Database Compatibility for Oracle® Developers Built-in Package Guide

EDB Postgres™ Advanced Server 10

November 1, 2017
Table of Contents

1  Introduction ................................................................................................................. 8
   1.1  Typographical Conventions Used in this Guide ................................................. 9
2  Packages .................................................................................................................... 10
   2.1  Package Components ......................................................................................... 10
        2.1.1  Package Specification Syntax ............................................................... 11
        2.1.2  Package Body Syntax .......................................................................... 14
   2.2  Creating Packages .............................................................................................. 20
        2.2.1  Creating the Package Specification ...................................................... 20
        2.2.2  Creating the Package Body ................................................................. 20
   2.3  Referencing a Package ....................................................................................... 23
   2.4  Using Packages With User Defined Types ...................................................... 24
   2.5  Dropping a Package ........................................................................................... 28
3  Built-In Packages ...................................................................................................... 29
   3.1  DBMS_ALERT .................................................................................................. 30
        3.1.1  REGISTER ......................................................................................... 31
        3.1.2  REMOVE ......................................................................................... 31
        3.1.3  REMOVEALL .................................................................................... 32
        3.1.4  SIGNAL .............................................................................................. 32
        3.1.5  WAITANY ............................................................................................ 32
        3.1.6  WAITONE ............................................................................................. 34
        3.1.7  Comprehensive Example ...................................................................... 35
   3.2  DBMS_AQ .......................................................................................................... 38
        3.2.1  ENQUEUE ............................................................................................ 39
        3.2.2  DEQUEUE ............................................................................................ 42
        3.2.3  REGISTER ......................................................................................... 46
        3.2.4  UNREGISTER ....................................................................................... 47
   3.3  DBMS_AQADM ................................................................................................. 49
        3.3.1  ALTER_QUEUE ................................................................................. 50
        3.3.2  ALTER_QUEUE_TABLE ................................................................. 51
        3.3.3  CREATE_QUEUE .............................................................................. 52
        3.3.4  CREATE_QUEUE_TABLE ............................................................... 54
        3.3.5  DROP_QUEUE .................................................................................... 56
        3.3.6  DROP_QUEUE_TABLE ................................................................. 57
        3.3.7  PURGE_QUEUE_TABLE ................................................................. 58
        3.3.8  START_QUEUE .................................................................................. 59
        3.3.9  STOP_QUEUE ..................................................................................... 59
   3.4  DBMS_CRYPTO ................................................................................................. 61
        3.4.1  DECrypt .............................................................................................. 62
        3.4.2  ENCRYPT ............................................................................................ 64
        3.4.3  HASH .................................................................................................. 66
        3.4.4  MAC ..................................................................................................... 66
        3.4.5  RANDOMBYTES .............................................................................. 67
        3.4.6  RANDOMINTEGER ......................................................................... 68
        3.4.7  RANDOMNUMBER .......................................................................... 69
   3.5  DBMS_JOB ........................................................................................................ 70
        3.5.1  BROKEN ............................................................................................. 71
        3.5.2  CHANGE ............................................................................................. 72
        3.5.3  INTERVAL ............................................................................................ 73
        3.5.4  NEXT_DATE ........................................................................................ 73
        3.5.5  REMOVE ............................................................................................. 74
        3.5.6  RUN ...................................................................................................... 74
        3.5.7  SUBMIT .............................................................................................. 75
<table>
<thead>
<tr>
<th>Section</th>
<th>Package</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.8</td>
<td>WHAT</td>
<td></td>
<td>76</td>
</tr>
<tr>
<td>3.6</td>
<td>DBMS_LOB</td>
<td></td>
<td>78</td>
</tr>
<tr>
<td>3.6.1</td>
<td>APPEND</td>
<td></td>
<td>79</td>
</tr>
<tr>
<td>3.6.2</td>
<td>COMPARE</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>3.6.3</td>
<td>CONVERTTOBLOB</td>
<td></td>
<td>81</td>
</tr>
<tr>
<td>3.6.4</td>
<td>CONVERTTOCLOB</td>
<td></td>
<td>82</td>
</tr>
<tr>
<td>3.6.5</td>
<td>COPY</td>
<td></td>
<td>84</td>
</tr>
<tr>
<td>3.6.6</td>
<td>ERASE</td>
<td></td>
<td>85</td>
</tr>
<tr>
<td>3.6.7</td>
<td>GET_STORAGE_LIMIT</td>
<td></td>
<td>85</td>
</tr>
<tr>
<td>3.6.8</td>
<td>GETLENGTH</td>
<td></td>
<td>86</td>
</tr>
<tr>
<td>3.6.9</td>
<td>INSTR</td>
<td></td>
<td>86</td>
</tr>
<tr>
<td>3.6.10</td>
<td>READ</td>
<td></td>
<td>87</td>
</tr>
<tr>
<td>3.6.11</td>
<td>SUBSTR</td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>3.6.12</td>
<td>TRIM</td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>3.6.13</td>
<td>WRITE</td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>3.6.14</td>
<td>WRITEAPPEND</td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>3.7</td>
<td>DBMS_LOCK</td>
<td></td>
<td>91</td>
</tr>
<tr>
<td>3.7.1</td>
<td>SLEEP</td>
<td></td>
<td>91</td>
</tr>
<tr>
<td>3.8</td>
<td>DBMS_MVIEW</td>
<td></td>
<td>92</td>
</tr>
<tr>
<td>3.8.1</td>
<td>GET_MV_DEPENDENCIES</td>
<td></td>
<td>93</td>
</tr>
<tr>
<td>3.8.2</td>
<td>REFRESH</td>
<td></td>
<td>93</td>
</tr>
<tr>
<td>3.8.3</td>
<td>REFRESH_ALL_MVIEWS</td>
<td></td>
<td>95</td>
</tr>
<tr>
<td>3.8.4</td>
<td>REFRESH_DEPENDENT</td>
<td></td>
<td>97</td>
</tr>
<tr>
<td>3.9</td>
<td>DBMS_OUTPUT</td>
<td></td>
<td>99</td>
</tr>
<tr>
<td>3.9.1</td>
<td>CHARARR</td>
<td></td>
<td>99</td>
</tr>
<tr>
<td>3.9.2</td>
<td>DISABLE</td>
<td></td>
<td>99</td>
</tr>
<tr>
<td>3.9.3</td>
<td>ENABLE</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>3.9.4</td>
<td>GET_LINE</td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>3.9.5</td>
<td>GET_LINES</td>
<td></td>
<td>103</td>
</tr>
<tr>
<td>3.9.6</td>
<td>NEW_LINE</td>
<td></td>
<td>104</td>
</tr>
<tr>
<td>3.9.7</td>
<td>PUT</td>
<td></td>
<td>104</td>
</tr>
<tr>
<td>3.9.8</td>
<td>PUT_LINE</td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>3.9.9</td>
<td>SERVEROUTPUT</td>
<td></td>
<td>106</td>
</tr>
<tr>
<td>3.10</td>
<td>DBMS_PIPE</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>3.10.1</td>
<td>CREATE_PIPE</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>3.10.2</td>
<td>NEXT_ITEM_TYPE</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>3.10.3</td>
<td>PACK_MESSAGE</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>3.10.4</td>
<td>PURGE</td>
<td></td>
<td>112</td>
</tr>
<tr>
<td>3.10.5</td>
<td>RECEIVE_MESSAGE</td>
<td></td>
<td>114</td>
</tr>
<tr>
<td>3.10.6</td>
<td>REMOVE_PIPE</td>
<td></td>
<td>114</td>
</tr>
<tr>
<td>3.10.7</td>
<td>RESET_BUFFER</td>
<td></td>
<td>116</td>
</tr>
<tr>
<td>3.10.8</td>
<td>SEND_MESSAGE</td>
<td></td>
<td>117</td>
</tr>
<tr>
<td>3.10.9</td>
<td>UNIQUE_SESSION_NAME</td>
<td></td>
<td>118</td>
</tr>
<tr>
<td>3.10.10</td>
<td>UNPACK_MESSAGE</td>
<td></td>
<td>118</td>
</tr>
<tr>
<td>3.10.11</td>
<td>Comprehensive Example</td>
<td></td>
<td>119</td>
</tr>
<tr>
<td>3.11</td>
<td>DBMS_PROFILER</td>
<td></td>
<td>122</td>
</tr>
<tr>
<td>3.11.1</td>
<td>FLUSH_DATA</td>
<td></td>
<td>122</td>
</tr>
<tr>
<td>3.11.2</td>
<td>GET_VERSION</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>3.11.3</td>
<td>INTERNAL_VERSION_CHECK</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>3.11.4</td>
<td>PAUSE_PROFILE</td>
<td></td>
<td>124</td>
</tr>
<tr>
<td>3.11.5</td>
<td>RESUME_PROFILE</td>
<td></td>
<td>124</td>
</tr>
<tr>
<td>3.11.6</td>
<td>START_PROFILE</td>
<td></td>
<td>124</td>
</tr>
<tr>
<td>3.11.7</td>
<td>STOP_PROFILE</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td>3.11.8</td>
<td>Using DBMS_PROFILER</td>
<td></td>
<td>126</td>
</tr>
<tr>
<td>3.11.8.1</td>
<td>Querying the DBMS_PROFILER Tables and View</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>3.11.8.2</td>
<td>DBMS_PROFILER - Reference</td>
<td></td>
<td>135</td>
</tr>
</tbody>
</table>

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.
3.12 DBMS_RANDOM ......................................................... 141
  3.12.1 INITIALIZE ...................................................... 141
  3.12.2 NORMAL ......................................................... 142
  3.12.3 RANDOM ......................................................... 142
  3.12.4 SEED ........................................................... 143
  3.12.5 SEED ........................................................... 143
  3.12.6 STRING ........................................................ 144
  3.12.7 TERMINATE ..................................................... 144
  3.12.8 VALUE .......................................................... 145
  3.12.9 VALUE .......................................................... 145

3.13 DBMS_RLS ............................................................ 147
  3.13.1 ADD_POLICY .................................................... 150
  3.13.2 DROP_POLICY ................................................... 157
  3.13.3 ENABLE_POLICY ............................................... 158

3.14 DBMS_SCHEDULER .................................................. 160
  3.14.1 Using Calendar Syntax to Specify a Repeating Interval .......... 161
  3.14.2 CREATE_JOB .................................................... 163
  3.14.3 CREATE_PROGRAM ............................................. 166
  3.14.4 CREATE_SCHEDULE .......................................... 168
  3.14.5 DEFINE_PROGRAM_ARGUMENT ................................ 170
  3.14.6 DISABLE ........................................................ 172
  3.14.7 DROP_JOB ....................................................... 173
  3.14.8 DROP_PROGRAM ................................................ 174
  3.14.9 DROP_PROGRAM_ARGUMENT .................................... 175
  3.14.10 DROP_SCHEDULE .............................................. 176
  3.14.11 ENABLE ........................................................ 176
  3.14.12 EVALUATECALENDAR_STRING ................................ 177
  3.14.13 RUN_JOB ....................................................... 179
  3.14.14 SET_JOB ARGUMENT_VALUE .................................. 179

3.15 DBMS_SESSION ...................................................... 181
  3.15.1 SET_ROLE ....................................................... 181

3.16 DBMS_SQL ........................................................... 182
  3.16.1 BIND_VARIABLE ................................................ 183
  3.16.2 BIND_VARIABLE_CHAR ....................................... 185
  3.16.3 BIND_VARIABLE_RAW ......................................... 185
  3.16.4 CLOSE_CURSOR ............................................... 186
  3.16.5 COLUMN_VALUE ............................................... 186
  3.16.6 COLUMN_VALUE_CHAR ......................................... 188
  3.16.7 COLUMN_VALUE_RAW .......................................... 188
  3.16.8 DEFINE_COLUMN ............................................... 189
  3.16.9 DEFINE_COLUMN_CHAR ....................................... 191
  3.16.10 DEFINE_COLUMN_RAW ....................................... 192
  3.16.11 DESCRIBE_COLUMNS ......................................... 192
  3.16.12 EXECUTE ........................................................ 193
  3.16.13 EXECUTE_AND_FETCH ....................................... 194
  3.16.14 FETCH_ROWS .................................................. 195
  3.16.15 IS_OPEN ...................................................... 197
  3.16.16 LAST_ROW_COUNT ............................................ 197
  3.16.17 OPEN_CURSOR ............................................... 199
  3.16.18 PARSE ........................................................ 199

3.17 DBMSUTILITY ...................................................... 202
  3.17.1 LNAME_ARRAY .................................................. 203
Database Compatibility for Oracle® Developers
Built-in Package Guide

3.17.2 UNCL_ARRAY .......................................................... 203
3.17.3 ANALYZE_DATABASE, ANALYZE_SCHEMA and ANALYZE_PART_OBJECT .. 203
3.17.4 CANONICALIZE ...................................................... 206
3.17.5 COMMA_TO_TABLE ................................................. 207
3.17.6 DB_VERSION .......................................................... 208
3.17.7 EXEC_DDL_STATEMENT .......................................... 209
3.17.8 FORMAT_CALL_STACK ........................................... 210
3.17.9 GET_CPU_TIME ..................................................... 210
3.17.10 GET_DEPENDENCY ................................................ 211
3.17.11 GET_HASH_VALUE ............................................... 212
3.17.12 GET_PARAMETER_VALUE ....................................... 213
3.17.13 GET_TIME .......................................................... 214
3.17.14 NAME_TOKENIZE ................................................ 215
3.17.15 TABLE_TO_COMMA ............................................... 217
3.18 UTL_ENCODE .................................................................. 219
3.18.1 BASE64_DECODER .................................................. 219
3.18.2 BASE64_ENCODE ..................................................... 220
3.18.3 MIME64_DECODE .................................................... 221
3.18.4 MIME64_ENCODE ..................................................... 222
3.18.5 QUOTED_PRINTABLE_DECODER .................................. 223
3.18.6 QUOTED_PRINTABLE_ENCODE ................................... 224
3.18.7 TEXT_ENCODE ......................................................... 225
3.18.8 TEXT_ENCODE ......................................................... 226
3.18.9 UUDECODE ............................................................ 227
3.18.10 UUENCODE ........................................................... 228
3.19 UTL_FILE .................................................................... 231
3.19.1 Setting File Permissions with utl_file.umask ... 232
3.19.2 FCLOSE ................................................................. 234
3.19.3 FCLOSE_ALL .......................................................... 234
3.19.4 FCOPY ................................................................. 234
3.19.5 FFLUSH ................................................................. 236
3.19.6 FOPEN ................................................................. 237
3.19.7 FREMOVE ............................................................. 238
3.19.8 FRENAM ............................................................... 238
3.19.9 GET_LINE .............................................................. 240
3.19.10 IS_OPEN .............................................................. 241
3.19.11 NEW_LINE ........................................................... 242
3.19.12 PUT ................................................................. 243
3.19.13 PUT_LINE ............................................................. 245
3.19.14 PUTF ................................................................. 246
3.20 UTL_HTTP .................................................................. 248
3.20.1 HTML_PIECES ......................................................... 251
3.20.2 REQ ................................................................. 251
3.20.3 RESP ................................................................. 251
3.20.4 BEGIN_REQUEST .................................................. 252
3.20.5 END_REQUEST ..................................................... 252
3.20.6 END_RESPONSE .................................................... 253
3.20.7 GET_BODY_CHARSET ............................................. 253
3.20.8 GET_FOLLOW_REDIRECT ........................................ 254
3.20.9 GET_HEADER ......................................................... 254
3.20.10 GET_HEADER_BY_NAME ......................................... 256
3.20.11 GET_HEADER_COUNT .......................................... 257
3.20.12 GET_RESPONSE .................................................. 257
3.20.13 GET_RESPONSE_ERROR_CHECK .............................. 257
3.20.14 GET_TRANSFER_TIMEOUT ..................................... 258
3.20.15 READ_LINE ........................................................ 258

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.
| 3.20.16 | READ_RAW | ................................................................. | .260 |
| 3.20.17 | READ_TEXT | ................................................................. | .261 |
| 3.20.18 | REQUEST | ................................................................. | .262 |
| 3.20.19 | REQUEST_PIECES | ................................................................. | .262 |
| 3.20.20 | SET_BODY_CHARSET | ................................................................. | .263 |
| 3.20.21 | SET_FOLLOW_REDIRECT | ................................................................. | .263 |
| 3.20.22 | SET_HEADER | ................................................................. | .264 |
| 3.20.23 | SET_RESPONSE_ERROR_CHECK | ................................................................. | .264 |
| 3.20.24 | SET_TRANSFER_TIMEOUT | ................................................................. | .265 |
| 3.20.25 | WRITE_LINE | ................................................................. | .265 |
| 3.20.26 | WRITE_RAW | ................................................................. | .266 |
| 3.20.27 | WRITE_TEXT | ................................................................. | .267 |
| 3.21 | UTL_MAIL | ................................................................. | .269 |
| 3.21.1 | SEND | ................................................................. | .269 |
| 3.21.2 | SEND_ATTACH_RAW | ................................................................. | .270 |
| 3.21.3 | SEND_ATTACH_VARCHAR2 | ................................................................. | .272 |
| 3.22 | UTL_RAW | ................................................................. | .274 |
| 3.22.1 | CAST_TO_RAW | ................................................................. | .274 |
| 3.22.2 | CAST_TO_VARCHAR2 | ................................................................. | .275 |
| 3.22.3 | CONCAT | ................................................................. | .276 |
| 3.22.4 | CONVERT | ................................................................. | .276 |
| 3.22.5 | LENGTH | ................................................................. | .277 |
| 3.22.6 | SUBSTR | ................................................................. | .278 |
| 3.23 | UTL_SMTP | ................................................................. | .280 |
| 3.23.1 | CONNECTION | ................................................................. | .281 |
| 3.23.2 | REPLY/REPLIES | ................................................................. | .281 |
| 3.23.3 | CLOSE_DATA | ................................................................. | .281 |
| 3.23.4 | COMMAND | ................................................................. | .282 |
| 3.23.5 | COMMAND_REPLIES | ................................................................. | .282 |
| 3.23.6 | DATA | ................................................................. | .283 |
| 3.23.7 | EHLO | ................................................................. | .283 |
| 3.23.8 | HELO | ................................................................. | .284 |
| 3.23.9 | HELP | ................................................................. | .284 |
| 3.23.10 | MAIL | ................................................................. | .285 |
| 3.23.11 | NOOP | ................................................................. | .285 |
| 3.23.12 | OPEN_CONNECTION | ................................................................. | .286 |
| 3.23.13 | OPEN_DATA | ................................................................. | .286 |
| 3.23.14 | QUIT | ................................................................. | .287 |
| 3.23.15 | RCPT | ................................................................. | .287 |
| 3.23.16 | RSET | ................................................................. | .288 |
| 3.23.17 | VRFY | ................................................................. | .288 |
| 3.23.18 | WRITE_DATA | ................................................................. | .288 |
| 3.23.19 | Comprehensive Example | ................................................................. | .289 |
| 3.24 | UTL_URL | ................................................................. | .291 |
| 3.24.1 | ESCAPE | ................................................................. | .291 |
| 3.24.2 | UNESCAPE | ................................................................. | .293 |

4 Acknowledgements ................................................................. .295
1 Introduction

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.

For more information about using other compatibility features offered by Advanced Server, please see the complete set of Advanced Server guides, available at:

http://www.enterprisedb.com/products-services-training/products/documentation
1.1 Typographical Conventions Used in this Guide

Certain typographical conventions are used in this manual to clarify the meaning and usage of various commands, statements, programs, examples, etc. This section provides a summary of these conventions.

In the following descriptions a term refers to any word or group of words which may be language keywords, user-supplied values, literals, etc. A term’s exact meaning depends upon the context in which it is used.

- *Italic font* introduces a new term, typically, in the sentence that defines it for the first time.
- *Fixed-width (mono-spaced) font* is used for terms that must be given literally such as SQL commands, specific table and column names used in the examples, programming language keywords, etc. For example, `SELECT * FROM emp;`
- *Italic fixed-width font* is used for terms for which the user must substitute values in actual usage. For example, `DELETE FROM table_name;`
- A vertical pipe | denotes a choice between the terms on either side of the pipe. A vertical pipe is used to separate two or more alternative terms within square brackets (optional choices) or braces (one mandatory choice).
- Square brackets [] denote that one or none of the enclosed term(s) may be substituted. For example, `[ a | b ]`, means choose one of “a” or “b” or neither of the two.
- Braces {} denote that exactly one of the enclosed alternatives must be specified. For example, `{ a | b }`, means exactly one of “a” or “b” must be specified.
- Ellipses ... denote that the proceeding term may be repeated. For example, `[ a | b ] ...` means that you may have the sequence, “b a a b a”.
2 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.

### 2.1 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.
2.1.1 Package Specification Syntax

The package specification defines the user interface for a package (the API). The specification lists the functions, procedures, types, exceptions and cursors that are visible to a user of the package.

The syntax used to define the interface for a package is:

```
CREATE [ OR REPLACE ] PACKAGE package_name
  [ authorization_clause ]
  { IS | AS }
  [ declaration; ] ...
  [ procedure_or_function_declaration ] ...
END [ package_name ] ;
```

Where `authorization_clause` :=

```
{ AUTHID DEFINER } | { AUTHID CURRENT_USER }
```

Where `procedure_or_function_declaration` :=

```
procedure_declaration | function_declaration
```

Where `procedure_declaration` :=

```
PROCEDURE proc_name [ argument_list ];
[ restrictionPragma; ]
```

Where `function_declaration` :=

```
FUNCTION func_name [ argument_list ]
  RETURN rettype [ DETERMINISTIC ];
  [ restrictionPragma; ]
```

Where `argument_list` :=

```
( argument_declaration [, ...] )
```

Where `argument_declaration` :=

```
argname [ IN | IN OUT | OUT ] argtype [ DEFAULT value ]
```

Where `restrictionPragma` :=

```
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 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.

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
2.1.2 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`:

```
[ declaration; ] [, ...]
BEGIN
  statement; [...] 
[ EXCEPTION
  { WHEN exception [OR exception] [...] THEN statement; }
  [...] ]
```

Where `function_definition`:

```
FUNCTION func_name [ argument_list ]
RETURN rettype [ DETERMINISTIC ]
{ options_list }
```
{ IS | AS }
    function_body
END [ func_name ] ;

Where function_body :=

[ declaration; ] [, ...]
BEGIN
    statement; [...] 
    [ EXCEPTION
      { WHEN exception [ OR exception ] [...] THEN statement; }
      [...] 
    ]

Where 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
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.

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.)

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.

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.
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`.

**result_rows**

`result_rows` is the estimated number of rows that the query planner should expect the function to return. The default is `1000`. 
Use the SET clause to specify a parameter value for the duration of the function:

\[ \text{config\_param} \text{ specifies the parameter name.} \]

\[ \text{value} \text{ specifies the parameter value.} \]

\[ \text{FROM CURRENT} \text{ guarantees that the parameter value is restored when the function ends.} \]

\text{package\_initializer}

The statements in the package\_initializer are executed once per user’s session when the package is first referenced.

Please Note: The STRICT, LEAKPROOF, PARALLEL, COST, ROWS and SET keywords provide extended functionality for Advanced Server and are not supported by Oracle.
2.2 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.

2.2.1 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:

```sql
-- Package specification for the 'emp_admin' package.
CREATE OR REPLACE PACKAGE emp_admin
IS
  FUNCTION get_dept_name (p_deptno NUMBER DEFAULT 10)
    RETURN VARCHAR2;
  FUNCTION update_emp_sal (p_empno NUMBER,
                           p_raise NUMBER)
    RETURN NUMBER;
  PROCEDURE hire_emp (p_empno NUMBER,
                      p_ename VARCHAR2,
                      p_job VARCHAR2,
                      p_sal NUMBER,
                      p_hiredate DATE DEFAULT sysdate,
                      p_comm NUMBER DEFAULT 0,
                      p_mgr NUMBER,
                      p_deptno NUMBER DEFAULT 10);
  PROCEDURE fire_emp (p_empno NUMBER);
END emp_admin;
```

This code sample creates the `emp_admin` package specification. This package specification consists of two functions and two stored procedures. We can also add the `OR REPLACE` clause to the `CREATE PACKAGE` statement for convenience.

2.2.2 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  
  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;
2.3 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
```

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.
2.4 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.

```sql
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`.

```sql
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
    CURSOR dept_cur IS SELECT * FROM dept;
    TYPE depttab_typ IS TABLE OF dept%ROWTYPE INDEX BY BINARY_INTEGER;
    t_dept          DEPTTAB_TYP;
    t_dept_max      INTEGER := 1;
    r_emp           EMPREC_TYP;

    FUNCTION get_dept_name (p_deptno IN NUMBER) RETURN VARCHAR2
    IS
    BEGIN
        FOR i IN 1..t_dept_max LOOP
            IF p_deptno = t_dept(i).deptno THEN
                RETURN t_dept(i).dname;
            END IF;
        END LOOP;
    END;
```
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 passed between the package function and procedures.

```sql
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.

```sql
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
    r_emp           emp_rpt.EMPREC_TYP;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno || ' : ' || emp_rpt.get_dept_name(v_deptno));
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    ------');
    LOOP
        FETCH v_emp_cur INTO r_emp;
        EXIT WHEN v_emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' || r_emp.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('***********
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    CLOSE v_emp_cur;
END;
```
The following is the result of this anonymous block.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7499</td>
<td>ALLEN</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
</tr>
</tbody>
</table>

********************
6 rows were retrieved
2.5 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;
```
3 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, please see the Database Compatibility for Oracle Developers Reference Guide, available at:

http://www.enterprisedb.com/products-services-training/products/documentation

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;
3.1 DBMS_ALERT

The DBMS_ALERT package provides the capability to register for, send, and receive alerts. The following table lists the supported procedures:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGISTER(name)</td>
<td>n/a</td>
<td>Register to be able to receive alerts named, name.</td>
</tr>
<tr>
<td>REMOVE(name)</td>
<td>n/a</td>
<td>Remove registration for the alert named, name.</td>
</tr>
<tr>
<td>REMOVEALL</td>
<td>n/a</td>
<td>Remove registration for all alerts.</td>
</tr>
<tr>
<td>SIGNAL(name, message)</td>
<td>n/a</td>
<td>Signals the alert named, name, with message.</td>
</tr>
<tr>
<td>WAITANY(name, OUT, message OUT, status OUT, timeout)</td>
<td>n/a</td>
<td>Wait for any registered alert to occur.</td>
</tr>
<tr>
<td>WAITONE(name, OUT, message OUT, status OUT, timeout)</td>
<td>n/a</td>
<td>Wait for the specified alert, name, to occur.</td>
</tr>
</tbody>
</table>

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/10AS/data) with your choice of editor, and edit the dbms_alert.max_alerts parameter as shown:

```
dbms_alert.max_alerts = alert_count
```

alert_count specifies the maximum number of concurrent alerts. By default, the value of dbms_alert.max_alerts is 100. To disable this feature, set dbms_alert.max_alerts to 0.

For the dbms_alert.max_alerts GUC to function correctly, the custom_variable_classes parameter must contain dbms_alerts:

```
custom_variable_classes = 'dbms_alert, ...
```

After editing the postgresql.conf file parameters, you must restart the server for the changes to take effect.
3.1.1 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
    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...

3.1.2 REMOVE

The REMOVE procedure unregisters the session for the named alert.

REMOVE(name VARCHAR2)

Parameters

name

Name of the alert to be unregistered.
3.1.3 REMOVEALL

The REMOVEALL procedure unregisters the session for all alerts.

REMOVEALL

3.1.4 SIGNAL

The SIGNAL procedure signals the occurrence of the named alert.

SIGNAL(name VARCHAR2, message VARCHAR2)

Parameters

name

Name of the alert.

message

Information to pass with this alert.

Examples

The following anonymous block signals an alert for alert_test.

```sql
DECLARE
  v_name   VARCHAR2(30) := 'alert_test';
BEGIN
  DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END;
```

3.1.5 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');
   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.REMOVEAL;
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

3.1.6 WAITONE

The WAITONE procedure waits for the specified registered alert to occur.

\[
\text{WAITONE}(\text{name VARCHAR2, message OUT VARCHAR2, status OUT INTEGER, timeout NUMBER})
\]

Parameters

name

Name of the alert.

message

Variable receiving the message sent by the SIGNAL procedure.

status

Status code returned by the operation. Possible values are: 0 – alert occurred; 1 – timeout occurred.

timeout

Time to wait for an alert in seconds.

Examples

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.

```sql
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');
END;
```
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

3.1.7 Comprehensive Example

The following example uses two triggers to send alerts when the dept table or the emp table is changed. An anonymous block listens for these alerts and displays messages when an alert is received.

The following are the triggers on the dept and emp tables:

CREATE OR REPLACE TRIGGER dept_alert_trig  
  AFTER INSERT OR UPDATE OR DELETE ON dept  
DECLARE  
  v_action VARCHAR2(25);
BEGIN  
  IF INSERTING THEN  
    v_action := ' added department(s) ';  
  ELSIF UPDATING THEN  
    v_action := ' updated department(s) ';  
  ELSIF DELETING THEN  
    v_action := ' deleted department(s) ';  
  END IF;
  DBMS_ALERT.SIGNAL('dept_alert',USER || v_action || 'on ' || SYSDATE);
END;

CREATE OR REPLACE TRIGGER emp_alert_trig  
  AFTER INSERT OR UPDATE OR DELETE ON emp  
DECLARE  
  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 \textit{dept} and \textit{emp} tables occur in other sessions:

\begin{verbatim}
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;
\end{verbatim}

Registered for alerts dept_alert and emp_alert
Waiting for signal...

The following changes are made by user, mary:

\begin{verbatim}
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);
\end{verbatim}

The following change is made by user, john:

\begin{verbatim}
INSERT INTO dept VALUES (60,'HR','LOS ANGELES');
\end{verbatim}

The following is the output displayed by the anonymous block receiving the signals from the triggers:

\begin{verbatim}
Registered for alerts dept_alert and emp_alert
Waiting for signal...
\end{verbatim}
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
3.2 **DBMS_AQ**

EDB Postgres Advanced Server Advanced Queueing provides message queuing 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. Please see the *Database Compatibility for Oracle Developers Reference 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:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENQUEUE</td>
<td>n/a</td>
<td>Post a message to a queue.</td>
</tr>
<tr>
<td>DEQUEUE</td>
<td>n/a</td>
<td>Retrieve a message from a queue if or when a message is available.</td>
</tr>
<tr>
<td>REGISTER</td>
<td>n/a</td>
<td>Register a callback procedure.</td>
</tr>
<tr>
<td>UNREGISTER</td>
<td>n/a</td>
<td>Unregister a callback procedure.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
<th>For Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_AQ.BROWSE (0)</td>
<td>Read the message without locking.</td>
<td>dequeue_options_t.dequeue_mode</td>
</tr>
<tr>
<td>DBMS_AQ.LOCKED (1)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>dequeue_options_t.dequeue_mode</td>
</tr>
<tr>
<td>DBMS_AQ.REMOVE (2)</td>
<td>Delete the message after reading; the default.</td>
<td>dequeue_options_t.dequeue_mode</td>
</tr>
<tr>
<td>Constant</td>
<td>Description</td>
<td>For Parameters</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>DBMS_AQ.REMOVE_NODATA (3)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>dequeue_options_t.dequeue_mode</td>
</tr>
<tr>
<td>DBMS_AQ-FIRST_MESSAGE (0)</td>
<td>Return the first available message that matches the search terms.</td>
<td>dequeue_options_t.navigation</td>
</tr>
<tr>
<td>DBMS_AQ.NEXT_MESSAGE (1)</td>
<td>Return the next available message that matches the search terms.</td>
<td>dequeue_options_t.navigation</td>
</tr>
<tr>
<td>DBMS_AQ.NEXT_TRANSACTION (2)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>dequeue_options_t.navigation</td>
</tr>
<tr>
<td>DBMS_AQ.FOREVER (0)</td>
<td>Wait forever if a message that matches the search term is not found, the default.</td>
<td>dequeue_options_t.wait</td>
</tr>
<tr>
<td>DBMS_AQ.NO_WAIT (1)</td>
<td>Do not wait if a message that matches the search term is not found.</td>
<td>dequeue_options_t.wait</td>
</tr>
<tr>
<td>DBMS_AQ.ON_COMMIT (0)</td>
<td>The dequeue is part of the current transaction.</td>
<td>enqueue_options_t.visibility,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enqueue_options_t.visibility</td>
</tr>
<tr>
<td>DBMS_AQ.IMMEDIATE (1)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>enqueue_options_t.visibility,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>enqueue_options_t.visibility</td>
</tr>
<tr>
<td>DBMS_AQ.PERSISTENT (0)</td>
<td>The message should be stored in a table.</td>
<td>enqueue_options_t.delivery_mode</td>
</tr>
<tr>
<td>DBMS_AQ.BUFFERED (1)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>enqueue_options_t.delivery_mode</td>
</tr>
<tr>
<td>DBMS_AQ.READY (0)</td>
<td>Specifies that the message is ready to process.</td>
<td>message_properties_t.state</td>
</tr>
<tr>
<td>DBMS_AQ.WAITING (1)</td>
<td>Specifies that the message is waiting to be processed.</td>
<td>message_properties_t.state</td>
</tr>
<tr>
<td>DBMS_AQ.PROCESSED (2)</td>
<td>Specifies that the message has been processed.</td>
<td>message_properties_t.state</td>
</tr>
<tr>
<td>DBMS_AQ.EXPIRED (3)</td>
<td>Specifies that the message is in the exception queue.</td>
<td>message_properties_t.state</td>
</tr>
<tr>
<td>DBMS_AQ.NO_DELAY (0)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>message_properties_t.delay</td>
</tr>
<tr>
<td>DBMS_AQ.NEVER (NULL)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>message_properties_t.expiration</td>
</tr>
<tr>
<td>DBMS_AQ.NAMESPACE_AQ (0)</td>
<td>Accept notifications from DBMS_AQ queues.</td>
<td>sys.aq$_reg_info.namespace</td>
</tr>
<tr>
<td>DBMS_AQ.NAMESPACE_ANONYMOUS (1)</td>
<td>This constant is defined, but will return an error if used</td>
<td>sys.aq$_reg_info.namespace</td>
</tr>
</tbody>
</table>

### 3.2.1 ENQUEUE

The **ENQUEUE** procedure adds an entry to a queue. The signature is:

```sql
ENQUEUE(
    queue_name IN VARCHAR2,
    enqueue_options IN DBMS_AQ.ENQUEUE_OPTIONS_T,
```
message_properties IN DBMS_AQ.MESSAGE_PROPERTIES_T,
payload IN <type_name>,
msgid OUT RAW)

Parameters

queue_name

The name (optionally schema-qualified) of an existing queue. If you omit the
schema name, the server 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, please 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>visibility</td>
<td>ON_COMMIT</td>
</tr>
<tr>
<td>delivery_mode</td>
<td>PERSISTENT</td>
</tr>
<tr>
<td>sequence_deviation</td>
<td>NULL</td>
</tr>
<tr>
<td>transformation</td>
<td>NULL</td>
</tr>
<tr>
<td>relative_msgid</td>
<td>NULL</td>
</tr>
</tbody>
</table>

message_properties

message_properties is a value of the type, message_properties_t:

message_properties_t IS RECORD(
    priority BINARY_INTEGER NOT NULL DEFAULT 1
    delay BINARY_INTEGER NOT NULL DEFAULT NO_DELAY,
    expiration BINARY_INTEGER NOT NULL DEFAULT NEVER,
    correlation VARCHAR2(128) DEFAULT NULL,
    attempts BINARY_INTEGER
    recipient_list AQ$_RECIPIENT_LIST_T,
    exception_queue VARCHAR2(61) DEFAULT NULL,
    enqueue_time DATE,
state BINARY_INTEGER,
sender_id SYS.AQ$_AGENT DEFAULT NULL,
original_msgid RAW(16) DEFAULT NULL,
signature aq$_sig_prop DEFAULT NULL,
transaction_group VARCHAR2(30) DEFAULT NULL,
user_property SYS.ANYDATA DEFAULT NULL,
delivery_mode PLS_INTEGER NOT NULL DEFAULT
DBMS_AQ.PERSISTENT);

The supported values for message_properties_t are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>If the queue table definition includes a sort_list that references priority, this parameter affects the order that messages are dequeued. A lower value indicates a higher dequeue priority.</td>
</tr>
<tr>
<td>delay</td>
<td>Specify the number of seconds that will pass before a message is available for dequeueing or NO_DELAY.</td>
</tr>
<tr>
<td>expiration</td>
<td>Use the expiration parameter to specify the number of seconds until a message expires.</td>
</tr>
<tr>
<td>correlation</td>
<td>Use correlation to specify a message that will be associated with the entry; the default is NULL.</td>
</tr>
<tr>
<td>attempts</td>
<td>This is a system-maintained value that specifies the number of attempts to dequeue the message.</td>
</tr>
<tr>
<td>recipient_list</td>
<td>This parameter is not supported.</td>
</tr>
<tr>
<td>exception_queue</td>
<td>Use the exception_queue parameter to specify the name of an exception queue to which a message will be moved if it expires or is dequeued by a transaction that rolls back too many times.</td>
</tr>
<tr>
<td>enqueue_time</td>
<td>enqueue_time is the time the record was added to the queue; this value is provided by the system.</td>
</tr>
<tr>
<td>state</td>
<td>This parameter is maintained by DBMS_AQ; state can be:</td>
</tr>
<tr>
<td></td>
<td>- DBMS_AQ.WAITING – the delay has not been reached.</td>
</tr>
<tr>
<td></td>
<td>- DBMS_AQ.READY – the queue entry is ready for processing.</td>
</tr>
<tr>
<td></td>
<td>- DBMS_AQ.PROCESSED – the queue entry has been processed.</td>
</tr>
<tr>
<td></td>
<td>- DBMS_AQ.EXPIRED – the queue entry has been moved to the exception queue.</td>
</tr>
<tr>
<td>sender_id</td>
<td>This parameter is not supported; specify a value of NULL.</td>
</tr>
<tr>
<td>original_msgid</td>
<td>This parameter is accepted for compatibility and ignored.</td>
</tr>
<tr>
<td>signature</td>
<td>This parameter is not supported; specify a value of NULL.</td>
</tr>
<tr>
<td>transaction_group</td>
<td>This parameter is accepted for compatibility and ignored.</td>
</tr>
<tr>
<td>user_property</td>
<td>This parameter is not supported; specify a value of NULL.</td>
</tr>
<tr>
<td>delivery_mode</td>
<td>This parameter is not supported; specify a value of DBMS_AQ.PERSISTENT.</td>
</tr>
</tbody>
</table>

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`:

```sql
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;
```

### 3.2.2 DEQUEUE

The `DEQUEUE` procedure dequeues a message. The signature is:

```sql
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, enclose the name in double quotes.
For detailed information about creating a queue, please see
DBMS_AQADM.CREATE_QUEUE.

dequeue_options

dequeue_options is a value of the type, dequeue_options_t:

DEQUEUE_OPTIONS_T IS RECORD(
  consumer_name VARCHAR2(30) DEFAULT NULL,
  dequeue_mode BINARY_INTEGER DEFAULT REMOVE,
  navigation BINARY_INTEGER DEFAULT NEXT_MESSAGE,
  visibility BINARY_INTEGER DEFAULT ON_COMMIT,
  wait BINARY_INTEGER DEFAULT FOREVER,
  msgid RAW(16) DEFAULT NULL,
  correlation VARCHAR2(128) DEFAULT NULL,
  deq_condition VARCHAR2(4000) DEFAULT NULL,
  signature aq$_sig_prop DEFAULT NULL,
  transformation VARCHAR2(61) DEFAULT NULL,
  delivery_mode PLS_INTEGER DEFAULT PERSISTENT)

Currently, the supported parameter values for dequeue_options_t are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>consumer_name</td>
<td>Must be NULL.</td>
</tr>
<tr>
<td>dequeue_mode</td>
<td>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.</td>
</tr>
<tr>
<td>navigation</td>
<td>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.</td>
</tr>
<tr>
<td>visibility</td>
<td>Must be ON_COMMIT – if you roll back the current transaction the dequeued item will remain in the queue.</td>
</tr>
<tr>
<td>wait</td>
<td>Must be a number larger than 0, or: DBMS_AQ.FOREVER – Wait indefinitely. DBMS_AQ.NO_WAIT – Do not wait.</td>
</tr>
<tr>
<td>msgid</td>
<td>The message ID of the message that will be dequeued.</td>
</tr>
<tr>
<td>correlation</td>
<td>Accepted for compatibility, and ignored.</td>
</tr>
<tr>
<td>deq_condition</td>
<td>A VARCHAR2 expression that evaluates to a BOOLEAN value, indicating if the message</td>
</tr>
<tr>
<td>Message Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>signature</td>
<td>Accepted for compatibility, and ignored.</td>
</tr>
<tr>
<td>transformation</td>
<td>Accepted for compatibility, and ignored.</td>
</tr>
<tr>
<td>delivery_mode</td>
<td>Must be PERSISTENT, buffered messages are not supported at this time.</td>
</tr>
</tbody>
</table>

`message_properties`

`message_properties` is a value of the type, `message_properties_t`:

```sql
message_properties_t IS RECORD(
    priority BINARY_INTEGER NOT NULL DEFAULT 1,
    delay BINARY_INTEGER NOT NULL DEFAULT NO_DELAY,
    expiration BINARY_INTEGER NOT NULL DEFAULT NEVER,
    correlation VARCHAR2(128) DEFAULT NULL,
    attempts BINARY_INTEGER,
    recipient_list AQ$_RECIPIENT_LIST_T,
    exception_queue VARCHAR2(61) DEFAULT NULL,
    enqueue_time DATE,
    state BINARY_INTEGER,
    sender_id SYS.AQ$_AGENT DEFAULT NULL,
    original_msgid RAW(16) DEFAULT NULL,
    signature AQ$_sig_prop DEFAULT NULL,
    transaction_group VARCHAR2(30) DEFAULT NULL,
    user_property SYS.ANYDATA DEFAULT NULL,
    delivery_mode PLS_INTEGER NOT NULL DEFAULT DBMS_AQ.PERSISTENT);
```

The supported values for `message_properties_t` are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>If the queue table definition includes a sort_list that references <code>priority</code>, this parameter affects the order that messages are dequeued. A lower value indicates a higher dequeue priority.</td>
</tr>
<tr>
<td>delay</td>
<td>Specify the number of seconds that will pass before a message is available for dequeueing or NO_DELAY.</td>
</tr>
<tr>
<td>expiration</td>
<td>Use the expiration parameter to specify the number of seconds until a message expires.</td>
</tr>
<tr>
<td>correlation</td>
<td>Use correlation to specify a message that will be associated with the entry; the default is NULL.</td>
</tr>
<tr>
<td>attempts</td>
<td>This is a system-maintained value that specifies the number of attempts to dequeue the message.</td>
</tr>
<tr>
<td>recipient_list</td>
<td>This parameter is not supported.</td>
</tr>
<tr>
<td>exception_queue</td>
<td>Use the exception_queue parameter to specify the name of an exception queue to which a message will be moved if it expires or is dequeued by a transaction that rolls back too many times.</td>
</tr>
<tr>
<td>enqueue_time</td>
<td>enqueue_time is the time the record was added to the queue; this value is provided by the system.</td>
</tr>
<tr>
<td>state</td>
<td>This parameter is maintained by DBMS_AQ; state can be:</td>
</tr>
</tbody>
</table>
payload

Use the `payload` parameter to retrieve the payload of a message with a dequeue operation. The payload type must match the type specified when creating the queue table.

msgid

Use the `msgid` parameter to retrieve a unique message identifier.

Example

The following anonymous block calls `DBMS_AQ.DEQUEUE`, retrieving a message from the queue and a payload:

```sql
DECLARE
  dequeue_options    DBMS_AQ.DEQUEUE_OPTIONS_T;
  message_properties DBMS_AQ.MESSAGE_PROPERTIES_T;
  message_handle     raw(16);
  payload            work_order;
BEGIN
  dequeue_options.dequeue_mode := DBMS_AQ.BROWSE;
  DECODE (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`.

### 3.2.3 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:

```sql
REGISTER(
    reg_list IN SYS.AQ$ REG_INFO_LIST,
    count IN NUMBER)
```

#### Parameters

**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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>VARCHAR2 (128)</td>
<td>The (optionally schema-qualified) name of the subscription.</td>
</tr>
<tr>
<td>namespace</td>
<td>NUMERIC</td>
<td>The only supported value is <code>DBMS_AQ_NAMESPACE_AQ(0)</code></td>
</tr>
<tr>
<td>callback</td>
<td>VARCHAR2 (4000)</td>
<td>Describes the action that will be performed upon notification. Currently, only calls to PL/SQL procedures are supported. The call should take the form: <code>plsql://schema.procedure</code> Where: <code>schema</code> specifies the schema in which the procedure resides. <code>procedure</code> specifies the name of the procedure that will be notified.</td>
</tr>
<tr>
<td>context</td>
<td>RAW (16)</td>
<td>Any user-defined value required by the callback procedure.</td>
</tr>
</tbody>
</table>

**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.

### 3.2.4 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)
```

**Parameters**

*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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>VARCHAR2(128)</td>
<td>The (optionally schema-qualified) name of the subscription.</td>
</tr>
</tbody>
</table>
**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.REREGISTER:

```sql
DECLARE
    subscription1 sys.aq$_reg_info;
    subscription2 sys.aq$_reg_info;
    subscription3 sys.aq$_reg_info;
    subscriptionlist sys.aq$_reg_info_list;
BEGIN
    subscription1 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ,
    'plsql://assign_worker?PR=0',HEXTORAW('FFFF'));
    subscription2 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ,
    'plsql://add_to_history?PR=1',HEXTORAW('FFFF'));
    subscription3 := sys.aq$_reg_info('q', DBMS_AQ.NAMESPACE_AQ,
    'plsql://reserve_parts?PR=2',HEXTORAW('FFFF'));

    subscriptionlist := sys.aq$_reg_info_list(subscription1, subscription2,
    subscription3);
    dbms_aq.unregister(subscriptionlist, 3);
    commit;
END;
/
```

The `subscriptionlist` is of type `sys.aq$_reg_info_list`, and contains the previously described `sys.aq$_reg_info` objects. The list name and an object count are passed to `dbms_aq.unregister`.  

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.
3.3 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 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. Please see the Database Compatibility for Oracle Developers Reference 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.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER_QUEUE</td>
<td>n/a</td>
<td>Modify an existing queue.</td>
</tr>
<tr>
<td>ALTER_QUEUE_TABLE</td>
<td>n/a</td>
<td>Modify an existing queue table.</td>
</tr>
<tr>
<td>CREATE_QUEUE</td>
<td>n/a</td>
<td>Create a queue.</td>
</tr>
<tr>
<td>CREATE_QUEUE_TABLE</td>
<td>n/a</td>
<td>Create a queue table.</td>
</tr>
<tr>
<td>DROP_QUEUE</td>
<td>n/a</td>
<td>Drop an existing queue.</td>
</tr>
<tr>
<td>DROP_QUEUE_TABLE</td>
<td>n/a</td>
<td>Drop an existing queue table.</td>
</tr>
<tr>
<td>PURGE_QUEUE_TABLE</td>
<td>n/a</td>
<td>Remove one or more messages from a queue table.</td>
</tr>
<tr>
<td>START_QUEUE</td>
<td>n/a</td>
<td>Make a queue available for enqueuing and dequeueing procedures.</td>
</tr>
<tr>
<td>STOP_QUEUE</td>
<td>n/a</td>
<td>Make a queue unavailable for enqueuing and dequeueing procedures.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
<th>For Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS_AQADM.TRANSACTIONAL(1)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>message_grouping</td>
</tr>
<tr>
<td>DBMS_AQADM.NONE(0)</td>
<td>Use to specify message grouping for a queue table.</td>
<td>message_grouping</td>
</tr>
<tr>
<td>DBMS_AQADM.NORMAL_QUEUE(0)</td>
<td>Use with create_queue to specify queue_type.</td>
<td>queue_type</td>
</tr>
<tr>
<td>DBMS_AQADM.EXCEPTION_QUEUE(1)</td>
<td>Use with create_queue to specify queue_type.</td>
<td>queue_type</td>
</tr>
<tr>
<td>DBMS_AQADM.INFINITE(-1)</td>
<td>Use with create_queue to specify retention time.</td>
<td>retention_time</td>
</tr>
<tr>
<td>DBMS_AQADM.PERSISTENT (0)</td>
<td>The message should be stored in a table.</td>
<td>enqueue_options.t.delivery_mode</td>
</tr>
<tr>
<td>DBMS_AQADM.BUFFERED (1)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>enqueue_options.t.delivery_mode</td>
</tr>
<tr>
<td>DBMS_AQADM.PERSISTENT_OR_BUFFERED (2)</td>
<td>This constant is defined, but will return an error if used.</td>
<td>enqueue_options.t.delivery_mode</td>
</tr>
</tbody>
</table>

### 3.3.1 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 dequeuing, 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:

```sql
EXEC DBMS_AQADM.ALTER_QUEUE(queue_name => 'work_order', retry_delay => 5);
```

3.3.2 ALTER_QUEUE_TABLE

Use the ALTER_QUEUE_TABLE procedure to modify an existing queue table. The signature is:

```sql
ALTER_QUEUE_TABLE (  
    queue_table IN VARCHAR2,
    comment IN VARCHAR2 DEFAULT NULL,
    primary_instance IN BINARY_INTEGER DEFAULT 0,
    secondary_instance IN BINARY_INTEGER DEFAULT 0,
)
```

Parameters

queue_table

The (optionally schema-qualified) name of the queue table.
comment

Use the comment parameter to provide a comment about the queue table.

primary_instance

primary_instance is accepted for compatibility and stored, but is ignored.

secondary_instance

secondary_instance is accepted for compatibility, but is ignored.

Example

The following command modifies a queue table named work_order_table:

```sql
EXEC DBMS_AQADM.ALTER_QUEUE_TABLE
(queue_table => 'work_order_table', comment => 'This queue table contains work orders for the shipping department.');
```

The queue table is named work_order_table; the command adds a comment to the definition of the queue table.

3.3.3 CREATE_QUEUE

Use the CREATE_QUEUE procedure to create a queue in an existing queue table. The signature is:

```sql
CREATE_QUEUE(
    queue_name IN VARCHAR2,
    queue_table IN VARCHAR2,
    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

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:

```sql
BEGIN
  DBMS_AQADM.CREATE_QUEUE ( queue_name => 'work_order', queue_table => 'work_order_table', comment => 'This queue contains pending work orders. ');
END;
```

### 3.3.4 CREATE_QUEUE_TABLE

Use the `CREATE_QUEUE_TABLE` procedure to create a queue table. The signature is:

```sql
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
- enq_time

(multiple_consumers)

If specified, (multiple_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)
**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.

### 3.3.5 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:

```sql
BEGIN
DBMS_AQADM.DROP_QUEUE(queue_name => 'work_order');
END;
```

3.3.6 DROP_QUEUE_TABLE

Use the DROP_QUEUE_TABLE procedure to delete a queue table. The signature is:

```sql
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.

auto_commit

auto_commit is accepted for compatibility, but is ignored.

Example

The following anonymous block drops a table named work_order_table:

```sql
BEGIN
```

```sql
```

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.
3.3.7 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>Boolean</td>
<td>Specify TRUE if an exclusive lock should be held on all queues within the table; the default is FALSE.</td>
</tr>
<tr>
<td>delivery_mode</td>
<td>INTEGER</td>
<td>delivery_mode specifies the type of message that will be purged. The only accepted value is DBMS_AQ.PERSISTENT.</td>
</tr>
</tbody>
</table>

Example

The following anonymous block removes any messages from the work_order_table with a value in the completed column of YES:

```
DECLARE
    purge_options dbms_aqadm.aq$_purge_options_t;
BEGIN
```

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.
3.3.8 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 enqueuing (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 enqueuing:

```
BEGIN
    DBMS_AQADM.START_QUEUE(
        queue_name => 'work_order');
END;
```

3.3.9 STOP_QUEUE

Use the STOP_QUEUE procedure to disable enqueuing or dequeueing on a specified queue. The signature is:

```
END;
```
STOP_QUEUE(
    queue_name IN VARCHAR2,
    enqueue IN BOOLEAN DEFAULT TRUE,
    dequeue IN BOOLEAN DEFAULT TRUE,
    wait IN BOOLEAN DEFAULT TRUE)

Parameters

queue_name

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.
3.4 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.

Table 7.7.2 DBMS_CRYPTO Functions and Procedures

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECRYPT(src, typ, key, iv)</td>
<td>RAW</td>
<td>Decrypts RAW data.</td>
</tr>
<tr>
<td>DECRYPT(dst INOUT, src, typ, key, iv)</td>
<td>N/A</td>
<td>Decrypts BLOB data.</td>
</tr>
<tr>
<td>DECRYPT(dst INOUT, src, typ, key, iv)</td>
<td>N/A</td>
<td>Decrypts CLOB data.</td>
</tr>
<tr>
<td>ENCRYPT(src, typ, key, iv)</td>
<td>RAW</td>
<td>Encrypts RAW data.</td>
</tr>
<tr>
<td>ENCRYPT(dst INOUT, src, typ, key, iv)</td>
<td>N/A</td>
<td>Encrypts BLOB data.</td>
</tr>
<tr>
<td>ENCRYPT(dst INOUT, src, typ, key, iv)</td>
<td>N/A</td>
<td>Encrypts CLOB data.</td>
</tr>
<tr>
<td>HASH(src, typ)</td>
<td>RAW</td>
<td>Applies a hash algorithm to RAW data.</td>
</tr>
<tr>
<td>HASH(src)</td>
<td>RAW</td>
<td>Applies a hash algorithm to CLOB data.</td>
</tr>
<tr>
<td>MAC(src, typ, key)</td>
<td>RAW</td>
<td>Returns the hashed MAC value of the given RAW data using the specified hash algorithm and key.</td>
</tr>
<tr>
<td>MAC(src, typ, key)</td>
<td>RAW</td>
<td>Returns the hashed MAC value of the given CLOB data using the specified hash algorithm and key.</td>
</tr>
<tr>
<td>RANDOMBYTES(number_bytes)</td>
<td>RAW</td>
<td>Returns a specified number of cryptographically strong random bytes.</td>
</tr>
<tr>
<td>RANDOMINTEGER()</td>
<td>INTEGER</td>
<td>Returns a random INTEGER.</td>
</tr>
<tr>
<td>RANDOMNUMBER()</td>
<td>NUMBER</td>
<td>Returns a random NUMBER.</td>
</tr>
</tbody>
</table>

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.
3.4.1 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 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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCRYPT DES</td>
<td>$\text{CONSTANT INTEGER := 1}$;</td>
</tr>
<tr>
<td>ENCRYPT 3DES</td>
<td>$\text{CONSTANT INTEGER := 3}$;</td>
</tr>
<tr>
<td>ENCRYPT AES</td>
<td>$\text{CONSTANT INTEGER := 4}$;</td>
</tr>
<tr>
<td>ENCRYPT AES128</td>
<td>$\text{CONSTANT INTEGER := 6}$;</td>
</tr>
</tbody>
</table>

### Block Cipher Modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAIN_CBC</td>
<td>$\text{CONSTANT INTEGER := 256}$;</td>
</tr>
<tr>
<td>CHAIN_ECB</td>
<td>$\text{CONSTANT INTEGER := 768}$;</td>
</tr>
</tbody>
</table>

### Block Cipher Padding Modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAD_PKCS5</td>
<td>$\text{CONSTANT INTEGER := 4096}$;</td>
</tr>
<tr>
<td>PAD_NONE</td>
<td>$\text{CONSTANT INTEGER := 8192}$;</td>
</tr>
</tbody>
</table>

### Block Cipher Suites

<table>
<thead>
<tr>
<th>Suite</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_CBC_PKCS5</td>
<td>$\text{CONSTANT INTEGER := ENCRYPT_DES + CHAIN_CBC + PAD_PKCS5}$;</td>
</tr>
<tr>
<td>DES3_CBC_PKCS5</td>
<td>$\text{CONSTANT INTEGER := ENCRYPT_3DES + CHAIN_CBC + PAD_PKCS5}$;</td>
</tr>
<tr>
<td>AES_CBC_PKCS5</td>
<td>$\text{CONSTANT INTEGER := ENCRYPT_AES + CHAIN_CBC + PAD_PKCS5}$;</td>
</tr>
</tbody>
</table>

The following example uses the `DBMS_CRYPTO.DECRYPT` function to decrypt an encrypted password retrieved from the `passwords` table:

```sql
CREATE TABLE passwords
(
  principal  VARCHAR2(90) PRIMARY KEY, -- username
ciphertext RAW(9) -- encrypted password
);
CREATE FUNCTION get_password(username VARCHAR2) RETURN RAW AS
typ INTEGER := DBMS_CRYPTO.DES_CBC_PKCS5;
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.
3.4.2 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:
### Block Cipher Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Constant Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCRYPT_DES</td>
<td>1</td>
</tr>
<tr>
<td>ENCRYPT_3DES</td>
<td>3</td>
</tr>
<tr>
<td>ENCRYPT_AES</td>
<td>4</td>
</tr>
<tr>
<td>ENCRYPT_AES128</td>
<td>6</td>
</tr>
</tbody>
</table>

### Block Cipher Modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Constant Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAIN_CBC</td>
<td>256</td>
</tr>
<tr>
<td>CHAIN_ECB</td>
<td>768</td>
</tr>
</tbody>
</table>

### Block Cipher Padding Modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Constant Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAD_PKCS5</td>
<td>4096</td>
</tr>
<tr>
<td>PAD_NONE</td>
<td>8192</td>
</tr>
</tbody>
</table>

### Block Cipher Suites

<table>
<thead>
<tr>
<th>Suite</th>
<th>Constant Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES_CBC_PKCS5</td>
<td>ENCRYPT_DES + CHAIN_CBC + PAD_PKCS5;</td>
</tr>
<tr>
<td>DES3_CBC_PKCS5</td>
<td>ENCRYPT_3DES + CHAIN_CBC + PAD_PKCS5;</td>
</tr>
<tr>
<td>AES_CBC_PKCS5</td>
<td>ENCRYPT_AES + CHAIN_CBC + PAD_PKCS5;</td>
</tr>
</tbody>
</table>

`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:

```sql
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`. 
3.4.3 HASH

The HASH function uses a user-specified algorithm to return the hash value of a RAW or CLOB value. The HASH function is available in three forms:

```sql
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:

<table>
<thead>
<tr>
<th>HASH Functions</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASH_MD4</td>
<td>CONSTANT INTEGER := 1;</td>
</tr>
<tr>
<td>HASH_MD5</td>
<td>CONSTANT INTEGER := 2;</td>
</tr>
<tr>
<td>HASH_SH1</td>
<td>CONSTANT INTEGER := 3;</td>
</tr>
</tbody>
</table>

Examples

The following example uses DBMS_CRYPTO.HASH to find the md5 hash value of the string *cleartext source*:

```sql
DECLARE
typ INTEGER := DBMS_CRYPTO.HASH_MD5;
hash_value RAW(100);
BEGIN
  hash_value := DBMS_CRYPTO.HASH('cleartext source', typ);
END;
```

3.4.4 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  

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.

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.

3.4.5 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

Parameters

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);
BEGIN
  result := DBMS_CRYPTO.RANDOMBYTES(1024);
END;
```

3.4.6 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;
```
3.4.7 RANDOMNUMBER

The `RANDOMNUMBER()` function returns a random `NUMBER` between 0 and 2,684,354,55. 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;
```
3.5 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. DBMS_JOB verifies that pgAgent is installed, but does not verify that the service is running.

The following table lists the supported DBMS_JOB procedures:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROKEN(job, broken [, next_date])</td>
<td>n/a</td>
<td>Specify that a given job is either broken or not broken.</td>
</tr>
<tr>
<td>CHANGE(job, what, next_date, interval, instance, force)</td>
<td>n/a</td>
<td>Change the job’s parameters.</td>
</tr>
<tr>
<td>INTERVAL(job, interval)</td>
<td>n/a</td>
<td>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.</td>
</tr>
<tr>
<td>NEXT_DATE(job, next_date)</td>
<td>n/a</td>
<td>Set the next date/time the job is to be run.</td>
</tr>
<tr>
<td>REMOVE(job)</td>
<td>n/a</td>
<td>Delete the job definition from the database.</td>
</tr>
<tr>
<td>RUN(job)</td>
<td>n/a</td>
<td>Forces execution of a job even if it is marked broken.</td>
</tr>
<tr>
<td>SUBMIT(job OUT, what [, next_date [, interval [, no_parse ]]]))</td>
<td>n/a</td>
<td>Creates a job and stores its definition in the database.</td>
</tr>
<tr>
<td>WHAT(job, what)</td>
<td>n/a</td>
<td>Change the stored procedure run by a job.</td>
</tr>
</tbody>
</table>

Advanced Server’s implementation of DBMS_JOB is a partial implementation when compared to Oracle’s version. Only those functions and procedures listed in the table above are supported.

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.
job is then executed. In this manner, the expression in \( \text{interval} \) is repeatedly re-evaluated prior to each job execution, supplying the \( \text{next\_date} \) date/time for the next execution.

The following examples use the following stored procedure, \text{job\_proc}, which simply inserts a timestamp into table, \text{jobrun}, containing a single VARCHAR2 column.

```sql
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;
```

### 3.5.1 BROKEN

The \text{BROKEN} procedure sets the state of a job to either broken or not broken. A broken job cannot be executed except by using the \text{RUN} procedure.

\text{BROKEN(\text{job} \text{BINARY\_INTEGER}, \text{broken} \text{BOOLEAN [, \text{next\_date} \text{DATE}]})}

**Parameters**

\text{job}

Identifier of the job to be set as broken or not broken.

\text{broken}

If set to \text{TRUE} the job’s state is set to broken. If set to \text{FALSE} the job’s state is set to not broken. Broken jobs cannot be run except by using the \text{RUN} procedure.

\text{next\_date}

Date/time when the job is to be run. The default is \text{SYSDATE}.

**Examples**

Set the state of a job with job identifier 104 to broken:

```sql
BEGIN
    DBMS_JOB.BROKEN(104, true);
END;
```
Change the state back to not broken:

```sql
BEGIN
    DBMS_JOB.BROKEN(104,false);
END;
```

### 3.5.2 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.

```sql
CHANGE(job BINARY_INTEGER what VARCHAR2, next_date DATE,
         interval VARCHAR2, instance BINARY_INTEGER, force BOOLEAN)
```

**Parameters**

`job`

Identifier of the job to modify.

`what`

Stored procedure name. Set this parameter to null if 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.

```sql
BEGIN
    DBMS_JOB.CHANGE(104, NULL, TO_DATE('13-DEC-07', 'DD-MON-YY'), NULL, NULL, NULL);
END;
```

3.5.3 INTERVAL

The `INTERVAL` procedure sets the frequency of how often a job is to be run.

```sql
INTERVAL(job BINARY_INTEGER, interval VARCHAR2)
```

Parameters

`job`

Identifier of the job to modify.

`interval`

Date function that when evaluated, provides the next date/time the job is to be run.

Examples

Change the job to run once a week:

```sql
BEGIN
    DBMS_JOB.INTERVAL(104,'SYSDATE + 7');
END;
```

3.5.4 NEXT_DATE

The `NEXT_DATE` procedure sets the date/time of when the job is to be run next.

```sql
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;
```

### 3.5.5 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)
```

**Parameters**

```
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;
```

### 3.5.6 RUN

The **RUN** procedure forces the job to be run, even if its state is broken.

```
RUN(job  BINARY_INTEGER)
```

**Parameters**
job

Identifier of the job to be run.

Examples

Force a job to be run.

```sql
BEGIN
    DBMS_JOB.RUN(104);
END;
```

3.5.7 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.

```sql
SUBMIT(job OUT BINARY_INTEGER, what VARCHAR2
    [, next_date DATE [, interval VARCHAR2 [, no_parse BOOLEAN ]]])
```

Parameters

job

Identifier assigned to the job.

what

Name of the stored procedure 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

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.

```sql
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:

```sql
SELECT * FROM jobrun;
```

<table>
<thead>
<tr>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_proc run at 2007-12-11 11:43:25</td>
</tr>
</tbody>
</table>

(1 row)

### 3.5.8 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;
3.6 DBMS_LOB

The DBMS_LOB package provides the capability to operate on large objects. The following table lists the supported functions and procedures:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPEND(dest_lob IN OUT, src_lob)</td>
<td>n/a</td>
<td>Appends one large object to another.</td>
</tr>
<tr>
<td>COMPARE(lob_1, lob_2 [, amount [, offset_1 [, offset_2 ]]])</td>
<td>INTEGER</td>
<td>Compares two large objects.</td>
</tr>
<tr>
<td>CONVERTOBLOB(dest_lob IN OUT, src_clob, amount, dest_offset IN OUT, src_offset IN OUT, blob_csid, lang_context IN OUT, warning OUT)</td>
<td>n/a</td>
<td>Converts character data to binary.</td>
</tr>
<tr>
<td>CONVERTTOCLOB(dest_lob IN OUT, src_blob, amount, dest_offset IN OUT, src_offset IN OUT, blob_csid, lang_context IN OUT, warning OUT)</td>
<td>n/a</td>
<td>Converts binary data to character.</td>
</tr>
<tr>
<td>COPY(dest_lob IN OUT, src_lob, amount [, dest_offset [, src_offset ]])</td>
<td>n/a</td>
<td>Copies one large object to another.</td>
</tr>
<tr>
<td>ERASE(lob_loc IN OUT, amount IN OUT [, offset ])</td>
<td>n/a</td>
<td>Erase a large object.</td>
</tr>
<tr>
<td>GET_STORAGE_LIMIT(lob_loc)</td>
<td>INTEGER</td>
<td>Get the storage limit for large objects.</td>
</tr>
<tr>
<td>GETLENGTH(lob_loc)</td>
<td>INTEGER</td>
<td>Get the length of the large object.</td>
</tr>
<tr>
<td>INSTR(lob_loc, pattern [, offset [, nth ]])</td>
<td>INTEGER</td>
<td>Get the position of the nth occurrence of a pattern in the large object starting at offset.</td>
</tr>
<tr>
<td>READ(lob_loc, amount IN OUT, offset, buffer OUT)</td>
<td>n/a</td>
<td>Read a large object.</td>
</tr>
<tr>
<td>SUBSTR(lob_loc [, amount [, offset ]])</td>
<td>RAW, VARCHAR2</td>
<td>Get part of a large object.</td>
</tr>
<tr>
<td>TRIM(lob_loc IN OUT, newlen)</td>
<td>n/a</td>
<td>Trim a large object to the specified length.</td>
</tr>
<tr>
<td>WRITE(lob_loc IN OUT, amount, offset, buffer)</td>
<td>n/a</td>
<td>Write data to a large object.</td>
</tr>
<tr>
<td>WRITEAPPEND(lob_loc IN OUT, amount, buffer)</td>
<td>n/a</td>
<td>Write data from the buffer to the end of a large object.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress_off</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>compress_on</td>
<td>INTEGER</td>
<td>1</td>
</tr>
</tbody>
</table>
### Public Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>deduplicate_off</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>deduplicate_on</td>
<td>INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>default_csid</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>default_lang_ctx</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>encrypt_off</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>encrypt_on</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>file_readonly</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>lobmaxsize</td>
<td>INTEGER</td>
<td>1073741823</td>
</tr>
<tr>
<td>lob_readonly</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>lob_readwrite</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>no_warning</td>
<td>INTEGER</td>
<td>0</td>
</tr>
<tr>
<td>opt_compress</td>
<td>INTEGER</td>
<td>1</td>
</tr>
<tr>
<td>opt_deduplicate</td>
<td>INTEGER</td>
<td>4</td>
</tr>
<tr>
<td>opt_encrypt</td>
<td>INTEGER</td>
<td>2</td>
</tr>
<tr>
<td>warn_inconvertible_char</td>
<td>INTEGER</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

#### 3.6.1 APPEND

The **APPEND** procedure provides the capability to append one large object to another. Both large objects must be of the same type.

```sql
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**.
3.6.2 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.

\[
\text{status INTEGER COMPARE(} \text{lob}_1 \{ \text{BLOB | CLOB} \}, \\
\quad \text{lob}_2 \{ \text{BLOB | CLOB} \} \\
\quad [, \text{amount INTEGER} [, \text{offset}_1 \text{ INTEGER} [, \text{offset}_2 \text{ INTEGER }]]])
\]

Parameters

\(\text{lob}_1\)

Large object locator of the first large object to be compared. Must be the same data type as \(\text{lob}_2\).

\(\text{lob}_2\)

Large object locator of the second large object to be compared. Must be the same data type as \(\text{lob}_1\).

\(\text{amount}\)

If the data type of the large objects is BLOB, then the comparison is made for \(\text{amount}\) bytes. If the data type of the large objects is CLOB, then the comparison is made for \(\text{amount}\) characters. The default is the maximum size of a large object.

\(\text{offset}_1\)

Position within the first large object to begin the comparison. The first byte/character is offset 1. The default is 1.

\(\text{offset}_2\)

Position within the second large object to begin the comparison. The first byte/character is offset 1. The default is 1.

\(\text{status}\)

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 \(\text{amount}\), \(\text{offset}_1\), or \(\text{offset}_2\) are less than zero.
3.6.3 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

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.

3.6.4 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

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.
3.6.5 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

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.
3.6.6 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.

3.6.7 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

Maximum allowable size of a large object in this database.
lob_loc

This parameter is ignored, but is included for compatibility.

3.6.8 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.

3.6.9 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 \textit{lob\_loc} to start search for \textit{pattern}. The first byte/character is position 1. The default is 1.

\textit{nth}

Search for \textit{pattern}, \textit{nth} number of times starting at the position given by \textit{offset}. The default is 1.

\textit{position}

Position within the large object where \textit{pattern} appears the \textit{nth} time specified by \textit{nth} starting from the position given by \textit{offset}.

\section*{3.6.10 \textbf{READ}}

The \textbf{READ} procedure provides the capability to read a portion of a large object into a buffer.

\texttt{READ(lob\_loc \{ BLOB | CLOB \}, amount IN OUT BINARY\_INTEGER,}
\texttt{ \quad offset INTEGER, buffer OUT \{ RAW \mid VARCHAR2 \})}

\textbf{Parameters}

\textit{lob\_loc}

Large object locator of the large object to be read.

\textit{amount IN}

Number of bytes/characters to read.

\textit{amount OUT}

Number of bytes/characters actually read. If there is no more data to be read, then \textit{amount} returns 0 and a \texttt{DATA\_NOT\_FOUND} exception is thrown.

\textit{offset}

Position to begin reading. The first byte/character is position 1.

\textit{buffer}
Variable to receive the large object. If \texttt{lob\_loc} is a BLOB, then \texttt{buffer} must be \texttt{RAW}. If \texttt{lob\_loc} is a CLOB, then \texttt{buffer} must be \texttt{VARCHAR2}.

### 3.6.11 \textbf{SUBSTR}

The \texttt{SUBSTR} function provides the capability to return a portion of a large object.

\begin{verbatim}
data \{ \texttt{RAW} | \texttt{VARCHAR2} \} \texttt{SUBSTR(lob\_loc} \{ \texttt{BLOB} | \texttt{CLOB} \}
\[, \texttt{amount} \texttt{INTEGER} [, \texttt{offset} \texttt{INTEGER}] \})
\end{verbatim}

**Parameters**

\textit{lob\_loc}

Large object locator of the large object to be read.

\textit{amount}

Number of bytes/characters to be returned. Default is 32,767.

\textit{offset}

Position within the large object to begin returning data. The first byte/character is position 1. The default is 1.

\textit{data}

Returned portion of the large object to be read. If \texttt{lob\_loc} is a BLOB, the return data type is \texttt{RAW}. If \texttt{lob\_loc} is a CLOB, the return data type is \texttt{VARCHAR2}.

### 3.6.12 \textbf{TRIM}

The \texttt{TRIM} procedure provides the capability to truncate a large object to the specified length.

\begin{verbatim}
\texttt{TRIM(lob\_loc} \texttt{IN OUT} \{ \texttt{BLOB} | \texttt{CLOB} \}, \texttt{newlen} \texttt{INTEGER})
\end{verbatim}

**Parameters**

\textit{lob\_loc}

Large object locator of the large object to be trimmed.

\textit{newlen}
3.6.13 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.

```sql
WRITE(lob_loc IN OUT { BLOB | CLOB },
       amount BINARY_INTEGER,
       offset INTEGER, buffer { RAW | VARCHAR2 })
```

**Parameters**

`lob_loc`

Large object locator of the large object to be written.

`amount`

The number of bytes/characters in `buffer` to be written to the large object.

`offset`

The offset in bytes/characters from the beginning of the large object (origin is 1) for the write operation to begin.

`buffer`

Contains data to 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.

3.6.14 WRITEAPPEND

The WRITEAPPEND procedure provides the capability to add data to the end of a large object.

```sql
WRITEAPPEND(lob_loc IN OUT { BLOB | CLOB },
             amount BINARY_INTEGER, buffer { RAW | VARCHAR2 })
```

**Parameters**

`lob_loc`

Number of bytes/characters to which the large object is to be trimmed.
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.
3.7 DBMS_LOCK

Advanced Server provides support for the DBMS_LOCK.SLEEP procedure.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLEEP(seconds)</td>
<td>n/a</td>
<td>Suspends a session for the specified number of seconds.</td>
</tr>
</tbody>
</table>

Advanced Server's implementation of DBMS_LOCK is a partial implementation when compared to Oracle's version. Only DBMS_LOCK.SLEEP is supported.

3.7.1 SLEEP

The SLEEP procedure suspends the current session for the specified number of seconds.

SLEEP(seconds NUMBER)

Parameters

seconds

seconds specifies the number of seconds for which you 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.
3.8 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:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_MV_DEPENDENCIES(list VARCHAR2, deplist VARCHAR2);</td>
<td>n/a</td>
<td>The GET_MV_DEPENDENCIES procedure returns a list of dependencies for a specified view.</td>
</tr>
<tr>
<td>REFRESH(list VARCHAR2, method VARCHAR2, rollback_seg VARCHAR2, push_deferred_rpc BOOLEAN, refresh_after_errors BOOLEAN, purge_option NUMBER, parallelism NUMBER, heap_size NUMBER, atomic_refresh BOOLEAN, nested BOOLEAN);</td>
<td>n/a</td>
<td>This variation of the REFRESH procedure refreshes all views named in a comma-separated list of view names.</td>
</tr>
<tr>
<td>REFRESH(tab dbms_utility.uncl_array, method VARCHAR2, rollback_seg VARCHAR2, push_deferred_rpc BOOLEAN, refresh_after_errors BOOLEAN, purge_option NUMBER, parallelism NUMBER, heap_size NUMBER, atomic_refresh BOOLEAN, nested BOOLEAN);</td>
<td>n/a</td>
<td>This variation of the REFRESH procedure refreshes all views named in a table of dbms_utility.uncl_array values.</td>
</tr>
<tr>
<td>REFRESH_ALL_MVIEWS(number_of_failures BINARY_INTEGER, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN);</td>
<td>n/a</td>
<td>The REFRESH_ALL_MVIEWS procedure refreshes all materialized views.</td>
</tr>
<tr>
<td>REFRESH_DEPENDENT(number_of_failures BINARY_INTEGER, list VARCHAR2, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN, nested BOOLEAN);</td>
<td>n/a</td>
<td>This variation of the REFRESH_DEPENDENT procedure refreshes all views that are dependent on the views listed in a comma-separated list.</td>
</tr>
<tr>
<td>REFRESH_DEPENDENT(number_of_failures BINARY_INTEGER, tab dbms_utility.uncl_array, method VARCHAR2, rollback_seg VARCHAR2, refresh_after_errors BOOLEAN, atomic_refresh BOOLEAN, nested BOOLEAN);</td>
<td>n/a</td>
<td>This variation of the REFRESH_DEPENDENT procedure refreshes all views that are dependent on the views listed in a table of dbms_utility.uncl_array values.</td>
</tr>
</tbody>
</table>

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.
3.8.1 GET_MV_DEPENDENCIES

When given the name of a materialized view, GET_MV_DEPENDENCIES returns a list of items that depend on the specified view. The signature is:

```
GET_MV_DEPENDENCIES(
    list IN VARCHAR2,
    deplist OUT VARCHAR2);
```

Parameters

`list`

`list` specifies the name of a materialized view, or a comma-separated list of materialized view names.

`deplist`

`deplist` is a comma-separated list of schema-qualified dependencies. `deplist` is a VARCHAR2 value.

Examples

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`.

3.8.2 REFRESH

Use the REFRESH procedure to refresh all views specified in either a comma-separated list of view names, or a table of DBMSUTILITY.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,
```

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.
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,
    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

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');

3.8.3 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:

```sql
DECLARE
  errors INTEGER;
BEGIN
  DBMS_MVIEW.REFRESH_ALL_MVIEWS(errors, method => 'C');
END;
```

Upon completion, errors contains the number of failures.
3.8.4 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 DBMSUTILITY.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 DBMSUTILITY.UNCL_ARRAY values:

```
REFRESH_DEPENDENT(
    number_of_failures OUT BINARY_INTEGER,
    tab IN DBMSUTILITY.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 DBMSUTILITY.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

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:

```sql
DECLARE
    errors INTEGER;
BEGIN
    DBMS_MVIEW.REFRESH_DEPENDENT(errors, list => 'public.emp_view', method => 'C');
END;
```

Upon completion, errors contains the number of failures.
### 3.9 **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.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISABLE</td>
<td>n/a</td>
<td>Disable the capability to send and receive messages.</td>
</tr>
<tr>
<td>ENABLE(buffer_size)</td>
<td>n/a</td>
<td>Enable the capability to send and receive messages.</td>
</tr>
<tr>
<td>GET_LINE(line OUT, status OUT)</td>
<td>n/a</td>
<td>Get a line from the message buffer.</td>
</tr>
<tr>
<td>GET_LINES(lines OUT, numlines IN OUT)</td>
<td>n/a</td>
<td>Get multiple lines from the message buffer.</td>
</tr>
<tr>
<td>NEW_LINE</td>
<td>n/a</td>
<td>Puts an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT(item)</td>
<td>n/a</td>
<td>Puts a partial line without an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT_LINE(item)</td>
<td>n/a</td>
<td>Puts a complete line with an end-of-line character sequence.</td>
</tr>
<tr>
<td>SERVEROUTPUT(stdout)</td>
<td>n/a</td>
<td>Direct messages from PUT, PUT_LINE, or NEW_LINE to either standard output or the message buffer.</td>
</tr>
</tbody>
</table>

The following table lists the public variables available in the `DBMS_OUTPUT` package.

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chararr</td>
<td>TABLE</td>
<td></td>
<td>For message lines.</td>
</tr>
</tbody>
</table>

### 3.9.1 **CHARARR**

The CHARARR is for storing multiple message lines.

```sql
TYPE chararr IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

### 3.9.2 **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.

```sql
BEGIN
    DBMS_OUTPUT.DISABLE;
END;
```

3.9.3 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) ]
```

Parameters

`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.

Examples
The following anonymous block enables messages. Setting `SERVEROUTPUT(TRUE)` forces them to standard output.

```sql
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Messages enabled');
END;
Messages enabled
```

The same effect could have been achieved by simply using `SERVEROUTPUT(TRUE)`.

```sql
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.

```sql
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    DBMS_OUTPUT.PUT_LINE('Message sent to buffer');
END;
```

### 3.9.4 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.

```sql
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.

```sql
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;
```

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.

```sql
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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20</td>
<td>7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30</td>
<td>7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30</td>
<td>7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10</td>
<td>7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7839, KING, PRESIDENT, 17-NOV-81 00:00:00,5000.00,,10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30</td>
<td>7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7902, FORD, ANALYST, 7566, 03-DEC-81 00:00:00,3000.00,,20</td>
<td>7934, MILLER, CLERK, 7782, 23-JAN-82 00:00:00,1300.00,,10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(14 rows)
```
3.9.5 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 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.CHARRR;
  v_numlines      INTEGER := 14;
  v_status        INTEGER := 0;
BEGIN
```
3.9.6 NEW_LINE

The NEW_LINE procedure writes an end-of-line character sequence in the message buffer.

NEW_LINE

Parameters

The NEW_LINE procedure expects no parameters.

3.9.7 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)

Parameters

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(',');
    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;
```

3.9.8 **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)
```

**Parameters**

- **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.

```sql
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;
```

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>First Name</th>
<th>Title</th>
<th>Manager ID</th>
<th>Hire Date</th>
<th>Salary</th>
<th>Commission</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>Smith</td>
<td>Clerk</td>
<td></td>
<td>17-Dec-80</td>
<td>800.00</td>
<td>0.00</td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>Allen</td>
<td>Salesman</td>
<td></td>
<td>20-Feb-81</td>
<td>1600.00</td>
<td>300.00</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>Ward</td>
<td>Salesman</td>
<td></td>
<td>22-Feb-81</td>
<td>1250.00</td>
<td>500.00</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>Jones</td>
<td>Manager</td>
<td></td>
<td>02-Apr-81</td>
<td>2975.00</td>
<td>0.00</td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>Martin</td>
<td>Salesman</td>
<td></td>
<td>28-Sep-81</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>Blake</td>
<td>Manager</td>
<td></td>
<td>01-May-81</td>
<td>2850.00</td>
<td>0.00</td>
<td>30</td>
</tr>
<tr>
<td>7728</td>
<td>Clark</td>
<td>Manager</td>
<td></td>
<td>09-Jun-81</td>
<td>2450.00</td>
<td>0.00</td>
<td>10</td>
</tr>
<tr>
<td>7788</td>
<td>Scott</td>
<td>Analyst</td>
<td></td>
<td>19-Apr-87</td>
<td>3000.00</td>
<td>0.00</td>
<td>20</td>
</tr>
<tr>
<td>7839</td>
<td>King</td>
<td>President</td>
<td></td>
<td>17-Nov-81</td>
<td>5000.00</td>
<td>0.00</td>
<td>10</td>
</tr>
<tr>
<td>7844</td>
<td>Turner</td>
<td>Salesman</td>
<td></td>
<td>08-Sep-81</td>
<td>1500.00</td>
<td>0.00</td>
<td>30</td>
</tr>
<tr>
<td>7976</td>
<td>Adams</td>
<td>Clerk</td>
<td></td>
<td>23-May-87</td>
<td>1100.00</td>
<td>0.00</td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>James</td>
<td>Clerk</td>
<td></td>
<td>03-Dec-81</td>
<td>950.00</td>
<td>0.00</td>
<td>30</td>
</tr>
<tr>
<td>7902</td>
<td>Ford</td>
<td>Analyst</td>
<td></td>
<td>03-Dec-81</td>
<td>3000.00</td>
<td>0.00</td>
<td>20</td>
</tr>
<tr>
<td>7934</td>
<td>Miller</td>
<td>Clerk</td>
<td></td>
<td>23-Jan-82</td>
<td>1300.00</td>
<td>0.00</td>
<td>10</td>
</tr>
</tbody>
</table>

3.9.9 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)
```

Parameters

`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;
```

This message goes to the message buffer
Flush messages from the buffer
3.10 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:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_PIPE(pipename [, maxpipesize ] [, private ])</td>
<td>INTEGER</td>
<td>Explicitly create a private pipe if private is “true” (the default) or a public pipe if private is “false”.</td>
</tr>
<tr>
<td>NEXT_ITEM_TYPE</td>
<td>INTEGER</td>
<td>Determine the data type of the next item in a received message.</td>
</tr>
<tr>
<td>PACK_MESSAGE(item)</td>
<td>n/a</td>
<td>Place item in the session’s local message buffer.</td>
</tr>
<tr>
<td>PURGE(pipename)</td>
<td>n/a</td>
<td>Remove unreceived messages from the specified pipe.</td>
</tr>
<tr>
<td>RECEIVE_MESSAGE(pipename [, timeout ])</td>
<td>INTEGER</td>
<td>Get a message from a specified pipe.</td>
</tr>
<tr>
<td>REMOVE_PIPE(pipename)</td>
<td>INTEGER</td>
<td>Delete an explicitly created pipe.</td>
</tr>
<tr>
<td>RESET_BUFFER</td>
<td>n/a</td>
<td>Reset the local message buffer.</td>
</tr>
<tr>
<td>SEND_MESSAGE(pipename [, timeout ] [, maxpipesize ])</td>
<td>INTEGER</td>
<td>Send a message on a pipe.</td>
</tr>
<tr>
<td>UNIQUE_SESSION_NAME</td>
<td>VARCHAR2</td>
<td>Obtain a unique session name.</td>
</tr>
<tr>
<td>UNPACK_MESSAGE(item OUT)</td>
<td>n/a</td>
<td>Retrieve the next data item from a message into a type-compatible variable, item.</td>
</tr>
</tbody>
</table>

Pipes are categorized as implicit or explicit. An implicit pipe is created if a reference is made to a pipe name that 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.

3.10.1 CREATE_PIPE

The CREATE Pipe function creates an explicit public pipe or an explicit private pipe with a specified name.

\[
status \text{ INTEGER CREATE_PIPE}(\text{pipename}\ \text{VARCHAR2} \\
\quad [, \ \text{maxpipesize} \ \text{INTEGER} ] [, \ \text{private} \ \text{BOOLEAN} ])
\]

Parameters

\text{pipename}

Name of the pipe.

\text{maxpipesize}

Maximum capacity of the pipe in bytes. Default is 8192 bytes.

\text{private}

Create a public pipe if set to FALSE. Create a private pipe if set to TRUE. This is the default.

\text{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,TRUE);
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END;
CREATE_PIPE status: 0
```

3.10.2 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 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 Table 7-3-1.

<table>
<thead>
<tr>
<th>Type Code</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No more data items</td>
</tr>
<tr>
<td>9</td>
<td>NUMBER</td>
</tr>
<tr>
<td>11</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>13</td>
<td>DATE</td>
</tr>
<tr>
<td>23</td>
<td>RAW</td>
</tr>
</tbody>
</table>

**Note:** The type codes 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.

```sql
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
  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);
```

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.
3.10.3 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.

```
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.

3.10.4 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:

```sql
DECLARE
  v_status        INTEGER;
BEGIN
  DBMS_PIPE.PACK_MESSAGE('Message #1');
  v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
  DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);

  DBMS_PIPE.PACK_MESSAGE('Message #2');
  v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
  DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END;
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

Receive the first message and unpack it:

```sql
DECLARE
  v_item          VARCHAR2(80);
  v_status        INTEGER;
BEGIN
  v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  DBMS_PIPE.UNPACK_MESSAGE(v_item);
  DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END;
RECEIVE_MESSAGE status: 0
Item: Message #1
```

Purge the pipe:

```sql
EXEC DBMS_PIPE.PURGE('pipe');
```

Try to retrieve the next message. The **RECEIVE_MESSAGE** call returns status code 1 indicating it timed out because no message was available.

```sql
DECLARE
  v_item          VARCHAR2(80);
  v_status        INTEGER;
```
BEGIN
    v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
    DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END;
RECEIVE_MESSAGE status: 1

3.10.5 RECEIVE_MESSAGE

The RECEIVE_MESSAGE function obtains a message from a specified pipe.

```sql
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:

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>1</td>
<td>Time out</td>
</tr>
<tr>
<td>2</td>
<td>Message too large for the buffer</td>
</tr>
</tbody>
</table>

3.10.6 REMOVE_PIPE

The REMOVE_PIPE function deletes an explicit private or explicit public pipe.

```sql
status INTEGER REMOVE_PIPE(pipename VARCHAR2)
```

Use the REMOVE_PIPE function to delete explicitly created pipes – i.e., pipes created with the CREATE_PIPE function.
Parameters

pipe

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
```

Remove the pipe:

```
SELECT DBMS_PIPE.REMOVE_PIPE('pipe') FROM DUAL;
```
Try to retrieve the next message. The RECEIVE_MESSAGE call returns status code 1 indicating it timed out because the pipe 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;
```

**3.10.7  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.

**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;
```

The message to Bob is in the received message.

```
DECLARE
  v_item          VARCHAR2(80);
  v_status        INTEGER;
BEGIN
  v_status := DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
```
3.10.8  SEND_MESSAGE

The SEND_MESSAGE function sends a message from the session’s local message buffer to the specified pipe.

```
status SEND_MESSAGE(pipename VARCHAR2 [, timeout INTEGER ] [, maxpipesize INTEGER ])
```

**Parameters**

* pipename
  
  Name of the pipe.

* timeout
  
  Wait time (seconds). Default is 86400000 (1000 days).

* maxpipesize
  
  Maximum capacity of the pipe in bytes. Default is 8192 bytes.

* status
  
  Status code returned by the operation.

The possible status codes are:

**Table 7-3-3 SEND_MESSAGE Status Codes**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Success</td>
</tr>
<tr>
<td>1</td>
<td>Time out</td>
</tr>
<tr>
<td>3</td>
<td>Function interrupted</td>
</tr>
</tbody>
</table>
### 3.10.9 UNIQUE_SESSION_NAME

The **UNIQUE_SESSION_NAME** function returns a name, unique to the current session.

```plaintext
name VARCHAR2 UNIQUE_SESSION_NAME
```

#### Parameters

**name**

Unique session name.

#### Examples

The following anonymous block retrieves and displays a unique session name.

```sql
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

### 3.10.10 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**.

```sql
UNPACK_MESSAGE(item OUT { DATE | NUMBER | VARCHAR2 | RAW })
```

#### Parameters

**item**

Type-compatible variable that receives a data item from the local message buffer.
3.10.11 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.

```sql
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,
                            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
```
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?
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
3.11 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.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLUSH_DATA</td>
<td>Status Code or Exception</td>
<td>Flushes performance data collected in the current session without terminating the session (profiling continues).</td>
</tr>
<tr>
<td>GET_VERSION(major OUT, minor OUT)</td>
<td>n/a</td>
<td>Returns the version number of this package.</td>
</tr>
<tr>
<td>INTERNAL_VERSION_CHECK</td>
<td>Status Code or Exception</td>
<td>Confirms that the current version of the profiler will work with the current database.</td>
</tr>
<tr>
<td>PAUSE_PROFILER</td>
<td>Status Code or Exception</td>
<td>Pause data collection.</td>
</tr>
<tr>
<td>RESUME_PROFILER</td>
<td>Status Code or Exception</td>
<td>Resume data collection.</td>
</tr>
<tr>
<td>START_PROFILER(run_comment, run_comment1 [, run_number OUT])</td>
<td>Status Code or Exception</td>
<td>Start data collection.</td>
</tr>
<tr>
<td>STOP_PROFILER</td>
<td>Status Code or Exception</td>
<td>Stop data collection and flush performance data to the PLSQL_PROFILER_RAWDATA table.</td>
</tr>
</tbody>
</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.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Message</th>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>error version</td>
<td>version_mismatch</td>
<td>The profiler version and the database are incompatible.</td>
</tr>
<tr>
<td>0</td>
<td>success</td>
<td>n/a</td>
<td>The operation completed successfully.</td>
</tr>
<tr>
<td>1</td>
<td>error_param</td>
<td>profiler_error</td>
<td>The operation received an incorrect parameter.</td>
</tr>
<tr>
<td>2</td>
<td>error_io</td>
<td>profiler_error</td>
<td>The data flush operation has failed.</td>
</tr>
</tbody>
</table>

3.11.1 FLUSH_DATA

The FLUSH_DATA function/procedure flushes the data collected in the current session without terminating the profiler session. The data is flushed to the tables described in the Advanced Server Performance Features Guide. The function and procedure signatures are:

```sql
status INTEGER FLUSH_DATA

FLUSH_DATA
```
Parameters

\textit{status}

Status code returned by the operation.

\subsection{3.11.2 \texttt{GET\_VERSION}}

The \texttt{GET\_VERSION} procedure returns the version of \texttt{DBMS\_PROFILER}. The procedure signature is:

\texttt{GET\_VERSION(major OUT INTEGER, minor OUT INTEGER)}

Parameters

\textit{major}

The major version number of \texttt{DBMS\_PROFILER}.

\textit{minor}

The minor version number of \texttt{DBMS\_PROFILER}.

\subsection{3.11.3 \texttt{INTERNAL\_VERSION\_CHECK}}

The \texttt{INTERNAL\_VERSION\_CHECK} function confirms that the current version of \texttt{DBMS\_PROFILER} will work with the current database. The function signature is:

\texttt{status INTEGER INTERNAL\_VERSION\_CHECK}

Parameters

\textit{status}

Status code returned by the operation.
3.11.4 PAUSE_PROFILER

The PAUSE_PROFILER function/procedure pauses a profiling session. The function and procedure signatures are:

\[ \text{status INTEGER PAUSE_PROFILER} \]

\[ \text{PAUSE_PROFILER} \]

Parameters

status

Status code returned by the operation.

3.11.5 RESUME_PROFILER

The RESUME_PROFILER function/procedure pauses a profiling session. The function and procedure signatures are:

\[ \text{status INTEGER RESUME_PROFILER} \]

\[ \text{RESUME_PROFILER} \]

Parameters

status

Status code returned by the operation.

3.11.6 START_PROFILER

The START_PROFILER function/procedure starts a data collection session. The function and procedure signatures are:

\[ \text{status INTEGER START_PROFILER(run_comment TEXT := SYSDATE, run_comment1 TEXT := '' [ , run_number OUT INTEGER ] )} \]

\[ \text{START_PROFILER(run_comment TEXT := SYSDATE, run_comment1 TEXT := '' [ , run_number OUT INTEGER ] )} \]

Parameters

run_comment
A user-defined comment for the profiler session. The default value is \texttt{SYSDATE}.

\texttt{run_comment1}

An additional user-defined comment for the profiler session. The default value is ''.

\texttt{run_number}

The session number of the profiler session.

\texttt{status}

Status code returned by the operation.

3.11.7 \textbf{STOP_PROFILER}

The \texttt{STOP_PROFILER} function/procedure stops a profiling session and flushes the performance information to the \texttt{DBMS_PROFILER} tables and view. The function and procedure signatures are:

\begin{verbatim}
status INTEGER STOP_PROFILER

STOP_PROFILER
\end{verbatim}

\textbf{Parameters}

\texttt{status}

Status code returned by the operation.
### 3.11.8 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:

```sql
1 - CREATE OR REPLACE FUNCTION getBalance(acctNumber INTEGER)
2 - RETURNS NUMERIC AS $$
3 - DECLARE
4 -   result NUMERIC;
5 - BEGIN
6 -   SELECT INTO result balance FROM acct WHERE id = acctNumber;
7 -   IF (result IS NULL) THEN
8 -     RAISE INFO 'Balance is null';
9 -   END IF;
10- RETURN result;
11- END;
12- $$ 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`).
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.
3.11.8.1 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
   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');
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;
   ```

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 | 4154 | enterprise | 04-FEB-14 09:32:48.874315 | profile list_emp
   | | | | |
   2 | 2088 | enterprise | 04-FEB-14 09:41:30.546503 | profile get_dept_name and
   | | | | emp_sal_trig
   (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 | enterprise | list_emp()
   | | | | |
   2 | 17004 | FUNCTION | enterprise | emp_sal_trig()
   | | | | |
   ```
10. Query the `plsql_profiler_rawdata` table to view a list of the wait event counters and wait event times:

```sql
acctg=# SELECT runid, sourcecode, func_oid, line_number, exec_count, tuples_returned, time_total FROM plsql_profiler_rawdata;
```

<table>
<thead>
<tr>
<th>runid</th>
<th>sourcecode</th>
<th>func_oid</th>
<th>line_number</th>
<th>exec_count</th>
<th>tuples_returned</th>
<th>time_total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>DECLARE</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>v_empno</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>v_ename</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>emp_cur CURSOR FOR</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SELECT empno, ename FROM memp ORDER BY empno;</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>BEGIN</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>OPEN emp_cur;</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>RAISE INFO 'EMPNO ENAME';</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>RAISE INFO '-----' '-----';</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LOOP</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>FETCH emp_cur INTO v_empno, v_ename;</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>EXIT WHEN NOT FOUND;</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>RAISE INFO '% %', v_empno, v_ename;</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>END LOOP;</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>CLOSE emp_cur;</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>RETURN;</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>END;</td>
<td>16999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
| Line | Code | Time | Cost
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>END IF;</td>
<td>17002</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>v_text := 'User '</td>
<td></td>
<td>USER</td>
</tr>
<tr>
<td>12</td>
<td>RAISE INFO ' %', v_text;</td>
<td>17002</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>RETURN NULL;</td>
<td>17002</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>END;</td>
<td>17002</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>DECLARE</td>
<td>17000</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>v_dname VARCHAR(1);</td>
<td>17000</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>BEGIN</td>
<td>17000</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>SELECT INTO v_dname dname FROM dept WHERE deptno = p_deptno;</td>
<td>17000</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>RETURN v_dname;</td>
<td>17000</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>IF NOT FOUND THEN</td>
<td>17000</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>RAISE INFO 'Invalid department number %', p_deptno;</td>
<td>17000</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>RETURN '';</td>
<td>17000</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>END IF;</td>
<td>17000</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>BEGIN</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>IF TG_OP = 'INSERT' THEN</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>RAISE INFO 'Inserting employee %', NEW.empno;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>RAISE INFO '..New salary: %', NEW.sal;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>RAISE INFO '..Raise     : %', sal_diff;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>RETURN NEW;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>END IF;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>IF TG_OP = 'UPDATE' THEN</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>sal_diff := NEW.sal - OLD.sal;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>RAISE INFO 'Updating employee %', OLD.empno;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>RAISE INFO '..Old salary: %', OLD.sal;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>35</td>
<td>RAISE INFO '..New salary: %', NEW.sal;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>RAISE INFO '..Raise     : %', sal_diff;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>RETURN NEW;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>38</td>
<td>END IF;</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>39</td>
<td>IF TG_OP = 'DELETE' THEN</td>
<td>17004</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>RAISE INFO 'Deleting employee %', OLD.empno;</td>
<td>17004</td>
<td>0</td>
</tr>
</tbody>
</table>
11. Query the `plsql_profiler_data` view to review a subset of the information found in `plsql_profiler_rawdata` table:
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>17002</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>9</td>
<td>1</td>
<td>3.2e-05</td>
<td>3.2e-05</td>
<td>3.2e-05</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>13</td>
<td>1</td>
<td>0.000383</td>
<td>0.000383</td>
<td>0.000383</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>14</td>
<td>1</td>
<td>6.3e-05</td>
<td>6.3e-05</td>
<td>6.3e-05</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>15</td>
<td>1</td>
<td>3.6e-05</td>
<td>3.6e-05</td>
<td>3.6e-05</td>
</tr>
<tr>
<td></td>
<td>17002</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>5</td>
<td>1</td>
<td>0.000647</td>
<td>0.000647</td>
<td>0.000647</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>6</td>
<td>1</td>
<td>2.6e-05</td>
<td>2.6e-05</td>
<td>2.6e-05</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>5</td>
<td>1</td>
<td>8.4e-05</td>
<td>8.4e-05</td>
<td>8.4e-05</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>10</td>
<td>1</td>
<td>0.000355</td>
<td>0.000355</td>
<td>0.000355</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>11</td>
<td>1</td>
<td>0.000177</td>
<td>0.000177</td>
<td>0.000177</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>12</td>
<td>1</td>
<td>5.5e-05</td>
<td>5.5e-05</td>
<td>5.5e-05</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>13</td>
<td>1</td>
<td>3.1e-05</td>
<td>3.1e-05</td>
<td>3.1e-05</td>
</tr>
<tr>
<td></td>
<td>17004</td>
<td>14</td>
<td>1</td>
<td>2.8e-05</td>
<td>2.8e-05</td>
<td>2.8e-05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>------</td>
<td>----</td>
<td>---</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>2</td>
<td>17004</td>
<td>15</td>
<td>1</td>
<td>2.7e-05</td>
<td>2.7e-05</td>
<td>2.7e-05</td>
</tr>
<tr>
<td>2</td>
<td>17004</td>
<td>16</td>
<td>1</td>
<td>1e-06</td>
<td>1e-06</td>
<td>1e-06</td>
</tr>
<tr>
<td>2</td>
<td>17004</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>17004</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>17004</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>17004</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>17004</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>17004</td>
<td>22</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>17004</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(68 rows)
3.11.8.2 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>
<thead>
<tr>
<th>Table Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLSQL_PROFILER_RUNS</td>
<td>Table containing information about all profiler runs, organized by runid.</td>
</tr>
<tr>
<td>PLSQL_PROFILER_UNITS</td>
<td>Table containing information about all profiler runs, organized by unit.</td>
</tr>
<tr>
<td>PLSQL_PROFILER_DATA</td>
<td>View containing performance statistics.</td>
</tr>
<tr>
<td>PLSQL_PROFILER_RAWDATA</td>
<td>Table containing the performance statistics and the extended performance statistics for DRITA counters and timers.</td>
</tr>
</tbody>
</table>

### 3.11.8.2.1 PLSQL_PROFILER_RUNS

The PLSQL_PROFILER_RUNS table contains the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runid</td>
<td>INTEGER (NOT NULL)</td>
<td>Unique identifier (plsql_profiler_runnumber)</td>
</tr>
<tr>
<td>related_run</td>
<td>INTEGER</td>
<td>The runid of a related run.</td>
</tr>
<tr>
<td>run_owner</td>
<td>TEXT</td>
<td>The role that recorded the profiling session.</td>
</tr>
<tr>
<td>run_date</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>The profiling session start time.</td>
</tr>
<tr>
<td>run_comment</td>
<td>TEXT</td>
<td>User comments relevant to this run</td>
</tr>
<tr>
<td>run_total_time</td>
<td>BIGINT</td>
<td>Run time (in microseconds)</td>
</tr>
<tr>
<td>run_system_info</td>
<td>TEXT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>run_comment1</td>
<td>TEXT</td>
<td>Additional user comments</td>
</tr>
<tr>
<td>spare1</td>
<td>TEXT</td>
<td>Currently Unused</td>
</tr>
</tbody>
</table>

### 3.11.8.2.2 PLSQL_PROFILER_UNITS

The PLSQL_PROFILER_UNITS table contains the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runid</td>
<td>INTEGER</td>
<td>Unique identifier (plsql_profiler_runnumber)</td>
</tr>
<tr>
<td>unit_number</td>
<td>OID</td>
<td>Corresponds to the OID of the row in the pg_proc table that identifies the unit.</td>
</tr>
<tr>
<td>unit_type</td>
<td>TEXT</td>
<td>PL/SQL function, procedure, trigger or anonymous block</td>
</tr>
<tr>
<td>unit_owner</td>
<td>TEXT</td>
<td>The identity of the role that owns the unit.</td>
</tr>
<tr>
<td>unit_name</td>
<td>TEXT</td>
<td>The complete signature of the unit.</td>
</tr>
<tr>
<td>unit_timestamp</td>
<td>TIMESTAMP WITHOUT TIME ZONE</td>
<td>Creation date of the unit (currently NULL).</td>
</tr>
</tbody>
</table>
### 3.11.8.2.3 PLSQL_PROFILER_DATA

The `PLSQL_PROFILER_DATA` view contains the following columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runid</td>
<td>INTEGER</td>
<td>Unique identifier (<code>plsql_profiler_runnumber</code>).</td>
</tr>
<tr>
<td>unit_number</td>
<td>OID</td>
<td>Object ID of the unit that contains the current line.</td>
</tr>
<tr>
<td>line_number</td>
<td>INTEGER</td>
<td>Current line number of the profiled workload.</td>
</tr>
<tr>
<td>total_occur</td>
<td>BIGINT</td>
<td>The number of times that the line was executed.</td>
</tr>
<tr>
<td>total_time</td>
<td>DOUBLE PRECISION</td>
<td>The amount of time spent executing the line (in seconds)</td>
</tr>
<tr>
<td>min_time</td>
<td>DOUBLE PRECISION</td>
<td>The minimum execution time for the line.</td>
</tr>
<tr>
<td>max_time</td>
<td>DOUBLE PRECISION</td>
<td>The maximum execution time for the line.</td>
</tr>
<tr>
<td>spare1</td>
<td>NUMBER</td>
<td>Currently Unused.</td>
</tr>
<tr>
<td>spare2</td>
<td>NUMBER</td>
<td>Currently Unused.</td>
</tr>
<tr>
<td>spare3</td>
<td>NUMBER</td>
<td>Currently Unused.</td>
</tr>
<tr>
<td>spare4</td>
<td>NUMBER</td>
<td>Currently Unused.</td>
</tr>
</tbody>
</table>

### 3.11.8.2.4 PLSQL_PROFILER_RAWDATA

The `PLSQL_PROFILER_RAWDATA` table contains the statistical information that is found in the `PLSQL_PROFILER_DATA` view, as well as the performance statistics returned by the DRITA counters and timers.

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runid</td>
<td>INTEGER</td>
<td>The run identifier (<code>plsql_profiler_runnumber</code>).</td>
</tr>
<tr>
<td>sourcecode</td>
<td>TEXT</td>
<td>The individual line of profiled code.</td>
</tr>
<tr>
<td>func_oid</td>
<td>OID</td>
<td>Object ID of the unit that contains the current line.</td>
</tr>
<tr>
<td>line_number</td>
<td>INTEGER</td>
<td>Current line number of the profiled workload.</td>
</tr>
<tr>
<td>exec_count</td>
<td>BIGINT</td>
<td>The number of times that the line was executed.</td>
</tr>
<tr>
<td>time_total</td>
<td>DOUBLE PRECISION</td>
<td>The amount of time spent executing the line (in seconds)</td>
</tr>
<tr>
<td>time_shortest</td>
<td>DOUBLE PRECISION</td>
<td>The minimum execution time for the line.</td>
</tr>
<tr>
<td>time_longest</td>
<td>DOUBLE PRECISION</td>
<td>The maximum execution time for the line.</td>
</tr>
<tr>
<td>tuples_returned</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>num_scans</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>tuples_fetched</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>tuples_inserted</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>tuples_updated</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>tuples_deleted</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>Column</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>blocks_fetched</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>blocks_hit</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>wal_write</td>
<td>BIGINT</td>
<td>The server has waited for a write to the write-ahead log buffer (expect this value to be high).</td>
</tr>
<tr>
<td>wal_flush</td>
<td>BIGINT</td>
<td>The server has waited for the write-ahead log to flush to disk.</td>
</tr>
<tr>
<td>wal_file_sync</td>
<td>BIGINT</td>
<td>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).</td>
</tr>
<tr>
<td>buffer_free_list_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory).</td>
</tr>
<tr>
<td>shmem_index_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the shared-memory map.</td>
</tr>
<tr>
<td>oid_gen_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available OID (object ID).</td>
</tr>
<tr>
<td>xid_gen_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available transaction ID.</td>
</tr>
<tr>
<td>proc_array_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the process array.</td>
</tr>
<tr>
<td>sinval_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the cache invalidation state.</td>
</tr>
<tr>
<td>freespace_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the freespace map.</td>
</tr>
<tr>
<td>wal_insert_lock_acquire</td>
<td>BIGINT</td>
<td>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.</td>
</tr>
<tr>
<td>wal_write_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes write-ahead log flushes.</td>
</tr>
<tr>
<td>control_file_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number).</td>
</tr>
<tr>
<td>checkpoint_lock_acquire</td>
<td>BIGINT</td>
<td>A server process has waited for the short-term lock that prevents simultaneous checkpoints.</td>
</tr>
<tr>
<td>clog_control_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the commit log.</td>
</tr>
<tr>
<td>subtrans_control_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the subtransaction log.</td>
</tr>
<tr>
<td>multi_xact_gen_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes).</td>
</tr>
<tr>
<td>multi_xact_offset_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes).</td>
</tr>
<tr>
<td>multi_xact_member_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes).</td>
</tr>
<tr>
<td>Column</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>rel_cache_init_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.</td>
</tr>
<tr>
<td>bgwriter_communication_lock_acquire</td>
<td>BIGINT</td>
<td>The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.</td>
</tr>
<tr>
<td>two_phase_state_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the list of prepared transactions.</td>
</tr>
<tr>
<td>tablespace_create_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands.</td>
</tr>
<tr>
<td>btree_vacuum_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.</td>
</tr>
<tr>
<td>add_in_shmem_lock_acquire</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>autovacuum_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the shared autovacuum state.</td>
</tr>
<tr>
<td>autovacuum_schedule_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the autovacuum schedule.</td>
</tr>
<tr>
<td>syncscan_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that coordinates synchronous scans.</td>
</tr>
<tr>
<td>icache_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to InfiniteCache state.</td>
</tr>
<tr>
<td>breakpoint_lock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for the short-term lock that synchronizes access to the debugger breakpoint list.</td>
</tr>
<tr>
<td>lwlock_acquire</td>
<td>BIGINT</td>
<td>The server has waited for a short-term lock that has not been described elsewhere in this section.</td>
</tr>
<tr>
<td>db_file_read</td>
<td>BIGINT</td>
<td>A server process has waited for the completion of a read (from disk).</td>
</tr>
<tr>
<td>db_file_write</td>
<td>BIGINT</td>
<td>A server process has waited for the completion of a write (to disk).</td>
</tr>
<tr>
<td>db_file_sync</td>
<td>BIGINT</td>
<td>A server process has waited for the operating system to flush all changes to disk.</td>
</tr>
<tr>
<td>db_file_extend</td>
<td>BIGINT</td>
<td>A server process has waited for the operating system while adding a new page to the end of a file.</td>
</tr>
<tr>
<td>sql_parse</td>
<td>BIGINT</td>
<td>Currently Unused</td>
</tr>
<tr>
<td>query_plan</td>
<td>BIGINT</td>
<td>The server has generated a query plan.</td>
</tr>
<tr>
<td>infinitecache_read</td>
<td>BIGINT</td>
<td>The server has waited for an Infinite Cache read request.</td>
</tr>
<tr>
<td>infinitecache_write</td>
<td>BIGINT</td>
<td>The server has waited for an Infinite Cache write request.</td>
</tr>
<tr>
<td>wal_write_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a write to the write-ahead log buffer (expect this value to be high).</td>
</tr>
<tr>
<td>wal_flush_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the write-ahead log to flush to disk.</td>
</tr>
<tr>
<td>wal_file_sync_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for 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_async).</td>
</tr>
<tr>
<td>buffer_free_list_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the list of</td>
</tr>
<tr>
<td>Column</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>shmem_index_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the shared-memory map.</td>
</tr>
<tr>
<td>oid_gen_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the next available OID (object ID).</td>
</tr>
<tr>
<td>xid_gen_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the next available transaction ID.</td>
</tr>
<tr>
<td>proc_array_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the process array.</td>
</tr>
<tr>
<td>sinval_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the cache invalidation state.</td>
</tr>
<tr>
<td>freespace_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the freespace map.</td>
</tr>
<tr>
<td>wal_insert_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the write-ahead log. A high number may indicate that WAL buffers are sized too small.</td>
</tr>
<tr>
<td>wal_write_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes write-ahead log flushes.</td>
</tr>
<tr>
<td>control_file_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the control file (this should usually be a low number).</td>
</tr>
<tr>
<td>checkpoint_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server process has waited for the short-term lock that prevents simultaneous checkpoints.</td>
</tr>
<tr>
<td>clog_control_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the commit log.</td>
</tr>
<tr>
<td>subtrans_control_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the subtransaction log.</td>
</tr>
<tr>
<td>multi_xact_gen_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes).</td>
</tr>
<tr>
<td>multi_xact_offset_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes).</td>
</tr>
<tr>
<td>multi_xact_member_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes).</td>
</tr>
<tr>
<td>rel_cache_init_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.</td>
</tr>
<tr>
<td>bgwriter_communication_lock</td>
<td>BIGINT</td>
<td>The amount of time that the bgwriter (background</td>
</tr>
<tr>
<td>Column</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>k_acquire_time</td>
<td></td>
<td>writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.</td>
</tr>
<tr>
<td>two_phase_state_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the list of prepared transactions.</td>
</tr>
<tr>
<td>tablespace_create_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands.</td>
</tr>
<tr>
<td>btree_vacuum_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.</td>
</tr>
<tr>
<td>add_in_shmem_lock_acquire_time</td>
<td>BIGINT</td>
<td>Obsolete/unused.</td>
</tr>
<tr>
<td>autovacuum_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the shared autovacuum state.</td>
</tr>
<tr>
<td>autovacuum_schedule_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the autovacuum schedule.</td>
</tr>
<tr>
<td>syncscan_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that coordinates synchronous scans.</td>
</tr>
<tr>
<td>icache_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to InfiniteCache state.</td>
</tr>
<tr>
<td>breakpoint_lock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for the short-term lock that synchronizes access to the debugger breakpoint list.</td>
</tr>
<tr>
<td>lwlock_acquire_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for a short-term lock that has not been described elsewhere in this section.</td>
</tr>
<tr>
<td>db_file_read_time</td>
<td>BIGINT</td>
<td>The amount of time that the server process has waited for the completion of a read (from disk).</td>
</tr>
<tr>
<td>db_file_write_time</td>
<td>BIGINT</td>
<td>The amount of time that the server process has waited for the completion of a write (to disk).</td>
</tr>
<tr>
<td>db_file_sync_time</td>
<td>BIGINT</td>
<td>The amount of time that the server process has waited for the operating system to flush all changes to disk.</td>
</tr>
<tr>
<td>db_file_extend_time</td>
<td>BIGINT</td>
<td>The amount of time that the server process has waited for the operating system while adding a new page to the end of a file.</td>
</tr>
<tr>
<td>sql_parse_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has parsed a SQL statement.</td>
</tr>
<tr>
<td>query_plan_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has computed the execution plan for a SQL statement.</td>
</tr>
<tr>
<td>infinitecache_read_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for an Infinite Cache read request.</td>
</tr>
<tr>
<td>infinitecache_write_time</td>
<td>BIGINT</td>
<td>The amount of time that the server has waited for an Infinite Cache write request.</td>
</tr>
<tr>
<td>totalwaits</td>
<td>BIGINT</td>
<td>The total number of event waits.</td>
</tr>
<tr>
<td>Totalwaittime</td>
<td>BIGINT</td>
<td>The total time spent waiting for an event.</td>
</tr>
</tbody>
</table>
### 3.12 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.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALIZE(val)</td>
<td>n/a</td>
<td>Initializes the DBMS_RANDOM package with the specified seed value. Deprecated, but supported for backward compatibility.</td>
</tr>
<tr>
<td>NORMAL()</td>
<td>NUMBER</td>
<td>Returns a random NUMBER.</td>
</tr>
<tr>
<td>RANDOM</td>
<td>INTEGER</td>
<td>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.</td>
</tr>
<tr>
<td>SEED(val)</td>
<td>n/a</td>
<td>Resets the seed with the specified value.</td>
</tr>
<tr>
<td>SEED(val)</td>
<td>n/a</td>
<td>Resets the seed with the specified value.</td>
</tr>
<tr>
<td>STRING(opt, len)</td>
<td>VARCHAR2</td>
<td>Returns a random string.</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>n/a</td>
<td>TERMINATE has no effect. Deprecated, but supported for backward compatibility.</td>
</tr>
<tr>
<td>VALUE</td>
<td>NUMBER</td>
<td>Returns a random number with a value greater than or equal to 0 and less than 1, with 38 digit precision.</td>
</tr>
<tr>
<td>VALUE(low, high)</td>
<td>NUMBER</td>
<td>Returns a random number with a value greater than or equal to low and less than high.</td>
</tr>
</tbody>
</table>

#### 3.12.1 INITIALIZE

The INITIALIZE procedure initializes the DBMS_RANDOM package with a seed value. The signature is:

```sql
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.

```
DBMS_RANDOM.INITIALIZE(6475);
```

### 3.12.2 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();
```

### 3.12.3 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:

```sql
x := DBMS_RANDOM.RANDOM();
```

3.12.4 SEED

The first form of the `SEED` procedure resets the seed value for the `DBMS_RANDOM` package with an `INTEGER` value. The `SEED` procedure is available in two forms; the signature of the first form is:

```sql
SEED(val IN INTEGER)
```

Parameters

`val`

`val` is the seed value used by the `DBMS_RANDOM` package algorithm.

Example

The following code snippet demonstrates a call to the `SEED` procedure; the call sets the seed value at 8495.

```sql
DBMS_RANDOM.SEED(8495);
```

3.12.5 SEED

The second form of the `SEED` procedure resets the seed value for the `DBMS_RANDOM` package with a string value. The `SEED` procedure is available in two forms; the signature of the second form is:

```sql
SEED(val IN VARCHAR2)
```

Parameters

`val`

`val` is the seed value used by the `DBMS_RANDOM` package algorithm.
Example

The following code snippet demonstrates a call to the `SEED` procedure; the call sets the seed value to `abc123`.

```sql
DBMS_RANDOM.SEED('abc123');
```

### 3.12.6 STRING

The `STRING` function returns a random `VARCHAR2` string in a user-specified format. The signature of the `STRING` function is:

```sql
result VARCHAR2 STRING(opt IN CHAR, len IN NUMBER)
```

#### Parameters

- **opt**
  
  Formatting option for the returned string. `option` may be:

<table>
<thead>
<tr>
<th>Option</th>
<th>Specifies Formatting Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>u or U</td>
<td>Uppercase alpha string</td>
</tr>
<tr>
<td>l or L</td>
<td>Lowercase alpha string</td>
</tr>
<tr>
<td>a or A</td>
<td>Mixed case string</td>
</tr>
<tr>
<td>x or X</td>
<td>Uppercase alpha-numeric string</td>
</tr>
<tr>
<td>p or P</td>
<td>Any printable characters</td>
</tr>
</tbody>
</table>

- **len**
  
  The length of the returned string.

- **result**
  
  `result` is a random value of type `VARCHAR2`.

Example

The following code snippet demonstrates a call to the `STRING` function; the call returns a random alpha-numeric character string that is 10 characters long.

```sql
x := DBMS_RANDOM.STRING('X', 10);
```

### 3.12.7 TERMINATE

The `TERMINATE` procedure has no effect. The signature is:
3.12.8 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:

\[
\text{result NUMBER VALUE()}
\]

Parameters

\(\text{result}\)

\(\text{result}\) 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:

\[
x := \text{DBMSRANDOM.VALUE();}
\]

3.12.9 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:

\[
\text{result NUMBER VALUE(low IN NUMBER, high IN NUMBER)}
\]

Parameters

\(\text{low}\)

\(\text{low}\) specifies the lower boundary for the random value. The random value may be equal to \(\text{low}\).

\(\text{high}\)
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:

```sql
x := DBMS_RANDOM.VALUE(1, 100);
```
3.13 DBMS_RLS

The DBMS_RLS package enables the implementation of Virtual Private Database on certain Advanced Server database objects.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_POLICY(object_schema, object_name, policy_name, function_schema, policy_function, statement_types, update_check, enable, static_policy, policy_type, long_predicate, sec_relevant_cols, sec_relevant_cols_opt)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Add a security policy to a database object.</td>
</tr>
<tr>
<td>DROP_POLICY(object_schema, object_name, policy_name)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Remove a security policy from a database object.</td>
</tr>
<tr>
<td>ENABLE_POLICY(object_schema, object_name, policy_name, enable)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Enable or disable a security policy.</td>
</tr>
</tbody>
</table>

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.

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 Reference Guide*, available at the EnterpriseDB website at:

http://www.enterprisedb.com/products-services-training/products/documentation

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:

<table>
<thead>
<tr>
<th>attribute Value</th>
<th>Equivalent Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESSION_USER</td>
<td>pg_catalog.session_user</td>
</tr>
<tr>
<td>CURRENT_USER</td>
<td>pg_catalog.current_user</td>
</tr>
<tr>
<td>CURRENT_SCHEMA</td>
<td>pg_catalog.current_schema</td>
</tr>
<tr>
<td>HOST</td>
<td>pg_catalog.inet_host</td>
</tr>
<tr>
<td>IP_ADDRESS</td>
<td>pg_catalog.inet_client_addr</td>
</tr>
<tr>
<td>SERVER_HOST</td>
<td>pg_catalog.inet_server_addr</td>
</tr>
</tbody>
</table>

**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:

```sql
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;
```

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
<th>authid</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>2450.00</td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>5000.00</td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>1300.00</td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>800.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>2975.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>1100.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>3000.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>1600.00</td>
<td>300.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>1250.00</td>
<td>500.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>2850.00</td>
<td></td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>1500.00</td>
<td>0.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
</tbody>
</table>
3.13.1 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, function_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.

`function_schema`

Name of the schema containing the policy function.
Note: The policy function may belong to a package in which case
\textit{function\_schema} must contain the name of the schema in which the package is
defined.

\textit{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
\textit{policy\_function} must also contain the package name in dot notation (that is,\
\textit{package\_name.function\_name}).

\textit{statement\_types}

Comma-separated list of SQL commands to which the policy applies. Valid SQL
commands are \texttt{INSERT, UPDATE, DELETE, and SELECT}. The default is
\texttt{INSERT, UPDATE, DELETE, SELECT}.

Note: Advanced Server accepts \texttt{INDEX} as a statement type, but it is ignored.
Policies are not applied to index operations in Advanced Server.

\textit{update\_check}

Applies to \texttt{INSERT} and \texttt{UPDATE} SQL commands only.

When set to \texttt{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 \texttt{INSERT} or \texttt{UPDATE}
command throws an exception and no rows are inserted or modified by the
\texttt{INSERT} or \texttt{UPDATE} command.

When set to \texttt{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 \texttt{UPDATE}
command may not appear in the result set of a subsequent SQL command that
invokes the same policy.

The default is \texttt{FALSE}.

\textit{enable}

When set to \texttt{TRUE}, the policy is enabled and applied to the SQL commands given
by the \textit{statement\_types} parameter. When set to \texttt{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 **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.

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.

```sql
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:

```sql
edb=# /c edb salesmgr
Password for user salesmgr:
You are now connected to database "edb" as user "salesmgr".
eedb=> SELECT * FROM vpemp;
<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>sal</th>
<th>comm</th>
<th>deptno</th>
<th>authid</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>2450.00</td>
<td></td>
<td>10</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>5000.00</td>
<td></td>
<td>10</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>1300.00</td>
<td></td>
<td>10</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>800.00</td>
<td></td>
<td>10</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>2975.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>1100.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>3000.00</td>
<td></td>
<td>20</td>
<td>researchmgr</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>1600.00</td>
<td>300.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>1250.00</td>
<td>500.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>1250.00</td>
<td>1400.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>2850.00</td>
<td></td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>1500.00</td>
<td>0.00</td>
<td>30</td>
<td>salesmgr</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>950.00</td>
<td></td>
<td>30</td>
<td>salesmgr</td>
</tr>
</tbody>
</table>
```

(14 rows)
An unqualified UPDATE command (no WHERE clause) is issued by the salesmgr user:

\[
\text{edb=> UPDATE vpemp SET comm = sal * .75;}
\]

Instead of updating all rows in the table, the policy restricts the effect of the update to only those rows where the \text{authid} column contains the value \text{salesmgr} as specified by the policy function predicate \text{authid = SYS_CONTEXT('USERENV', 'SESSION_USER')}.

The following query shows that the \text{comm} column has been changed only for those rows where \text{authid} contains \text{salesmgr}. All other rows are unchanged.

\[
\text{edb=> SELECT * FROM vpemp;}
\]

Furthermore, since the \text{update_check} parameter was set to \text{TRUE} in the \text{ADD_POLICY} procedure, the following INSERT command throws an exception since the value given for the \text{authid} column, \text{researchmgr}, does not match the session user, which is \text{salesmgr}, and hence, fails the policy.

If \text{update_check} was set to \text{FALSE}, the preceding INSERT command would have succeeded.

The following example illustrates the use of the \text{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.

\[
\text{CREATE OR REPLACE FUNCTION sal_lt_2000 (}
\]

\[
\text{p_schema VARCHAR2,}
\]

\[
\text{p_job VARCHAR2,}
\]

\[
\text{p_sal NUMBER,}
\]

\[
\text{RETURN BOOLEAN;}
\]

\[
\text{BEGIN}
\]

\[
\text{RETURN } \text{SYS_CONTEXT('USERENV', 'SESSION_USER')} = \text{p_schema}.
\]

\[
\text{END;}
\]

\[
\text{END;}
\]

\[
\text{FILE 'policies/oracle全球最大安全的数据库.xml';}
\]

\[
\text{ADD_POLICY sal_lt_2000, DELETE, \text{sold_price}}
\]

\[
\text{sec_relevant_cols='p_job,p_sal'};
\]

\[
\text{ON \text{vpemp}}
\]

\[
\text{WITH sec_relevant_cols}
\]

\[
\text{UPDATE \text{vpemp} SET \text{sold_price} = \text{p_sal} * .95;}
\]

\[
\text{INSERT INTO vpemp VALUES (9001,'SMITH','ANALYST',3200.00,NULL,20,'researchmgr');}
\]

\[
\text{ERROR: policy with check option violation}
\]

\[
\text{DETAIL: Policy predicate was evaluated to FALSE with the updated values}
\]

\[
\text{If update_check was set to FALSE, the preceding INSERT command would have succeeded.}
\]

\[
\text{The following example illustrates the use of the sec_relevant_cols parameter to apply a policy only when certain columns are referenced in the SQL command. The following policy function is used for this example, which selects rows where the employee salary is less than 2000.}
\]

\[
\text{CREATE OR REPLACE FUNCTION sal lt 2000 (}
\]

\[
\text{p_schema VARCHAR2,}
\]

\[
\text{p_job VARCHAR2,}
\]

\[
\text{p_sal NUMBER,}
\]

\[
\text{RETURN BOOLEAN;}
\]

\[
\text{BEGIN}
\]

\[
\text{RETURN } \text{SYS_CONTEXT('USERENV', 'SESSION_USER')} = \text{p_schema}.
\]

\[
\text{END;}
\]

\[
\text{END;}
\]

\[
\text{FILE 'policies/oracle全球最大安全的数据库.xml';}
\]

\[
\text{ADD_POLICY sal lt 2000, DELETE, \text{sold_price}}
\]

\[
\text{sec_relevant_cols='p_job,p_sal'};
\]

\[
\text{ON \text{vpemp}}
\]

\[
\text{WITH sec_relevant_cols}
\]

\[
\text{UPDATE \text{vpemp} SET \text{sold_price} = \text{p_sal} * .95;}
\]

\[
\text{INSERT INTO vpemp VALUES (9001,'SMITH','ANALYST',3200.00,NULL,20,'researchmgr');}
\]

\[
\text{ERROR: policy with check option violation}
\]

\[
\text{DETAIL: Policy predicate was evaluated to FALSE with the updated values}
\]

\[
\text{If update_check was set to FALSE, the preceding INSERT command would have succeeded.}
\]

\[
\text{The following example illustrates the use of the sec_relevant_cols parameter to apply a policy only when certain columns are referenced in the SQL command. The following policy function is used for this example, which selects rows where the employee salary is less than 2000.}
\]
The policy is created so that it is enforced only if a SELECT command includes columns sal or comm:

```sql
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:

```sql
edb=# SELECT empno, ename, job, deptno, authid FROM vpemp;
```

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:

```sql
edb=# SELECT empno, ename, job, sal, comm, deptno, authid FROM vpemp;
```
### 3.13.2 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;
```

3.13.3  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**

- **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;
END;
```
);  
END;

3.14 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:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_JOB(job_name, job_type, job_action, number_of_arguments, start_date, repeat_interval, end_date, job_class, enabled, auto_drop, comments)</td>
<td>n/a</td>
<td>Use the first form of the CREATE_JOB procedure to create a job, specifying program and schedule details by means of parameters.</td>
</tr>
<tr>
<td>CREATE_JOB(job_name, program_name, schedule_name, job_class, enabled, auto_drop, comments)</td>
<td>n/a</td>
<td>Use the second form of CREATE_JOB to create a job that uses a named program and named schedule.</td>
</tr>
<tr>
<td>CREATE_PROGRAM(program_name, program_type, program_action, number_of_arguments, enabled, comments)</td>
<td>n/a</td>
<td>Use CREATE_PROGRAM to create a program.</td>
</tr>
<tr>
<td>CREATE_SCHEDULE(schedule_name, start_date, repeat_interval, end_date, comments)</td>
<td>n/a</td>
<td>Use the CREATE_SCHEDULE procedure to create a schedule.</td>
</tr>
<tr>
<td>DEFINE_PROGRAM_ARGUMENT(program_name, argument_position, argument_name, argument_type, default_value, out_argument)</td>
<td>n/a</td>
<td>Use the first form of the DEFINE_PROGRAM_ARGUMENT procedure to define a program argument that has a default value.</td>
</tr>
<tr>
<td>DEFINE_PROGRAM_ARGUMENT(program_name, argument_position, argument_name, argument_type, out_argument)</td>
<td>n/a</td>
<td>Use the first form of the DEFINE_PROGRAM_ARGUMENT procedure to define a program argument that does not have a default value.</td>
</tr>
<tr>
<td>DISABLE(name, force, commit_semantics)</td>
<td>n/a</td>
<td>Use the DISABLE procedure to disable a job or program.</td>
</tr>
<tr>
<td>DROP_JOB(job_name, force, defer, commit_semantics)</td>
<td>n/a</td>
<td>Use the DROP_JOB procedure to drop a job.</td>
</tr>
<tr>
<td>DROP_PROGRAM(program_name, force)</td>
<td>n/a</td>
<td>Use the DROP_PROGRAM procedure to drop a program.</td>
</tr>
<tr>
<td>DROP_PROGRAM_ARGUMENT(program_name, argument_position)</td>
<td>n/a</td>
<td>Use the first form of DROP_PROGRAM_ARGUMENT to drop a program argument by specifying the argument position.</td>
</tr>
<tr>
<td>DROP_PROGRAM_ARGUMENT(program_name, argument_name)</td>
<td>n/a</td>
<td>Use the second form of DROP_PROGRAM_ARGUMENT to drop a program argument by specifying the argument name.</td>
</tr>
<tr>
<td>DROP_SCHEDULE(schedule_name, force)</td>
<td>n/a</td>
<td>Use the DROP_SCHEDULE procedure to drop a schedule.</td>
</tr>
<tr>
<td>ENABLE(name, commit_semantics)</td>
<td>n/a</td>
<td>Use the ENABLE command to enable a program or job.</td>
</tr>
<tr>
<td>EVALUATECALENDAR_STRING()</td>
<td>n/a</td>
<td>Use EVALUATECALENDAR_STRING to review the</td>
</tr>
<tr>
<td>Function/Procedure</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>calendar_string, start_date, return_date_after, next_run_date</td>
<td></td>
<td>execution date described by a user-defined calendar schedule.</td>
</tr>
<tr>
<td>RUN_JOB(job_name, use_current_session, manually)</td>
<td>n/a</td>
<td>Use the RUN_JOB procedure to execute a job immediately.</td>
</tr>
<tr>
<td>SET_JOB_ARGUMENT_VALUE(job_name, argument_position, argument_value)</td>
<td>n/a</td>
<td>Use the first form of SET_JOB_ARGUMENT value to set the value of a job argument described by the argument's position.</td>
</tr>
<tr>
<td>SET_JOB_ARGUMENT_VALUE(job_name, argument_name, argument_value)</td>
<td>n/a</td>
<td>Use the second form of SET_JOB_ARGUMENT value to set the value of a job argument described by the argument's name.</td>
</tr>
</tbody>
</table>

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.

### 3.14.1 Using Calendar Syntax to Specify a Repeating Interval

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:

\[
\text{FREQ=\text{DAILY}; BYDAY=\text{MON,TUE,WED,THU,FRI}; BYHOUR=17; BYMINUTE=45}
\]

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:

<table>
<thead>
<tr>
<th>Token type</th>
<th>Syntax</th>
<th>Valid Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREQ</td>
<td>FREQ=predefined_interval</td>
<td>Where predefined_interval is one of the following: YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY. The SECONDLY keyword is not supported.</td>
</tr>
<tr>
<td>BYMONTH</td>
<td>BYMONTH=month(, month)...</td>
<td>Where month is the three-letter abbreviation of the month name: JAN</td>
</tr>
<tr>
<td>BYMONTH</td>
<td>BYMONTH=month(, month)...</td>
<td>Where month is the numeric value representing the month: 1</td>
</tr>
<tr>
<td>Bymonthday</td>
<td>Bymonthday=day_of_month</td>
<td>Where day_of_month is a value from 1 through 31</td>
</tr>
<tr>
<td>BYDAY</td>
<td>BYDAY=weekday</td>
<td>Where weekday is a three-letter abbreviation or single-digit value representing the day of the week.</td>
</tr>
<tr>
<td>BYDATE</td>
<td>BYDATE=date(, date)...</td>
<td>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.</td>
</tr>
<tr>
<td>BYDATE</td>
<td>BYDATE=date(, date)...</td>
<td>Where date is MMDD. MM is a two-digit representation of the month, and DD is a two-digit day representation of the day</td>
</tr>
<tr>
<td>BYHOUR</td>
<td>BYHOUR=hour</td>
<td>Where hour is a value from 0 through 23.</td>
</tr>
<tr>
<td>BYMINUTE</td>
<td>BYMINUTE=minute</td>
<td>Where minute is a value from 0 through 59.</td>
</tr>
</tbody>
</table>
3.14.2 CREATE_JOB

Use the CREATE_JOB procedure to create a job. The procedure comes in two forms; the first form of the procedure specifies a schedule 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`

`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:

```sql
EXEC DBMS_SCHEDULER.CREATE_JOB (  
    job_name => 'update_log',  
    job_type => 'PLSQL_BLOCK',  
    job_action => 'BEGIN INSERT INTO my_log VALUES(current_timestamp);  
                       END;',  
    start_date => '01-JUN-15 09:00:00.000000',  
    repeat_interval => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',  
    end_date => NULL,  
    enabled => TRUE,  
    comments => 'This job adds a row to the my_log table.');
```

The code fragment creates a job named update_log that executes each weeknight at 5:00. The job executes a PL/SQL block that inserts the current timestamp into a logfile (my_log). Since no end_date is specified, the job will execute until it is disabled by the DBMS_SCHEDULER.DISABLE procedure.
3.14.3 CREATE_PROGRAM

Use the CREATE_PROGRAM procedure to create a DBMS_SCHEDULER program. The signature is:

```sql
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

 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,
   comment          => 'This program adds a row to the my_log table.');

 update_log is a PL/SQL block that adds a row containing the current date and time to
the my_log table. The program will be enabled when the CREATE_PROGRAM procedure
executes.
3.14.4 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` value that specifies the date and time that the schedule is eligible to execute. If `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`.

For information about defining a repeating schedule for a job, see Section 3.14.1.

Please 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` 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`.

Please note that if a `repeat_interval` is specified and an `end_date` is not specified, the schedule will repeat indefinitely until it is disabled.
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:

```sql
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.
3.14.5  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:

```sql
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:

```sql
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.

*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:

```sql
EXEC DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
    program_name         => 'add_emp',
    argument_position    => 1,
    argument_name        => 'dept_no',
    argument_type        => 'INTEGER',
    default_value        => '20');
EXEC DBMS_SCHEDULER.DEFINE_PROGRAM_ARGUMENT(
    program_name         => 'add_emp',
    argument_position    => 2,
    argument_name        => 'emp_name',
    argument_type        => 'VARCHAR2');
```

The first argument is an INTEGER value named dept_no that has a default value of 20. The second argument is a VARCHAR2 value named emp_name; the second argument does not have a default value.
3.14.6  DISABLE

Use the DISABLE procedure to disable a program or a job. The signature of the DISABLE procedure is:

```sql
DISABLE(
    name IN VARCHAR2,
    force IN BOOLEAN DEFAULT FALSE,
    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

`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`:

```sql
DBMS_SCHEDULER.DISABLE('update_emp');
```
### 3.14.7 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:

```sql
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`:

```sql
DBMS_SCHEDULER.DROP_JOB('update_log');
```
3.14.8 DROP_PROGRAM

The DROP_PROGRAM procedure

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');
```
3.14.9 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**

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');
```
3.14.10 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 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);
```

The server will disable any jobs that use the schedule before dropping the schedule.

3.14.11 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:

```sql
DBMS_SCHEDULER.ENABLE('update_emp');
```

3.14.12 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:

```sql
EVALUATE_CALENDAR_STRING(
    calendar_string IN VARCHAR2,
    start_date IN TIMESTAMP WITH TIME ZONE,
    return_date_after IN TIMESTAMP WITH TIME ZONE,
    next_run_date OUT TIMESTAMP WITH TIME ZONE)
```

Parameters

calendar_string

calendar_string is the calendar string that describes a repeat_interval (see Section 3.14.1 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:

```sql
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.
3.14.13 RUN_JOB

Use the RUN_JOB procedure to execute a job immediately. The signature of the RUN_JOB procedure is:

```sql
RUN_JOB(
    job_name IN VARCHAR2,
    use_current_session IN BOOLEAN DEFAULT TRUE
)
```

**Parameters**

*job_name*

*job_name* specifies the name of the job 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:

```sql
DBMS_SCHEDULER.RUN_JOB('update_log', TRUE);
```

Passing a value of TRUE as the second argument instructs the server to invoke the job in the current session.

3.14.14 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:

```sql
SET_JOB_ARGUMENT_VALUE(
    job_name IN VARCHAR2,
    argument_position IN PLS_INTEGER,
    argument_value IN VARCHAR2)
```

The second form uses an argument name to specify which argument to modify:

```sql
SET_JOB_ARGUMENT_VALUE(
    job_name IN VARCHAR2,
```
argument_name IN VARCHAR2,
argument_value IN VARCHAR2)

Argument values set by the SET_JOB_ARGUMENT_VALUE procedure override any values set by default.

Parameters

job_name

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.

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');
```
3.15 DBMS_SESSION

Advanced Server provides support for the following `DBMS_SESSION.SET_ROLE` procedure:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SET_ROLE(role_cmd)</code></td>
<td>n/a</td>
<td>Executes a <code>SET ROLE</code> statement followed by the string value specified in <code>role_cmd</code>.</td>
</tr>
</tbody>
</table>

Advanced Server's implementation of `DBMS_SESSION` is a partial implementation when compared to Oracle's version. Only `DBMS_SESSION.SET_ROLE` is supported.

3.15.1 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');
```
3.16 DBMS_SQL

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.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIND_VARIABLE(c, name, value [, out_value_size ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Bind a value to a variable.</td>
</tr>
<tr>
<td>BIND_VARIABLE_CHAR(c, name, value [, out_value_size ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Bind a CHAR value to a variable.</td>
</tr>
<tr>
<td>BIND_VARIABLE_RAW(c, name, value [, out_value_size ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Bind a RAW value to a variable.</td>
</tr>
<tr>
<td>CLOSE_CURSOR(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Close a cursor.</td>
</tr>
<tr>
<td>COLUMN_VALUE(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Return a column value into a variable.</td>
</tr>
<tr>
<td>COLUMN_VALUE_CHAR(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Return a CHAR column value into a variable.</td>
</tr>
<tr>
<td>COLUMN_VALUE_RAW(c, position, value OUT [, column_error OUT [, actual_length OUT ]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Return a RAW column value into a variable.</td>
</tr>
<tr>
<td>DEFINE_COLUMN(c, position, column [, column_size ])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define a column in the SELECT list.</td>
</tr>
<tr>
<td>DEFINE_COLUMN_CHAR(c, position, column, column_size)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define a CHAR column in the SELECT list.</td>
</tr>
<tr>
<td>DEFINE_COLUMN_RAW(c, position, column, column_size)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Define a RAW column in the SELECT list.</td>
</tr>
<tr>
<td>DESCRIBE_COLUMNS</td>
<td>Procedure</td>
<td>n/a</td>
<td>Defines columns to hold a cursor result set.</td>
</tr>
<tr>
<td>EXECUTE(c)</td>
<td>Function</td>
<td>INTEGER</td>
<td>Execute a cursor.</td>
</tr>
<tr>
<td>EXECUTE_AND_FETCH(c [, exact ])</td>
<td>Function</td>
<td>INTEGER</td>
<td>Execute a cursor and fetch a single row.</td>
</tr>
<tr>
<td>FETCH_ROWS(c)</td>
<td>Function</td>
<td>INTEGER</td>
<td>Fetch rows from the cursor.</td>
</tr>
<tr>
<td>IS_OPEN(c)</td>
<td>Function</td>
<td>BOOLEAN</td>
<td>Check if a cursor is open.</td>
</tr>
<tr>
<td>LAST_ROW_COUNT</td>
<td>Function</td>
<td>INTEGER</td>
<td>Return cumulative number of rows fetched.</td>
</tr>
<tr>
<td>OPEN_CURSOR</td>
<td>Function</td>
<td>INTEGER</td>
<td>Open a cursor.</td>
</tr>
<tr>
<td>PARSE(c, statement, language_flag)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Parse a statement.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>native</td>
<td>INTEGER</td>
<td>1</td>
<td>Provided for compatibility with Oracle syntax. See <code>DBMS_SQL.PARSE</code> for more information.</td>
</tr>
<tr>
<td>V6</td>
<td>INTEGER</td>
<td>2</td>
<td>Provided for compatibility with Oracle syntax. See <code>DBMS_SQL.PARSE</code> for more information.</td>
</tr>
<tr>
<td>V7</td>
<td>INTEGER</td>
<td>3</td>
<td>Provided for compatibility with Oracle syntax. See <code>DBMS_SQL.PARSE</code> for more information.</td>
</tr>
</tbody>
</table>

### 3.16.1 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.

```sql
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.

```sql
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;
    v_ename    := 'JONES';
    v_job      := 'SALESMAN';
    v_mgr      := 7369;
    v_hiredate := TO_DATE('13-DEC-07','DD-MON-YY');
    v_sal      := 8500.00;
    v_comm     := 1500.00;
    v_deptno   := 40;
    DBMS_SQL.BIND_VARIABLE(curid,':p_empno',v_empno);
    DBMS_SQL.BIND_VARIABLE(curid,':p_ename',v_ename);
    DBMS_SQL.BIND_VARIABLE(curid,':p_job',v_job);
    DBMS_SQL.BIND_VARIABLE(curid,':p_mgr',v_mgr);
    DBMS_SQL.BIND_VARIABLE(curid,':p_hiredate',v_hiredate);
    DBMS_SQL.BIND_VARIABLE(curid,':p_sal',v_sal);
    DBMS_SQL.BIND_VARIABLE(curid,':p_comm',v_comm);
    DBMS_SQL.BIND_VARIABLE(curid,':p_deptno',v_deptno);
    v_status := DBMS_SQL.EXECUTE(curid);
    DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
END;
```

Number of rows processed: 1
3.16.2 **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.

```sql
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.

3.16.3 **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.

```sql
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.
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.

3.16.4  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;

3.16.5  COLUMN_VALUE

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,5,v_comm);
        DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,10) || '  ' ||
                             TO_CHAR(v_hiredate,'yyyy-mm-dd') || '   ' ||   
```
3.16.6 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.

3.16.7 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
Cursors id of the cursor returning data to the variable being defined.

Position within the cursor of the returned data. The first value in the cursor is position 1.

Variable of data type RAW receiving the data returned in the cursor by a prior fetch call.

Error number associated with the column, if any.

Actual length of the data prior to any truncation.

### 3.16.8 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.

```sql
DEFINE_COLUMN(c INTEGER, position INTEGER, column { BLOB | CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMPT | 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);
  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;

3.16.9 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

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.
3.16.10  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.

3.16.11  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);
```

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:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>col type</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col max len</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col name</td>
<td>VARCHAR2(128)</td>
</tr>
<tr>
<td>col name len</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col schema name</td>
<td>VARCHAR2(128)</td>
</tr>
<tr>
<td>col schema name len</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col precision</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col scale</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col charsetid</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col charsetform</td>
<td>INTEGER</td>
</tr>
<tr>
<td>col null_ok</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>

3.16.12 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.

```sql
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;
```
3.16.13  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.

```sql
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 ' ||
    'WHERE ename = :p_ename ' ||
    'AND e.deptno = d.deptno';

  v_status INTEGER;
BEGIN
```
```sql
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
```

### 3.16.14 FETCH_ROWS

The `FETCH_ROWS` function retrieves a row from a cursor.

```
status INTEGER FETCH_ROWS(c INTEGER)
```

**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,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;
END;
```

```
3.16.15 IS_OPEN

The IS_OPEN function provides the capability to test if the given cursor is open.

\[
\text{status BOOLEAN IS_OPEN}(c \text{ INTEGER})
\]

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.

3.16.16 LAST_ROW_COUNT

The LAST_ROW_COUNT function returns the number of rows that have been currently fetched.

\[
\text{rowcnt INTEGER LAST_ROW_COUNT}
\]

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.

```sql
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,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;
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
3.16.17 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.

\[
c \text{ INTEGER OPEN_CURSOR}
\]

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;
```

3.16.18 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.

\[
\text{PARSE}(c \text{ INTEGER}, \text{ statement VARCHAR2}, \text{ language_flag INTEGER})
\]

Parameters

\[
c
\]

Cursor ID of an open cursor.

\[
\text{statement}
\]
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.

```sql
DECLARE
   curid           INTEGER;
BEGIN
   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.

```sql
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.

```sql
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
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;
3.17 DBMS_UTILITY

The DBMS_UTILITY package provides support for the following various utility programs:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANALYZE_DATABASE(method [, estimate_rows [, estimate_percent [, method_opt ]]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Analyze database tables.</td>
</tr>
<tr>
<td>ANALYZE_PART_OBJECT(schema, object_name [, object_type [, command_type [, command_opt [, sample_clause ]]]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Analyze a partitioned table.</td>
</tr>
<tr>
<td>ANALYZE_SCHEMA(schema, method [, estimate_rows [, estimate_percent [, method_opt ]]])</td>
<td>Procedure</td>
<td>n/a</td>
<td>Analyze schema tables.</td>
</tr>
<tr>
<td>CANONICALIZE(name, canon_name OUT, canon_len)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Canonicalizes a string – e.g., strips off white space.</td>
</tr>
<tr>
<td>COMMA_TO_TABLE(list, tablen OUT, tab OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Convert a comma-delimited list of names to a table of names.</td>
</tr>
<tr>
<td>DB_VERSION(version OUT, compatibility OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Get the database version.</td>
</tr>
<tr>
<td>EXEC_DDL_STATEMENT(parse_string)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Execute a DDL statement.</td>
</tr>
<tr>
<td>FORMAT_CALL_STACK</td>
<td>Function</td>
<td>TEXT</td>
<td>Formats the current call stack.</td>
</tr>
<tr>
<td>GET_CPU_TIME</td>
<td>Function</td>
<td>NUMBER</td>
<td>Get the current CPU time.</td>
</tr>
<tr>
<td>GET_DEPENDENCY(type, schema, name)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Get objects that are dependent upon the given object.</td>
</tr>
<tr>
<td>GET_HASH_VALUE(name, base, hash_size)</td>
<td>Function</td>
<td>NUMBER</td>
<td>Compute a hash value.</td>
</tr>
<tr>
<td>GET_PARAMETER_VALUE(parnam, intval OUT, strval OUT)</td>
<td>Procedure</td>
<td>BINARY_INTEGER</td>
<td>Get database initialization parameter settings.</td>
</tr>
<tr>
<td>GET_TIME</td>
<td>Function</td>
<td>NUMBER</td>
<td>Get the current time.</td>
</tr>
<tr>
<td>NAME_TOKENIZE(name, a OUT, b OUT, c OUT, dblink OUT, nextpos OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Parse the given name into its component parts.</td>
</tr>
<tr>
<td>TABLE_TO_COMMA(tab, tablen OUT, list OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Convert a table of names to a comma-delimited list.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv_error_on_restrictions</td>
<td>PLS_INTEGER</td>
<td>1</td>
<td>Used by the INVALIDATE procedure.</td>
</tr>
<tr>
<td>lname_array</td>
<td>TABLE</td>
<td></td>
<td>For lists of long names.</td>
</tr>
</tbody>
</table>
3.17.1  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;
```

3.17.2  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;
```

3.17.3  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
```
Parameters - `ANALYZE_DATABASE` and `ANALYZE_SCHEMA`

`method` determines whether the `ANALYZE` procedure populates the `pg_statistics` table or removes entries from the `pg_statistics` table. If you specify a method of `DELETE`, the `ANALYZE` procedure removes the relevant rows from `pg_statistics`. If you specify a method of `COMPUTE` or `ESTIMATE`, the `ANALYZE` procedure analyzes a table (or multiple tables) and records the distribution information in `pg_statistics`. There 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`.

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.

SAMPLE n { ROWS | PERCENT }

This argument is ignored, but is included for compatibility.
3.17.4 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.

```sql
CREATE OR REPLACE PROCEDURE canonicalize (p_name VARCHAR2,
p_length BINARY_INTEGER DEFAULT 30)
IS
  v_canon VARCHAR2(100);
BEGIN

```

Copyright © 2007 - 2017 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

### 3.17.5 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 \textit{tab}.

\textit{tab}

Table containing the individual names in \textit{list}.

\textbf{LNAME\_ARRAY}

A \texttt{DBMS\_UTILITY.LNAME\_ARRAY} (as described in Section 3.17.1).

\textbf{UNCL\_ARRAY}

A \texttt{DBMS\_UTILITY.UNCL\_ARRAY} (as described in Section 3.17.2).

\textbf{Examples}

The following procedure uses the \texttt{COMMA\_TO\_TABLE} procedure to convert a list of names to a table. The table entries are then displayed.

```sql
CREATE OR REPLACE PROCEDURE comma_to_table (  
p_list      VARCHAR2  )  
IS  
    r_lname     DBMS_UTILITY.LNAME_ARRAY;  
    v_length    BINARY_INTEGER;  
BEGIN  
    DBMS_UTILITY.COMMA_TO_TABLE(p_list,v_length,r_lname);  
    FOR i IN 1..v_length LOOP  
        DBMS_OUTPUT.PUT_LINE(r_lname(i));  
    END LOOP;  
END;  
EXEC comma_to_table('edb.dept, edb.emp, edb.jobhist')
edb.dept  
edb.emp  
edb.jobhist
```

\textbf{3.17.6 DB\_VERSION}

The \texttt{DB\_VERSION} procedure returns the version number of the database.

\texttt{DB\_VERSION(version OUT VARCHAR2, compatibility OUT VARCHAR2)}

\textbf{Parameters}

\textit{version}
Database version number.

compatibility

Compatibility setting of the database. (To be implementation-defined as to its meaning.)

Examples

The following anonymous block displays the database version information.

```sql
DECLARE
  v_version       VARCHAR2(150);
  v_compat        VARCHAR2(150);
BEGIN
  DBMS_UTILITY.DB_VERSION(v_version,v_compat);
  DBMS_OUTPUT.PUT_LINE('Version: '       || v_version);
  DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END;
```

Version: EnterpriseDB 10.0.0 on i686-pc-linux-gnu, compiled by GCC gcc (GCC) 4.1.2 20080704 (Red Hat 4.1.2-48), 32-bit
Compatibility: EnterpriseDB 10.0.0 on i686-pc-linux-gnu, compiled by GCC gcc (GCC) 4.1.220080704 (Red Hat 4.1.2-48), 32-bit

3.17.7 EXEC_DDL_STATEMENT

The EXEC_DDL_STATEMENT provides the capability to execute a DDL command.

```sql
EXEC_DDL_STATEMENT(parse_string VARCHAR2)
```

Parameters

parse_string

The DDL command to be executed.

Examples

The following anonymous block creates the job table.

```sql
BEGIN
  DBMS_UTILITY.EXEC_DDL_STATEMENT(
    'CREATE TABLE job (' ||
    'jobno NUMBER(3),' ||
    'jname VARCHAR2(9))' ||
  );
```
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.

### 3.17.8 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.

### 3.17.9 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`  
--------------
3.17.10   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;
```

```
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()
```
### 3.17.11 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.

```sql
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) || ' ' || r_hash(r_emp.ename));
  END LOOP;
END;
SMITH 377
```
3.17.12 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);
  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

### 3.17.13 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
```
3.17.14  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:

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      :
dbлик  :
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      :
dbлик  :
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
dbлик  :
nextpos: 32

Tokenize the name edb.emp@edb_dblink:

BEGIN
3.17.15 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**

Table containing names.

**LNAME_ARRAY**

A DBMSUTILITY LNAME_ARRAY (as described in Section 3.17.1).

**UNCL_ARRAY**

A DBMSUTILITY UNCL_ARRAY (as described in Section 3.17.2).

**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.

```sql
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
### 3.18 UTL_ENCODE

The UTL_ENCODE package provides a way to encode and decode data. Advanced Server supports the following functions and procedures:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE64_DECODE(r)</td>
<td>RAW</td>
<td>Use the BASE64_DECODE function to translate a Base64 encoded string to the original RAW value.</td>
</tr>
<tr>
<td>BASE64_ENCODE(r)</td>
<td>RAW</td>
<td>Use the BASE64_ENCODE function to translate a RAW string to an encoded Base64 value.</td>
</tr>
<tr>
<td>BASE64_ENCODE(loid)</td>
<td>TEXT</td>
<td>Use the BASE64_ENCODE function to translate a TEXT string to an encoded Base64 value.</td>
</tr>
<tr>
<td>MIMEHEADER_DECODE(buf)</td>
<td>VARCHAR2</td>
<td>Use the MIMEHEADER_DECODE function to translate an encoded MIMEHEADER formatted string to its original value.</td>
</tr>
<tr>
<td>MIMEHEADER_ENCODE(buf, encode_charset, encoding)</td>
<td>VARCHAR2</td>
<td>Use the MIMEHEADER_ENCODE function to convert and encode a string in MIMEHEADER format.</td>
</tr>
<tr>
<td>QUOTED_PRINTABLE_DECODE(r)</td>
<td>RAW</td>
<td>Use the QUOTED_PRINTABLE_DECODE function to translate an encoded string to a RAW value.</td>
</tr>
<tr>
<td>QUOTED_PRINTABLE_ENCODE(r)</td>
<td>RAW</td>
<td>Use the QUOTED_PRINTABLE_ENCODE function to translate a string to a quoted-printable formatted RAW value.</td>
</tr>
<tr>
<td>TEXT_DECODE(buf, encode_charset, encoding)</td>
<td>VARCHAR2</td>
<td>Use the TEXT_DECODE function to decode a string encoded by TEXT_ENCODE.</td>
</tr>
<tr>
<td>TEXT_ENCODE(buf, encode_charset, encoding)</td>
<td>VARCHAR2</td>
<td>Use the TEXT_ENCODE function to translate a string to a user-specified character set, and then encode the string.</td>
</tr>
<tr>
<td>UUDECODE(r)</td>
<td>RAW</td>
<td>Use the UUDECODE function to translate a uuencode encoded string to a RAW value.</td>
</tr>
<tr>
<td>UUENCODE(r, type, filename, permission)</td>
<td>RAW</td>
<td>Use the UUENCODE function to translate a RAW string to an encoded uuencode value.</td>
</tr>
</tbody>
</table>

#### 3.18.1 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

Note: Before executing the following example, invoke the command:

```sql
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, please refer to the Postgres Core Documentation available at:


The following example first encodes (using `BASE64_ENCODE`), and then decodes (using `BASE64_DECODE`) a string that contains the text `abc`:

```sql
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)
```

3.18.2 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 \texttt{RAW} value or an \texttt{OID}.

**Parameters**

\texttt{r}

\texttt{r} specifies the \texttt{RAW} string that will be translated to Base64.

\texttt{loid}

\texttt{loid} specifies the object ID of a large object that will be translated to Base64.

**Examples**

Note: Before executing the following example, invoke the command:

\begin{verbatim}
SET bytea_output = escape;
\end{verbatim}

This command instructs the server to escape any non-printable characters, and to display \texttt{BYTEA} or \texttt{RAW} values onscreen in readable form. For more information, please refer to the Postgres Core Documentation available at:

\url{http://www.postgresql.org/docs/9.5/static/datatype-binary.html}

The following example first encodes (using \texttt{BASE64_ENCODE}), and then decodes (using \texttt{BASE64_DECODE}) a string that contains the text \texttt{abc}:

\begin{verbatim}
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)
\end{verbatim}

3.18.3 \quad \textbf{MIMEHEADER_DECODE}

Use the \texttt{MIMEHEADER_DECODE} function to decode values that are encoded by the \texttt{MIMEHEADER_ENCODE} function. The signature is:

\begin{verbatim}
MIMEHEADER_DECODE(buf IN VARCHAR2)
\end{verbatim}

This function returns a \texttt{VARCHAR2} value.
Parameters

\( \text{buf} \)

\( \text{buf} \) contains the value (encoded by \texttt{MIMEHEADER ENCODE}) that will be decoded.

Examples

The following examples use the \texttt{MIMEHEADER ENCODE} and \texttt{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)
```

3.18.4  \texttt{MIMEHEADER ENCODE}

Use the \texttt{MIMEHEADER ENCODE} function to convert a string into mime header format, and then encode the string. The signature is:

\[
\texttt{MIMEHEADER ENCODE(buf IN VARCHAR2, encode_charset IN VARCHAR2 DEFAULT NULL, encoding IN INTEGER DEFAULT NULL)}
\]

This function returns a \texttt{VARCHAR2} value.

Parameters

\( \text{buf} \)

\( \text{buf} \) contains the string that will be formatted and encoded. The string is a \texttt{VARCHAR2} value.

\( \text{encode_charset} \)

\( \text{encode_charset} \) specifies the character set to which the string will be converted before being formatted and encoded. The default value is \texttt{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)
```

3.18.5 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.

Examples
Note: 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, please refer to the Postgres Core Documentation available at:


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)
```

### 3.18.6 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**

Note: 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, please refer to the Postgres Core Documentation available at:

http://www.postgresql.org/docs/9.5/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)
```

### 3.18.7 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:

```sql
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)
```

### 3.18.8 **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 \texttt{TEXT\_ENCODE}. Specify:

- \texttt{UTL\_ENCODE.BASE64} to specify base-64 encoding.
- \texttt{UTL\_ENCODE.QUOTED\_PRINTABLE} to specify quoted printable encoding.
  This is the default.

**Examples**

The following example uses the \texttt{TEXT\_ENCODE} and \texttt{TEXT\_DECODE} functions to first encode, and then decode a string:

```sql
edb=# SELECT UTL\_ENCODE\_.\_TEXT\_ENCODE('What is the date?', 'BIG5', UTL\_ENCODE\_.BASE64) FROM DUAL;
  text\_encode
  --------------------
  V2hhdCBpcyB0aGUgZGF0ZT8=
  (1 row)

deb=# SELECT UTL\_ENCODE\_.\_TEXT\_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5', UTL\_ENCODE\_.BASE64) FROM DUAL;
  text\_decode
  -----------------
  What is the date?
  (1 row)
```

### 3.18.9 \texttt{UUDECODE}

Use the \texttt{UUDECODE} function to translate and decode a uuencode encoded string to the \texttt{RAW} value that was originally encoded by the \texttt{UUENCODE} function. The signature is:

\[
\texttt{UUDECODE}(r \ \text{IN} \ \texttt{RAW})
\]

This function returns a \texttt{RAW} value.

**Note:** If you are using the Advanced Server \texttt{UUDECODE} function to decode uuencoded data that was created by the Oracle implementation of the \texttt{UTL\_ENCODE\_.UUENCODE} function, then you must first set the Advanced Server configuration parameter \texttt{utl\_encode\_.uudecode\_redwood} to \texttt{TRUE} before invoking the Advanced Server \texttt{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 \texttt{UUENCODE} function results in a format that differs from the uuencoded data created by the Advanced Server \texttt{UUENCODE} function. As a result, attempting to use the Advanced Server \texttt{UUDECODE} function on the Oracle uuencoded data results in an error unless the configuration parameter \texttt{utl\_encode\_.uudecode\_redwood} is set to \texttt{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**

Note: Before executing the following example, invoke the command:

```sql
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, please refer to the Postgres Core Documentation available at:

'http://www.postgresql.org/docs/9.5/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;

begin 0 uuencode.txt
begin 0 uuencode.txt
end
(1 row)

edb=# SELECT UTL_ENCODE.UUDECODE('begin 0 uuencode.txt
\01215VAA="!I\r\n\n\t012`
\012`
012"
\012"
012end`012`
012end`012')

What is the date?
(1 row)
```

### 3.18.10 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>complete</td>
</tr>
<tr>
<td>2</td>
<td>header_piece</td>
</tr>
<tr>
<td>3</td>
<td>middle_piece</td>
</tr>
<tr>
<td>4</td>
<td>end_piece</td>
</tr>
</tbody>
</table>

$filename$

$filename$ is a VARCHAR2 value that specifies the file name that you want to embed in the encoded form; if you 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**

Note: 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, please refer to the Postgres Core Documentation available at:


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="!T;&9&%T93\"\012\012end\012
(1 row)

edb=# SELECT UTL_ENCODE.UUDECODE
edb-
(1 row)

What is the date?
3.19 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:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCLOSE(file IN OUT)</td>
<td>n/a</td>
<td>Closes the specified file identified by file.</td>
</tr>
<tr>
<td>FCLOSE_ALL</td>
<td>n/a</td>
<td>Closes all open files.</td>
</tr>
<tr>
<td>FCOPY(location, filename, dest_dir, dest_file [, start_line [ , end_line ]])</td>
<td>n/a</td>
<td>Copies filename in the directory identified by location to file, dest_file, in directory, dest_dir, starting from line, start_line, to line, end_line.</td>
</tr>
<tr>
<td>FFLUSH(file)</td>
<td>n/a</td>
<td>Forces data in the buffer to be written to disk in the file identified by file.</td>
</tr>
<tr>
<td>FOPEN(location, filename, open_mode [, max_linesize ])</td>
<td>FILE_TYPE</td>
<td>Opens file, filename, in the directory identified by location.</td>
</tr>
<tr>
<td>FREMOVE(location, filename)</td>
<td>n/a</td>
<td>Removes the specified file from the file system.</td>
</tr>
<tr>
<td>FRENAMEN(location, filename, dest_dir, dest_file [, overwrite ])</td>
<td>n/a</td>
<td>Renames the specified file.</td>
</tr>
<tr>
<td>GET_LINE(file, buffer OUT)</td>
<td>n/a</td>
<td>Reads a line of text into variable, buffer, from the file identified by file.</td>
</tr>
<tr>
<td>IS_OPEN(file)</td>
<td>BOOLEAN</td>
<td>Determines whether or not the given file is open.</td>
</tr>
<tr>
<td>NEW_LINE(file [, lines ])</td>
<td>n/a</td>
<td>Writes an end-of-line character sequence into the file.</td>
</tr>
</tbody>
</table>
### Function/Procedure Description

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT(file, buffer)</td>
<td>n/a</td>
<td>Writes buffer to the given file. PUT does not write an end-of-line character sequence.</td>
</tr>
<tr>
<td>PUT_LINE(file, buffer)</td>
<td>n/a</td>
<td>Writes buffer to the given file. An end-of-line character sequence is added by the PUT_LINE procedure.</td>
</tr>
<tr>
<td>PUTF(file, format [, arg1 ] [, ...])</td>
<td>n/a</td>
<td>Writes a formatted string to the given file. Up to five substitution parameters, arg1,...,arg5 may be specified for replacement in format.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Exception Code</th>
<th>Condition name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-29283</td>
<td>invalid operation</td>
</tr>
<tr>
<td>-29285</td>
<td>write error</td>
</tr>
<tr>
<td>-29284</td>
<td>read error</td>
</tr>
<tr>
<td>-29282</td>
<td>invalid filehandle</td>
</tr>
<tr>
<td>-29287</td>
<td>invalid maxlinesize</td>
</tr>
<tr>
<td>-29281</td>
<td>invalid mode</td>
</tr>
<tr>
<td>-29280</td>
<td>invalid path</td>
</tr>
</tbody>
</table>

### 3.19.1 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.

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.

```sql
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.

```sql
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.

```sql
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.

```sql
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-r-- 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.

3.19.2 FCLOSE

The FCLOSE procedure closes an open file.

    FCLOSE(file IN OUT FILE_TYPE)

Parameters

file

Variable of type FILE_TYPE containing a file handle of the file to be closed.

3.19.3 FCLOSE_ALL

The FCLOSE_ALL procedure closes all open files. The procedure executes successfully even if there are no open files to close.

    FCLOSE_ALL

3.19.4 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 ] ])
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**

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.

```sql
CREATE DIRECTORY empdir AS 'C:/TEMP/EMPDIR';

DECLARE
  v_empfile       UTL_FILE.FILE_TYPE;
  v_src_dir       VARCHAR2(50) := 'empdir';
  v_src_file      VARCHAR2(20) := 'empfile.csv';
  v_dest_dir      VARCHAR2(50) := 'empdir';
  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-81,1600,30,30
7521,WARD,SALESMAN,7698,22-FEB-81,1250,50,30
7566,JOHNSON,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,WHITE,CLERK,7788,10-SEP-81,950,,30
7934,MILLER,CLERK,7788,23-JAN-82,1300,,10
14 records retrieved

3.19.5 FFLUSH

The FFLUSH procedure flushes unwritten data from the write buffer to the file.

    FFLUSH(file FILE_TYPE)

Parameters

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);
    END LOOP;
END;
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;

3.19.6 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

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.

### 3.19.7 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

### 3.19.8 FRENAME

The FRENAME procedure renames a given file. This effectively moves a file from one location to another.

FRENAME(location VARCHAR2, filename VARCHAR2,
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);
    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'

<table>
<thead>
<tr>
<th>Employee_ID, Name, Title, Manager_ID, Hired_Date, Salary, Commision_Pct, Manager_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369, SMITH, CLERK, 7698, 17-DEC-80 00:00:00, 800.00, 20</td>
</tr>
<tr>
<td>7499, ALLEN, SALESMAN, 7698, 20-FEB-81 00:00:00, 1600.00, 300.00, 30</td>
</tr>
<tr>
<td>7521, WARD, SALESMAN, 7698, 22-FEB-81 00:00:00, 1250.00, 500.00, 30</td>
</tr>
<tr>
<td>7566, JONES, MANAGER, 7839, 02-APR-81 00:00:00, 2975.00, 20</td>
</tr>
<tr>
<td>7654, MARTIN, SALESMAN, 7698, 28-SEP-81 00:00:00, 1250.00, 1400.00, 30</td>
</tr>
<tr>
<td>7698, BLAKE, MANAGER, 7839, 01-MAY-81 00:00:00, 2850.00, 30</td>
</tr>
<tr>
<td>7782, CLARK, MANAGER, 7839, 09-JUN-81 00:00:00, 2450.00, 10</td>
</tr>
<tr>
<td>7788, SCOTT, ANALYST, 7566, 19-APR-87 00:00:00, 3000.00, 20</td>
</tr>
<tr>
<td>7839, KING, PRESIDENT, 17-NOV-81 00:00:00, 5000.00, 10</td>
</tr>
<tr>
<td>7844, TURNER, SALESMAN, 7698, 08-SEP-81 00:00:00, 1500.00, 0.00, 30</td>
</tr>
<tr>
<td>7876, ADAMS, CLERK, 7788, 23-MAY-87 00:00:00, 1100.00, 20</td>
</tr>
<tr>
<td>7900, JAMES, CLERK, 7698, 03-DEC-81 00:00:00, 950.00, 30</td>
</tr>
<tr>
<td>7902, FORD, ANALYST, 7566, 03-DEC-81 00:00:00, 3000.00, 20</td>
</tr>
<tr>
<td>7934, MILLER, CLERK, 7782, 23-JAN-82 00:00:00, 1300.00, 10</td>
</tr>
</tbody>
</table>
14 records retrieved

3.19.9 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;
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
```

3.19.10 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.

3.19.11 NEW_LINE

The NEW_LINE procedure writes an end-of-line character sequence in the file.

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);
END;
```
UTL_FILE.FCLOSE(v_empfile);
END;

Created file: empfile.csv

This file is then displayed:

```sql
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
```

### 3.19.12 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.

```sql
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.
**Examples**

The following example uses the `PUT` procedure to create a comma-delimited file of employees from the `emp` table.

```sql
DECLARE
    v_empfile    UTL_FILE.FILE_TYPE;
    v_directory  VARCHAR2(50) := 'empdir';
    v_filename   VARCHAR2(20) := 'empfile.csv';
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
    FOR i IN emp_cur LOOP
        UTL_FILE.PUT(v_empfile,i.empno);
        UTL_FILE.PUT(v_empfile,',');
        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);
    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
```
3.19.13 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**

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 empnum;
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
```
3.19.14 **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`

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
Salary: $%s Commission: $%s
';
  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
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
### 3.20 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:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_REQUEST(url, method, http_version)</td>
<td>UTL_HTTP.REQ</td>
<td>Initiates a new HTTP request.</td>
</tr>
<tr>
<td>END_REQUEST(r IN OUT)</td>
<td>n/a</td>
<td>Ends an HTTP request before allowing it to complete.</td>
</tr>
<tr>
<td>END_RESPONSE(r IN OUT)</td>
<td>n/a</td>
<td>Ends the HTTP response.</td>
</tr>
<tr>
<td>GET_BODY_CHARSET</td>
<td>VARCHAR2</td>
<td>Returns the default character set of the body of future HTTP requests.</td>
</tr>
<tr>
<td>GET_BODY_CHARSET(charset OUT)</td>
<td>n/a</td>
<td>Returns the default character set of the body of future HTTP requests.</td>
</tr>
<tr>
<td>GET_FOLLOW_REDIRECT(max_redirects OUT)</td>
<td>n/a</td>
<td>Current setting for the maximum number of redirections allowed.</td>
</tr>
<tr>
<td>GET_HEADER(r IN OUT, n, name OUT, value OUT)</td>
<td>n/a</td>
<td>Returns the $n$th header of the HTTP response.</td>
</tr>
<tr>
<td>GET_HEADER_BY_NAME(r IN OUT, name, value OUT, n)</td>
<td>n/a</td>
<td>Returns the HTTP response header for the specified name.</td>
</tr>
<tr>
<td>GET_HEADER_COUNT(r IN OUT)</td>
<td>INTEGER</td>
<td>Returns the number of HTTP response headers.</td>
</tr>
<tr>
<td>GET_RESPONSE(r IN OUT)</td>
<td>UTL_HTTP.RESP</td>
<td>Returns the HTTP response.</td>
</tr>
<tr>
<td>GET_RESPONSE_ERROR_CHECK(enable OUT)</td>
<td>n/a</td>
<td>Returns whether or not response error check is set.</td>
</tr>
<tr>
<td>GET_TRANSFER_TIMEOUT(timeout OUT)</td>
<td>n/a</td>
<td>Returns the transfer timeout setting for HTTP requests.</td>
</tr>
<tr>
<td>READ_LINE(r IN OUT, data OUT, remove_crlf)</td>
<td>n/a</td>
<td>Returns the HTTP response body in text form until the end of line.</td>
</tr>
<tr>
<td>READ_RAW(r IN OUT, data OUT, len)</td>
<td>n/a</td>
<td>Returns the HTTP response body in binary form for a specified number of bytes.</td>
</tr>
<tr>
<td>READ_TEXT(r IN OUT, data OUT, len)</td>
<td>n/a</td>
<td>Returns the HTTP response body in text form for a specified number of characters.</td>
</tr>
<tr>
<td>REQUEST(url)</td>
<td>VARCHAR2</td>
<td>Returns the content of a web page.</td>
</tr>
<tr>
<td>REQUESTPieces(url, max_pieces)</td>
<td>UTL_HTTP.HTML_PIECES</td>
<td>Returns a table of 2000-byte segments retrieved from an URL.</td>
</tr>
<tr>
<td>SET_BODY_CHARSET(charset)</td>
<td>n/a</td>
<td>Sets the default character set of the body of future HTTP requests.</td>
</tr>
<tr>
<td>SET_FOLLOW_REDIRECT(max_redirects)</td>
<td>n/a</td>
<td>Sets the maximum number of times to follow the redirect instruction.</td>
</tr>
<tr>
<td>SET_FOLLOW_REDIRECT(r IN OUT, max_redirects)</td>
<td>n/a</td>
<td>Sets the maximum number of times to follow the redirect instruction for an individual request.</td>
</tr>
<tr>
<td>SET_HEADER(r IN OUT, name, value)</td>
<td>n/a</td>
<td>Sets the HTTP request header.</td>
</tr>
<tr>
<td>SET_RESPONSE_ERROR_CHECK(enable)</td>
<td>n/a</td>
<td>Determines whether or not HTTP 4xx and 5xx status codes are to be treated as errors.</td>
</tr>
</tbody>
</table>
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.

Please 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.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_TRANSFER_TIMEOUT(timeout)</td>
<td>n/a</td>
<td>Sets the default, transfer timeout value for HTTP requests.</td>
</tr>
<tr>
<td>SET_TRANSFER_TIMEOUT(r IN OUT, timeout)</td>
<td>n/a</td>
<td>Sets the transfer timeout value for an individual HTTP request.</td>
</tr>
<tr>
<td>WRITE_LINE(r IN OUT, data)</td>
<td>n/a</td>
<td>Writes CRLF terminated data to the HTTP request body in TEXT form.</td>
</tr>
<tr>
<td>WRITE_RAW(r IN OUT, data)</td>
<td>n/a</td>
<td>Writes data to the HTTP request body in BINARY form.</td>
</tr>
<tr>
<td>WRITE_TEXT(r IN OUT, data)</td>
<td>n/a</td>
<td>Writes data to the HTTP request body in TEXT form.</td>
</tr>
</tbody>
</table>

The following table contains UTL_HTTP public constants defining HTTP versions and port assignments.

<table>
<thead>
<tr>
<th>HTTP VERSIONS</th>
<th>Standard Port Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_VERSION_1_0</td>
<td>DEFAULT_HTTP_PORT</td>
</tr>
<tr>
<td>HTTP_VERSION_1_1</td>
<td>DEFAULT_HTTPS_PORT</td>
</tr>
<tr>
<td>CONSTANT VARCHAR2(64) := 'HTTP/1.0'</td>
<td>CONSTANT INTEGER := 80;</td>
</tr>
<tr>
<td>CONSTANT VARCHAR2(64) := 'HTTP/1.1'</td>
<td>CONSTANT INTEGER := 443;</td>
</tr>
</tbody>
</table>
The following table contains `UTL_HTTP` public status code constants.

<table>
<thead>
<tr>
<th>Category</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFORMATIONAL</td>
<td>1XX</td>
<td>HTTP_CONTINUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_SWITCHING_PROTOCOLS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_PROCESSING</td>
</tr>
<tr>
<td>SUCCESS</td>
<td>2XX</td>
<td>HTTP_OK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_CREATED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_ACCEPTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_NON_AUTHORITATIVE_INFO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_NO_CONTENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_RESET_CONTENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_PARTIAL_CONTENT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_MULTI_STATUS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_ALREADY_REPORTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_IM_USED</td>
</tr>
<tr>
<td>REDIRECTION</td>
<td>3XX</td>
<td>HTTP_MULTIPLE_CHOICES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_MOVED_PERMANENTLY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_FOUND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_SEE_OTHER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_NOT_MODIFIED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_USE_PROXY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_SWITCH_PROXY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_TEMPORARY_REDIRECT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_PERMANENT_REDIRECT</td>
</tr>
<tr>
<td>CLIENT ERROR</td>
<td>4XX</td>
<td>HTTP_BAD_REQUEST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_UNAUTHORIZED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_PAYMENT_REQUIRED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_FORBIDDEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_NOT_FOUND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_METHOD_NOT_ALLOWED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_NOT_ACCEPTABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_PROXY_AUTH_REQUIRED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_REQUEST_TIME_OUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_CONFLICT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_GONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_LENGTH_REQUIRED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_PRECONDITION_FAILED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_REQUEST_ENTITY_TOO_LARGE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_REQUEST_URI_TOO_LARGE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_UNSUPPORTED_MEDIA_TYPE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_REQUEST_RANGE_NOT_SATISFIABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_EXPECTATION_FAILED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_I_AM_A_TEAPOT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_AUTHENTICATION_TIME_OUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_ENHANCE_YOUR_CALM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_UNPROCESSABLE_ENTITY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_LOCKED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_FAILED_DEPENDENCY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_UNORDERED_COLLECTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_UPGRADE_REQUIRED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_PRECONDITION_REQUIRED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_TOO_MANY_REQUESTS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_REQUEST_HEADER_FIELDS_TOO_LARGE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_NO_RESPONSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HTTP_RETRY_WITH</td>
</tr>
</tbody>
</table>

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.
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;

3.20.1 HTML_PIECES

The UTL_HTTP package declares a type named HTML_PIECES, which is a table of type VARCHAR2 (2000) indexed by BINARY INTEGER. A value of this type is returned by the REQUEST_PIECES function.

```sql
TYPE html_pieces IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
```

3.20.2 REQ

The REQ record type holds information about each HTTP request.

```sql
TYPE req IS RECORD (
    url VARCHAR2(32767), -- URL to be accessed
    method VARCHAR2(64), -- HTTP method
    http_version VARCHAR2(64), -- HTTP version
    private_hdl INTEGER -- Holds handle for this request
);
```

3.20.3 RESP

The RESP record type holds information about the response from each HTTP request.
3.20.4 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:

```sql
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.

3.20.5 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:

```sql
END_REQUEST(r IN OUT UTL_HTTP.REQ)
```
Parameters

\( r \)

\( r \) is the HTTP request record.

### 3.20.6 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:

\[
\text{END\_RESPONSE}(r \ \text{IN} \ \text{OUT} \ \text{UTL\_HTTP\_RESP})
\]

Parameters

\( r \)

\( r \) is the HTTP response record.

### 3.20.7 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:

\[
\text{GET\_BODY\_CHARSET}(\text{charset} \ \text{OUT} \ \text{VARCHAR2})
\]

The function signature is:

\[
\text{GET\_BODY\_CHARSET()} \ \text{RETURN} \ \text{VARCHAR2}
\]

This function returns a `VARCHAR2` value.

Parameters

\( \text{charset} \)

\( \text{charset} \) is the character set of the body.
Examples

The following is an example of the \texttt{GET\_BODY\_CHARSET} function.

\begin{verbatim}
edb=# SELECT UTL_HTTP.GET_BODY_CHARSET() FROM DUAL;
get_body_charset
------------------
ISO-8859-1
(1 row)
\end{verbatim}

3.20.8 \hspace{1em} GET\_FOLLOW\_REDIRECT

The \texttt{GET\_FOLLOW\_REDIRECT} procedure returns the current setting for the maximum number of redirections allowed. The signature is:

\begin{verbatim}
GET\_FOLLOW\_REDIRECT(max\_redirects \hspace{1em} OUT \hspace{1em} INTEGER)
\end{verbatim}

Parameters

\begin{itemize}
    \item \texttt{max\_redirects}
\end{itemize}

\texttt{max\_redirects} is maximum number of redirections allowed.

3.20.9 \hspace{1em} GET\_HEADER

The \texttt{GET\_HEADER} procedure returns the \textit{n}th header of the HTTP response. The signature is:

\begin{verbatim}
GET\_HEADER(r \hspace{1em} IN \hspace{1em} OUT \hspace{1em} UTL\_HTTP\_RESP, \hspace{1em} n \hspace{1em} INTEGER, \hspace{1em} name \hspace{1em} OUT \hspace{1em} VARCHAR2, \hspace{1em} value \hspace{1em} OUT \hspace{1em} VARCHAR2)
\end{verbatim}

Parameters

\begin{itemize}
    \item \texttt{r}
\end{itemize}

\texttt{r} is the HTTP response record.

\begin{itemize}
    \item \texttt{n}
\end{itemize}

\texttt{n} is the \textit{n}th header of the HTTP response record to retrieve.

\begin{itemize}
    \item \texttt{name}
\end{itemize}

\texttt{name} is the name of the response header.
value is the value of the response header.

Examples

The following example retrieves the header count, then the headers.

```sql
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=jn1uljletmd15gg41h8hakvs01;
            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
```
3.20.10 GET_HEADER_BY_NAME

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 `n`th header of the HTTP response record to retrieve according to the values specified by `name`. The default is 1.

Examples

The following example retrieves the header for Content-Type.

```
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
3.20.11 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**

`r`

`r` is the HTTP response record.

3.20.12 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.

3.20.13 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`

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.
enable returns TRUE if response error check is set, otherwise it returns FALSE.

3.20.14 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.
```

3.20.15 READ_LINE

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;
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, 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" />
    <script 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>
  </head>
3.20.16 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;
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.

```

``
3.20.17 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;
```

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/
```
3.20.18 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;
```

3.20.19 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:

```sql
DECLARE
    result UTL_HTTP.HTML_PIECES;
BEGIN
    result := UTL_HTTP.REQUEST_PIECES('http://www.enterprisedb.com/', 4);
END;
```

3.20.20 SET_BODY_CHARSET

The SET_BODY_CHARSET procedure sets the default character set of the body of future HTTP requests. The signature is:

```sql
SET_BODY_CHARSET(charset VARCHAR2 DEFAULT NULL)
```

Parameters

ccharset
ccharset is the character set of the body of future requests. The default is null in which case the database character set is assumed.

3.20.21 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:

```sql
SET_FOLLOW_REDIRECT(max_redirects IN INTEGER DEFAULT 3)
```

and

```sql
SET_FOLLOW_REDIRECT(r IN OUT UTL_HTTP.REQ, max_redirects IN INTEGER DEFAULT 3)
```

Use the second form to change the maximum number of redirections for an individual request that a request inherits from the session default settings.

Parameters

r
$r$ 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.

### 3.20.22 SET_HEADER

The SET_HEADER procedure sets the HTTP request header. The signature is:

```sql
SET_HEADER($r$ IN OUT UTL_HTTP.REQ, $name$ IN VARCHAR2, $value$ IN VARCHAR2 DEFAULT NULL)
```

**Parameters**

$r$

$r$ 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.

### 3.20.23 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:

```sql
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.
### 3.20.24 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`

  `r` is the HTTP request record.

- `timeout`

  `timeout` is the transfer timeout setting in seconds for HTTP requests. The default is 60 seconds.

### 3.20.25 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**

```sql
/* 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.

```sql
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.example.com/post.php', 'POST');
    UTL_HTTP.SET_HEADER(v_req, 'Content-Length', '23');
    UTL_HTTP.WRITE_LINE(v_req, 'Account balance $500.00');
    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
```

### 3.20.26 WRITE_RAW

The WRITE_RAW procedure writes data to the HTTP request body in binary form. The signature is:

```
WRITE_RAW(r IN OUT UTL_HTTP.REQ, data IN RAW)
```

**Parameters**

- `r`

  `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.

```sql
DECLARE
```

Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved. 266
v_req           UTL_HTTP.REQ;
v_resp          UTL_HTTP.RESP;
BEGIN
  v_req := UTL_HTTP.BEGIN_REQUEST('http://www.example.com/post.php', 'POST');
  UTL_HTTP.SET_HEADER(v_req, 'Content-Length', '23');
  UTL_HTTP.WRITE_RAW(v_req, HEXTORAW('54657374696e6720504f53545020696e20485454502072657175657374'));
  v_resp := UTL_HTTP.GET_RESPONSE(v_req);
  DBMS_OUTPUT.PUT_LINE('Status Code: ' || v_resp.status_code);
  DBMS_OUTPUT.PUT_LINE('Reason Phrase: ' || v_resp.reason_phrase);
  UTL_HTTP.END_RESPONSE(v_resp);
END;

The text string shown in the HEXTORAW function is the hexadecimal translation of the text
Testing POST method in HTTP request.

Assuming the web application successfully processed the POST method, the following
output would be displayed:

```
Status Code: 200
Reason Phrase: OK
```

### 3.20.27 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`

`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;
END;
```
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
3.21 UTL_MAIL

The UTL_MAIL package provides the capability to manage e-mail. Advanced Server supports the following procedures:

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND(sender, recipients, cc, bcc, subject, message [, mime_type [, priority ]])</td>
<td>n/a</td>
<td>Packages and sends an e-mail to an SMTP server.</td>
</tr>
<tr>
<td>SEND_ATTACHMENT_RAW(sender, recipients, cc, bcc, subject, message, mime_type, priority, attachment [, att_inline [, att_mime_type [, att_filename ]]])</td>
<td>n/a</td>
<td>Same as the SEND procedure, but with BYTEA or large object attachments.</td>
</tr>
<tr>
<td>SEND_ATTACHMENT_VARCHAR2(sender, recipients, cc, bcc, subject, message, mime_type, priority, attachment [, att_inline [, att_mime_type [, att_filename ]]])</td>
<td>n/a</td>
<td>Same as the SEND procedure, but with VARCHAR2 attachments.</td>
</tr>
</tbody>
</table>

Note: An administrator must grant execute privileges to each user or group before they can use this package.

3.21.1 SEND

The SEND procedure provides the capability to send an e-mail to an SMTP server.

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.

```sql
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;
```

3.21.2 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 ]]])

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

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.

3.21.3 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

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.
3.22 UTL_RAW

The UTL_RAW package allows you to manipulate or retrieve the length of raw data types.

**Note:** An administrator must grant execute privileges to each user or group before they can use this package.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAST_TO_RAW(c IN VARCHAR2)</td>
<td>Function</td>
<td>RAW</td>
<td>Converts a VARCHAR2 string to a RAW value.</td>
</tr>
<tr>
<td>CAST_TO_VARCHAR2(r IN RAW)</td>
<td>Function</td>
<td>VARCHAR2</td>
<td>Converts a RAW value to a VARCHAR2 string.</td>
</tr>
<tr>
<td>CONCAT(r1 IN RAW, r2 IN RAW, r3 IN RAW, ...)</td>
<td>Function</td>
<td>RAW</td>
<td>Concatenate multiple RAW values into a single RAW value.</td>
</tr>
<tr>
<td>CONVERT(r IN RAW, to_charset IN VARCHAR2, from_charset IN VARCHAR2)</td>
<td>Function</td>
<td>RAW</td>
<td>Converts encoded data from one encoding to another, and returns the result as a RAW value.</td>
</tr>
<tr>
<td>LENGTH(r IN RAW)</td>
<td>Function</td>
<td>NUMBER</td>
<td>Returns the length of a RAW value.</td>
</tr>
<tr>
<td>SUBSTR(r IN RAW, pos IN INTEGER, len IN INTEGER)</td>
<td>Function</td>
<td>RAW</td>
<td>Returns a portion of a RAW value.</td>
</tr>
</tbody>
</table>

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.

3.22.1 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
```

### 3.22.2 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:

```
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);
```
The result set includes the content of the original string and the converted RAW value:

\x4163636f756e7473
Accounts

3.22.3 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
```

The result (the concatenated values) is then converted to VARCHAR2 format by the CAST_TO_VARCHAR2 function.

3.22.4 CONVERT

The CONVERT function converts a string from one encoding to another encoding and returns the result as a RAW value. The signature is:
The function returns a \texttt{RAW} value.

\textbf{Parameters}

\texttt{r}

The \texttt{RAW} value that will be converted.

\texttt{to\_charset}

The name of the encoding to which \texttt{r} will be converted.

\texttt{from\_charset}

The name of the encoding from which \texttt{r} will be converted.

\textbf{Example}

The following example uses the \texttt{UTL\_RAW.CAST\_TO\_RAW} function to convert a \texttt{VARCHAR2} string (Accounts) to a raw value, and then convert the value from \texttt{UTF8} to \texttt{LATIN7}, and then from \texttt{LATIN7} to \texttt{UTF8}:

\begin{verbatim}
DECLARE
  r RAW;
  v VARCHAR2;
BEGIN
  v := 'Accounts';
  dbms_output.put_line(v);
  r := UTL_RAW.CAST_TO_RAW(v);
  dbms_output.put_line(r);
  r := UTL_RAW.CONVERT(r, 'UTF8', 'LATIN7');
  dbms_output.put_line(r);
  r := UTL_RAW.CONVERT(r, 'LATIN7', 'UTF8');
  dbms_output.put_line(r);
\end{verbatim}

The example returns the \texttt{VARCHAR2} value, the \texttt{RAW} value, and the converted values:

\begin{verbatim}
Accounts
\x4163636f756e7473
\x4163636f756e7473
\x4163636f756e7473
\end{verbatim}

\section{3.22.5 \textbf{LENGTH}}

The \texttt{LENGTH} function returns the length of a \texttt{RAW} value. The signature is:
LENGTH(r RAW)

The function returns a RAW value.

Parameters

r

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('独孤求败'));
length-------
12
(1 row)
```

3.22.6 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.

\textit{len}

The maximum number of bytes that will be returned.

\textbf{Example}

The following example uses the \texttt{SUBSTR} function to select a substring that begins 3 bytes from the start of a RAW value:

```
SELECT UTL_RAW.SUBSTR(UTL_RAW.CAST_TO_RAW('Accounts'), 3, 5) FROM DUAL;
```

```
substr
--------
count
(1 row)
```

The following example uses the \texttt{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
--------
coun
(1 row)
```
3.23 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.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Function or Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE_DATA(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Ends an e-mail message.</td>
</tr>
<tr>
<td>COMMAND(c IN OUT, cmd [, arg ] )</td>
<td>Both</td>
<td>REPLY</td>
<td>Execute an SMTP command.</td>
</tr>
<tr>
<td>COMMAND_REPLIES(c IN OUT, cmd [, arg ] )</td>
<td>Function</td>
<td>REPLIES</td>
<td>Execute an SMTP command where multiple reply lines are expected.</td>
</tr>
<tr>
<td>DATA(c IN OUT, body VARCHAR2)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Specify the body of an e-mail message.</td>
</tr>
<tr>
<td>EHLO(c IN OUT, domain)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Perform initial handshaking with an SMTP server and return extended information.</td>
</tr>
<tr>
<td>HELO(c IN OUT, domain)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Perform initial handshaking with an SMTP server</td>
</tr>
<tr>
<td>HELP(c IN OUT [, command ] )</td>
<td>Function</td>
<td>REPLIES</td>
<td>Send the HELP command.</td>
</tr>
<tr>
<td>MAIL(c IN OUT, sender [, parameters ] )</td>
<td>Procedure</td>
<td>n/a</td>
<td>Start a mail transaction.</td>
</tr>
<tr>
<td>NOOP(c IN OUT)</td>
<td>Both</td>
<td>REPLY</td>
<td>Send the null command.</td>
</tr>
<tr>
<td>OPEN_CONNECTION(host [, port [, tx_timeout ] ] )</td>
<td>Function</td>
<td>CONNECTION</td>
<td>Open a connection.</td>
</tr>
<tr>
<td>OPEN_DATA(c IN OUT)</td>
<td>Both</td>
<td>REPLY</td>
<td>Send the DATA command.</td>
</tr>
<tr>
<td>QUIT(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Terminate the SMTP session and disconnect.</td>
</tr>
<tr>
<td>RCPT(c IN OUT, recipient [, parameters ] )</td>
<td>Procedure</td>
<td>n/a</td>
<td>Specify the recipient of an e-mail message.</td>
</tr>
<tr>
<td>RSET(c IN OUT)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Terminate the current mail transaction.</td>
</tr>
<tr>
<td>VRFY(c IN OUT, recipient )</td>
<td>Function</td>
<td>REPLIES</td>
<td>Validate an e-mail address.</td>
</tr>
<tr>
<td>WRITE_DATA(c IN OUT, data)</td>
<td>Procedure</td>
<td>n/a</td>
<td>Write a portion of the e-mail message.</td>
</tr>
</tbody>
</table>

Advanced Server's implementation of UTL_SMTP 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 UTL_SMTP package.

<table>
<thead>
<tr>
<th>Public Variables</th>
<th>Data Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>RECORD</td>
<td></td>
<td>Description of an SMTP connection.</td>
</tr>
<tr>
<td>reply</td>
<td>RECORD</td>
<td></td>
<td>SMTP reply line.</td>
</tr>
</tbody>
</table>
3.23.1 CONNECTION

The CONNECTION record type provides a description of an SMTP connection.

```plaintext
TYPE connection IS RECORD (
  host            VARCHAR2(255),
  port            PLS_INTEGER,
  tx_timeout      PLS_INTEGER
);
```

3.23.2 REPLY/REPLIES

The REPLY record type provides a description of an SMTP reply line. REPLIES is a table of multiple SMTP reply lines.

```plaintext
TYPE reply IS RECORD (
  code            INTEGER,
  text            VARCHAR2(508)
);
TYPE replies IS TABLE OF reply INDEX BY BINARY_INTEGER;
```

3.23.3 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.
3.23.4 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 Section 3.23.2 for a description of REPLY and REPLIES.

3.23.5 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

SMTP reply lines to the command. See Section 3.23.2 for a description of `REPLY` and `REPLIES`.

3.23.6 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.

3.23.7 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.

3.23.8 HELO

The HELO procedure performs initial handshaking with the SMTP server after establishing the connection. The HELO procedure allows the client to identify itself to the SMTP server according to RFC 821. The EHLO procedure performs the equivalent functionality, but returns more information about the server.

HELO(c IN OUT, domain VARCHAR2)

Parameters

c
The connection to the SMTP server over which to perform handshaking.

domain

Domain name of the sending host.

3.23.9 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 Section 3.23.2 for a description of \texttt{REPLY} and \texttt{REPLIES}.

3.23.10 MAIL

The \texttt{MAIL} procedure initiates a mail transaction.

\texttt{MAIL(c IN OUT CONNECTION, sender VARCHAR2}\n \texttt{[,, 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, \texttt{key=value} as defined in RFC 1869.

3.23.11 NOOP

The \texttt{NOOP} function/procedure sends the null command to the SMTP server. The \texttt{NOOP} has no effect upon the server except to obtain a successful response.

\texttt{reply REPLY NOOP(c IN OUT CONNECTION)}

\texttt{NOOP(c IN OUT CONNECTION)}

Parameters
The SMTP connection on which to send the command.

\texttt{reply} 

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in \texttt{reply}. See Section 3.23.2 for a description of \texttt{REPLY} and \texttt{REPLIES}.

### 3.23.12 OPEN\_CONNECTION

The \texttt{OPEN\_CONNECTION} functions open a connection to an SMTP server.

\begin{verbatim}
c CONNECTION OPEN\_CONNECTION(host VARCHAR2 [, port PLS\_INTEGER [, tx\_timeout PLS\_INTEGER DEFAULT NULL]])
\end{verbatim}

**Parameters**

\textit{host} 

Name of the SMTP server.

\textit{port} 

Port number on which the SMTP server is listening. The default is 25.

\textit{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.

\texttt{c} 

Connection handle returned by the SMTP server.

### 3.23.13 OPEN\_DATA

The \texttt{OPEN\_DATA} procedure sends the \texttt{DATA} command to the SMTP server.

\begin{verbatim}
OPEN\_DATA(c IN OUT CONNECTION)
\end{verbatim}
Parameters

\( c \)

SMTP connection on which to send the command.

### 3.23.14 QUIT

The **QUIT** procedure closes the session with an SMTP server.

\[
\text{QUIT}(c \text{ IN OUT CONNECTION})
\]

Parameters

\( c \)

SMTP connection to be terminated.

### 3.23.15 RCPT

The **RCPT** procedure provides the e-mail address of the recipient. To schedule multiple recipients, invoke **RCPT** multiple times.

\[
\text{RCPT}(c \text{ IN OUT CONNECTION, recipient VARCHAR2}\ 
\qquad [, \text{parameters VARCHAR2 }] )
\]

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.
3.23.16 RSET

The RSET procedure provides the capability to terminate the current mail transaction.

\[ \text{RSET}(c \text{ IN OUT CONNECTION}) \]

**Parameters**

\( c \)

SMTP connection on which to cancel the mail transaction.

3.23.17 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.

\[ \text{reply REPLY VRFY}(c \text{ IN OUT CONNECTION, recipient VARCHAR2}) \]

**Parameters**

\( c \)

The SMTP connection on which to verify the e-mail address.

\( \text{recipient} \)

The recipient’s e-mail address to be verified.

\( \text{reply} \)

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in \( \text{reply} \). See Section 3.23.2 for a description of \( \text{REPLY} \) and \( \text{REPLIES} \).

3.23.18 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.

\[ \text{WRITE_DATA}(c \text{ IN OUT CONNECTION, data VARCHAR2}) \]
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.

3.23.19 Comprehensive Example

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);
    '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.CLOSE_DATA(v_conn);
  UTL_SMTP.QUIT(v_conn);
END;
EXEC send_mail_2('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday Party','Are you planning to attend?','smtp.enterprisedb.com');
IS
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 Party','Are you planning to attend?','smtp.enterprisedb.com');
3.24 UTL_URL

The UTL_URL package provides a way to escape illegal and reserved characters within an URL.

<table>
<thead>
<tr>
<th>Function/Procedure</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESCAPE(url, escape_reserved_chars, url_charset)</td>
<td>VARCHAR2</td>
<td>Use the ESCAPE function to escape any illegal and reserved characters in a URL.</td>
</tr>
<tr>
<td>UNESCAPE(url, url_charset)</td>
<td>VARCHAR2</td>
<td>The UNESCAPE function to convert an URL to its original form.</td>
</tr>
</tbody>
</table>

The UTL_URL package will return the BAD_URL exception if the call to a function includes an incorrectly-formed URL.

3.24.1 ESCAPE

Use the ESCAPE function to escape illegal and reserved characters within an URL. The signature is:

```
ESCAPE(url VARCHAR2, escapeReservedChars 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:

<table>
<thead>
<tr>
<th>Uppercase A through Z</th>
<th>Lowercase a through z</th>
<th>0 through 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>asterisk (*)</td>
<td>exclamation point (!)</td>
<td>hyphen (-)</td>
</tr>
<tr>
<td>left parenthesis ()</td>
<td>period (.)</td>
<td>right parenthesis ()</td>
</tr>
<tr>
<td>single-quote ('')</td>
<td>tilde (~)</td>
<td>underscore (_)</td>
</tr>
</tbody>
</table>

Some characters are legal in some parts of an URL, while illegal in others; to review comprehensive rules about illegal characters, please refer to RFC 2396. Some *examples* of characters that are considered illegal in any part of an URL are:

<table>
<thead>
<tr>
<th>Illegal Character</th>
<th>Escape Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>a blank space ( )</td>
<td>%20</td>
</tr>
<tr>
<td>curly braces ({ or })</td>
<td>%7b and %7d</td>
</tr>
<tr>
<td>hash mark (#)</td>
<td>%23</td>
</tr>
</tbody>
</table>

The **ESCAPE** function considers the following characters to be reserved, and will escape them if `escape_reserved_chars` is set to **TRUE**:

<table>
<thead>
<tr>
<th>Reserved Character</th>
<th>Escape Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>ampersand (&amp;)</td>
<td>%5C</td>
</tr>
<tr>
<td>at sign (@)</td>
<td>%25</td>
</tr>
<tr>
<td>colon (;)</td>
<td>%3a</td>
</tr>
<tr>
<td>comma (,)</td>
<td>%2c</td>
</tr>
<tr>
<td>dollar sign ($)</td>
<td>%24</td>
</tr>
<tr>
<td>equal sign (=)</td>
<td>%3d</td>
</tr>
<tr>
<td>plus sign (+)</td>
<td>%2b</td>
</tr>
<tr>
<td>question mark (?)</td>
<td>%3f</td>
</tr>
<tr>
<td>semi-colon (;)</td>
<td>%3b</td>
</tr>
<tr>
<td>slash (/)</td>
<td>%2f</td>
</tr>
</tbody>
</table>

**url_charset**

`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:

```sql
DECLARE
    result varchar2(400);
BEGIN
    result := UTL_URL.ESCAPE('http://www.example.com/Using the ESCAPE function.html');
    DBMS_OUTPUT.PUT_LINE(result);
END;
```
The resulting (escaped) URL is:

```
http://www.example.com/Using%20the%20ESCAPE%20function.html
```

If you include a value of TRUE for the escape_reserved_chars parameter when invoking the function:

```
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
```

### 3.24.2 UNESCAPE

The UNESCAPE function removes escape characters added to an URL by the ESCAPE function, converting the URL to its original form.

The signature is:

```
UNESCAPE(url VARCHAR2, url_charset VARCHAR2)
```

**Parameters**

`url`

`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 (unescape) URL is:

http://www.example.com/Using the UNESCAPE function.html
4 Acknowledgements

The PostgreSQL 8.3, 8.4, 9.0, 9.1, 9.2, 9.3, 9.4 and 9.5 Documentation provided the baseline for the portions of this guide that are common to PostgreSQL, and is hereby acknowledged:

Portions of this EnterpriseDB Software and Documentation may utilize the following copyrighted material, the use of which is hereby acknowledged.

PostgreSQL Documentation, Database Management System

PostgreSQL is Copyright © 1996-2016 by the PostgreSQL Global Development Group and is distributed under the terms of the license of the University of California below.

Postgres95 is Copyright © 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS-IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.