully copied data
into tgtschema
```

Note that the `DATA-COPY` log message includes two, square bracket numbers (for example, `[0][3]`).

The first number is the job index whereas 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 will have `0` as the job index whereas the second will have `1` as the job index.

remotecopyschema_nb

The `remotecopyschema_nb` function copies a schema and its database objects from a source schema in the remote source database specified within the `source_fdw` foreign server to a target schema in the local target database specified within the `target_fdw` foreign server, but in a non-blocking manner as a job submitted to pgAgent.

```
remotecopyschema_nb(
  <source_fdw> TEXT,
  <target_fdw> TEXT,
  <source> TEXT,
  <target> TEXT,
  <log_filename> TEXT
  [, <on_tblspace> BOOLEAN
  [, <verbose_on> BOOLEAN
  [, <copy_acls> BOOLEAN
  [, <worker_count> INTEGER ]]]]
)
```

An `INTEGER` value job ID is returned by the function for the job submitted to pgAgent. If the function fails, then null is returned.

The `source_fdw, target_fdw, source, target`, and `log_filename` are required parameters while all other parameters are optional.

After completion of the pgAgent job, remove the job 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 database objects are to be cloned.

`target_fdw`

Name of the foreign server managed by the `postgres_fdw` foreign data wrapper to which database objects are to be cloned.

`source`

Name of the schema from which database objects are to be cloned.

`target`

Name of the schema into which database objects are to be cloned from the source schema.

`log_filename`

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 or not database objects are to be created within their tablespaces. If `false` is specified, then the `TABLESPACE` clause is not included in the applicable `CREATE` DDL statement when added to the target schema. If `true` is specified, then the `TABLESPACE` clause is included in the `CREATE` DDL statement when added to the target schema. If the `on_tblspace` parameter is omitted, the default value is `false`.

> **Note**
>
> If `true` is specified and a database object has a `TABLESPACE` clause, but that tablespace does not exist in the target database cluster, then the cloning function fails.

`verbose_on`

`BOOLEAN` value to specify whether or not the DDLs are to be printed in `log_filename` when creating objects in the target schema. If `false` is specified, then DDLs are not printed. If `true` is specified, then DDLs are printed. If omitted, the default value is `false`.

`copy_acls`

`BOOLEAN` value to specify whether or not the access control list (ACL) is to be included while creating objects in the target schema. The access control list is the set of `GRANT` privilege statements. If `false` is specified, then the access control list is not included for the target schema. If `true` is specified, then the access control list is included for the target schema. If the `copy_acls` parameter is omitted, the default value is `false`. **Note:** If `true` is specified and a role with `GRANT` privilege does not exist in the target database cluster, then the cloning function fails.

`worker_count`

Number of background workers to perform the clone in parallel. If omitted, 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`.

The following command starts pgAgent on the target database `tgtdb`. The `pgagent` program file is located in the `bin` subdirectory of the 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
```

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 completed status of the job is shown by the following:

```
tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                                process_status_from_log
-------------------------------------------------------------------------------------------------------------------
-
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 (FINISH,"29-JUN-17 13:16:00.100284 -04:00",3849,INFO,"STAGE: FINAL","successfully clone schema into
tgtschema")
(1 row)
```

The following 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)
```

process_status_from_log

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:

Field Name	Description
`status`	Displays either `STARTING`, `RUNNING`, `FINISH`, or `FAILED`.
`execution_time`	When the command was executed. Displayed in timestamp format.
`pid`	Session process ID in which clone schema is getting called.
`level`	Displays either `INFO`, `ERROR`, or `SUCCESSFUL`.
`stage`	Displays either `STARTUP`, `INITIAL`, `DDL-COLLECTION`, `PRE-DATA`, `DATA-COPY`, `POST-DATA`, or `FINAL`.
`message`	Information respective to each command or failure.

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 usage 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)
```

remove_log_file_and_job

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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 |
    <log_file> TEXT, <job_id> INTEGER
  }
)
```

Values for any or both of the two parameters may be specified when invoking the `remove_log_file_and_job` function:

- If only `log_file` is specified, then the function will only remove the log file.
- If only `job_id` is specified, then the function will only remove the job.
- If both are specified, then the function will remove the log file and the job.

Parameters

`log_file`

Name of the log file to be removed.

`job_id`

Job ID of the job to be removed.

Example

The following examples removes only the log file, given the log filename.

```
edb=# SELECT edb_util.remove_log_file_and_job ('clone_edb_edbcopy');
 remove_log_file_and_job
-------------------------
 t
(1 row)
```

The following 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)
```

The following example removes the log file and the job, given both values:

```
tgtdb=# SELECT edb_util.remove_log_file_and_job ('clone_rmt_src_tgt',2);
 remove_log_file_and_job
-------------------------
 t
(1 row)
```

20.9 PL/Java

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The PL/Java package provides access to Java stored procedures, triggers, and functions via the JDBC interface. Unless otherwise noted, the commands and paths noted in the following section assume that you have performed an installation with the `edb-asxx-pljava` RPM package where `xx` is the Advanced Server version number.

Before installing PL/Java for use with a standard Java virtual machine (JVM) on a Linux system, you must first confirm that a Java runtime environment (version 1.8) is installed on your system. Installation of a Java development kit also provides a Java runtime environment.

Installing PL/Java on Linux

The following steps outline the process of installing PL/Java on a Linux system:

Step 1: Edit the `postgresql.conf` file located under the `data` directory of your Advanced Server installation and add (or modify) the following settings:

```
pljava.classpath = 'path_to_pljava.jar'
pljava.libjvm_location = 'path_to_libjvm.so'
```

Where `path_to_pljava.jar` specifies the location of the `pljava.jar` file and `path_to_libjvm.so` specifies the location of the `libjvm.so` file.

For example, the following lists the paths for a default installation with Java version 1.8:

```
pljava.classpath = '/usr/edb/as11/share/pljava/pljava-1.5.0.jar'
pljava.libjvm_location = '/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.91-
1.b14.el6.x86_64/jre/lib/amd64/server/libjvm.so'
```

Step 2: Restart the database server.

Step 3: You can use the `CREATE EXTENSION` command to install PL/Java. To install the PL/Java extension, login to the database in which you want to install PL/Java with the psql or pgAdmin client, and invoke the following command:

```
CREATE EXTENSION pljava;
```

Step 4: To confirm that PL/Java is installed, invoke the following command:

```
SELECT * FROM pg_language WHERE lanname LIKE 'java%';
```

The `edb-psql` client displays two rows indicating that `java` and `javau` (Java Untrusted) have been installed in the database.

```
edb=# SELECT * FROM pg_language WHERE lanname LIKE 'java%';
lanname  | lanowner | lanispl | lanpltrusted | lanplcallfoid | laninline | lanvalidator | lanacl
---------+----------+---------+--------------+---------------+-----------+--------------+---------------------
--------
java     | 10       | t       | t            | 16462         | 0         | 0            |
{enterprisedb=U/enterprisedb}
javau    | 10       | t       | f            | 16463         | 0         | 0            |
(2 rows)
```

Installing PL/Java on Windows

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The following steps outline the process of installing PL/Java on a Windows system.

Step 1: Edit the `postgresql.conf` file and add (or modify) the following settings:

```
pljava.classpath = 'POSTGRES_INSTALL_HOME\lib\pljava.jar'
pljava.libjvm_location = 'path_to_libjvm.so'
```

Where `POSTGRES_INSTALL_HOME` specifies the location of the Advanced Server installation. For example, the following is the configuration setting for a default installation:

```
pljava.classpath = 'C:\Program Files\edb\as11\lib\pljava.jar'
```

Step 2: Restart the database server.

Step 3: Modify the PATH setting used by the server, adding the following two entries:

```
%JRE_HOME%\bin;%JRE_HOME%\bin\client
```

Where `JRE_HOME` specifies the installation directory of your Java runtime environment. If you have a Java development kit, substitute the location of `$JDK_HOME/jre` for `JRE_HOME`.

Step 4: Use the Postgres `CREATE EXTENSION` command to install PL/Java. To run the installation script, use the psql or pgAdmin client to connect to the database in which you wish to install PL/Java and invoke the following command:

```
CREATE EXTENSION pljava;
```

Step 5: To confirm that PL/Java is installed, invoke the following command:

```
SELECT * FROM pg_language WHERE lanname LIKE 'java%';
```

The client will return a result set that includes java and javau (Java Untrusted).

Using PL/Java

To create a PL/Java program, you must first create a Java class that contains at least one static method, and then you must compile that class into a `.class` or `.jar` file. Next, you declare the Java function within SQL using the `CREATE FUNCTION` command. The `CREATE FUNCTION` command gives a SQL name to the function and associates the compiled class (and method name) with that function name.

For example, the following `CREATE FUNCTION` statement creates a function named `getsysprop`:

```
CREATE FUNCTION getsysprop(VARCHAR)
RETURNS VARCHAR
AS 'java.lang.System.getProperty'
LANGUAGE java;
```

When invoked, `getsysprop` will execute the `getProperty` (static) method defined within the `java.lang.System` class.

```
SELECT getsysprop('user.home');

 getsysprop
---------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
 /usr/edb/as11
(1 row)
```

The example that follows demonstrates the procedures used to create and install a simple `HelloWorld` program:

Step 1: Save the following code sample to a file named `HelloWorld.java`:

```
package com.mycompany.helloworld;
public class HelloWorld
{
    public static String helloWorld()
    {
      return "Hello World";
    }
}
```

Step 2: Compile the file.

```
$ javac HelloWorld.java
```

And save it in a folder hierarchy as:

```
com/mycompany/helloworld/HelloWorld.class
```

Step 3: Create an archive file (a JAR file) named `helloworld.jar`:

```
jar cf helloworld.jar com/mycompany/helloworld/HelloWorld.class
```

Step 4: Open the `edb-psql` client, and install the jar file with the following command:

```
SELECT sqlj.install_jar('file:///file_path/helloworld.jar', 'helloworld', true);
```

Where `file_path` is the directory containing the `helloworld.jar` file. For example, if the `/tmp` directory is the `file_path`:

```
SELECT sqlj.install_jar('file:///tmp/helloworld.jar', 'helloworld', true);
```

To confirm that the jar file has been loaded correctly, perform a `SELECT` statement on the `sqlj.jar_entry` and `sqlj.jar_repository` tables.

```
edb=# SELECT entryid, entryname FROM sqlj.jar_entry;
 entryid | entryname
---------+-------------------------------------------
     1   | com/mycompany/helloworld/HelloWorld.class
(1 row)

edb=# SELECT jarid, jarname, jarorigin, jarowner FROM sqlj.jar_repository;
 jarid | jarname    | jarorigin                  | jarowner
-------+------------+----------------------------+--------------
    4  | helloworld | file:///tmp/helloworld.jar | enterprisedb
(1 row)
```

Step 5: Set the classpath as:

```
edb=# SELECT sqlj.set_classpath('public', 'helloworld');
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

The `sqlj.classpath_entry` table will now include an entry for the `helloworld` class file.

```
edb=# SELECT * FROM sqlj.classpath_entry;
 schemaname | ordinal | jarid
------------+---------+-------
 public     | 1       | 4
(1 row)
```

Step 6: Create a function that uses Java to call the static function declared in the jar file:

```
CREATE OR REPLACE FUNCTION helloworld()
RETURNS "varchar"
AS
     'com.mycompany.helloworld.HelloWorld.helloWorld'
LANGUAGE 'java' VOLATILE;
```

Step 7: Execute the function:

```
edb=# SELECT * FROM helloworld();
```

You should see the output:

```
helloworld
-------------
 Hello World
 (1 row)
```

The official PL/Java distribution is distributed with examples and documentation. For more information about using PL/Java, see the project page at:

https://github.com/tada/pljava/wiki

> **Note**
>
> The PL/Java package is deprecated in Advanced Server 11 and will be unavailable in server versions 12 or later.

20.10 Enhanced SQL and Other Miscellaneous Features

Advanced Server includes enhanced SQL functionality and various other features that provide additional flexibility and convenience. This chapter discusses some of these additions.

COMMENT

In addition to commenting on objects supported by the PostgreSQL `COMMENT` command, Advanced Server supports comments on additional object types. The complete supported syntax is:

```
COMMENT ON
{
  AGGREGATE <aggregate_name> ( <aggregate_signature> ) |
  CAST (<source_type> AS <target_type>) |
  COLLATION <object_name> |
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  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 are 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, and `aggregate_signature` specifies the associated signature in one of the following forms:

```
 * |
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
[ <argmode> ] [ <argname> ] <argtype> [ , ... ] |
[ [ <argmode> ] [ <argname> ] <argtype> [ , ... ] ]
ORDER BY [ <argmode> ] [ <argname> ] <argtype> [ , ... ]
```

Where `argmode` is the mode of a function, procedure, or aggregate argument, argmode may be `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, 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 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 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` may be `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, 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 are 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 are commenting. `left_type` and `right_type` are the (optionally schema-qualified) data type(s) 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 are 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 are commenting. `index_method` specifies the associated index method of the operator family.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

`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, and `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` may be `IN` , `OUT` , `INOUT` , or `VARIADIC` . If omitted, 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, and `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 and `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, and `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.

Notes:

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:

The following 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 at:

https://www.postgresql.org/docs/11/static/sql-comment.html

Output of Function version()

The text string output of the `version()` function displays the name of the product, its version, and the host system on which it has been installed.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

For Advanced Server, the `version()` output is in a format similar to the PostgreSQL community version in that the first text word is *PostgreSQL* instead of *EnterpriseDB* as in Advanced Server version 10 and earlier.

The general format of the version() output is the following:

```
PostgreSQL $PG_VERSION_EXT (EnterpriseDB Advanced Server $PG_VERSION) on $host
```

So for the current Advanced Server the version string appears as follows:

```
edb@45032=#select version();
version
-----------------------------------------------------------------------------------------------------
-----------------------------------------------
PostgreSQL 11.0 (EnterpriseDB Advanced Server 11.0.0) on x86_64-pc-linux-gnu, compiled by gcc
(GCC) 4.8.5 20150623 (Red Hat 4.8.5-11), 64-bit
(1 row)
```

In contrast, for Advanced Server 10, the version string was the following:

```
edb=# select version();
                                                  version
-----------------------------------------------------------------------------------------------
-----------------
EnterpriseDB 10.4.9 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.4.7 20120313 (Red Hat
4.4.7-18), 64-bit
(1 row)
```

SQL Server dbo schema

Prior to Advanced Server 11, a system catalog named `dbo` was available. The `dbo` system catalog contained views of database objects for similarity with Microsoft SQL Server.

Now, for Advanced Server neither the `dbo` system catalog nor a schema named `dbo` exist in any database.

If it is desired to have such a schema with its SQL Server compatible views, use the `CREATE EXTENSION edb_dbo` command to create the `dbo` schema and its content.

The following example shows the creation of the `dbo` schema and its views:

```
edb=# CREATE EXTENSION edb_dbo;
CREATE EXTENSION
edb=# \dn
    List of Schemas
  Name   |    Owner
  -------+--------------
  dbo    | enterprisedb
  public | enterprisedb

edb=# SET search_path TO dbo;
SET
edb=# \dv
                List of relations
  Schema |   Name      | Type |   Owner
---------+-------------+------+-------------
```

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

```
  dbo    | sysinedexes | view | enterprisedb
  dbo    | sysobjects  | view | enterprisedb
  dbo    | systables   | view | enterprisedb
  dbo    | systypes    | view | enterprisedb
  dbo    | sysusers    | view | enterprisedb
(5 rows)
```

20.11 Advanced Server Keywords

A keyword is a word that is recognized by the 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 Advanced Server keywords:

```
 acctg=#
acctg=# SELECT * FROM pg_get_keywords();
        word         | catcode |          catdesc
---------------------+---------+-----------------------------------
 abort               | U       | unreserved
 absolute            | U       | unreserved
 access              | U       | unreserved
...
```

`pg_get_keywords` returns a table containing the keywords recognized by 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.

Note that any character can be used 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 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. Reserved keywords may never be used 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 may be used as a name for a database object.

`T` - The word is used internally, but may be used as a name for a function or type.

`C` - The word is used internally, and may not be used as a name for a function or type.

For more information about Advanced Server identifiers and keywords, see to the PostgreSQL core documentation at:

https://www.postgresql.org/docs/11/static/sql-syntax-lexical.html

20.12 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 are writing an application that depends on information stored in the system tables, it would be prudent to use an existing catalog view, or create a catalog view to isolate

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

the application from changes to the system 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 within the file system. For example, the `UTL_FILE` package offers functions that permit a user to read and write files and directories in the host file system, but only allows access to paths that the database administrator has granted access to via a `CREATE DIRECTORY` command.

Column	Type	Modifiers	Description
`dirname`	`"name"`	`not null`	The name of the alias.
`dirowner`	`oid`	`not null`	The OID of the user that owns the alias.
`dirpath`	`text`		The directory name to which access is granted.
`diracl`	`aclitem[]`		The access control list that determines which users may access the alias.

edb_all_resource_groups

The `edb_all_resource_groups` table contains one row for each resource group created with the `CREATE RESOURCE GROUP` command and displays the number of active processes in each resource group.

Column	Type	Modifiers	Description
`group_name`	`"name"`		The name of the resource group.
`active_processes`	`integer`		Number of currently active processes in the resource group.
`cpu_rate_limit`	`float8`		Maximum CPU rate limit for the resource group. `0` means no limit.
`per_process_cpu_rate_limit`	`float8`		Maximum CPU rate limit per currently active process in the resource group.
`dirty_rate_limit`	`float8`		Maximum dirty rate limit for a resource group. `0` means no limit.
`per_process_dirty_rate_limit`	`float8`		Maximum dirty rate limit per currently active process in the resource group.

edb_policy

The `edb_policy` table contains one row for each policy.

Column	Type	Modifiers	Description
`policyname`	`name`	`not null`	The policy name.
`policygroup`	`oid`	`not null`	Currently unused.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	Modifiers	Description
policyobject	oid	not null	The OID of the table secured by this policy (the object_schema plus the object_name).
policykind	char	not null	The kind of object secured by this policy: 'r' for a table 'v' for a view = for a synonym Currently always 'r'.
policyproc	oid	not null	The OID of the policy function (function_schema plus policy_function).
policyinsert	boolean	not null	True if the policy is enforced by INSERT statements.
policyselect	boolean	not null	True if the policy is enforced by SELECT statements.
policydelete	boolean	not null	True if the policy is enforced by DELETE statements.
policyupdate	boolean	not null	True if the policy is enforced by UPDATE statements.
policyindex	boolean	not null	Currently unused.
policyenabled	boolean	not null	True if the policy is enabled.
policyupdatecheck	boolean	not null	True if rows updated by an UPDATE statement must satisfy the policy.
policystatic	boolean	not null	Currently unused.
policytype	integer	not null	Currently unused.
policyopts	integer	not null	Currently unused.
policyseccols	int2vector	not null	The column numbers for columns listed in sec_relevant_cols .

edb_profile

The edb_profile table stores information about the available profiles. edb_profiles is shared across all databases within a cluster.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute, must be explicitly selected).
prfname	name		The name of the profile.
prffailedloginattempts	integer		The number of failed login attempts allowed by the profile. -1 indicates that the value from the default profile should be used. -2 indicates no limit on failed login attempts.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	References	Description
prfpasswordlocktime	integer		The password lock time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the account should be locked permanently.
prfpasswordlifetime	integer		The password life time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
prfpasswordgracetime	integer		The password grace time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
prfpasswordreusetime	integer		The number of seconds that a user must wait before reusing a password. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordreusemax	integer		The number of password changes that have to occur before a password can be reused. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordverifyfuncdb	oid	pg_database.oid	The OID of the database in which the password verify function exists.
prfpasswordverifyfunc	oid	pg_proc.oid	The OID of the password verify function associated with the profile.

edb_redaction_column

The catalog edb_redaction_column stores information of data redaction policy attached to the columns of the table.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute, must be explicitly selected).
rdpolicyid	oid	edb_redaction_policy.oid	The data redaction policy applies to the described column.
rdrelid	oid	pg_class.oid	The table that the described column belongs to.
rdattnum	int2	pg_attribute.attnum	The number of the described column.
rdscope	int2		The redaction scope: 1 = query, 2 = top_tlist, 4 = top_tlist_or_error
rdexception	int2		The redaction exception: 8 = none, 16 = equal, 32 = leakproof
rdfuncexpr	pg_node_tree		Data redaction function expression

> **Note**
>
> The described column will be 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 of the redaction policies for tables.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute, must be explicitly selected).
rdname	name		The name of the data redaction policy
rdrelid	oid	pg_class.oid	The table to which the data redaction policy applies.
rdenable	boolean		Is the data redaction policy enabled?
rdexpr	pg_node_tree		The data redaction policy expression.

> **Note**
>
> The data redaction policy applies for the table if it is enabled and the expression ever evaluated true.

edb_resource_group

The `edb_resource_group` table contains one row for each resource group created with the `CREATE RESOURCE GROUP` command.

Column	Type	Modifiers	Description
rgrpname	"name"	not null	The name of the resource group.
rgrpcpuratelimit	float8	not null	Maximum CPU rate limit for a resource group. `0` means no limit.
rgrpdirtyratelimit	float8	not null	Maximum dirty rate limit for a resource group. `0` means no limit.

edb_variable

The `edb_variable` table contains one row for each package level variable (each variable declared within a package).

Column	Type	Modifiers	Description
varname	"name"	not null	The name of the variable.
varpackage	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 only visible within the package. Note: Public variables are declared within the package header, private variables are declared within the package body.
varsrc	text		Contains the source of the variable declaration, including any default value expressions for the variable.
varseq	smallint	not null	The order in which the variable was declared in the package.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.

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.
`synobjschema`	`"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 at installation or run time 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.

Copyright © 2009 - 2024 EnterpriseDB Corporation. All rights reserved.