

**Database Compatibility for Oracle® Developers Built-in Package Guide**

EDB Postgres™ Advanced Server 10

November 1, 2017

Database Compatibility for Oracle® Developers   
Built-in Package Guide  
by EnterpriseDB® Corporation  
Copyright © 2007 - 2017 EnterpriseDB Corporation. All rights reserved.

EnterpriseDB Corporation, 34 Crosby Drive, Suite 100, Bedford, MA 01730, USA  
**T**  +1 781 357 3390 **F** +1 978 589 5701 **E** info@enterprisedb.com [**www**.enterprisedb.com](http://www.enterprisedb.com)

Table of Contents

[1 Introduction 8](#_Toc497305650)

[1.1 Typographical Conventions Used in this Guide 9](#_Toc497305651)

[2 Packages 10](#_Toc497305652)

[2.1 Package Components 10](#_Toc497305653)

[2.1.1 Package Specification Syntax 11](#_Toc497305654)

[2.1.2 Package Body Syntax 14](#_Toc497305655)

[2.2 Creating Packages 20](#_Toc497305656)

[2.2.1 Creating the Package Specification 20](#_Toc497305657)

[2.2.2 Creating the Package Body 20](#_Toc497305658)

[2.3 Referencing a Package 23](#_Toc497305659)

[2.4 Using Packages With User Defined Types 24](#_Toc497305660)

[2.5 Dropping a Package 28](#_Toc497305661)

[3 Built-In Packages 29](#_Toc497305662)

[3.1 DBMS\_ALERT 30](#_Toc497305663)

[3.1.1 REGISTER 31](#_Toc497305664)

[3.1.2 REMOVE 31](#_Toc497305665)

[3.1.3 REMOVEALL 32](#_Toc497305666)

[3.1.4 SIGNAL 32](#_Toc497305667)

[3.1.5 WAITANY 32](#_Toc497305668)

[3.1.6 WAITONE 34](#_Toc497305669)

[3.1.7 Comprehensive Example 35](#_Toc497305670)

[3.2 DBMS\_AQ 38](#_Toc497305671)

[3.2.1 ENQUEUE 39](#_Toc497305672)

[3.2.2 DEQUEUE 42](#_Toc497305673)

[3.2.3 REGISTER 46](#_Toc497305674)

[3.2.4 UNREGISTER 47](#_Toc497305675)

[3.3 DBMS\_AQADM 49](#_Toc497305676)

[3.3.1 ALTER\_QUEUE 50](#_Toc497305677)

[3.3.2 ALTER\_QUEUE\_TABLE 51](#_Toc497305678)

[3.3.3 CREATE\_QUEUE 52](#_Toc497305679)

[3.3.4 CREATE\_QUEUE\_TABLE 54](#_Toc497305680)

[3.3.5 DROP\_QUEUE 56](#_Toc497305681)

[3.3.6 DROP\_QUEUE\_TABLE 57](#_Toc497305682)

[3.3.7 PURGE\_QUEUE\_TABLE 58](#_Toc497305683)

[3.3.8 START\_QUEUE 59](#_Toc497305684)

[3.3.9 STOP\_QUEUE 59](#_Toc497305685)

[3.4 DBMS\_CRYPTO 61](#_Toc497305686)

[3.4.1 DECRYPT 62](#_Toc497305687)

[3.4.2 ENCRYPT 64](#_Toc497305688)

[3.4.3 HASH 66](#_Toc497305689)

[3.4.4 MAC 66](#_Toc497305690)

[3.4.5 RANDOMBYTES 67](#_Toc497305691)

[3.4.6 RANDOMINTEGER 68](#_Toc497305692)

[3.4.7 RANDOMNUMBER 69](#_Toc497305693)

[3.5 DBMS\_JOB 70](#_Toc497305694)

[3.5.1 BROKEN 71](#_Toc497305695)

[3.5.2 CHANGE 72](#_Toc497305696)

[3.5.3 INTERVAL 73](#_Toc497305697)

[3.5.4 NEXT\_DATE 73](#_Toc497305698)

[3.5.5 REMOVE 74](#_Toc497305699)

[3.5.6 RUN 74](#_Toc497305700)

[3.5.7 SUBMIT 75](#_Toc497305701)

[3.5.8 WHAT 76](#_Toc497305702)

[3.6 DBMS\_LOB 78](#_Toc497305703)

[3.6.1 APPEND 79](#_Toc497305704)

[3.6.2 COMPARE 80](#_Toc497305705)

[3.6.3 CONVERTTOBLOB 81](#_Toc497305706)

[3.6.4 CONVERTTOCLOB 82](#_Toc497305707)

[3.6.5 COPY 84](#_Toc497305708)

[3.6.6 ERASE 85](#_Toc497305709)

[3.6.7 GET\_STORAGE\_LIMIT 85](#_Toc497305710)

[3.6.8 GETLENGTH 86](#_Toc497305711)

[3.6.9 INSTR 86](#_Toc497305712)

[3.6.10 READ 87](#_Toc497305713)

[3.6.11 SUBSTR 88](#_Toc497305714)

[3.6.12 TRIM 88](#_Toc497305715)

[3.6.13 WRITE 89](#_Toc497305716)

[3.6.14 WRITEAPPEND 89](#_Toc497305717)

[3.7 DBMS\_LOCK 91](#_Toc497305718)

[3.7.1 SLEEP 91](#_Toc497305719)

[3.8 DBMS\_MVIEW 92](#_Toc497305720)

[3.8.1 GET\_MV\_DEPENDENCIES 93](#_Toc497305721)

[3.8.2 REFRESH 93](#_Toc497305722)

[3.8.3 REFRESH\_ALL\_MVIEWS 95](#_Toc497305723)

[3.8.4 REFRESH\_DEPENDENT 97](#_Toc497305724)

[3.9 DBMS\_OUTPUT 99](#_Toc497305725)

[3.9.1 CHARARR 99](#_Toc497305726)

[3.9.2 DISABLE 99](#_Toc497305727)

[3.9.3 ENABLE 100](#_Toc497305728)

[3.9.4 GET\_LINE 101](#_Toc497305729)

[3.9.5 GET\_LINES 103](#_Toc497305730)

[3.9.6 NEW\_LINE 104](#_Toc497305731)

[3.9.7 PUT 104](#_Toc497305732)

[3.9.8 PUT\_LINE 105](#_Toc497305733)

[3.9.9 SERVEROUTPUT 106](#_Toc497305734)

[3.10 DBMS\_PIPE 108](#_Toc497305735)

[3.10.1 CREATE\_PIPE 109](#_Toc497305736)

[3.10.2 NEXT\_ITEM\_TYPE 110](#_Toc497305737)

[3.10.3 PACK\_MESSAGE 112](#_Toc497305738)

[3.10.4 PURGE 112](#_Toc497305739)

[3.10.5 RECEIVE\_MESSAGE 114](#_Toc497305740)

[3.10.6 REMOVE\_PIPE 114](#_Toc497305741)

[3.10.7 RESET\_BUFFER 116](#_Toc497305742)

[3.10.8 SEND\_MESSAGE 117](#_Toc497305743)

[3.10.9 UNIQUE\_SESSION\_NAME 118](#_Toc497305744)

[3.10.10 UNPACK\_MESSAGE 118](#_Toc497305745)

[3.10.11 Comprehensive Example 119](#_Toc497305746)

[3.11 DBMS\_PROFILER 122](#_Toc497305747)

[3.11.1 FLUSH\_DATA 122](#_Toc497305748)

[3.11.2 GET\_VERSION 123](#_Toc497305749)

[3.11.3 INTERNAL\_VERSION\_CHECK 123](#_Toc497305750)

[3.11.4 PAUSE\_PROFILER 124](#_Toc497305751)

[3.11.5 RESUME\_PROFILER 124](#_Toc497305752)

[3.11.6 START\_PROFILER 124](#_Toc497305753)

[3.11.7 STOP\_PROFILER 125](#_Toc497305754)

[3.11.8 Using DBMS\_PROFILER 126](#_Toc497305755)

[3.11.8.1 Querying the DBMS\_PROFILER Tables and View 128](#_Toc497305756)

[3.11.8.2 DBMS\_PROFILER - Reference 135](#_Toc497305757)

[3.11.8.2.1 PLSQL\_PROFILER\_RUNS 135](#_Toc497305758)

[3.11.8.2.2 PLSQL\_PROFILER\_UNITS 135](#_Toc497305759)

[3.11.8.2.3 PLSQL\_PROFILER\_DATA 136](#_Toc497305760)

[3.11.8.2.4 PLSQL\_PROFILER\_RAWDATA 136](#_Toc497305761)

[3.12 DBMS\_RANDOM 141](#_Toc497305762)

[3.12.1 INITIALIZE 141](#_Toc497305763)

[3.12.2 NORMAL 142](#_Toc497305764)

[3.12.3 RANDOM 142](#_Toc497305765)

[3.12.4 SEED 143](#_Toc497305766)

[3.12.5 SEED 143](#_Toc497305767)

[3.12.6 STRING 144](#_Toc497305768)

[3.12.7 TERMINATE 144](#_Toc497305769)

[3.12.8 VALUE 145](#_Toc497305770)

[3.12.9 VALUE 145](#_Toc497305771)

[3.13 DBMS\_RLS 147](#_Toc497305772)

[3.13.1 ADD\_POLICY 150](#_Toc497305773)

[3.13.2 DROP\_POLICY 157](#_Toc497305774)

[3.13.3 ENABLE\_POLICY 158](#_Toc497305775)

[3.14 DBMS\_SCHEDULER 160](#_Toc497305776)

[3.14.1 Using Calendar Syntax to Specify a Repeating Interval 161](#_Toc497305777)

[3.14.2 CREATE\_JOB 163](#_Toc497305778)

[3.14.3 CREATE\_PROGRAM 166](#_Toc497305779)

[3.14.4 CREATE\_SCHEDULE 168](#_Toc497305780)

[3.14.5 DEFINE\_PROGRAM\_ARGUMENT 170](#_Toc497305781)

[3.14.6 DISABLE 172](#_Toc497305782)

[3.14.7 DROP\_JOB 173](#_Toc497305783)

[3.14.8 DROP\_PROGRAM 174](#_Toc497305784)

[3.14.9 DROP\_PROGRAM\_ARGUMENT 175](#_Toc497305785)

[3.14.10 DROP\_SCHEDULE 176](#_Toc497305786)

[3.14.11 ENABLE 176](#_Toc497305787)

[3.14.12 EVALUATE\_CALENDAR\_STRING 177](#_Toc497305788)

[3.14.13 RUN\_JOB 179](#_Toc497305789)

[3.14.14 SET\_JOB\_ARGUMENT\_VALUE 179](#_Toc497305790)

[3.15 DBMS\_SESSION 181](#_Toc497305791)

[3.15.1 SET\_ROLE 181](#_Toc497305792)

[3.16 DBMS\_SQL 182](#_Toc497305793)

[3.16.1 BIND\_VARIABLE 183](#_Toc497305794)

[3.16.2 BIND\_VARIABLE\_CHAR 185](#_Toc497305795)

[3.16.3 BIND VARIABLE RAW 185](#_Toc497305796)

[3.16.4 CLOSE\_CURSOR 186](#_Toc497305797)

[3.16.5 COLUMN\_VALUE 186](#_Toc497305798)

[3.16.6 COLUMN\_VALUE\_CHAR 188](#_Toc497305799)

[3.16.7 COLUMN VALUE RAW 188](#_Toc497305800)

[3.16.8 DEFINE\_COLUMN 189](#_Toc497305801)

[3.16.9 DEFINE\_COLUMN\_CHAR 191](#_Toc497305802)

[3.16.10 DEFINE COLUMN RAW 192](#_Toc497305803)

[3.16.11 DESCRIBE COLUMNS 192](#_Toc497305804)

[3.16.12 EXECUTE 193](#_Toc497305805)

[3.16.13 EXECUTE\_AND\_FETCH 194](#_Toc497305806)

[3.16.14 FETCH\_ROWS 195](#_Toc497305807)

[3.16.15 IS\_OPEN 197](#_Toc497305808)

[3.16.16 LAST\_ROW\_COUNT 197](#_Toc497305809)

[3.16.17 OPEN\_CURSOR 199](#_Toc497305810)

[3.16.18 PARSE 199](#_Toc497305811)

[3.17 DBMS\_UTILITY 202](#_Toc497305812)

[3.17.1 LNAME\_ARRAY 203](#_Toc497305813)

[3.17.2 UNCL\_ARRAY 203](#_Toc497305814)

[3.17.3 ANALYZE\_DATABASE, ANALYZE SCHEMA and ANALYZE PART\_OBJECT 203](#_Toc497305815)

[3.17.4 CANONICALIZE 206](#_Toc497305816)

[3.17.5 COMMA\_TO\_TABLE 207](#_Toc497305817)

[3.17.6 DB\_VERSION 208](#_Toc497305818)

[3.17.7 EXEC\_DDL\_STATEMENT 209](#_Toc497305819)

[3.17.8 FORMAT\_CALL\_STACK 210](#_Toc497305820)

[3.17.9 GET\_CPU\_TIME 210](#_Toc497305821)

[3.17.10 GET\_DEPENDENCY 211](#_Toc497305822)

[3.17.11 GET\_HASH\_VALUE 212](#_Toc497305823)

[3.17.12 GET\_PARAMETER\_VALUE 213](#_Toc497305824)

[3.17.13 GET\_TIME 214](#_Toc497305825)

[3.17.14 NAME\_TOKENIZE 215](#_Toc497305826)

[3.17.15 TABLE\_TO\_COMMA 217](#_Toc497305827)

[3.18 UTL\_ENCODE 219](#_Toc497305828)

[3.18.1 BASE64\_DECODE 219](#_Toc497305829)

[3.18.2 BASE64\_ENCODE 220](#_Toc497305830)

[3.18.3 MIMEHEADER\_DECODE 221](#_Toc497305831)

[3.18.4 MIMEHEADER\_ENCODE 222](#_Toc497305832)

[3.18.5 QUOTED\_PRINTABLE\_DECODE 223](#_Toc497305833)

[3.18.6 QUOTED\_PRINTABLE\_ENCODE 224](#_Toc497305834)

[3.18.7 TEXT\_DECODE 225](#_Toc497305835)

[3.18.8 TEXT\_ENCODE 226](#_Toc497305836)

[3.18.9 UUDECODE 227](#_Toc497305837)

[3.18.10 UUENCODE 228](#_Toc497305838)

[3.19 UTL\_FILE 231](#_Toc497305839)

[3.19.1 Setting File Permissions with utl\_file.umask 232](#_Toc497305840)

[3.19.2 FCLOSE 234](#_Toc497305841)

[3.19.3 FCLOSE\_ALL 234](#_Toc497305842)

[3.19.4 FCOPY 234](#_Toc497305843)

[3.19.5 FFLUSH 236](#_Toc497305844)

[3.19.6 FOPEN 237](#_Toc497305845)

[3.19.7 FREMOVE 238](#_Toc497305846)

[3.19.8 FRENAME 238](#_Toc497305847)

[3.19.9 GET\_LINE 240](#_Toc497305848)

[3.19.10 IS\_OPEN 241](#_Toc497305849)

[3.19.11 NEW\_LINE 242](#_Toc497305850)

[3.19.12 PUT 243](#_Toc497305851)

[3.19.13 PUT\_LINE 245](#_Toc497305852)

[3.19.14 PUTF 246](#_Toc497305853)

[3.20 UTL\_HTTP 248](#_Toc497305854)

[3.20.1 HTML\_PIECES 251](#_Toc497305855)

[3.20.2 REQ 251](#_Toc497305856)

[3.20.3 RESP 251](#_Toc497305857)

[3.20.4 BEGIN\_REQUEST 252](#_Toc497305858)

[3.20.5 END\_REQUEST 252](#_Toc497305859)

[3.20.6 END\_RESPONSE 253](#_Toc497305860)

[3.20.7 GET\_BODY\_CHARSET 253](#_Toc497305861)

[3.20.8 GET\_FOLLOW\_REDIRECT 254](#_Toc497305862)

[3.20.9 GET\_HEADER 254](#_Toc497305863)

[3.20.10 GET\_HEADER\_BY\_NAME 256](#_Toc497305864)

[3.20.11 GET\_HEADER\_COUNT 257](#_Toc497305865)

[3.20.12 GET\_RESPONSE 257](#_Toc497305866)

[3.20.13 GET\_RESPONSE\_ERROR\_CHECK 257](#_Toc497305867)

[3.20.14 GET\_TRANSFER\_TIMEOUT 258](#_Toc497305868)

[3.20.15 READ\_LINE 258](#_Toc497305869)

[3.20.16 READ\_RAW 260](#_Toc497305870)

[3.20.17 READ\_TEXT 261](#_Toc497305871)

[3.20.18 REQUEST 262](#_Toc497305872)

[3.20.19 REQUEST\_PIECES 262](#_Toc497305873)

[3.20.20 SET\_BODY\_CHARSET 263](#_Toc497305874)

[3.20.21 SET\_FOLLOW\_REDIRECT 263](#_Toc497305875)

[3.20.22 SET\_HEADER 264](#_Toc497305876)

[3.20.23 SET\_RESPONSE\_ERROR\_CHECK 264](#_Toc497305877)

[3.20.24 SET\_TRANSFER\_TIMEOUT 265](#_Toc497305878)

[3.20.25 WRITE\_LINE 265](#_Toc497305879)

[3.20.26 WRITE\_RAW 266](#_Toc497305880)

[3.20.27 WRITE\_TEXT 267](#_Toc497305881)

[3.21 UTL\_MAIL 269](#_Toc497305882)

[3.21.1 SEND 269](#_Toc497305883)

[3.21.2 SEND\_ATTACH\_RAW 270](#_Toc497305884)

[3.21.3 SEND\_ATTACH\_VARCHAR2 272](#_Toc497305885)

[3.22 UTL\_RAW 274](#_Toc497305886)

[3.22.1 CAST\_TO\_RAW 274](#_Toc497305887)

[3.22.2 CAST\_TO\_VARCHAR2 275](#_Toc497305888)

[3.22.3 CONCAT 276](#_Toc497305889)

[3.22.4 CONVERT 276](#_Toc497305890)

[3.22.5 LENGTH 277](#_Toc497305891)

[3.22.6 SUBSTR 278](#_Toc497305892)

[3.23 UTL\_SMTP 280](#_Toc497305893)

[3.23.1 CONNECTION 281](#_Toc497305894)

[3.23.2 REPLY/REPLIES 281](#_Toc497305895)

[3.23.3 CLOSE\_DATA 281](#_Toc497305896)

[3.23.4 COMMAND 282](#_Toc497305897)

[3.23.5 COMMAND\_REPLIES 282](#_Toc497305898)

[3.23.6 DATA 283](#_Toc497305899)

[3.23.7 EHLO 283](#_Toc497305900)

[3.23.8 HELO 284](#_Toc497305901)

[3.23.9 HELP 284](#_Toc497305902)

[3.23.10 MAIL 285](#_Toc497305903)

[3.23.11 NOOP 285](#_Toc497305904)

[3.23.12 OPEN\_CONNECTION 286](#_Toc497305905)

[3.23.13 OPEN\_DATA 286](#_Toc497305906)

[3.23.14 QUIT 287](#_Toc497305907)

[3.23.15 RCPT 287](#_Toc497305908)

[3.23.16 RSET 288](#_Toc497305909)

[3.23.17 VRFY 288](#_Toc497305910)

[3.23.18 WRITE\_DATA 288](#_Toc497305911)

[3.23.19 Comprehensive Example 289](#_Toc497305912)

[3.24 UTL\_URL 291](#_Toc497305913)

[3.24.1 ESCAPE 291](#_Toc497305914)

[3.24.2 UNESCAPE 293](#_Toc497305915)

[4 Acknowledgements 295](#_Toc497305916)

# 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>

## 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”.

# 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.

## Package Components

Packages consist of two main components:

* The package specification: This is the public interface, (these are the elements which can be referenced outside the package). We declare all database objects that are to be a part of our package within the specification.
* The package body: This contains the actual implementation of all the database objects declared within the package specification.

The package body implements the specifications in the package specification. It contains implementation details and private declarations which are invisible to the application. You can debug, enhance or replace a package body without changing the specifications. Similarly, you can change the body without recompiling the calling programs because the implementation details are invisible to the application.

### Package Specification Syntax

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

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

CREATE [ OR REPLACE ] PACKAGE package\_name

[ *authorization\_clause* ]

{ IS | AS }

[ declaration; ] ...

[ *procedure\_or\_function\_declaration* ] *...*

END [ *package\_name* ] ;

Where *authorization\_clause* :=

{ AUTHID DEFINER } | { AUTHID CURRENT\_USER }

Where *procedure\_or\_function\_declaration* :=

*procedure\_declaration* | *function\_declaration*

Where *procedure\_declaration* :=

PROCEDURE *proc\_name* [ *argument\_list* ];

[ *restriction\_pragma*; ]

Where *function\_declaration* :=

FUNCTION *func\_name* [ *argument\_list* ]

RETURN *rettype* [ DETERMINISTIC ];

[ *restriction\_pragma*; ]

Where *argument\_list* :=

( *argument\_declaration* [, ...] )

Where *argument\_declaration* :=

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

Where restriction\_pragma :=

PRAGMA RESTRICT\_REFERENCES(*name*, *restrictions*)

Where *restrictions* :=

*restriction* [, ... ]

**Parameters**

package\_name

package\_name is an identifier assigned to the package - each package must have a name unique within the schema.

AUTHID DEFINER

If you omit the AUTHID clause or specify AUTHID DEFINER, the privileges of the package owner are used to determine access privileges to database 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

TRUST

WNDS

WNPS

### Package Body Syntax

Package implementation details reside in the package body; the package body may contain objects that are not visible to the package user. Advanced Server supports the following syntax for the package body:

CREATE [ OR REPLACE ] PACKAGE BODY package\_name

{ IS | AS }

[ private\_declaration; ] ...

[ *procedure\_or\_function\_definition* ] *...*

[ *package\_initializer* ]

END [ *package\_name* ] ;

Where *procedure\_or\_function\_definition* :=

*procedure\_definition* | *function\_definition*

Where *procedure\_definition* :=

PROCEDURE *proc\_name*[ *argument\_list* ]

[ *options\_list* ]

{ IS | AS }

*procedure\_body*

END [ *proc\_name* ] ;

Where *procedure\_body* :=

[ 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.

DEFAULT value

The DEFAULT clause supplies a default value for an input argument if one is not supplied in the procedure call. DEFAULT may not be specified for arguments with modes IN OUT or OUT.

Please note: the following options are not compatible with Oracle databases; they are extensions to Oracle package syntax provided by Advanced Server only.

STRICT

The STRICT keyword specifies that the function will not be executed if called with a NULL argument; instead the function will return NULL.

LEAKPROOF

The LEAKPROOF keyword specifies that the function will not reveal any information about arguments, other than through a return value.

PARALLEL { UNSAFE | RESTRICTED | SAFE }

The PARALLEL clause enables the use of parallel sequential scans (parallel mode). A parallel sequential scan uses multiple workers to scan a relation in parallel during a query in contrast to a serial sequential scan.

When set to UNSAFE, the procedure or function cannot be executed in parallel mode. The presence of such a procedure or function forces a serial execution plan. This is the default setting if the PARALLEL clause is omitted.

When set to RESTRICTED, the procedure or function can be executed in parallel mode, but the execution is restricted to the parallel group leader. If the qualification for any particular relation has anything that is parallel restricted, that relation won't be chosen for parallelism.

When set to SAFE, the procedure or function can be executed in parallel mode with no restriction.

*execution\_cost*

*execution\_cost* specifies a positive number giving the estimated execution cost for the function, in units of cpu\_operator\_cost. If the function returns a set, this is the cost per returned row. The default is 0.0025.

*result\_rows*

*result\_rows* is the estimated number of rows that the query planner should expect the function to return. The default is 1000.

SET

Use the SET clause to specify a parameter value for the duration of the function:

*config\_param* specifies the parameter name.

*value* specifies the parameter value.

FROM CURRENT guarantees that the parameter value is restored when the function ends.

package\_initializer

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

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

## Creating Packages

A package is not an executable piece of code; rather it is a repository of code. When you use a package, you actually execute or make reference to an element within a package.

### Creating the Package Specification

The package specification contains the definition of all the elements in the package that can be referenced from outside of the package. These are called the public elements of the package, and they act as the package interface. The following code sample is a package specification:

--

-- Package specification for the 'emp\_admin' package.

--

CREATE OR REPLACE PACKAGE emp\_admin

IS

FUNCTION get\_dept\_name (

p\_deptno NUMBER DEFAULT 10

)

RETURN VARCHAR2;

FUNCTION update\_emp\_sal (

p\_empno NUMBER,

p\_raise NUMBER

)

RETURN NUMBER;

PROCEDURE hire\_emp (

p\_empno NUMBER,

p\_ename VARCHAR2,

p\_job VARCHAR2,

p\_sal NUMBER,

p\_hiredate DATE DEFAULT sysdate,

p\_comm NUMBER DEFAULT 0,

p\_mgr NUMBER,

p\_deptno NUMBER DEFAULT 10

);

PROCEDURE fire\_emp (

p\_empno NUMBER

);

END emp\_admin;

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

### Creating the Package Body

The body of the package contains the actual implementation behind the package specification. For the above emp\_admin package specification, we shall now create a package body which will implement the specifications. The body will contain the implementation of the functions and stored procedures in the specification.

--

-- Package body for the 'emp\_admin' package.

--

CREATE OR REPLACE PACKAGE BODY emp\_admin

IS

--

-- Function that queries the 'dept' table based on the department

-- number and returns the corresponding department name.

--

FUNCTION get\_dept\_name (

p\_deptno IN NUMBER DEFAULT 10

)

RETURN VARCHAR2

IS

v\_dname VARCHAR2(14);

BEGIN

SELECT dname INTO v\_dname FROM dept WHERE deptno = p\_deptno;

RETURN v\_dname;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Invalid department number ' || p\_deptno);

RETURN '';

END;

--

-- Function that updates an employee's salary based on the

-- employee number and salary increment/decrement passed

-- as IN parameters. Upon successful completion the function

-- returns the new updated salary.

--

FUNCTION update\_emp\_sal (

p\_empno IN NUMBER,

p\_raise IN NUMBER

)

RETURN NUMBER

IS

v\_sal NUMBER := 0;

BEGIN

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;

## 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.

## Using Packages With User Defined Types

The following example incorporates the various user-defined types discussed in earlier chapters within the context of a package.

The package specification of emp\_rpt shows the declaration of a record type, emprec\_typ, and a weakly-typed REF CURSOR, emp\_refcur, as publicly accessible along with two functions and two procedures. Function, open\_emp\_by\_dept, returns the REF CURSOR type, EMP\_REFCUR. Procedures, fetch\_emp and close\_refcur, both declare a weakly-typed REF CURSOR as a formal parameter.

CREATE OR REPLACE PACKAGE emp\_rpt

IS

TYPE emprec\_typ IS RECORD (

empno NUMBER(4),

ename VARCHAR(10)

);

TYPE emp\_refcur IS REF CURSOR;

FUNCTION get\_dept\_name (

p\_deptno IN NUMBER

) RETURN VARCHAR2;

FUNCTION open\_emp\_by\_dept (

p\_deptno IN emp.deptno%TYPE

) RETURN EMP\_REFCUR;

PROCEDURE fetch\_emp (

p\_refcur IN OUT SYS\_REFCURSOR

);

PROCEDURE close\_refcur (

p\_refcur IN OUT SYS\_REFCURSOR

);

END emp\_rpt;

The package body shows the declaration of several private variables - a static cursor, dept\_cur, a table type, depttab\_typ, a table variable, t\_dept, an integer variable, t\_dept\_max, and a record variable, r\_emp.

CREATE OR REPLACE PACKAGE BODY emp\_rpt

IS

CURSOR dept\_cur IS SELECT \* FROM dept;

TYPE depttab\_typ IS TABLE of dept%ROWTYPE

INDEX BY BINARY\_INTEGER;

t\_dept DEPTTAB\_TYP;

t\_dept\_max INTEGER := 1;

r\_emp EMPREC\_TYP;

FUNCTION get\_dept\_name (

p\_deptno IN NUMBER

) RETURN VARCHAR2

IS

BEGIN

FOR i IN 1..t\_dept\_max LOOP

IF p\_deptno = t\_dept(i).deptno THEN

RETURN t\_dept(i).dname;

END IF;

END LOOP;

RETURN 'Unknown';

END;

FUNCTION open\_emp\_by\_dept(

p\_deptno IN emp.deptno%TYPE

) RETURN EMP\_REFCUR

IS

emp\_by\_dept EMP\_REFCUR;

BEGIN

OPEN emp\_by\_dept FOR SELECT empno, ename FROM emp

WHERE deptno = p\_deptno;

RETURN emp\_by\_dept;

END;

PROCEDURE fetch\_emp (

p\_refcur IN OUT SYS\_REFCURSOR

)

IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE('EMPNO ENAME');

DBMS\_OUTPUT.PUT\_LINE('----- -------');

LOOP

FETCH p\_refcur INTO r\_emp;

EXIT WHEN p\_refcur%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(r\_emp.empno || ' ' || r\_emp.ename);

END LOOP;

END;

PROCEDURE close\_refcur (

p\_refcur IN OUT SYS\_REFCURSOR

)

IS

BEGIN

CLOSE p\_refcur;

END;

BEGIN

OPEN dept\_cur;

LOOP

FETCH dept\_cur INTO t\_dept(t\_dept\_max);

EXIT WHEN dept\_cur%NOTFOUND;

t\_dept\_max := t\_dept\_max + 1;

END LOOP;

CLOSE dept\_cur;

t\_dept\_max := t\_dept\_max - 1;

END emp\_rpt;

This package contains an initialization section that loads the private table variable, t\_dept, using the private static cursor, dept\_cur. t\_dept serves as a department name lookup table in function, get\_dept\_name.

Function, open\_emp\_by\_dept returns a REF CURSOR variable for a result set of employee numbers and names for a given department. This REF CURSOR variable can then be passed to procedure, fetch\_emp, to retrieve and list the individual rows of the result set. Finally, procedure, close\_refcur, can be used to close the REF CURSOR variable associated with this result set.

The following anonymous block runs the package function and procedures. In the anonymous block's declaration section, note the declaration of cursor variable, v\_emp\_cur, using the package’s public REF CURSOR type, EMP\_REFCUR. v\_emp\_cur contains the pointer to the result set that is passed between the package function and procedures.

DECLARE

v\_deptno dept.deptno%TYPE DEFAULT 30;

v\_emp\_cur emp\_rpt.EMP\_REFCUR;

BEGIN

v\_emp\_cur := emp\_rpt.open\_emp\_by\_dept(v\_deptno);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEES IN DEPT #' || v\_deptno ||

': ' || emp\_rpt.get\_dept\_name(v\_deptno));

emp\_rpt.fetch\_emp(v\_emp\_cur);

DBMS\_OUTPUT.PUT\_LINE('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

DBMS\_OUTPUT.PUT\_LINE(v\_emp\_cur%ROWCOUNT || ' rows were retrieved');

emp\_rpt.close\_refcur(v\_emp\_cur);

END;

The following is the result of this anonymous block.

EMPLOYEES IN DEPT #30: SALES

EMPNO ENAME

----- -------

7499 ALLEN

7521 WARD

7654 MARTIN

7698 BLAKE

7844 TURNER

7900 JAMES

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

6 rows were retrieved

The following anonymous block illustrates another means of achieving the same result. Instead of using the package procedures, fetch\_emp and close\_refcur, the logic of these programs is coded directly into the anonymous block. In the anonymous block’s declaration section, note the addition of record variable, r\_emp, declared using the package’s public record type, EMPREC\_TYP.

DECLARE

v\_deptno dept.deptno%TYPE DEFAULT 30;

v\_emp\_cur emp\_rpt.EMP\_REFCUR;

r\_emp emp\_rpt.EMPREC\_TYP;

BEGIN

v\_emp\_cur := emp\_rpt.open\_emp\_by\_dept(v\_deptno);

DBMS\_OUTPUT.PUT\_LINE('EMPLOYEES IN DEPT #' || v\_deptno ||

': ' || emp\_rpt.get\_dept\_name(v\_deptno));

DBMS\_OUTPUT.PUT\_LINE('EMPNO ENAME');

DBMS\_OUTPUT.PUT\_LINE('----- -------');

LOOP

FETCH v\_emp\_cur INTO r\_emp;

EXIT WHEN v\_emp\_cur%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(r\_emp.empno || ' ' ||

r\_emp.ename);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

DBMS\_OUTPUT.PUT\_LINE(v\_emp\_cur%ROWCOUNT || ' rows were retrieved');

CLOSE v\_emp\_cur;

END;

The following is the result of this anonymous block.

EMPLOYEES IN DEPT #30: SALES

EMPNO ENAME

----- -------

7499 ALLEN

7521 WARD

7654 MARTIN

7698 BLAKE

7844 TURNER

7900 JAMES

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

6 rows were retrieved

## 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;

# 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;

## DBMS\_ALERT

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

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

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

Advanced Server allows a maximum of 500 concurrent alerts. You can use the dbms\_alert.max\_alerts GUC variable (located in the postgresql.conf file) to specify the maximum number of concurrent alerts allowed on a system.

To set a value for the dbms\_alert.max\_alerts variable, open the postgresql.conf file (located by default in /opt/PostgresPlus/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

alert\_count specifies the maximum number of concurrent alerts. By default, the value of dbms\_alert.max\_alerts is 100. To disable this feature, set dbms\_alert.max\_alerts to 0.

For the dbms\_alert.max\_alerts GUC to function correctly, the custom\_variable\_classes parameter must contain dbms\_alerts:

custom\_variable\_classes = 'dbms\_alert, …'

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

### REGISTER

The REGISTER procedure enables the current session to be notified of the specified alert.

REGISTER(name VARCHAR2)

Parameters

*name*

Name of the alert to be registered.

Examples

The following anonymous block registers for an alert named, alert\_test, then waits for the signal.

DECLARE

v\_name VARCHAR2(30) := 'alert\_test';

v\_msg VARCHAR2(80);

v\_status INTEGER;

v\_timeout NUMBER(3) := 120;

BEGIN

DBMS\_ALERT.REGISTER(v\_name);

DBMS\_OUTPUT.PUT\_LINE('Registered for alert ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Waiting for signal...');

DBMS\_ALERT.WAITONE(v\_name,v\_msg,v\_status,v\_timeout);

DBMS\_OUTPUT.PUT\_LINE('Alert name : ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Alert msg : ' || v\_msg);

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('Alert timeout: ' || v\_timeout || ' seconds');

DBMS\_ALERT.REMOVE(v\_name);

END;

Registered for alert alert\_test

Waiting for signal...

### REMOVE

The REMOVE procedure unregisters the session for the named alert.

REMOVE(name VARCHAR2)

Parameters

name

Name of the alert to be unregistered.

### REMOVEALL

The REMOVEALL procedure unregisters the session for all alerts.

REMOVEALL

### SIGNAL

The SIGNAL procedure signals the occurrence of the named alert.

SIGNAL(name VARCHAR2, message VARCHAR2)

Parameters

name

Name of the alert.

message

Information to pass with this alert.

Examples

The following anonymous block signals an alert for alert\_test.

DECLARE

v\_name VARCHAR2(30) := 'alert\_test';

BEGIN

DBMS\_ALERT.SIGNAL(v\_name,'This is the message from ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Issued alert for ' || v\_name);

END;

Issued alert for alert\_test

### WAITANY

The WAITANY procedure waits for any of the registered alerts to occur.

WAITANY(name OUT VARCHAR2, message OUT VARCHAR2,

status OUT INTEGER, timeout NUMBER)

Parameters

name

Variable receiving the name of the alert.

message

Variable receiving the message sent by the SIGNAL procedure.

status

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

timeout

Time to wait for an alert in seconds.

Examples

The following anonymous block uses the WAITANY procedure to receive an alert named, alert\_test or any\_alert:

DECLARE

v\_name VARCHAR2(30);

v\_msg VARCHAR2(80);

v\_status INTEGER;

v\_timeout NUMBER(3) := 120;

BEGIN

DBMS\_ALERT.REGISTER('alert\_test');

DBMS\_ALERT.REGISTER('any\_alert');

DBMS\_OUTPUT.PUT\_LINE('Registered for alert alert\_test and any\_alert');

DBMS\_OUTPUT.PUT\_LINE('Waiting for signal...');

DBMS\_ALERT.WAITANY(v\_name,v\_msg,v\_status,v\_timeout);

DBMS\_OUTPUT.PUT\_LINE('Alert name : ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Alert msg : ' || v\_msg);

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('Alert timeout: ' || v\_timeout || ' seconds');

DBMS\_ALERT.REMOVEALL;

END;

Registered for alert alert\_test and any\_alert

Waiting for signal...

An anonymous block in a second session issues a signal for any\_alert:

DECLARE

v\_name VARCHAR2(30) := 'any\_alert';

BEGIN

DBMS\_ALERT.SIGNAL(v\_name,'This is the message from ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Issued alert for ' || v\_name);

END;

Issued alert for any\_alert

Control returns to the first anonymous block and the remainder of the code is executed:

Registered for alert alert\_test and any\_alert

Waiting for signal...

Alert name : any\_alert

Alert msg : This is the message from any\_alert

Alert status : 0

Alert timeout: 120 seconds

### WAITONE

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

WAITONE(name VARCHAR2, message OUT VARCHAR2,

status OUT INTEGER, timeout NUMBER)

Parameters

name

Name of the alert.

message

Variable receiving the message sent by the SIGNAL procedure.

status

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

timeout

Time to wait for an alert in seconds.

Examples

The following anonymous block is similar to the one used in the WAITANY example except the WAITONE procedure is used to receive the alert named, alert\_test.

DECLARE

v\_name VARCHAR2(30) := 'alert\_test';

v\_msg VARCHAR2(80);

v\_status INTEGER;

v\_timeout NUMBER(3) := 120;

BEGIN

DBMS\_ALERT.REGISTER(v\_name);

DBMS\_OUTPUT.PUT\_LINE('Registered for alert ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Waiting for signal...');

DBMS\_ALERT.WAITONE(v\_name,v\_msg,v\_status,v\_timeout);

DBMS\_OUTPUT.PUT\_LINE('Alert name : ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Alert msg : ' || v\_msg);

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('Alert timeout: ' || v\_timeout || ' seconds');

DBMS\_ALERT.REMOVE(v\_name);

END;

Registered for alert alert\_test

Waiting for signal...

Signal sent for alert\_test sent by an anonymous block in a second session:

DECLARE

v\_name VARCHAR2(30) := 'alert\_test';

BEGIN

DBMS\_ALERT.SIGNAL(v\_name,'This is the message from ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Issued alert for ' || v\_name);

END;

Issued alert for alert\_test

First session is alerted, control returns to the anonymous block, and the remainder of the code is executed:

Registered for alert alert\_test

Waiting for signal...

Alert name : alert\_test

Alert msg : This is the message from alert\_test

Alert status : 0

Alert timeout: 120 seconds

### Comprehensive Example

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

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

CREATE OR REPLACE TRIGGER dept\_alert\_trig

AFTER INSERT OR UPDATE OR DELETE ON dept

DECLARE

v\_action VARCHAR2(25);

BEGIN

IF INSERTING THEN

v\_action := ' added department(s) ';

ELSIF UPDATING THEN

v\_action := ' updated department(s) ';

ELSIF DELETING THEN

v\_action := ' deleted department(s) ';

END IF;

DBMS\_ALERT.SIGNAL('dept\_alert',USER || v\_action || 'on ' ||

SYSDATE);

END;

CREATE OR REPLACE TRIGGER emp\_alert\_trig

AFTER INSERT OR UPDATE OR DELETE ON emp

DECLARE

v\_action VARCHAR2(25);

BEGIN

IF INSERTING THEN

v\_action := ' added employee(s) ';

ELSIF UPDATING THEN

v\_action := ' updated employee(s) ';

ELSIF DELETING THEN

v\_action := ' deleted employee(s) ';

END IF;

DBMS\_ALERT.SIGNAL('emp\_alert',USER || v\_action || 'on ' ||

SYSDATE);

END;

The following anonymous block is executed in a session while updates to the dept and emp tables occur in other sessions:

DECLARE

v\_dept\_alert VARCHAR2(30) := 'dept\_alert';

v\_emp\_alert VARCHAR2(30) := 'emp\_alert';

v\_name VARCHAR2(30);

v\_msg VARCHAR2(80);

v\_status INTEGER;

v\_timeout NUMBER(3) := 60;

BEGIN

DBMS\_ALERT.REGISTER(v\_dept\_alert);

DBMS\_ALERT.REGISTER(v\_emp\_alert);

DBMS\_OUTPUT.PUT\_LINE('Registered for alerts dept\_alert and emp\_alert');

DBMS\_OUTPUT.PUT\_LINE('Waiting for signal...');

LOOP

DBMS\_ALERT.WAITANY(v\_name,v\_msg,v\_status,v\_timeout);

EXIT WHEN v\_status != 0;

DBMS\_OUTPUT.PUT\_LINE('Alert name : ' || v\_name);

DBMS\_OUTPUT.PUT\_LINE('Alert msg : ' || v\_msg);

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('------------------------------------' ||

'-------------------------');

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Alert status : ' || v\_status);

DBMS\_ALERT.REMOVEALL;

END;

Registered for alerts dept\_alert and emp\_alert

Waiting for signal...

The following changes are made by user, mary:

INSERT INTO dept VALUES (50,'FINANCE','CHICAGO');

INSERT INTO emp (empno,ename,deptno) VALUES (9001,'JONES',50);

INSERT INTO emp (empno,ename,deptno) VALUES (9002,'ALICE',50);

The following change is made by user, john:

INSERT INTO dept VALUES (60,'HR','LOS ANGELES');

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

Registered for alerts dept\_alert and emp\_alert

Waiting for signal...

Alert name : dept\_alert

Alert msg : mary added department(s) on 25-OCT-07 16:41:01

Alert status : 0

-------------------------------------------------------------

Alert name : emp\_alert

Alert msg : mary added employee(s) on 25-OCT-07 16:41:02

Alert status : 0

-------------------------------------------------------------

Alert name : dept\_alert

Alert msg : john added department(s) on 25-OCT-07 16:41:22

Alert status : 0

-------------------------------------------------------------

Alert status : 1

## DBMS\_AQ

EDB Postgres Advanced Server Advanced Queueing provides message queueing and message processing for the Advanced Server database. User-defined messages are stored in a queue; a collection of queues is stored in a queue table. Procedures in the DBMS\_AQADM package create and manage message queues and queue tables. Use the DBMS\_AQ package to add messages to a queue or remove messages from a queue, or register or unregister a PL/SQL callback procedure.

Advanced Server also provides extended (non-compatible) functionality for the DBMS\_AQ package with SQL commands. 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:

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| ENQUEUE | n/a | Post a message to a queue. |
| DEQUEUE | n/a | Retrieve a message from a queue if or when a message is available. |
| REGISTER | n/a | Register a callback procedure. |
| UNREGISTER | n/a | Unregister a callback procedure. |

Advanced Server's implementation of DBMS\_AQ is a partial implementation when compared to Oracle's version. Only those procedures listed in the table above are supported.

Advanced Server supports use of the constants listed below:

| Constant | Description | For Parameters |
| --- | --- | --- |
| DBMS\_AQ.BROWSE (0) | Read the message without locking. | dequeue\_options\_t.dequeue\_mode |
| DBMS\_AQ.LOCKED (1) | This constant is defined, but will return an error if used. | dequeue\_options\_t.dequeue\_mode |
| DBMS\_AQ.REMOVE (2) | Delete the message after reading; the default. | dequeue\_options\_t.dequeue\_mode |
| DBMS\_AQ.REMOVE\_NODATA (3) | This constant is defined, but will return an error if used. | dequeue\_options\_t.dequeue\_mode |
| DBMS\_AQ.FIRST\_MESSAGE (0) | Return the first available message that matches the search terms. | dequeue\_options\_t.navigation |
| DBMS\_AQ.NEXT\_MESSAGE (1) | Return the next available message that matches the search terms. | dequeue\_options\_t.navigation |
| DBMS\_AQ.NEXT\_TRANSACTION (2) | This constant is defined, but will return an error if used. | dequeue\_options\_t.navigation |
| DBMS\_AQ.FOREVER (0) | Wait forever if a message that matches the search term is not found, the default. | dequeue\_options\_t.wait |
| DBMS\_AQ.NO\_WAIT (1) | Do not wait if a message that matches the search term is not found. | dequeue\_options\_t.wait |
| DBMS\_AQ.ON\_COMMIT (0) | The dequeue is part of the current transaction. | enqueue\_options\_t.visibility, dequeue\_options\_t.visibility |
| DBMS\_AQ.IMMEDIATE (1) | This constant is defined, but will return an error if used. | enqueue\_options\_t.visibility, dequeue\_options\_t.visibility |
| DBMS\_AQ.PERSISTENT (0) | The message should be stored in a table. | enqueue\_options\_t.delivery\_mode |
| DBMS\_AQ.BUFFERED (1) | This constant is defined, but will return an error if used. | enqueue\_options\_t.delivery\_mode |
| DBMS\_AQ.READY (0) | Specifies that the message is ready to process. | message\_properties\_t.state |
| DBMS\_AQ.WAITING (1) | Specifies that the message is waiting to be processed. | message\_properties\_t.state |
| DBMS\_AQ.PROCESSED (2) | Specifies that the message has been processed. | message\_properties\_t.state |
| DBMS\_AQ.EXPIRED (3) | Specifies that the message is in the exception queue. | message\_properties\_t.state |
| DBMS\_AQ.NO\_DELAY (0) | This constant is defined, but will return an error if used | message\_properties\_t.delay |
| DBMS\_AQ.NEVER (NULL) | This constant is defined, but will return an error if used | message\_properties\_t.expiration |
| DBMS\_AQ.NAMESPACE\_AQ (0) | Accept notifications from DBMS\_AQ queues. | sys.aq$\_reg\_info.namespace |
| DBMS\_AQ.NAMESPACE\_ANONYMOUS (1) | This constant is defined, but will return an error if used | sys.aq$\_reg\_info.namespace |

### ENQUEUE

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

ENQUEUE(  
 *queue*\_*name* IN VARCHAR2,  
 *enqueue*\_*options* IN DBMS\_AQ.ENQUEUE\_OPTIONS\_T,  
 *message*\_*properties* IN DBMS\_AQ.MESSAGE\_PROPERTIES\_T,  
 *payload* IN <*type*\_*name*>,  
 *msgid* OUT RAW)

Parameters

*queue\_name*

The name (optionally schema-qualified) of an existing queue. If you omit the schema name, the server will use the schema specified in the SEARCH\_PATH. Please note that unlike Oracle, unquoted identifiers are converted to lower case before storing. To include special characters or use a case-sensitive name, enclose the name in double quotes.

For detailed information about creating a queue, 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:

|  |  |
| --- | --- |
| visibility | ON\_COMMIT. |
| delivery\_mode | PERSISTENT |
| sequence\_deviation | NULL |
| transformation | NULL |
| relative\_msgid | NULL |

*message\_properties*

*message\_properties* is a value of the type, message*\_*properties*\_*t:

message\_properties\_t IS RECORD(  
 priority 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:

|  |  |
| --- | --- |
| priority | If the queue table definition includes a sort\_list that references priority, this parameter affects the order that messages are dequeued. A lower value indicates a higher dequeue priority. |
| delay | Specify the number of seconds that will pass before a message is available for dequeueing or NO\_DELAY. |
| expiration | Use the expiration parameter to specify the number of seconds until a message expires. |
| correlation | Use correlation to specify a message that will be associated with the entry; the default is NULL. |
| attempts | This is a system-maintained value that specifies the number of attempts to dequeue the message. |
| recipient\_list | This parameter is not supported. |
| exception\_queue | Use the exception\_queue parameter to specify the name of an exception queue to which a message will be moved if it expires or is dequeued by a transaction that rolls back too many times. |
| enqueue\_time | enqueue\_time is the time the record was added to the queue; this value is provided by the system. |
| state | This parameter is maintained by DBMS\_AQ; state can be: DBMS\_AQ.WAITING – the delay has not been reached. DBMS\_AQ.READY – the queue entry is ready for processing. DBMS\_AQ.PROCESSED – the queue entry has been processed. DBMS\_AQ.EXPIRED – the queue entry has been moved to the exception queue. |
| sender\_id | This parameter is not supported; specify a value of NULL. |
| original\_msgid | This parameter is accepted for compatibility and ignored. |
| signature | This parameter is not supported; specify a value of NULL. |
| transaction\_group | This parameter is accepted for compatibility and ignored. |
| user\_property | This parameter is not supported; specify a value of NULL. |
| delivery\_mode | This parameter is not supported; specify a value of DBMS\_AQ.PERSISTENT. |

*payload*

Use the *payload* parameter to provide the data that will be associated with the queue entry. The payload type must match the type specified when creating the corresponding queue table (see DBMS\_AQADM.CREATE\_QUEUE\_TABLE).

*msgid*

Use the *msgid* parameter to retrieve a unique (system-generated) message identifier.

Example

The following anonymous block calls DBMS\_AQ.ENQUEUE, adding a message to a queue named work\_order:

DECLARE

enqueue\_options DBMS\_AQ.ENQUEUE\_OPTIONS\_T;

message\_properties DBMS\_AQ.MESSAGE\_PROPERTIES\_T;

message\_handle raw(16);

payload work\_order;

BEGIN

payload := work\_order('Smith', 'system upgrade');

DBMS\_AQ.ENQUEUE(

queue\_name => 'work\_order',

enqueue\_options => enqueue\_options,

message\_properties => message\_properties,

payload => payload,

msgid => message\_handle

);

END;

### DEQUEUE

The DEQUEUE procedure dequeues a message. The signature is:

DEQUEUE(  
 *queue*\_*name* IN VARCHAR2,  
 dequeue\_*options* IN DBMS\_AQ.DEQUEUE\_OPTIONS\_T,  
 *message*\_*properties* OUT DBMS\_AQ.MESSAGE\_PROPERTIES\_T,  
 *payload* OUT *type*\_*name*,  
 *msgid* OUT RAW)

Parameters

*queue\_name*

The name (optionally schema-qualified) of an existing queue. If you omit the schema name, the server will use the schema specified in the SEARCH\_PATH. Please note that unlike Oracle, unquoted identifiers are converted to lower case before storing. To include special characters or use a case-sensitive name, 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:

|  |  |
| --- | --- |
| consumer\_name | Must be NULL. |
| dequeue\_mode | The locking behavior of the dequeue operation. Must be either:  DBMS\_AQ.BROWSE – Read the message without obtaining a lock. DBMS\_AQ.LOCKED – Read the message after acquiring a lock. DBMS\_AQ.REMOVE – Read the message before deleting the message. DBMS\_AQ.REMOVE\_NODATA – Read the message, but do not delete the message. |
| navigation | Identifies the message that will be retrieved. Must be either: FIRST\_MESSAGE – The first message within the queue that matches the search term.  NEXT\_MESSAGE – The next message that is available that matches the first term. |
| visibility | Must be ON\_COMMIT – if you roll back the current transaction the dequeued item will remain in the queue. |
| wait | Must be a number larger than 0, or:  DBMS\_AQ.FOREVER – Wait indefinitely. DBMS\_AQ.NO\_WAIT – Do not wait. |
| msgid | The message ID of the message that will be dequeued. |
| correlation | Accepted for compatibility, and ignored. |
| deq\_condition | A VARCHAR2 expression that evaluates to a BOOLEAN value, indicating if the message should be dequeued. |
| signature | Accepted for compatibility, and ignored. |
| transformation | Accepted for compatibility, and ignored. |
| delivery\_mode | Must be PERSISTENT; buffered messages are not supported at this time. |

*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:

|  |  |
| --- | --- |
| priority | If the queue table definition includes a sort\_list that references priority, this parameter affects the order that messages are dequeued. A lower value indicates a higher dequeue priority. |
| delay | Specify the number of seconds that will pass before a message is available for dequeueing or NO\_DELAY. |
| expiration | Use the expiration parameter to specify the number of seconds until a message expires. |
| correlation | Use correlation to specify a message that will be associated with the entry; the default is NULL. |
| attempts | This is a system-maintained value that specifies the number of attempts to dequeue the message. |
| recipient\_list | This parameter is not supported. |
| exception\_queue | Use the exception\_queue parameter to specify the name of an exception queue to which a message will be moved if it expires or is dequeued by a transaction that rolls back too many times. |
| enqueue\_time | enqueue\_time is the time the record was added to the queue; this value is provided by the system. |
| state | This parameter is maintained by DBMS\_AQ; state can be: DBMS\_AQ.WAITING – the delay has not been reached. DBMS\_AQ.READY – the queue entry is ready for processing. DBMS\_AQ.PROCESSED – the queue entry has been processed. DBMS\_AQ.EXPIRED – the queue entry has been moved to the exception queue. |
| sender\_id | This parameter is not supported; specify a value of NULL. |
| original\_msgid | This parameter is accepted for compatibility and ignored. |
| signature | This parameter is not supported; specify a value of NULL. |
| transaction\_group | This parameter is accepted for compatibility and ignored. |
| user\_property | This parameter is not supported; specify a value of NULL. |
| delivery\_mode | This parameter is not supported; specify a value of DBMS\_AQ.PERSISTENT. |

*payload*

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

*msgid*

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

Example

The following anonymous block calls DBMS\_AQ.DEQUEUE, retrieving a message from the queue and a payload:

DECLARE

dequeue\_options DBMS\_AQ.DEQUEUE\_OPTIONS\_T;

message\_properties DBMS\_AQ.MESSAGE\_PROPERTIES\_T;

message\_handle raw(16);

payload work\_order;

BEGIN

dequeue\_options.dequeue\_mode := DBMS\_AQ.BROWSE;

DBMS\_AQ.DEQUEUE(

queue\_name => 'work\_queue',

dequeue\_options => dequeue\_options,

message\_properties => message\_properties,

payload => payload,

msgid => message\_handle

);

DBMS\_OUTPUT.PUT\_LINE(

'The next work order is [' || payload.subject || '].'

);

END;

The payload is displayed by DBMS\_OUTPUT.PUT\_LINE.

### REGISTER

Use the REGISTER procedure to register an email address, procedure or URL that will be notified when an item is enqueued or dequeued. The signature is:

REGISTER(  
 *reg*\_*list* IN SYS.AQ$\_REG\_INFO\_LIST,  
 *count* IN NUMBER)

Parameters

*reg\_list*

*reg\_list* is a list of type AQ$\_REG\_INFO\_LIST; that provides information about each subscription that you would like to register. Each entry within the list is of the type AQ$\_REG\_INFO, and may contain:

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

*count*

*count* is the number of entries in *reg*\_*list*.

Example

The following anonymous block calls DBMS\_AQ.REGISTER, registering procedures that will be notified when an item is added to or removed from a queue. A set of attributes (of sys.aq$\_reg\_info type) is provided for each subscription identified in the DECLARE section:

DECLARE

subscription1 sys.aq$\_reg\_info;

subscription2 sys.aq$\_reg\_info;

subscription3 sys.aq$\_reg\_info;

subscriptionlist sys.aq$\_reg\_info\_list;

BEGIN

subscription1 := sys.aq$\_reg\_info('q', DBMS\_AQ.NAMESPACE\_AQ, 'plsql://assign\_worker?PR=0',HEXTORAW('FFFF'));

subscription2 := sys.aq$\_reg\_info('q', DBMS\_AQ.NAMESPACE\_AQ, 'plsql://add\_to\_history?PR=1',HEXTORAW('FFFF'));

subscription3 := sys.aq$\_reg\_info('q', DBMS\_AQ.NAMESPACE\_AQ, 'plsql://reserve\_parts?PR=2',HEXTORAW('FFFF'));

subscriptionlist := sys.aq$\_reg\_info\_list(subscription1, subscription2, subscription3);

dbms\_aq.register(subscriptionlist, 3);

commit;

END;

/

The subscriptionlist is of type sys.aq$\_reg\_info\_list, and contains the previously described sys.aq$\_reg\_info objects. The list name and an object count are passed to dbms\_aq.register.

### 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:

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

*count*

*count* is the number of entries in *reg*\_*list*.

Example

The following anonymous block calls DBMS\_AQ.UNREGISTER, disabling the notifications specified in the example for DBMS\_AQ.REGISTER:

DECLARE

subscription1 sys.aq$\_reg\_info;

subscription2 sys.aq$\_reg\_info;

subscription3 sys.aq$\_reg\_info;

subscriptionlist sys.aq$\_reg\_info\_list;

BEGIN

subscription1 := sys.aq$\_reg\_info('q', DBMS\_AQ.NAMESPACE\_AQ, 'plsql://assign\_worker?PR=0',HEXTORAW('FFFF'));

subscription2 := sys.aq$\_reg\_info('q', DBMS\_AQ.NAMESPACE\_AQ, 'plsql://add\_to\_history?PR=1',HEXTORAW('FFFF'));

subscription3 := sys.aq$\_reg\_info('q', DBMS\_AQ.NAMESPACE\_AQ, 'plsql://reserve\_parts?PR=2',HEXTORAW('FFFF'));

subscriptionlist := sys.aq$\_reg\_info\_list(subscription1, subscription2, subscription3);

dbms\_aq.unregister(subscriptionlist, 3);

commit;

END;

/

The subscriptionlist is of type sys.aq$\_reg\_info\_list, and contains the previously described sys.aq$\_reg\_info objects. The list name and an object count are passed to dbms\_aq.unregister.

## 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.

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| ALTER\_QUEUE | n/a | Modify an existing queue. |
| ALTER\_QUEUE\_TABLE | n/a | Modify an existing queue table. |
| CREATE\_QUEUE | n/a | Create a queue. |
| CREATE\_QUEUE\_TABLE | n/a | Create a queue table. |
| DROP\_QUEUE | n/a | Drop an existing queue. |
| DROP\_QUEUE\_TABLE | n/a | Drop an existing queue table. |
| PURGE\_QUEUE\_TABLE | n/a | Remove one or more messages from a queue table. |
| START\_QUEUE | n/a | Make a queue available for enqueueing and dequeueing procedures. |
| STOP\_QUEUE | n/a | Make a queue unavailable for enqueueing and dequeueing procedures |

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

Advanced Server supports use of the arguments listed below:

| Constant | Description | For Parameters |
| --- | --- | --- |
| DBMS\_AQADM.TRANSACTIONAL(1) | This constant is defined, but will return an error if used. | message\_grouping |
| DBMS\_AQADM.NONE(0) | Use to specify message grouping for a queue table. | message\_grouping |
| DBMS\_AQADM.NORMAL\_QUEUE(0) | Use with create\_queue to specify queue\_type. | queue\_type |
| DBMS\_AQADM.EXCEPTION\_QUEUE (1) | Use with create\_queue to specify queue\_type. | queue\_type |
| DBMS\_AQADM.INFINITE(-1) | Use with create\_queue to specify retention\_time. | retention\_time |
| DBMS\_AQADM.PERSISTENT (0) | The message should be stored in a table. | enqueue\_options\_t.delivery\_mode |
| DBMS\_AQADM.BUFFERED (1) | This constant is defined, but will return an error if used. | enqueue\_options\_t.delivery\_mode |
| DBMS\_AQADM.PERSISTENT\_OR\_BUFFERED (2) | This constant is defined, but will return an error if used. | enqueue\_options\_t.delivery\_mode |

### ALTER\_QUEUE

Use the ALTER\_QUEUE procedure to modify an existing queue. The signature is:

ALTER\_QUEUE(  
 *max*\_*retries* IN NUMBER DEFAULT NULL,  
 *retry*\_*delay* IN NUMBER DEFAULT 0  
 *retention*\_*time* IN NUMBER DEFAULT 0,  
 *auto*\_*commit* IN BOOLEAN DEFAULT TRUE)  
 *comment* IN VARCHAR2 DEFAULT NULL,

Parameters

*queue\_name*

The name of the new queue.

*max\_retries*

*max*\_*retries* specifies the maximum number of attempts to remove a message with a dequeue statement. The value of *max*\_*retries* is incremented with each ROLLBACK statement. When the number of failed attempts reaches the value specified by *max*\_*retries*, the message is moved to the exception queue. Specify 0 to indicate that no retries are allowed.

*retry\_delay*

*retry\_delay* specifies the number of seconds until a message is scheduled for re-processing after a ROLLBACK. Specify 0 to indicate that the message should be retried immediately (the default).

*retention\_time*

*retention\_time* specifies the length of time (in seconds) that a message will be stored after being dequeued. You can also specify 0 (the default) to indicate the message should not be retained after dequeueing, or INFINITE to retain the message forever.

*auto\_commit*

This parameter is accepted for compatibility and ignored.

*comment*

*comment* specifies a comment associated with the queue.

Example

The following command alters a queue named work\_order, setting the retry\_delay parameter to 5 seconds:

EXEC DBMS\_AQADM.ALTER\_QUEUE(queue\_name => 'work\_order', retry\_delay => 5);

### ALTER\_QUEUE\_TABLE

Use the ALTER\_QUEUE\_TABLE procedure to modify an existing queue table. The signature is:

ALTER\_QUEUE\_TABLE (  
 *queue\_table* IN VARCHAR2,  
 *comment* IN VARCHAR2 DEFAULT NULL,  
 *primary*\_*instance* IN BINARY\_INTEGER DEFAULT 0,  
 *secondary*\_*instance* IN BINARY\_INTEGER DEFAULT 0,

Parameters

*queue\_table*

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

*comment*

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

*primary\_instance*

*primary\_instance* is accepted for compatibility and stored, but is ignored.

*secondary\_instance*

*secondary\_instance* is accepted for compatibility, but is ignored.

Example

The following command modifies a queue table named work\_order\_table:

EXEC DBMS\_AQADM.ALTER\_QUEUE\_TABLE

(queue\_table => 'work\_order\_table', comment => 'This queue table contains work orders for the shipping department.');

The queue table is named work\_order\_table; the command adds a comment to the definition of the queue table.

### CREATE\_QUEUE

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

CREATE\_QUEUE(  
 queue\_name IN VARCHAR2  
 *queue\_table* IN VARCHAR2,  
 *queue*\_*type* IN BINARY\_INTEGER DEFAULT NORMAL\_QUEUE,  
 *max*\_*retries* IN NUMBER DEFAULT 5,  
 *retry*\_*delay* IN NUMBER DEFAULT 0  
 *retention*\_*time* IN NUMBER DEFAULT 0,  
 *dependency*\_*tracking* IN BOOLEAN DEFAULT FALSE,  
 *comment* IN VARCHAR2 DEFAULT NULL,  
 *auto*\_*commit* IN BOOLEAN DEFAULT TRUE)

Parameters

*queue\_name*

The name of the new queue.

*queue\_table*

The name of the table in which the new queue will reside.

*queue\_type*

The type of the new queue. The valid values for *queue*\_*type* are:

DBMS\_AQADM.NORMAL\_QUEUE – This value specifies a normal queue (the default).

DBMS\_AQADM.EXCEPTION\_QUEUE – This value specifies that the new queue is an exception queue. An exception queue will support only dequeue operations.

*max\_retries*

*max*\_*retries* specifies the maximum number of attempts to remove a message with a dequeue statement. The value of *max*\_*retries* is incremented with each ROLLBACK statement. When the number of failed attempts reaches the value specified by *max*\_*retries*, the message is moved to the exception queue. The default value for a system table is 0; the default value for a user created table is 5.

*retry\_delay*

*retry\_delay* specifies the number of seconds until a message is scheduled for re-processing after a ROLLBACK. Specify 0 to indicate that the message should be retried immediately (the default).

*retention\_time*

*retention\_time* specifies the length of time (in seconds) that a message will be stored after being dequeued. You can also specify 0 (the default) to indicate the message should not be retained after dequeueing, or INFINITE to retain the message forever.

*dependency\_tracking*

This parameter is accepted for compatibility and ignored.

*comment*

*comment* specifies a comment associated with the queue.

*auto\_commit*

This parameter is accepted for compatibility and ignored.

Example

The following anonymous block creates a queue named work\_order in the work\_order\_table table:

BEGIN

DBMS\_AQADM.CREATE\_QUEUE ( queue\_name => 'work\_order', queue\_table => 'work\_order\_table', comment => 'This queue contains pending work orders.');

END;

### CREATE\_QUEUE\_TABLE

Use the CREATE\_QUEUE\_TABLE procedure to create a queue table. The signature is:

CREATE\_QUEUE\_TABLE (  
 *queue\_table* IN VARCHAR2,  
 *queue*\_*payload*\_*type* IN VARCHAR2,  
 *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*

*secondary\_instance* is accepted for compatibility, but is ignored.

*compatible*

*compatible* is accepted for compatibility, but is ignored.

*secure*

*secure* is accepted for compatibility, but is ignored.

Example

The following anonymous block first creates a type (work\_order) with attributes that hold a name (a VARCHAR2), and a project description (a TEXT). The block then uses that type to create a queue table:

BEGIN

CREATE TYPE work\_order AS (name VARCHAR2, project TEXT, completed BOOLEAN);

EXEC DBMS\_AQADM.CREATE\_QUEUE\_TABLE

(queue\_table => 'work\_order\_table',

queue\_payload\_type => 'work\_order',

comment => 'Work order message queue table');

END;

The queue table is named work\_order\_table, and contains a payload of a type work\_order. A comment notes that this is the Work order message queue table.

### DROP\_QUEUE

Use the DROP\_QUEUE procedure to delete a queue. The signature is:

DROP\_QUEUE(  
 *queue*\_*name* IN VARCHAR2,  
 *auto*\_*commit* IN BOOLEAN DEFAULT TRUE)

Parameters

*queue\_name*

The name of the queue that you wish to drop.

*auto\_commit*

*auto\_commit* is accepted for compatibility, but is ignored.

Example

The following anonymous block drops the queue named work\_order:

BEGIN

DBMS\_AQADM.DROP\_QUEUE(queue\_name => 'work\_order');

END;

### DROP\_QUEUE\_TABLE

Use the DROP\_QUEUE\_TABLE procedure to delete a queue table. The signature is:

DROP\_QUEUE\_TABLE(  
 *queue*\_*table* IN VARCHAR2,  
 *force* IN BOOLEAN default FALSE,  
 *auto*\_*commit* IN BOOLEAN default TRUE)

Parameters

*queue\_table*

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

*force*

The *force* keyword determines the behavior of the DROP\_QUEUE\_TABLE command when dropping a table that contain entries:

If the target table contains entries and force is FALSE, the command will fail, and the server will issue an error.

If the target table contains entries and force is TRUE, the command will drop the table and any dependent objects.

*auto\_commit*

*auto\_commit* is accepted for compatibility, but is ignored.

Example

The following anonymous block drops a table named work\_order\_table:

BEGIN

DBMS\_AQADM.DROP\_QUEUE\_TABLE ('work\_order\_table', force => TRUE);

END;

### PURGE\_QUEUE\_TABLE

Use the PURGE\_QUEUE\_TABLE procedure to delete messages from a queue table. The signature is:

PURGE\_QUEUE\_TABLE(  
 *queue*\_*table* IN VARCHAR2,  
 *purge\_condition* IN VARCHAR2,  
 *purge\_options* IN aq$\_purge\_options\_t)

Parameters

*queue\_table*

*queue\_table* specifies the name of the queue table from which you are deleting a message.

*purge\_condition*

Use *purge*\_*condition* to specify a condition (a SQL WHERE clause) that the server will evaluate when deciding which messages to purge.

*purge\_options*

*purge\_options* is an object of the type aq$\_purge\_options\_t. An aq$\_purge\_options\_t object contains:

|  |  |  |
| --- | --- | --- |
| Attribute | Type | Description |
| block | Boolean | Specify TRUE if an exclusive lock should be held on all queues within the table; the default is FALSE. |
| delivery\_mode | INTEGER | *delivery*\_*mode* specifies the type of message that will be purged. The only accepted value is DBMS\_AQ.PERSISTENT. |

Example

The following anonymous block removes any messages from the work\_order\_table with a value in the completed column of YES:

DECLARE

purge\_options dbms\_aqadm.aq$\_purge\_options\_t;

BEGIN

dbms\_aqadm.purge\_queue\_table('work\_order\_table', 'completed = YES', purge\_options);

END;

### START\_QUEUE

Use the START\_QUEUE procedure to make a queue available for enqueuing and dequeueing. The signature is:

START\_QUEUE(  
 *queue*\_*name* IN VARCHAR2,  
 *enqueue* IN BOOLEAN DEFAULT TRUE,  
 *dequeue* IN BOOLEAN DEFAULT TRUE)

Parameters

*queue\_name*

*queue\_name* specifies the name of the queue that you are starting.

*enqueue*

Specify TRUE to enable enqueueing (the default), or FALSE to leave the current setting unchanged.

*dequeue*

Specify TRUE to enable dequeueing (the default), or FALSE to leave the current setting unchanged.

Example

The following anonymous block makes a queue named work\_order available for enqueueing:

BEGIN

DBMS\_AQADM.START\_QUEUE

(queue\_name => 'work\_order);

END;

### STOP\_QUEUE

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

STOP\_QUEUE(  
 *queue*\_*name* IN VARCHAR2,  
 *enqueue* IN BOOLEAN DEFAULT TRUE,  
 *dequeue* IN BOOLEAN DEFAULT TRUE,  
 *wait* IN BOOLEAN DEFAULT TRUE)

Parameters

*queue\_name*

*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.

## 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

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| DECRYPT(*src*, *typ*, *key*, *iv*) | RAW | Decrypts RAW data. |
| DECRYPT(*dst* INOUT, *src*, *typ*, *key*, *iv*) | N/A | Decrypts BLOB data. |
| DECRYPT(*dst* INOUT, *src*, *typ*, *key*, *iv*) | N/A | Decrypts CLOB data. |
| ENCRYPT(*src*, *typ*, *key*, *iv*) | RAW | Encrypts RAW data. |
| ENCRYPT(*dst* INOUT, *src*, *typ*, *key*, *iv*) | N/A | Encrypts BLOB data. |
| ENCRYPT(*dst* INOUT, *src*, *typ*, *key*, *iv*) | N/A | Encrypts CLOB data. |
| HASH(*src*, *typ*) | RAW | Applies a hash algorithm to RAW data. |
| HASH(*src*) | RAW | Applies a hash algorithm to CLOB data. |
| MAC(*src*, *typ*, *key*) | RAW | Returns the hashed MAC value of the given RAW data using the specified hash algorithm and key. |
| MAC(*src*, *typ*, *key*) | RAW | Returns the hashed MAC value of the given CLOB data using the specified hash algorithm and key. |
| RANDOMBYTES(*number*\_*bytes*) | RAW | Returns a specified number of cryptographically strong random bytes. |
| RANDOMINTEGER() | INTEGER | Returns a random INTEGER. |
| RANDOMNUMBER() | NUMBER | Returns a random NUMBER. |

DBMS\_CRYPTO functions and procedures support the following error messages:

ORA-28239 - DBMS\_CRYPTO.KeyNull

ORA-28829 - DBMS\_CRYPTO.CipherSuiteNull

ORA-28827 - DBMS\_CRYPTO.CipherSuiteInvalid

Unlike Oracle, Advanced Server will *not* return error ORA-28233 if you re-encrypt previously encrypted information.

Please note that RAW and BLOB are synonyms for the PostgreSQL BYTEA data type, and CLOB is a synonym for TEXT.

### 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 | |
| ENCRYPT\_DES | CONSTANT INTEGER := 1; |
| ENCRYPT\_3DES | CONSTANT INTEGER := 3; |
| ENCRYPT\_AES | CONSTANT INTEGER := 4; |
| ENCRYPT\_AES128 | CONSTANT INTEGER := 6; |
| Block Cipher Modifiers | |
| CHAIN\_CBC | CONSTANT INTEGER := 256; |
| CHAIN\_ECB | CONSTANT INTEGER := 768; |
| Block Cipher Padding Modifiers | |
| PAD\_PKCS5 | CONSTANT INTEGER := 4096; |
| PAD\_NONE | CONSTANT INTEGER := 8192; |
| Block Cipher Suites | |
| DES\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_DES + CHAIN\_CBC + PAD\_PKCS5; |
| DES3\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_3DES + CHAIN\_CBC + PAD\_PKCS5; |
| AES\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_AES + CHAIN\_CBC + PAD\_PKCS5; |

*key*

*key* specifies the user-defined decryption key. This should match the key specified when the *src* was encrypted.

*iv*

*iv* (optional) specifies an initialization vector. If an initialization vector was specified when the *src* was encrypted, you must specify an initialization vector when decrypting the *src*. The default is NULL.

Examples

The following example uses the DBMS\_CRYPTO.DECRYPT function to decrypt an encrypted password retrieved from the passwords table:

CREATE TABLE passwords  
(  
 principal VARCHAR2(90) PRIMARY KEY, -- username  
 ciphertext RAW(9) -- encrypted password  
);

CREATE FUNCTION get\_password(username VARCHAR2) RETURN RAW AS  
 typ INTEGER := DBMS\_CRYPTO.DES\_CBC\_PKCS5;  
 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.

### 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 | |
| ENCRYPT\_DES | CONSTANT INTEGER := 1; |
| ENCRYPT\_3DES | CONSTANT INTEGER := 3; |
| ENCRYPT\_AES | CONSTANT INTEGER := 4; |
| ENCRYPT\_AES128 | CONSTANT INTEGER := 6; |
| Block Cipher Modifiers | |
| CHAIN\_CBC | CONSTANT INTEGER := 256; |
| CHAIN\_ECB | CONSTANT INTEGER := 768; |
| Block Cipher Padding Modifiers | |
| PAD\_PKCS5 | CONSTANT INTEGER := 4096; |
| PAD\_NONE | CONSTANT INTEGER := 8192; |
| Block Cipher Suites | |
| DES\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_DES + CHAIN\_CBC + PAD\_PKCS5; |
| DES3\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_3DES + CHAIN\_CBC + PAD\_PKCS5; |
| AES\_CBC\_PKCS5 | CONSTANT INTEGER := ENCRYPT\_AES + CHAIN\_CBC + PAD\_PKCS5; |

*key*

*key* specifies the encryption key.

*iv*

*iv* (optional) specifies an initialization vector. By default, iv is NULL.

Examples

The following example uses the DBMS\_CRYPTO.DES\_CBC\_PKCS5 Block Cipher Suite (a pre-defined set of algorithms and modifiers) to encrypt a value retrieved from the passwords table:

CREATE TABLE passwords  
(  
 principal VARCHAR2(90) PRIMARY KEY, -- username  
 ciphertext RAW(9) -- encrypted password  
);  
CREATE PROCEDURE set\_password(username VARCHAR2, cleartext RAW) AS  
 typ INTEGER := DBMS\_CRYPTO.DES\_CBC\_PKCS5;  
 key RAW(128) := 'my secret key';  
 iv RAW(100) := 'my initialization vector';  
 encrypted RAW(2048);  
BEGIN  
 encrypted := dbms\_crypto.encrypt(cleartext, typ, key, iv);   
 UPDATE passwords SET ciphertext = encrypted WHERE principal = username;  
END;

ENCRYPT uses a key value of my secret key and an initialization vector of my initialization vector when encrypting the password; specify the same key and initialization vector when decrypting the password.

### 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:

HASH  
 (*src* IN RAW, *typ* IN INTEGER) RETURN RAW

HASH  
 (*src* IN CLOB, *typ* IN INTEGER) RETURN RAW

Parameters

*src*

*src* specifies the value for which the hash value will be generated. You can specify a RAW, a BLOB, or a CLOB value.

*typ*

*typ* specifies the HASH function type. Advanced Server supports the HASH function types listed below:

|  |  |
| --- | --- |
| HASH Functions | |
| HASH\_MD4 | CONSTANT INTEGER := 1; |
| HASH\_MD5 | CONSTANT INTEGER := 2; |
| HASH\_SH1 | CONSTANT INTEGER := 3; |

Examples

The following example uses DBMS\_CRYPTO.HASH to find the md5 hash value of the string, cleartext source:

DECLARE  
 typ INTEGER := DBMS\_CRYPTO.HASH\_MD5;  
 hash\_value RAW(100);  
BEGIN  
  
 hash\_value := DBMS\_CRYPTO.HASH('cleartext source', typ);  
  
END;

### 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.

|  |  |
| --- | --- |
| MAC Functions | |
| HMAC\_MD5 | CONSTANT INTEGER := 1; |
| HMAC\_SH1 | CONSTANT INTEGER := 2; |

*key*

*key* specifies the key that will be used to calculate the hashed MAC value.

Examples

The following example finds the hashed MAC value of the string cleartext source:

DECLARE  
 typ INTEGER := DBMS\_CRYPTO.HMAC\_MD5;  
 key RAW(100) := 'my secret key';  
 mac\_value RAW(100);  
BEGIN  
  
 mac\_value := DBMS\_CRYPTO.MAC('cleartext source', typ, key);  
  
END;

DBMS\_CRYPTO.MAC uses a key value of my secret key when calculating the MAC value of cleartext source.

### 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;

### 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;

### RANDOMNUMBER

The RANDOMNUMBER() function returns a random NUMBER between 0 and 268,435,455. The signature is:

RANDOMNUMBER() RETURNS NUMBER

Examples

The following example uses the RANDOMNUMBER function to return a cryptographically strong random number:

DECLARE  
 result NUMBER;  
BEGIN  
 result := DBMS\_CRYPTO.RANDOMNUMBER();  
 DBMS\_OUTPUT.PUT\_LINE(result);  
END;

## 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:

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| BROKEN(job, broken [, next\_date ]) | n/a | Specify that a given job is either broken or not broken. |
| CHANGE(job, what, next\_date, interval, instance, force) | n/a | Change the job’s parameters. |
| INTERVAL(job, interval) | n/a | Set the execution frequency by means of a date function that is recalculated each time the job is run. This value becomes the next date/time for execution. |
| NEXT\_DATE(job, next\_date) | n/a | Set the next date/time the job is to be run. |
| REMOVE(job) | n/a | Delete the job definition from the database. |
| RUN(job) | n/a | Forces execution of a job even if it is marked broken. |
| SUBMIT(job OUT, what [, next\_date [, interval [, no\_parse ]]]) | n/a | Creates a job and stores its definition in the database. |
| WHAT(job, what) | n/a | Change the stored procedure run by a job. |

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

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

CREATE TABLE jobrun (

runtime VARCHAR2(40)

);

CREATE OR REPLACE PROCEDURE job\_proc

IS

BEGIN

INSERT INTO jobrun VALUES ('job\_proc run at ' || TO\_CHAR(SYSDATE,

'yyyy-mm-dd hh24:mi:ss'));

END;

### BROKEN

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

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

Parameters

job

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

broken

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

next\_date

Date/time when the job is to be run. The default is SYSDATE.

Examples

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

BEGIN

DBMS\_JOB.BROKEN(104,true);

END;

Change the state back to not broken:

BEGIN

DBMS\_JOB.BROKEN(104,false);

END;

### CHANGE

The CHANGE procedure modifies certain job attributes including the stored procedure to be run, the next date/time the job is to be run, and how often it is to be run.

CHANGE(job BINARY\_INTEGER what VARCHAR2, next\_date DATE,

interval VARCHAR2, *instance* BINARY\_INTEGER, *force* BOOLEAN)

Parameters

job

Identifier of the job to modify.

what

Stored procedure name. Set this parameter to null if the existing value is to remain unchanged.

next\_date

Date/time when the job is to be run next. Set this parameter to null if the existing value is to remain unchanged.

interval

Date function that when evaluated, provides the next date/time the job is to run. Set this parameter to null if the existing value is to remain unchanged.

instance

This argument is ignored, but is included for compatibility.

force

This argument is ignored, but is included for compatibility.

Examples

Change the job to run next on December 13, 2007. Leave other parameters unchanged.

BEGIN

DBMS\_JOB.CHANGE(104,NULL,TO\_DATE('13-DEC-07','DD-MON-YY'),NULL, NULL,

NULL);

END;

### INTERVAL

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

INTERVAL(job BINARY\_INTEGER, interval VARCHAR2)

Parameters

job

Identifier of the job to modify.

interval

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

Examples

Change the job to run once a week:

BEGIN

DBMS\_JOB.INTERVAL(104,'SYSDATE + 7');

END;

### NEXT\_DATE

The NEXT\_DATE procedure sets the date/time of when the job is to be run next.

NEXT\_DATE(job BINARY\_INTEGER, next\_date DATE)

Parameters

job

Identifier of the job whose next run date is to be set.

next\_date

Date/time when the job is to be run next.

Examples

Change the job to run next on December 14, 2007:

BEGIN

DBMS\_JOB.NEXT\_DATE(104, TO\_DATE('14-DEC-07','DD-MON-YY'));

END;

### 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;

### 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.

BEGIN

DBMS\_JOB.RUN(104);

END;

### SUBMIT

The SUBMIT procedure creates a job definition and stores it in the database. A job consists of a job identifier, the stored procedure to be executed, when the job is to be first run, and a date function that calculates the next date/time the job is to be run.

SUBMIT(job OUT BINARY\_INTEGER, what VARCHAR2

[, next\_date DATE [, interval VARCHAR2 [, no\_parse BOOLEAN ]]])

Parameters

job

Identifier assigned to the job.

what

Name of the stored procedure to be executed by the job.

next\_date

Date/time when the job is to be run next. The default is SYSDATE.

interval

Date function that when evaluated, provides the next date/time the job is to run. If interval is set to null, then the job is run only once. Null is the default.

no\_parse

If set to TRUE, do not syntax-check the stored procedure upon job creation – check only when the job first executes. If set to FALSE, check the procedure upon job creation. The default is FALSE.

Note: The *no\_parse* option is not supported in this implementation of SUBMIT(). It is included for compatibility only.

Examples

The following example creates a job using stored procedure, job\_proc. The job will execute immediately and run once a day thereafter as set by the interval parameter, SYSDATE + 1.

DECLARE

jobid INTEGER;

BEGIN

DBMS\_JOB.SUBMIT(jobid,'job\_proc;',SYSDATE,

'SYSDATE + 1');

DBMS\_OUTPUT.PUT\_LINE('jobid: ' || jobid);

END;

jobid: 104

The job immediately executes procedure, job\_proc, populating table, jobrun, with a row:

SELECT \* FROM jobrun;

runtime

-------------------------------------

job\_proc run at 2007-12-11 11:43:25

(1 row)

### 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;

## DBMS\_LOB

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

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| APPEND(dest\_lob IN OUT, src\_lob) | n/a | Appends one large object to another. |
| COMPARE(lob\_1, lob\_2 [, amount [, offset\_1 [, offset\_2 ]]]) | INTEGER | Compares two large objects. |
| CONVERTOBLOB(dest\_lob IN OUT, src\_clob, amount, dest\_offset IN OUT, src\_offset IN OUT, blob\_csid, lang\_context IN OUT, warning OUT) | n/a | Converts character data to binary. |
| CONVERTTOCLOB(dest\_lob IN OUT, src\_blob, amount, dest\_offset IN OUT, src\_offset IN OUT, blob\_csid, lang\_context IN OUT, warning OUT) | n/a | Converts binary data to character. |
| COPY(dest\_lob IN OUT, src\_lob, amount [, dest\_offset [, src\_offset ]]) | n/a | Copies one large object to another. |
| ERASE(lob\_loc IN OUT, amount IN OUT [, offset ]) | n/a | Erase a large object. |
| GET\_STORAGE\_LIMIT(lob\_loc) | INTEGER | Get the storage limit for large objects. |
| GETLENGTH(lob\_loc) | INTEGER | Get the length of the large object. |
| INSTR(lob\_loc, pattern [, offset [, nth ]]) | INTEGER | Get the position of the nth occurrence of a pattern in the large object starting at offset. |
| READ(lob\_loc, amount IN OUT, offset, buffer OUT) | n/a | Read a large object. |
| SUBSTR(lob\_loc [, amount [, offset ]]) | RAW, VARCHAR2 | Get part of a large object. |
| TRIM(lob\_loc IN OUT, newlen) | n/a | Trim a large object to the specified length. |
| WRITE(lob\_loc IN OUT, amount, offset, buffer) | n/a | Write data to a large object. |
| WRITEAPPEND(lob\_loc IN OUT, amount, buffer) | n/a | Write data from the buffer to the end of a large object. |

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

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

| Public Variables | Data Type | Value |
| --- | --- | --- |
| compress off | INTEGER | 0 |
| compress\_on | INTEGER | 1 |
| deduplicate\_off | INTEGER | 0 |
| deduplicate\_on | INTEGER | 4 |
| default\_csid | INTEGER | 0 |
| default\_lang\_ctx | INTEGER | 0 |
| encrypt\_off | INTEGER | 0 |
| encrypt\_on | INTEGER | 1 |
| file\_readonly | INTEGER | 0 |
| lobmaxsize | INTEGER | 1073741823 |
| lob\_readonly | INTEGER | 0 |
| lob\_readwrite | INTEGER | 1 |
| no\_warning | INTEGER | 0 |
| opt\_compress | INTEGER | 1 |
| opt\_deduplicate | INTEGER | 4 |
| opt\_encrypt | INTEGER | 2 |
| warn\_inconvertible\_char | INTEGER | 1 |

In the following sections, lengths and offsets are measured in bytes if the large objects are BLOBs. Lengths and offsets are measured in characters if the large objects are CLOBs.

### APPEND

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

APPEND(dest\_lob IN OUT { BLOB | CLOB }, src\_lob { BLOB | CLOB })

Parameters

dest\_lob

Large object locator for the destination object. Must be the same data type as src\_lob.

src\_lob

Large object locator for the source object. Must be the same data type as dest\_lob.

### COMPARE

The COMPARE procedure performs an exact byte-by-byte comparison of two large objects for a given length at given offsets. The large objects being compared must be the same data type.

status INTEGER COMPARE(lob\_1 { BLOB | CLOB },

lob\_2 { BLOB | CLOB }

[, amount INTEGER [, offset\_1 INTEGER [, offset\_2 INTEGER ]]])

Parameters

lob\_1

Large object locator of the first large object to be compared. Must be the same data type as lob\_2.

lob\_2

Large object locator of the second large object to be compared. Must be the same data type as lob\_1.

amount

If the data type of the large objects is BLOB, then the comparison is made for amount bytes. If the data type of the large objects is CLOB, then the comparison is made for amount characters. The default it the maximum size of a large object.

offset\_1

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

offset\_2

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

status

Zero if both large objects are exactly the same for the specified length for the specified offsets. Non-zero, if the objects are not the same. NULL if amount, offset\_1, or offset\_2 are less than zero.

### 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.

### 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.

### 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.

### 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.

### 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.

### 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.

### INSTR

The INSTR function returns the location of the nth occurrence of a given pattern within a large object.

position INTEGER INSTR(lob\_loc { BLOB | CLOB },

pattern { RAW | VARCHAR2 } [, offset INTEGER [, nth INTEGER ]])

Parameters

lob\_loc

Large object locator of the large object in which to search for pattern.

pattern

Pattern of bytes or characters to match against the large object, lob. pattern must be RAW if lob\_loc is a BLOB. pattern must be VARCHAR2 if lob\_loc is a CLOB.

offset

Position within lob\_loc to start search for pattern. The first byte/character is position 1. The default is 1.

nth

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

position

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

### READ

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

READ(lob\_loc { BLOB | CLOB }, amount IN OUT BINARY\_INTEGER,

offset INTEGER, buffer OUT { RAW | VARCHAR2 })

Parameters

lob\_loc

Large object locator of the large object to be read.

amount IN

Number of bytes/characters to read.

amount OUT

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

offset

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

buffer

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

### SUBSTR

The SUBSTR function provides the capability to return a portion of a large object.

data { RAW | VARCHAR2 } SUBSTR(lob\_loc { BLOB | CLOB }

[, amount INTEGER [, offset INTEGER ]])

Parameters

lob\_loc

Large object locator of the large object to be read.

amount

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

offset

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

data

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

### TRIM

The TRIM procedure provides the capability to truncate a large object to the specified length.

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

Parameters

lob\_loc

Large object locator of the large object to be trimmed.

newlen

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

### WRITE

The WRITE procedure provides the capability to write data into a large object. Any existing data in the large object at the specified offset for the given length is overwritten by data given in the buffer.

WRITE(lob\_loc IN OUT { BLOB | CLOB },  
 amount BINARY\_INTEGER,

offset INTEGER, buffer { RAW | VARCHAR2 })

Parameters

lob\_loc

Large object locator of the large object to be written.

amount

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

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.

### WRITEAPPEND

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

WRITEAPPEND(lob\_loc IN OUT { BLOB | CLOB },

amount BINARY\_INTEGER, buffer { RAW | VARCHAR2 })

Parameters

lob\_loc

Large object locator of the large object to which data is to be appended.

amount

Number of bytes/characters from buffer to be appended the large object.

buffer

Data to be appended to the large object. If lob\_loc is a BLOB, then buffer must be RAW. If lob\_loc is a CLOB, then buffer must be VARCHAR2.

## DBMS\_LOCK

Advanced Server provides support for the DBMS\_LOCK.SLEEP procedure.

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| SLEEP(seconds) | n/a | Suspends a session for the specified number of seconds. |

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

### SLEEP

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

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.

## DBMS\_MVIEW

Use procedures in the DBMS\_MVIEW package to manage and refresh materialized views and their dependencies. Advanced Server provides support for the following DBMS\_MVIEW procedures:

| Procedure | Return Type | Description |
| --- | --- | --- |
| GET\_MV\_DEPENDENCIES(*list* VARCHAR2, *deplist* VARCHAR2); | n/a | The GET\_MV\_DEPENDENCIES procedure returns a list of dependencies for a specified view. |
| REFRESH(*list* VARCHAR2, *method* VARCHAR2, *rollback\_seg* VARCHAR2 , *push\_deferred\_rpc* BOOLEAN, *refresh\_after\_errors* BOOLEAN , *purge\_option* NUMBER, *parallelism* NUMBER, *heap\_size* NUMBER , *atomic\_refresh* BOOLEAN , *nested* BOOLEAN); | n/a | This variation of the REFRESH procedure refreshes all views named in a comma-separated list of view names. |
| REFRESH(*tab* dbms\_utility.uncl\_array, *method* VARCHAR2, *rollback\_seg* VARCHAR2, *push\_deferred\_rpc* BOOLEAN, *refresh\_after\_errors* BOOLEAN, *purge\_option* NUMBER, *parallelism* NUMBER, *heap\_size* NUMBER, *atomic\_refresh* BOOLEAN, *nested* BOOLEAN); | n/a | This variation of the REFRESH procedure refreshes all views named in a table of dbms\_utility.uncl\_array values. |
| REFRESH\_ALL\_MVIEWS(*number\_of\_failures* BINARY\_INTEGER, *method* VARCHAR2, *rollback\_seg* VARCHAR2, *refresh\_after\_errors* BOOLEAN, *atomic\_refresh* BOOLEAN); | n/a | The REFRESH\_ALL\_MVIEWS procedure refreshes all materialized views. |
| REFRESH\_DEPENDENT(*number\_of\_failures* BINARY\_INTEGER, *list* VARCHAR2, *method* VARCHAR2, *rollback\_seg* VARCHAR2, *refresh\_after\_errors* BOOLEAN, *atomic\_refresh* BOOLEAN, *nested* BOOLEAN); | n/a | This variation of the REFRESH\_DEPENDENT procedure refreshes all views that are dependent on the views listed in a comma-separated list. |
| REFRESH\_DEPENDENT(*number\_of\_failures* BINARY\_INTEGER, *tab* dbms\_utility.uncl\_array, *method* VARCHAR2, *rollback\_seg* VARCHAR2, *refresh\_after\_errors* BOOLEAN, *atomic\_refresh* BOOLEAN, *nested* BOOLEAN); | n/a | This variation of the REFRESH\_DEPENDENT procedure refreshes all views that are dependent on the views listed in a table of dbms\_utility.uncl\_array values. |

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

### 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.

### REFRESH

Use the REFRESH procedure to refresh all views specified in either a comma-separated list of view names, or a table of DBMS\_UTILITY.UNCL\_ARRAY values. The procedure has two signatures; use the first form when specifying a comma-separated list of view names:

REFRESH(  
 *list* IN VARCHAR2,   
 *method* IN VARCHAR2 DEFAULT NULL,   
 *rollback\_seg* IN VARCHAR2 DEFAULT NULL,   
 *push\_deferred\_rpc* IN BOOLEAN DEFAULT TRUE,   
 *refresh\_after\_errors* IN BOOLEAN DEFAULT FALSE,   
 *purge\_option* IN NUMBER DEFAULT 1,   
 *parallelism* IN NUMBER DEFAULT 0,   
 *heap\_size* IN NUMBER DEFAULT 0,   
 *atomic\_refresh* IN BOOLEAN DEFAULT TRUE,   
 *nested* IN BOOLEAN DEFAULT FALSE);

Use the second form to specify view names in a table of DBMS\_UTILITY.UNCL\_ARRAY values:

REFRESH(  
 *tab* IN OUT DBMS\_UTILITY.UNCL\_ARRAY,   
 *method* IN VARCHAR2 DEFAULT NULL,   
 *rollback\_seg* IN VARCHAR2 DEFAULT NULL,   
 *push\_deferred\_rpc* IN BOOLEAN DEFAULT TRUE,   
 *refresh\_after\_errors* IN BOOLEAN DEFAULT FALSE,   
 *purge\_option* IN NUMBER DEFAULT 1,   
 *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');

### REFRESH\_ALL\_MVIEWS

Use the REFRESH\_ALL\_MVIEWS procedure to refresh any materialized views that have not been refreshed since the table or view on which the view depends has been modified. The signature is:

REFRESH\_ALL\_MVIEWS(  
 *number\_of\_failures* OUT BINARY\_INTEGER,   
 *method* IN VARCHAR2 DEFAULT NULL,   
 *rollback*\_*seg* IN VARCHAR2 DEFAULT NULL,   
 *refresh*\_*after*\_*errors* IN BOOLEAN DEFAULT FALSE,   
 *atomic\_refresh* IN BOOLEAN DEFAULT TRUE);

Parameters

*number\_of\_failures*

*number\_of\_failures* is a BINARY\_INTEGER that specifies the number of failures that occurred during the refresh operation.

*method*

method is a VARCHAR2 value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is C; this performs a complete refresh of the view.

*rollback\_seg*

*rollback*\_*seg* is accepted for compatibility and ignored. The default is NULL.

*refresh\_after\_errors*

*refresh\_after\_errors* is accepted for compatibility and ignored. The default is FALSE.

*atomic\_refresh*

*atomic\_refresh* is accepted for compatibility and ignored. The default is TRUE.

Examples

The following example performs a COMPLETE refresh on all materialized views:

DECLARE  
 errors INTEGER;  
BEGIN  
 DBMS\_MVIEW.REFRESH\_ALL\_MVIEWS(errors, method => 'C');  
END;

Upon completion, errors contains the number of failures.

### REFRESH\_DEPENDENT

Use the REFRESH\_DEPENDENT procedure to refresh all material views that are dependent on the views specified in the call to the procedure. You can specify a comma-separated list or provide the view names in a table of DBMS\_UTILITY.UNCL\_ARRAY values.

Use the first form of the procedure to refresh all material views that are dependent on the views specified in a comma-separated list:

REFRESH\_DEPENDENT(  
 *number\_of\_failures* OUT BINARY\_INTEGER,  
 *list* IN VARCHAR2,  
 *method* IN VARCHAR2 DEFAULT NULL,  
 *rollback\_seg* IN VARCHAR2 DEFAULT NULL  
 *refresh\_after\_errors* IN BOOLEAN DEFAULT FALSE,  
 *atomic\_refresh* IN BOOLEAN DEFAULT TRUE,  
 *nested* IN BOOLEAN DEFAULT FALSE);

Use the second form of the procedure to refresh all material views that are dependent on the views specified in a table of DBMS\_UTILITY.UNCL\_ARRAY values:

REFRESH\_DEPENDENT(  
 *number\_of\_failures* OUT BINARY\_INTEGER,   
 *tab* IN DBMS\_UTILITY.UNCL\_ARRAY,   
 *method* IN VARCHAR2 DEFAULT NULL,   
 *rollback\_seg* IN VARCHAR2 DEFAULT NULL,  
 *refresh\_after\_errors* IN BOOLEAN DEFAULT FALSE,   
 *atomic\_refresh* IN BOOLEAN DEFAULT TRUE,   
 *nested* IN BOOLEAN DEFAULT FALSE);

Parameters

*number\_of\_failures*

*number\_of\_failures* is a BINARY\_INTEGER that contains the number of failures that occurred during the refresh operation.

*list*

*list* is a VARCHAR2 value that specifies the name of a materialized view, or a comma-separated list of materialized view names. The names may be schema-qualified.

*tab*

*tab* is a table of DBMS\_UTILITY.UNCL\_ARRAY values that specify the name (or names) of a materialized view.

*method*

method is a VARCHAR2 value that specifies the refresh method that will be applied to the specified view (or views). The only supported method is C; this performs a complete refresh of the view.

*rollback\_seg*

*rollback*\_*seg* is accepted for compatibility and ignored. The default is NULL.

*refresh\_after\_errors*

*refresh\_after\_errors* is accepted for compatibility and ignored. The default is FALSE.

*atomic\_refresh*

*atomic\_refresh* is accepted for compatibility and ignored. The default is TRUE.

*nested*

*nested* is accepted for compatibility and ignored. The default is FALSE.

Examples

The following example performs a COMPLETE refresh on all materialized views dependent on a materialized view named emp\_view that resides in the public schema:

DECLARE  
 errors INTEGER;  
BEGIN  
 DBMS\_MVIEW.REFRESH\_DEPENDENT(errors, list => 'public.emp\_view', method => 'C');  
END;

Upon completion, errors contains the number of failures.

## DBMS\_OUTPUT

The DBMS\_OUTPUT package provides the capability to send messages (lines of text) to a message buffer, or get messages from the message buffer. A message buffer is local to a single session. Use the DBMS\_PIPE package to send messages between sessions.

The procedures and functions available in the DBMS\_OUTPUT package are listed in the following table.

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| DISABLE | n/a | Disable the capability to send and receive messages. |
| ENABLE(buffer\_size) | n/a | Enable the capability to send and receive messages. |
| GET\_LINE(line OUT, status OUT) | n/a | Get a line from the message buffer. |
| GET\_LINES(lines OUT, numlines IN OUT) | n/a | Get multiple lines from the message buffer. |
| NEW\_LINE | n/a | Puts an end-of-line character sequence. |
| PUT(item) | n/a | Puts a partial line without an end-of-line character sequence. |
| PUT\_LINE(item) | n/a | Puts a complete line with an end-of-line character sequence. |
| SERVEROUTPUT(stdout) | n/a | Direct messages from PUT, PUT\_LINE, or NEW\_LINE to either standard output or the message buffer. |

The following table lists the public variables available in the DBMS\_OUTPUT package.

| Public Variables | Data Type | Value | Description |
| --- | --- | --- | --- |
| chararr | TABLE |  | For message lines. |

### CHARARR

The CHARARR is for storing multiple message lines.

TYPE chararr IS TABLE OF VARCHAR2(32767) INDEX BY BINARY\_INTEGER;

### DISABLE

The DISABLE procedure clears out the message buffer. Any messages in the buffer at the time the DISABLE procedure is executed will no longer be accessible. Any messages subsequently sent with the PUT, PUT\_LINE, or NEW\_LINE procedures are discarded. No error is returned to the sender when the PUT, PUT\_LINE, or NEW\_LINE procedures are executed and messages have been disabled.

Use the ENABLE procedure or SERVEROUTPUT(TRUE) procedure to re-enable the sending and receiving of messages.

DISABLE

Examples

This anonymous block disables the sending and receiving messages in the current session.

BEGIN

DBMS\_OUTPUT.DISABLE;

END;

### ENABLE

The ENABLE procedure enables the capability to send messages to the message buffer or retrieve messages from the message buffer. Running SERVEROUTPUT(TRUE) also implicitly performs the ENABLE procedure.

The destination of a message sent with PUT, PUT\_LINE, or NEW\_LINE depends upon the state of SERVEROUTPUT.

* If the last state of SERVEROUTPUT is TRUE, the message goes to standard output of the command line.
* If the last state of SERVEROUTPUT is FALSE, the message goes to the message buffer.

ENABLE [ (buffer\_size INTEGER) ]

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.

BEGIN

DBMS\_OUTPUT.ENABLE;

DBMS\_OUTPUT.SERVEROUTPUT(TRUE);

DBMS\_OUTPUT.PUT\_LINE('Messages enabled');

END;

Messages enabled

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

BEGIN

DBMS\_OUTPUT.SERVEROUTPUT(TRUE);

DBMS\_OUTPUT.PUT\_LINE('Messages enabled');

END;

Messages enabled

The following anonymous block enables messages, but setting SERVEROUTPUT(FALSE) directs messages to the message buffer.

BEGIN

DBMS\_OUTPUT.ENABLE;

DBMS\_OUTPUT.SERVEROUTPUT(FALSE);

DBMS\_OUTPUT.PUT\_LINE('Message sent to buffer');

END;

### GET\_LINE

The GET\_LINE procedure provides the capability to retrieve a line of text from the message buffer. Only text that has been terminated by an end-of-line character sequence is retrieved – that is complete lines generated using PUT\_LINE, or by a series of PUT calls followed by a NEW\_LINE call.

GET\_LINE(line OUT VARCHAR2, status OUT INTEGER)

Parameters

line

Variable receiving the line of text from the message buffer.

status

0 if a line was returned from the message buffer, 1 if there was no line to return.

Examples

The following anonymous block writes the emp table out to the message buffer as a comma-delimited string for each row.

EXEC DBMS\_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE

v\_emprec VARCHAR2(120);

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

DBMS\_OUTPUT.ENABLE;

FOR i IN emp\_cur LOOP

v\_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||

NVL(LTRIM(TO\_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||

',' || i.sal || ',' ||

NVL(LTRIM(TO\_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

END LOOP;

END;

The following anonymous block reads the message buffer and inserts the messages written by the prior example into a table named messages. The rows in messages are then displayed.

CREATE TABLE messages (

status INTEGER,

msg VARCHAR2(100)

);

DECLARE

v\_line VARCHAR2(100);

v\_status INTEGER := 0;

BEGIN

DBMS\_OUTPUT.GET\_LINE(v\_line,v\_status);

WHILE v\_status = 0 LOOP

INSERT INTO messages VALUES(v\_status, v\_line);

DBMS\_OUTPUT.GET\_LINE(v\_line,v\_status);

END LOOP;

END;

SELECT msg FROM messages;

msg

-----------------------------------------------------------------

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

(14 rows)

### GET\_LINES

The GET\_LINES procedure provides the capability to retrieve one or more lines of text from the message buffer into a collection. Only text that has been terminated by an end-of-line character sequence is retrieved – that is complete lines generated using PUT\_LINE, or by a series of PUT calls followed by a NEW\_LINE call.

GET\_LINES(lines OUT CHARARR, numlines IN OUT INTEGER)

Parameters

lines

Table receiving the lines of text from the message buffer. See CHARARR for a description of lines.

numlines IN

Number of lines to be retrieved from the message buffer.

numlines OUT

Actual number of lines retrieved from the message buffer. If the output value of numlines is less than the input value, then there are no more lines left in the message buffer.

Examples

The following example uses the GET\_LINES procedure to store all rows from the emp table that were placed on the message buffer, into an array.

EXEC DBMS\_OUTPUT.SERVEROUTPUT(FALSE);

DECLARE

v\_emprec VARCHAR2(120);

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

DBMS\_OUTPUT.ENABLE;

FOR i IN emp\_cur LOOP

v\_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||

NVL(LTRIM(TO\_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||

',' || i.sal || ',' ||

NVL(LTRIM(TO\_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

END LOOP;

END;

DECLARE

v\_lines DBMS\_OUTPUT.CHARARR;

v\_numlines INTEGER := 14;

v\_status INTEGER := 0;

BEGIN

DBMS\_OUTPUT.GET\_LINES(v\_lines,v\_numlines);

FOR i IN 1..v\_numlines LOOP

INSERT INTO messages VALUES(v\_numlines, v\_lines(i));

END LOOP;

END;

SELECT msg FROM messages;

msg

-----------------------------------------------------------------

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

(14 rows)

### NEW\_LINE

The NEW\_LINE procedure writes an end-of-line character sequence in the message buffer.

NEW\_LINE

Parameters

The NEW\_LINE procedure expects no parameters.

### PUT

The PUT procedure writes a string to the message buffer. No end-of-line character sequence is written at the end of the string. Use the NEW\_LINE procedure to add an end-of-line character sequence.

PUT(item VARCHAR2)

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;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### PUT\_LINE

The PUT\_LINE procedure writes a single line to the message buffer including an end-of-line character sequence.

PUT\_LINE(item VARCHAR2)

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.

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;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### SERVEROUTPUT

The SERVEROUTPUT procedure provides the capability to direct messages to standard output of the command line or to the message buffer. Setting SERVEROUTPUT(TRUE) also performs an implicit execution of ENABLE.

The default setting of SERVEROUTPUT is implementation dependent. For example, in Oracle SQL\*Plus, SERVEROUTPUT(FALSE) is the default. In PSQL, SERVEROUTPUT(TRUE) is the default. Also note that in Oracle SQL\*Plus, this setting is controlled using the SQL\*Plus SET command, not by a stored procedure as implemented in Advanced Server.

SERVEROUTPUT(stdout BOOLEAN)

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

## DBMS\_PIPE

The DBMS\_PIPE package provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.

The procedures and functions available in the DBMS\_PIPE package are listed in the following table:

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| CREATE\_PIPE(pipename [, maxpipesize ] [, private ]) | INTEGER | Explicitly create a private pipe if private is “true” (the default) or a public pipe if private is “false”. |
| NEXT\_ITEM\_TYPE | INTEGER | Determine the data type of the next item in a received message. |
| PACK\_MESSAGE(item) | n/a | Place item in the session’s local message buffer. |
| PURGE(pipename) | n/a | Remove unreceived messages from the specified pipe. |
| RECEIVE\_MESSAGE(pipename [, timeout ]) | INTEGER | Get a message from a specified pipe. |
| REMOVE\_PIPE(pipename) | INTEGER | Delete an explicitly created pipe. |
| RESET\_BUFFER | n/a | Reset the local message buffer. |
| SEND\_MESSAGE(pipename [, timeout ] [, maxpipesize ]) | INTEGER | Send a message on a pipe. |
| UNIQUE\_SESSION\_NAME | VARCHAR2 | Obtain a unique session name. |
| UNPACK\_MESSAGE(item OUT) | n/a | Retrieve the next data item from a message into a type-compatible variable, item. |

Pipes are categorized as implicit or explicit. An implicit pipe is created if a reference is made to a pipe name that was not previously created by the CREATE\_PIPE function. For example, if the SEND\_MESSAGE function is executed using a non-existent pipe name, a new implicit pipe is created with that name. An explicit pipe is created using the CREATE\_PIPE function whereby the first parameter specifies the pipe name for the new pipe.

Pipes are also categorized as private or public. A private pipe can only be accessed by the user who created the pipe. Even a superuser cannot access a private pipe that was created by another user. A public pipe can be accessed by any user who has access to the DBMS\_PIPE package.

A public pipe can only be created by using the CREATE\_PIPE function with the third parameter set to FALSE. The CREATE\_PIPE function can be used to create a private pipe by setting the third parameter to TRUE or by omitting the third parameter. All implicit pipes are private.

The individual data items or “lines” of a message are first built-in a local message buffer, unique to the current session. The PACK\_MESSAGE procedure builds the message in the session’s local message buffer. The SEND\_MESSAGE function is then used to send the message through the pipe.

Receipt of a message involves the reverse operation. The RECEIVE\_MESSAGE function is used to get a message from the specified pipe. The message is written to the session’s local message buffer. The UNPACK\_MESSAGE procedure is then used to transfer the message data items from the message buffer to program variables. If a pipe contains multiple messages, RECEIVE\_MESSAGE gets the messages in FIFO (first-in-first-out) order.

Each session maintains separate message buffers for messages created with the PACK\_MESSAGE procedure and messages retrieved by the RECEIVE\_MESSAGE function. Thus messages can be both built and received in the same session. However, if consecutive RECEIVE\_MESSAGE calls are made, only the message from the last RECEIVE\_MESSAGE call will be preserved in the local message buffer.

### CREATE\_PIPE

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

status INTEGER CREATE\_PIPE(pipename VARCHAR2

[, maxpipesize INTEGER ] [, private BOOLEAN ])

Parameters

pipename

Name of the pipe.

maxpipesize

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

private

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

status

Status code returned by the operation. 0 indicates successful creation.

Examples

The following example creates a private pipe named messages:

DECLARE

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.CREATE\_PIPE('messages');

DBMS\_OUTPUT.PUT\_LINE('CREATE\_PIPE status: ' || v\_status);

END;

CREATE\_PIPE status: 0

The following example creates a public pipe named mailbox:

DECLARE

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.CREATE\_PIPE('mailbox',8192,FALSE);

DBMS\_OUTPUT.PUT\_LINE('CREATE\_PIPE status: ' || v\_status);

END;

CREATE\_PIPE status: 0

### 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 7‑3‑1 NEXT\_ITEM\_TYPE Data Type Codes

| Type Code | Data Type |
| --- | --- |
| 0 | No more data items |
| 9 | NUMBER |
| 11 | VARCHAR2 |
| 13 | DATE |
| 23 | RAW |

Note: The type codes list in the table are not compatible with Oracle databases. Oracle assigns a different numbering sequence to the data types.

Examples

The following example shows a pipe packed with a NUMBER item, a VARCHAR2 item, a DATE item, and a RAW item. A second anonymous block then uses the NEXT\_ITEM\_TYPE function to display the type code of each item.

DECLARE

v\_number NUMBER := 123;

v\_varchar VARCHAR2(20) := 'Character data';

v\_date DATE := SYSDATE;

v\_raw RAW(4) := '21222324';

v\_status INTEGER;

BEGIN

DBMS\_PIPE.PACK\_MESSAGE(v\_number);

DBMS\_PIPE.PACK\_MESSAGE(v\_varchar);

DBMS\_PIPE.PACK\_MESSAGE(v\_date);

DBMS\_PIPE.PACK\_MESSAGE(v\_raw);

v\_status := DBMS\_PIPE.SEND\_MESSAGE('datatypes');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

SEND\_MESSAGE status: 0

DECLARE

v\_number NUMBER;

v\_varchar VARCHAR2(20);

v\_date DATE;

v\_timestamp TIMESTAMP;

v\_raw RAW(4);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('datatypes');

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_number);

DBMS\_OUTPUT.PUT\_LINE('NUMBER Item : ' || v\_number);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_varchar);

DBMS\_OUTPUT.PUT\_LINE('VARCHAR2 Item : ' || v\_varchar);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_date);

DBMS\_OUTPUT.PUT\_LINE('DATE Item : ' || v\_date);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_raw);

DBMS\_OUTPUT.PUT\_LINE('RAW Item : ' || v\_raw);

DBMS\_OUTPUT.PUT\_LINE('----------------------------------');

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

DBMS\_OUTPUT.PUT\_LINE('NEXT\_ITEM\_TYPE: ' || v\_status);

DBMS\_OUTPUT.PUT\_LINE('---------------------------------');

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

RECEIVE\_MESSAGE status: 0

----------------------------------

NEXT\_ITEM\_TYPE: 9

NUMBER Item : 123

----------------------------------

NEXT\_ITEM\_TYPE: 11

VARCHAR2 Item : Character data

----------------------------------

NEXT\_ITEM\_TYPE: 13

DATE Item : 02-OCT-07 11:11:43

----------------------------------

NEXT\_ITEM\_TYPE: 23

RAW Item : 21222324

----------------------------------

NEXT\_ITEM\_TYPE: 0

### 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.

### PURGE

The PURGE procedure removes the unreceived messages from a specified implicit pipe.

PURGE(pipename VARCHAR2)

Use the REMOVE\_PIPE function to delete an explicit pipe.

Parameters

pipename

Name of the pipe.

Examples

Two messages are sent on a pipe:

DECLARE

v\_status INTEGER;

BEGIN

DBMS\_PIPE.PACK\_MESSAGE('Message #1');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.PACK\_MESSAGE('Message #2');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

END;

SEND\_MESSAGE status: 0

SEND\_MESSAGE status: 0

Receive the first message and unpack it:

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

DBMS\_OUTPUT.PUT\_LINE('Item: ' || v\_item);

END;

RECEIVE\_MESSAGE status: 0

Item: Message #1

Purge the pipe:

EXEC DBMS\_PIPE.PURGE('pipe');

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

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

END;

RECEIVE\_MESSAGE status: 1

### RECEIVE\_MESSAGE

The RECEIVE\_MESSAGE function obtains a message from a specified pipe.

status INTEGER RECEIVE\_MESSAGE(pipename VARCHAR2

[, timeout INTEGER ])

Parameters

pipename

Name of the pipe.

timeout

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

status

Status code returned by the operation.

The possible status codes are:

Table 7‑3‑2 RECEIVE\_MESSAGE Status Codes

| Status Code | Description |
| --- | --- |
| 0 | Success |
| 1 | Time out |
| 2 | Message too large .for the buffer |

### REMOVE\_PIPE

The REMOVE\_PIPE function deletes an explicit private or explicit public pipe.

status INTEGER REMOVE\_PIPE(pipename VARCHAR2)

Use the REMOVE\_PIPE function to delete explicitly created pipes – i.e., pipes created with the CREATE\_PIPE function.

Parameters

pipename

Name of the pipe.

status

Status code returned by the operation. A status code of 0 is returned even if the named pipe is non-existent.

Examples

Two messages are sent on a pipe:

DECLARE

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.CREATE\_PIPE('pipe');

DBMS\_OUTPUT.PUT\_LINE('CREATE\_PIPE status : ' || v\_status);

DBMS\_PIPE.PACK\_MESSAGE('Message #1');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.PACK\_MESSAGE('Message #2');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

END;

CREATE\_PIPE status : 0

SEND\_MESSAGE status: 0

SEND\_MESSAGE status: 0

Receive the first message and unpack it:

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

DBMS\_OUTPUT.PUT\_LINE('Item: ' || v\_item);

END;

RECEIVE\_MESSAGE status: 0

Item: Message #1

Remove the pipe:

SELECT DBMS\_PIPE.REMOVE\_PIPE('pipe') FROM DUAL;

remove\_pipe

-------------

0

(1 row)

Try to retrieve the next message. The RECEIVE\_MESSAGE call returns status code 1 indicating it timed out because the pipe had been deleted.

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

END;

RECEIVE\_MESSAGE status: 1

### RESET\_BUFFER

The RESET\_BUFFER procedure resets a “pointer” to the session’s local message buffer back to the beginning of the buffer. This has the effect of causing subsequent PACK\_MESSAGE calls to overwrite any data items that existed in the message buffer prior to the RESET\_BUFFER call.

RESET\_BUFFER

Examples

A message to John is written to the local message buffer. It is replaced by a message to Bob by calling RESET\_BUFFER. The message is sent on the pipe.

DECLARE

v\_status INTEGER;

BEGIN

DBMS\_PIPE.PACK\_MESSAGE('Hi, John');

DBMS\_PIPE.PACK\_MESSAGE('Can you attend a meeting at 3:00, today?');

DBMS\_PIPE.PACK\_MESSAGE('If not, is tomorrow at 8:30 ok with you?');

DBMS\_PIPE.RESET\_BUFFER;

DBMS\_PIPE.PACK\_MESSAGE('Hi, Bob');

DBMS\_PIPE.PACK\_MESSAGE('Can you attend a meeting at 9:30, tomorrow?');

v\_status := DBMS\_PIPE.SEND\_MESSAGE('pipe');

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE status: ' || v\_status);

END;

SEND\_MESSAGE status: 0

The message to Bob is in the received message.

DECLARE

v\_item VARCHAR2(80);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE('pipe',1);

DBMS\_OUTPUT.PUT\_LINE('RECEIVE\_MESSAGE status: ' || v\_status);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

DBMS\_OUTPUT.PUT\_LINE('Item: ' || v\_item);

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

DBMS\_OUTPUT.PUT\_LINE('Item: ' || v\_item);

END;

RECEIVE\_MESSAGE status: 0

Item: Hi, Bob

Item: Can you attend a meeting at 9:30, tomorrow?

### 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

| Status Code | Description |
| --- | --- |
| 0 | Success |
| 1 | Time out |
| 3 | Function interrupted |

### UNIQUE\_SESSION\_NAME

The UNIQUE\_SESSION\_NAME function returns a name, unique to the current session.

name VARCHAR2 UNIQUE\_SESSION\_NAME

Parameters

name

Unique session name.

Examples

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

DECLARE

v\_session VARCHAR2(30);

BEGIN

v\_session := DBMS\_PIPE.UNIQUE\_SESSION\_NAME;

DBMS\_OUTPUT.PUT\_LINE('Session Name: ' || v\_session);

END;

Session Name: PG$PIPE$5$2752

### UNPACK\_MESSAGE

The UNPACK\_MESSAGE procedure copies the data items of a message from the local message buffer to a specified program variable. The message must be placed in the local message buffer with the RECEIVE\_MESSAGE function before using UNPACK\_MESSAGE.

UNPACK\_MESSAGE(item OUT { DATE | NUMBER | VARCHAR2 | RAW })

Parameters

item

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

### Comprehensive Example

The following example uses a pipe as a “mailbox”. The procedures to create the mailbox, add a multi-item message to the mailbox (up to three items), and display the full contents of the mailbox are enclosed in a package named, mailbox.

CREATE OR REPLACE PACKAGE mailbox

IS

PROCEDURE create\_mailbox;

PROCEDURE add\_message (

p\_mailbox VARCHAR2,

p\_item\_1 VARCHAR2,

p\_item\_2 VARCHAR2 DEFAULT 'END',

p\_item\_3 VARCHAR2 DEFAULT 'END'

);

PROCEDURE empty\_mailbox (

p\_mailbox VARCHAR2,

p\_waittime INTEGER DEFAULT 10

);

END mailbox;

CREATE OR REPLACE PACKAGE BODY mailbox

IS

PROCEDURE create\_mailbox

IS

v\_mailbox VARCHAR2(30);

v\_status INTEGER;

BEGIN

v\_mailbox := DBMS\_PIPE.UNIQUE\_SESSION\_NAME;

v\_status := DBMS\_PIPE.CREATE\_PIPE(v\_mailbox,1000,FALSE);

IF v\_status = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Created mailbox: ' || v\_mailbox);

ELSE

DBMS\_OUTPUT.PUT\_LINE('CREATE\_PIPE failed - status: ' ||

v\_status);

END IF;

END create\_mailbox;

PROCEDURE add\_message (

p\_mailbox VARCHAR2,

p\_item\_1 VARCHAR2,

p\_item\_2 VARCHAR2 DEFAULT 'END',

p\_item\_3 VARCHAR2 DEFAULT 'END'

)

IS

v\_item\_cnt INTEGER := 0;

v\_status INTEGER;

BEGIN

DBMS\_PIPE.PACK\_MESSAGE(p\_item\_1);

v\_item\_cnt := 1;

IF p\_item\_2 != 'END' THEN

DBMS\_PIPE.PACK\_MESSAGE(p\_item\_2);

v\_item\_cnt := v\_item\_cnt + 1;

END IF;

IF p\_item\_3 != 'END' THEN

DBMS\_PIPE.PACK\_MESSAGE(p\_item\_3);

v\_item\_cnt := v\_item\_cnt + 1;

END IF;

v\_status := DBMS\_PIPE.SEND\_MESSAGE(p\_mailbox);

IF v\_status = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Added message with ' || v\_item\_cnt ||

' item(s) to mailbox ' || p\_mailbox);

ELSE

DBMS\_OUTPUT.PUT\_LINE('SEND\_MESSAGE in add\_message failed - ' ||

'status: ' || v\_status);

END IF;

END add\_message;

PROCEDURE empty\_mailbox (

p\_mailbox VARCHAR2,

p\_waittime INTEGER DEFAULT 10

)

IS

v\_msgno INTEGER DEFAULT 0;

v\_itemno INTEGER DEFAULT 0;

v\_item VARCHAR2(100);

v\_status INTEGER;

BEGIN

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE(p\_mailbox,p\_waittime);

WHILE v\_status = 0 LOOP

v\_msgno := v\_msgno + 1;

DBMS\_OUTPUT.PUT\_LINE('\*\*\*\*\*\* Start message #' || v\_msgno ||

' \*\*\*\*\*\*');

BEGIN

LOOP

v\_status := DBMS\_PIPE.NEXT\_ITEM\_TYPE;

EXIT WHEN v\_status = 0;

DBMS\_PIPE.UNPACK\_MESSAGE(v\_item);

v\_itemno := v\_itemno + 1;

DBMS\_OUTPUT.PUT\_LINE('Item #' || v\_itemno || ': ' ||

v\_item);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('\*\*\*\*\*\*\* End message #' || v\_msgno ||

' \*\*\*\*\*\*\*');

DBMS\_OUTPUT.PUT\_LINE('\*');

v\_itemno := 0;

v\_status := DBMS\_PIPE.RECEIVE\_MESSAGE(p\_mailbox,1);

END;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Number of messages received: ' || v\_msgno);

v\_status := DBMS\_PIPE.REMOVE\_PIPE(p\_mailbox);

IF v\_status = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('Deleted mailbox ' || p\_mailbox);

ELSE

DBMS\_OUTPUT.PUT\_LINE('Could not delete mailbox - status: '

|| v\_status);

END IF;

END empty\_mailbox;

END mailbox;

The following demonstrates the execution of the procedures in mailbox. The first procedure creates a public pipe using a name generated by the UNIQUE\_SESSION\_NAME function.

EXEC mailbox.create\_mailbox;

Created mailbox: PG$PIPE$13$3940

Using the mailbox name, any user in the same database with access to the mailbox package and DBMS\_PIPE package can add messages:

EXEC mailbox.add\_message('PG$PIPE$13$3940','Hi, John','Can you attend a meeting at 3:00, today?','-- Mary');

Added message with 3 item(s) to mailbox PG$PIPE$13$3940

EXEC mailbox.add\_message('PG$PIPE$13$3940','Don''t forget to submit your report','Thanks,','-- Joe');

Added message with 3 item(s) to mailbox PG$PIPE$13$3940

Finally, the contents of the mailbox can be emptied:

EXEC mailbox.empty\_mailbox('PG$PIPE$13$3940');

\*\*\*\*\*\* Start message #1 \*\*\*\*\*\*

Item #1: Hi, John

Item #2: Can you attend a meeting at 3:00, today?

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

## DBMS\_PROFILER

The DBMS\_PROFILER package collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a performance profiling session; use the functions and procedures listed below to control the profiling tool.

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| FLUSH\_DATA | Status Code or Exception | Flushes performance data collected in the current session without terminating the session (profiling continues). |
| GET\_VERSION(major OUT, minor OUT) | n/a | Returns the version number of this package. |
| INTERNAL\_VERSION\_CHECK | Status Code | Confirms that the current version of the profiler will work with the current database. |
| PAUSE\_PROFILER | Status Code or Exception | Pause data collection. |
| RESUME\_PROFILER | Status Code or Exception | Resume data collection. |
| START\_PROFILER(run\_comment, run\_comment1 [, run\_number OUT ]) | Status Code or Exception | Start data collection. |
| STOP\_PROFILER | Status Code or Exception | Stop data collection and flush performance data to the PLSQL\_PROFILER\_RAWDATA table. |

The functions within the DBMS\_PROFILER package return a status code to indicate success or failure; the DBMS\_PROFILER procedures raise an exception only if they encounter a failure. The status codes and messages returned by the functions, and the exceptions raised by the procedures are listed in the table below.

| Status Code | Message | Exception | Description |
| --- | --- | --- | --- |
| -1 | error version | version\_mismatch | The profiler version and the database are incompatible. |
| 0 | success | n/a | The operation completed successfully. |
| 1 | error\_param | profiler\_error | The operation received an incorrect parameter. |
| 2 | error\_io | profiler\_error | The data flush operation has failed. |

### FLUSH\_DATA

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

status INTEGER FLUSH\_DATA

FLUSH\_DATA

Parameters

status

Status code returned by the operation.

### GET\_VERSION

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

GET\_VERSION(major OUT INTEGER, minor OUT INTEGER)

Parameters

major

The major version number of DBMS\_PROFILER.

minor

The minor version number of DBMS\_PROFILER.

### INTERNAL\_VERSION\_CHECK

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

status INTEGER INTERNAL\_VERSION\_CHECK

Parameters

status

Status code returned by the operation.

### PAUSE\_PROFILER

The PAUSE\_PROFILER function/procedure pauses a profiling session. The function and procedure signatures are:

status INTEGER PAUSE\_PROFILER

PAUSE\_PROFILER

Parameters

status

Status code returned by the operation.

### RESUME\_PROFILER

The RESUME\_PROFILER function/procedure pauses a profiling session. The function and procedure signatures are:

status INTEGER RESUME\_PROFILER

RESUME\_PROFILER

Parameters

status

Status code returned by the operation.

### START\_PROFILER

The START\_PROFILER function/procedure starts a data collection session. The function and procedure signatures are:

status INTEGER START\_PROFILER(run\_comment TEXT := SYSDATE,

run\_comment1 TEXT := '' [, run\_number OUT INTEGER ])

START\_PROFILER(run\_comment TEXT := SYSDATE,

run\_comment1 TEXT := '' [, run\_number OUT INTEGER ])

Parameters

run\_comment

A user-defined comment for the profiler session. The default value is SYSDATE.

run\_comment1

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

run\_number

The session number of the profiler session.

status

Status code returned by the operation.

### STOP\_PROFILER

The STOP\_PROFILER function/procedure stops a profiling session and flushes the performance information to the DBMS\_PROFILER tables and view. The function and procedure signatures are:

status INTEGER STOP\_PROFILER

STOP\_PROFILER

Parameters

status

Status code returned by the operation.

### Using DBMS\_PROFILER

The DBMS\_PROFILER package collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a profiling session; you can review the performance information in the tables and views provided by the profiler.

DBMS\_PROFILER works by recording a set of performance-related counters and timers for each line of PL/pgSQL or SPL statement that executes within a profiling session. The counters and timers are stored in a table named SYS.PLSQL\_PROFILER\_DATA. When you complete a profiling session, DBMS\_PROFILER will write a row to the performance statistics table for each line of PL/pgSQL or SPL code that executed within the session. For example, if you execute the following function:

1 - CREATE OR REPLACE FUNCTION getBalance(acctNumber INTEGER)

2 - RETURNS NUMERIC AS $$

3 - DECLARE

4 - result NUMERIC;

5 - BEGIN

6 - SELECT INTO result balance FROM acct WHERE id = acctNumber;

7 -

8 - IF (result IS NULL) THEN

9 - RAISE INFO 'Balance is null';

10- END IF;

11-

12- RETURN result;

13- END;

14- $$ LANGUAGE 'plpgsql';

DBMS\_PROFILER adds one PLSQL\_PROFILER\_DATA entry for each line of code within the getBalance() function (including blank lines and comments). The entry corresponding to the SELECT statement executed exactly one time; and required a very small amount of time to execute. On the other hand, the entry corresponding to the RAISE INFO statement executed once or not at all (depending on the value for the balance column).

Some of the lines in this function contain no executable code so the performance statistics for those lines will always contain zero values.

To start a profiling session, invoke the DBMS\_PROFILER.START\_PROFILER function (or procedure). Once you've invoked START\_PROFILER, Advanced Server will profile every PL/pgSQL or SPL function, procedure, trigger, or anonymous block that your session executes until you either stop or pause the profiler (by calling STOP\_PROFILER or PAUSE\_PROFILER).

It is important to note that when you start (or resume) the profiler, the profiler will only gather performance statistics for functions/procedures/triggers that start after the call to START\_PROFILER (or RESUME\_PROFILER).

While the profiler is active, Advanced Server records a large set of timers and counters in memory; when you invoke the STOP\_PROFILER (or FLUSH\_DATA) function/procedure, DBMS\_PROFILER writes those timers and counters to a set of three tables:

* SYS.PLSQL\_PROFILER\_RAWDATA  
  Contains the performance counters and timers for each statement executed within the session.
* SYS.PLSQL\_PROFILER\_RUNS  
  Contains a summary of each run (aggregating the information found in PLSQL\_PROFILER\_RAWDATA).
* SYS.PLSQL\_PROFILER\_UNITS  
  Contains a summary of each code unit (function, procedure, trigger, or anonymous block) executed within a session.

In addition, DBMS\_PROFILER defines a view, SYS.PLSQL\_PROFILER\_DATA, which contains a subset of the PLSQL\_PROFILER\_RAWDATA table.

Please note that a non-superuser may gather profiling information, but may not view that profiling information unless a superuser grants specific privileges on the profiling tables (stored in the SYS schema). This permits a non-privileged user to gather performance statistics without exposing information that the administrator may want to keep secret.

#### Querying the DBMS\_PROFILER Tables and View

The following step-by-step example uses DBMS\_PROFILER to retrieve performance information for procedures, functions, and triggers included in the sample data distributed with Advanced Server.

1. Open the EDB-PSQL command line, and establish a connection to the Advanced Server database. Use an EXEC statement to start the profiling session:

acctg=# EXEC dbms\_profiler.start\_profiler('profile list\_emp');

EDB-SPL Procedure successfully completed

(Note: the call to start\_profiler() includes a comment that DBMS\_PROFILER associates with the profiler session).

1. 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)

1. Stop the profiling session with a call to dbms\_profiler.stop\_profiler:

acctg=# EXEC dbms\_profiler.stop\_profiler;

EDB-SPL Procedure successfully completed

1. Start a new session with the dbms\_profiler.start\_profiler function (followed by a new comment):

acctg=# EXEC dbms\_profiler.start\_profiler('profile get\_dept\_name and emp\_sal\_trig');

EDB-SPL Procedure successfully completed

1. Invoke the get\_dept\_name function:

acctg=# SELECT get\_dept\_name(10);

get\_dept\_name

---------------

ACCOUNTING

(1 row)

1. 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

1. Terminate the profiling session and flush the performance information to the profiling tables:

acctg=# EXEC dbms\_profiler.stop\_profiler;

EDB-SPL Procedure successfully completed

1. Now, query the plsql\_profiler\_runs table to view a list of the profiling sessions, arranged by runid:

acctg=# SELECT \* FROM plsql\_profiler\_runs;

runid | related\_run | run\_owner | run\_date | run\_comment | run\_total\_time | run\_system\_info | run\_comment1 | spare1

-------+-------------+--------------+---------------------------+----------------------------------------+----------------+-----------------+--------------+--------

1 | | enterprisedb | 04-FEB-14 09:32:48.874315 | profile list\_emp | 4154 | | |

2 | | enterprisedb | 04-FEB-14 09:41:30.546503 | profile get\_dept\_name and emp\_sal\_trig | 2088 | | |

(2 rows)

1. Query the plsql\_profiler\_units table to view the amount of time consumed by each unit (each function, procedure, or trigger):

acctg=# SELECT \* FROM plsql\_profiler\_units;

runid | unit\_number | unit\_type | unit\_owner | unit\_name | unit\_timestamp | total\_time | spare1 | spare2

-------+-------------+-----------+--------------+---------------------------------+----------------+------------+--------+--------

1 | 16999 | FUNCTION | enterprisedb | list\_emp() | | 4 | |

2 | 17002 | FUNCTION | enterprisedb | user\_audit\_trig() | | 1 | |

2 | 17000 | FUNCTION | enterprisedb | get\_dept\_name(p\_deptno numeric) | | 1 | |

2 | 17004 | FUNCTION | enterprisedb | emp\_sal\_trig() | | 1 | |

(4 rows)

1. Query the plsql\_profiler\_rawdata table to view a list of the wait event counters and wait event times:

acctg=# SELECT runid, sourcecode, func\_oid, line\_number, exec\_count, tuples\_returned, time\_total FROM plsql\_profiler\_rawdata;

runid | sourcecode | func\_oid | line\_number | exec\_count | tuples\_returned | time\_total

-------+------------------------------------------------------------------+----------+-------------+------------+-----------------+------------

1 | DECLARE | 16999 | 1 | 0 | 0 | 0

1 | v\_empno NUMERIC(4); | 16999 | 2 | 0 | 0 | 0

1 | v\_ename VARCHAR(10); | 16999 | 3 | 0 | 0 | 0

1 | emp\_cur CURSOR FOR | 16999 | 4 | 0 | 0 | 0

1 | SELECT empno, ename FROM memp ORDER BY empno; | 16999 | 5 | 0 | 0 | 0

1 | BEGIN | 16999 | 6 | 0 | 0 | 0

1 | OPEN emp\_cur; | 16999 | 7 | 0 | 0 | 0

1 | RAISE INFO 'EMPNO ENAME'; | 16999 | 8 | 1 | 0 | 0.001621

1 | RAISE INFO '----- -------'; | 16999 | 9 | 1 | 0 | 0.000301

1 | LOOP | 16999 | 10 | 1 | 0 | 4.6e-05

1 | FETCH emp\_cur INTO v\_empno, v\_ename; | 16999 | 11 | 1 | 0 | 0.001114

1 | EXIT WHEN NOT FOUND; | 16999 | 12 | 15 | 0 | 0.000206

1 | RAISE INFO '% %', v\_empno, v\_ename; | 16999 | 13 | 15 | 0 | 8.3e-05

1 | END LOOP; | 16999 | 14 | 14 | 0 | 0.000773

1 | CLOSE emp\_cur; | 16999 | 15 | 0 | 0 | 0

1 | RETURN; | 16999 | 16 | 1 | 0 | 1e-05

1 | END; | 16999 | 17 | 1 | 0 | 0

1 | | 16999 | 18 | 0 | 0 | 0

2 | DECLARE | 17002 | 1 | 0 | 0 | 0

2 | v\_action VARCHAR(24); | 17002 | 2 | 0 | 0 | 0

2 | v\_text TEXT; | 17002 | 3 | 0 | 0 | 0

2 | BEGIN | 17002 | 4 | 0 | 0 | 0

2 | IF TG\_OP = 'INSERT' THEN | 17002 | 5 | 0 | 0 | 0

2 | v\_action := ' added employee(s) on '; | 17002 | 6 | 1 | 0 | 0.000143

2 | ELSIF TG\_OP = 'UPDATE' THEN | 17002 | 7 | 0 | 0 | 0

2 | v\_action := ' updated employee(s) on '; | 17002 | 8 | 0 | 0 | 0

2 | ELSIF TG\_OP = 'DELETE' THEN | 17002 | 9 | 1 | 0 | 3.2e-05

2 | v\_action := ' deleted employee(s) on '; | 17002 | 10 | 0 | 0 | 0

2 | END IF; | 17002 | 11 | 0 | 0 | 0

2 | v\_text := 'User ' || USER || v\_action || CURRENT\_DATE; | 17002 | 12 | 0 | 0 | 0

2 | RAISE INFO ' %', v\_text; | 17002 | 13 | 1 | 0 | 0.000383

2 | RETURN NULL; | 17002 | 14 | 1 | 0 | 6.3e-05

2 | END; | 17002 | 15 | 1 | 0 | 3.6e-05

2 | | 17002 | 16 | 0 | 0 | 0

2 | DECLARE | 17000 | 1 | 0 | 0 | 0

2 | v\_dname VARCHAR(14); | 17000 | 2 | 0 | 0 | 0

2 | BEGIN | 17000 | 3 | 0 | 0 | 0

2 | SELECT INTO v\_dname dname FROM dept WHERE deptno = p\_deptno; | 17000 | 4 | 0 | 0 | 0

2 | RETURN v\_dname; | 17000 | 5 | 1 | 0 | 0.000647

2 | IF NOT FOUND THEN | 17000 | 6 | 1 | 0 | 2.6e-05

2 | RAISE INFO 'Invalid department number %', p\_deptno; | 17000 | 7 | 0 | 0 | 0

2 | RETURN ''; | 17000 | 8 | 0 | 0 | 0

2 | END IF; | 17000 | 9 | 0 | 0 | 0

2 | END; | 17000 | 10 | 0 | 0 | 0

2 | | 17000 | 11 | 0 | 0 | 0

2 | DECLARE | 17004 | 1 | 0 | 0 | 0

2 | sal\_diff NUMERIC(7,2); | 17004 | 2 | 0 | 0 | 0

2 | BEGIN | 17004 | 3 | 0 | 0 | 0

2 | IF TG\_OP = 'INSERT' THEN | 17004 | 4 | 0 | 0 | 0

2 | RAISE INFO 'Inserting employee %', NEW.empno; | 17004 | 5 | 1 | 0 | 8.4e-05

2 | RAISE INFO '..New salary: %', NEW.sal; | 17004 | 6 | 0 | 0 | 0

2 | RETURN NEW; | 17004 | 7 | 0 | 0 | 0

2 | END IF; | 17004 | 8 | 0 | 0 | 0

2 | IF TG\_OP = 'UPDATE' THEN | 17004 | 9 | 0 | 0 | 0

2 | sal\_diff := NEW.sal - OLD.sal; | 17004 | 10 | 1 | 0 | 0.000355

2 | RAISE INFO 'Updating employee %', OLD.empno; | 17004 | 11 | 1 | 0 | 0.000177

2 | RAISE INFO '..Old salary: %', OLD.sal; | 17004 | 12 | 1 | 0 | 5.5e-05

2 | RAISE INFO '..New salary: %', NEW.sal; | 17004 | 13 | 1 | 0 | 3.1e-05

2 | RAISE INFO '..Raise : %', sal\_diff; | 17004 | 14 | 1 | 0 | 2.8e-05

2 | RETURN NEW; | 17004 | 15 | 1 | 0 | 2.7e-05

2 | END IF; | 17004 | 16 | 1 | 0 | 1e-06

2 | IF TG\_OP = 'DELETE' THEN | 17004 | 17 | 0 | 0 | 0

2 | RAISE INFO 'Deleting employee %', OLD.empno; | 17004 | 18 | 0 | 0 | 0

2 | RAISE INFO '..Old salary: %', OLD.sal; | 17004 | 19 | 0 | 0 | 0

2 | RETURN OLD; | 17004 | 20 | 0 | 0 | 0

2 | END IF; | 17004 | 21 | 0 | 0 | 0

2 | END; | 17004 | 22 | 0 | 0 | 0

2 | | 17004 | 23 | 0 | 0 | 0

(68 rows)

1. Query the plsql\_profiler\_data view to review a subset of the information found in plsql\_profiler\_rawdata table:

acctg=# SELECT \* FROM plsql\_profiler\_data;

runid | unit\_number | line# | total\_occur | total\_time | min\_time | max\_time | spare1 | spare2 | spare3 | spare4

-------+-------------+-------+-------------+------------+----------+----------+--------+--------+--------+--------

1 | 16999 | 1 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 2 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 3 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 4 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 5 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 6 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 7 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 8 | 1 | 0.001621 | 0.001621 | 0.001621 | | | |

1 | 16999 | 9 | 1 | 0.000301 | 0.000301 | 0.000301 | | | |

1 | 16999 | 10 | 1 | 4.6e-05 | 4.6e-05 | 4.6e-05 | | | |

1 | 16999 | 11 | 1 | 0.001114 | 0.001114 | 0.001114 | | | |

1 | 16999 | 12 | 15 | 0.000206 | 5e-06 | 7.8e-05 | | | |

1 | 16999 | 13 | 15 | 8.3e-05 | 2e-06 | 4.7e-05 | | | |

1 | 16999 | 14 | 14 | 0.000773 | 4.7e-05 | 0.000116 | | | |

1 | 16999 | 15 | 0 | 0 | 0 | 0 | | | |

1 | 16999 | 16 | 1 | 1e-05 | 1e-05 | 1e-05 | | | |

1 | 16999 | 17 | 1 | 0 | 0 | 0 | | | |

1 | 16999 | 18 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 1 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 2 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 3 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 4 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 5 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 6 | 1 | 0.000143 | 0.000143 | 0.000143 | | | |

2 | 17002 | 7 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 8 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 9 | 1 | 3.2e-05 | 3.2e-05 | 3.2e-05 | | | |

2 | 17002 | 10 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 11 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 12 | 0 | 0 | 0 | 0 | | | |

2 | 17002 | 13 | 1 | 0.000383 | 0.000383 | 0.000383 | | | |

2 | 17002 | 14 | 1 | 6.3e-05 | 6.3e-05 | 6.3e-05 | | | |

2 | 17002 | 15 | 1 | 3.6e-05 | 3.6e-05 | 3.6e-05 | | | |

2 | 17002 | 16 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 1 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 2 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 3 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 4 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 5 | 1 | 0.000647 | 0.000647 | 0.000647 | | | |

2 | 17000 | 6 | 1 | 2.6e-05 | 2.6e-05 | 2.6e-05 | | | |

2 | 17000 | 7 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 8 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 9 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 10 | 0 | 0 | 0 | 0 | | | |

2 | 17000 | 11 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 1 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 2 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 3 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 4 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 5 | 1 | 8.4e-05 | 8.4e-05 | 8.4e-05 | | | |

2 | 17004 | 6 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 7 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 8 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 9 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 10 | 1 | 0.000355 | 0.000355 | 0.000355 | | | |

2 | 17004 | 11 | 1 | 0.000177 | 0.000177 | 0.000177 | | | |

2 | 17004 | 12 | 1 | 5.5e-05 | 5.5e-05 | 5.5e-05 | | | |

2 | 17004 | 13 | 1 | 3.1e-05 | 3.1e-05 | 3.1e-05 | | | |

2 | 17004 | 14 | 1 | 2.8e-05 | 2.8e-05 | 2.8e-05 | | | |

2 | 17004 | 15 | 1 | 2.7e-05 | 2.7e-05 | 2.7e-05 | | | |

2 | 17004 | 16 | 1 | 1e-06 | 1e-06 | 1e-06 | | | |

2 | 17004 | 17 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 18 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 19 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 20 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 21 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 22 | 0 | 0 | 0 | 0 | | | |

2 | 17004 | 23 | 0 | 0 | 0 | 0 | | | |

(68 rows)

#### DBMS\_PROFILER - Reference

The Advanced Server installer creates the following tables and views that you can query to review PL/SQL performance profile information:

| Table Name | Description |
| --- | --- |
| PLSQL\_PROFILER\_RUNS | Table containing information about all profiler runs, organized by runid. |
| PLSQL\_PROFILER\_UNITS | Table containing information about all profiler runs, organized by unit. |
| PLSQL\_PROFILER\_DATA | View containing performance statistics. |
| PLSQL\_PROFILER\_RAWDATA | Table containing the performance statistics and the extended performance statistics for DRITA counters and timers. |

##### PLSQL\_PROFILER\_RUNS

The PLSQL\_PROFILER\_RUNS table contains the following columns:

| Column | Data Type | Description |
| --- | --- | --- |
| runid | INTEGER (NOT NULL) | Unique identifier (plsql\_profiler\_runnumber) |
| related\_run | INTEGER | The runid of a related run. |
| run\_owner | TEXT | The role that recorded the profiling session. |
| run\_date | TIMESTAMP WITHOUT TIME ZONE | The profiling session start time. |
| run\_comment | TEXT | User comments relevant to this run |
| run\_total\_time | BIGINT | Run time (in microseconds) |
| run\_system\_info | TEXT | Currently Unused |
| run\_comment1 | TEXT | Additional user comments |
| spare1 | TEXT | Currently Unused |

##### PLSQL\_PROFILER\_UNITS

The PLSQL\_PROFILER\_UNITS table contains the following columns:

| Column | Data Type | Description |
| --- | --- | --- |
| runid | INTEGER | Unique identifier (plsql\_profiler\_runnumber) |
| unit\_number | OID | Corresponds to the OID of the row in the pg\_proc table that identifies the unit. |
| unit\_type | TEXT | PL/SQL function, procedure, trigger or anonymous block |
| unit\_owner | TEXT | The identity of the role that owns the unit. |
| unit\_name | TEXT | The complete signature of the unit. |
| unit\_timestamp | TIMESTAMP WITHOUT TIME ZONE | Creation date of the unit (currently NULL). |
| total\_time | BIGINT | Time spent within the unit (in milliseconds) |
| spare1 | BIGINT | Currently Unused |
| spare2 | BIGINT | Currently Unused |

##### PLSQL\_PROFILER\_DATA

The PLSQL\_PROFILER\_DATA view contains the following columns:

| Column | Data Type | Description |
| --- | --- | --- |
| runid | INTEGER | Unique identifier (plsql\_profiler\_runnumber) |
| unit\_number | OID | Object ID of the unit that contains the current line. |
| line# | INTEGER | Current line number of the profiled workload. |
| total\_occur | BIGINT | The number of times that the line was executed. |
| total\_time | DOUBLE PRECISION | The amount of time spent executing the line (in seconds) |
| min\_time | DOUBLE PRECISION | The minimum execution time for the line. |
| max\_time | DOUBLE PRECISION | The maximum execution time for the line. |
| spare1 | NUMBER | Currently Unused |
| spare2 | NUMBER | Currently Unused |
| spare3 | NUMBER | Currently Unused |
| spare4 | NUMBER | Currently Unused |

##### PLSQL\_PROFILER\_RAWDATA

The PLSQL\_PROFILER\_RAWDATA table contains the statistical information that is found in the PLSQL\_PROFILER\_DATA view, as well as the performance statistics returned by the DRITA counters and timers.

| Column | Data Type | Description |
| --- | --- | --- |
| runid | INTEGER | The run identifier (plsql\_profiler\_runnumber). |
| sourcecode | TEXT | The individual line of profiled code. |
| func\_oid | OID | Object ID of the unit that contains the current line. |
| line\_number | INTEGER | Current line number of the profiled workload. |
| exec\_count | BIGINT | The number of times that the line was executed. |
| time\_total | DOUBLE PRECISION | The amount of time spent executing the line (in seconds) |
| time\_shortest | DOUBLE PRECISION | The minimum execution time for the line. |
| time\_longest | DOUBLE PRECISION | The maximum execution time for the line. |
| tuples\_returned | BIGINT | Currently Unused |
| num\_scans | BIGINT | Currently Unused |
| tuples\_fetched | BIGINT | Currently Unused |
| tuples\_inserted | BIGINT | Currently Unused |
| tuples\_updated | BIGINT | Currently Unused |
| tuples\_deleted | BIGINT | Currently Unused |
| blocks\_fetched | BIGINT | Currently Unused |
| blocks\_hit | BIGINT | Currently Unused |
| wal\_write | BIGINT | The server has waited for a write to the write-ahead log buffer (expect this value to be high). |
| wal\_flush | BIGINT | The server has waited for the write-ahead log to flush to disk. |
| wal\_file\_sync | BIGINT | 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). |
| buffer\_free\_list\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory). |
| shmem\_index\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the shared-memory map. |
| oid\_gen\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the next available OID (object ID). |
| xid\_gen\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the next available transaction ID. |
| proc\_array\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the process array |
| sinval\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the cache invalidation state. |
| freespace\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the freespace map. |
| wal\_insert\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small. |
| wal\_write\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes write-ahead log flushes. |
| control\_file\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number). |
| checkpoint\_lock\_acquire | BIGINT | A server process has waited for the short-term lock that prevents simultaneous checkpoints. |
| clog\_control\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the commit log. |
| subtrans\_control\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the subtransaction log. |
| multi\_xact\_gen\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes). |
| multi\_xact\_offset\_lock\_acquire | BIGINT | 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). |
| multi\_xact\_member\_lock\_acquire | BIGINT | 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). |
| rel\_cache\_init\_lock\_acquire | BIGINT | The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads. |
| bgwriter\_communication\_lock\_acquire | BIGINT | The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process. |
| two\_phase\_state\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the list of prepared transactions. |
| tablespace\_create\_lock\_acquire | BIGINT | The server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands. |
| btree\_vacuum\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID. |
| add\_in\_shmem\_lock\_acquire | BIGINT | Currently Unused |
| autovacuum\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the shared autovacuum state. |
| autovacuum\_schedule\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the autovacuum schedule. |
| syncscan\_lock\_acquire | BIGINT | The server has waited for the short-term lock that coordinates synchronous scans. |
| icache\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to InfiniteCache state |
| breakpoint\_lock\_acquire | BIGINT | The server has waited for the short-term lock that synchronizes access to the debugger breakpoint list. |
| lwlock\_acquire | BIGINT | The server has waited for a short-term lock that has not been described elsewhere in this section. |
| db\_file\_read | BIGINT | A server process has waited for the completion of a read (from disk). |
| db\_file\_write | BIGINT | A server process has waited for the completion of a write (to disk). |
| db\_file\_sync | BIGINT | A server process has waited for the operating system to flush all changes to disk. |
| db\_file\_extend | BIGINT | A server process has waited for the operating system while adding a new page to the end of a file. |
| sql\_parse | BIGINT | Currently Unused |
| query\_plan | BIGINT | The server has generated a query plan. |
| infinitecache\_read | BIGINT | The server has waited for an Infinite Cache read request. |
| infinitecache\_write | BIGINT | The server has waited for an Infinite Cache write request. |
| wal\_write\_time | BIGINT | The amount of time that the server has waited for a write to the write-ahead log buffer (expect this value to be high). |
| wal\_flush\_time | BIGINT | The amount of time that the server has waited for the write-ahead log to flush to disk. |
| wal\_file\_sync\_time | BIGINT | 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\_sync). |
| buffer\_free\_list\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory). |
| shmem\_index\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the shared-memory map. |
| oid\_gen\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the next available OID (object ID). |
| xid\_gen\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the next available transaction ID. |
| proc\_array\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the process array. |
| sinval\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the cache invalidation state. |
| freespace\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the freespace map. |
| wal\_insert\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small. |
| wal\_write\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes write-ahead log flushes. |
| control\_file\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number). |
| checkpoint\_lock\_acquire\_time | BIGINT | The amount of time that the server process has waited for the short-term lock that prevents simultaneous checkpoints. |
| clog\_control\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the commit log. |
| subtrans\_control\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the subtransaction log. |
| multi\_xact\_gen\_lock\_acquire\_time | BIGINT | 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). |
| multi\_xact\_offset\_lock\_acquire\_time | BIGINT | 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). |
| multi\_xact\_member\_lock\_acquire\_time | BIGINT | 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). |
| rel\_cache\_init\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads. |
| bgwriter\_communication\_lock\_acquire\_time | BIGINT | The amount of time that the bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process. |
| two\_phase\_state\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the list of prepared transactions. |
| tablespace\_create\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands. |
| btree\_vacuum\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID. |
| add\_in\_shmem\_lock\_acquire\_time | BIGINT | Obsolete/unused |
| autovacuum\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the shared autovacuum state. |
| autovacuum\_schedule\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the autovacuum schedule. |
| syncscan\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that coordinates synchronous scans. |
| icache\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to InfiniteCache state |
| breakpoint\_lock\_acquire\_time | BIGINT | The amount of time that the server has waited for the short-term lock that synchronizes access to the debugger breakpoint list. |
| lwlock\_acquire\_time | BIGINT | The amount of time that the server has waited for a short-term lock that has not been described elsewhere in this section. |
| db\_file\_read\_time | BIGINT | The amount of time that the server process has waited for the completion of a read (from disk). |
| db\_file\_write\_time | BIGINT | The amount of time that the server process has waited for the completion of a write (to disk). |
| db\_file\_sync\_time | BIGINT | The amount of time that the server process has waited for the operating system to flush all changes to disk. |
| db\_file\_extend\_time | BIGINT | 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. |
| sql\_parse\_time | BIGINT | The amount of time that the server has parsed a SQL statement. |
| query\_plan\_time | BIGINT | The amount of time that the server has computed the execution plan for a SQL statement. |
| infinitecache\_read\_time | BIGINT | The amount of time that the server has waited for an Infinite Cache read request. |
| infinitecache\_write\_time | BIGINT | The amount of time that the server has waited for an Infinite Cache write request. |
| totalwaits | BIGINT | The total number of event waits. |
| Totalwaittime | BIGINT | The total time spent waiting for an event. |

## DBMS\_RANDOM

The DBMS\_RANDOM package provides a number of methods to generate random values. The procedures and functions available in the DBMS\_RANDOM package are listed in the following table.

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| INITIALIZE(val) | n/a | Initializes the DBMS\_RANDOM package with the specified seed *value*. Deprecated, but supported for backward compatibility. |
| NORMAL() | NUMBER | Returns a random NUMBER. |
| RANDOM | INTEGER | Returns a random INTEGER with a value greater than or equal to -2^31 and less than 2^31. Deprecated, but supported for backward compatibility. |
| SEED(*val*) | n/a | Resets the seed with the specified *value*. |
| SEED(*val*) | n/a | Resets the seed with the specified *value*. |
| STRING(opt, len) | VARCHAR2 | Returns a random string. |
| TERMINATE | n/a | TERMINATE has no effect. Deprecated, but supported for backward compatibility. |
| VALUE | NUMBER | Returns a random number with a value greater than or equal to 0 and less than 1, with 38 digit precision. |
| VALUE(*low*, *high*) | NUMBER | Returns a random number with a value greater than or equal to low and less than high. |

### INITIALIZE

The INITIALIZE procedure initializes the DBMS\_RANDOM package with a seed value. The signature is:

INITIALIZE(val IN INTEGER)

This procedure should be considered deprecated; it is included for backward compatibility only.

Parameters

*val*

*val* is the seed value used by the DBMS\_RANDOM package algorithm.

Example

The following code snippet demonstrates a call to the INITIALIZE procedure that initializes the DBMS\_RANDOM package with the seed value, 6475.

DBMS\_RANDOM.INITIALIZE(6475);

### NORMAL

The NORMAL function returns a random number of type NUMBER. The signature is:

*result* NUMBER NORMAL()

Parameters

*result*

*result* is a random value of type NUMBER.

Example

The following code snippet demonstrates a call to the NORMAL function:

x:= DBMS\_RANDOM.NORMAL();

### RANDOM

The RANDOM function returns a random INTEGER value that is greater than or equal to -2 ^31 and less than 2 ^31. The signature is:

*result* INTEGER RANDOM()

This function should be considered deprecated; it is included for backward compatibility only.

Parameters

*result*

*result* is a random value of type INTEGER.

Example

The following code snippet demonstrates a call to the RANDOM function. The call returns a random number:

x := DBMS\_RANDOM.RANDOM();

### SEED

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

SEED(*val* IN INTEGER)

Parameters

*val*

*val* is the seed value used by the DBMS\_RANDOM package algorithm.

Example

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

DBMS\_RANDOM.SEED(8495);

### SEED

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

SEED(*val* IN VARCHAR2)

Parameters

*val*

*val* is the seed value used by the DBMS\_RANDOM package algorithm.

Example

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

DBMS\_RANDOM.SEED('abc123');

### STRING

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

*result* VARCHAR2 STRING(*opt* IN CHAR, *len* IN NUMBER)

Parameters

*opt*

Formatting option for the returned string. *option* may be:

|  |  |
| --- | --- |
| **Option** | **Specifies Formatting Option** |
| u or U | Uppercase alpha string |
| l or L | Lowercase alpha string |
| a or A | Mixed case string |
| x or X | Uppercase alpha-numeric string |
| p or P | Any printable characters |

*len*

The length of the returned string.

*result*

*result* is a random value of type VARCHAR2.

Example

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

x := DBMS\_RANDOM.STRING('X', 10);

### TERMINATE

The TERMINATE procedure has no effect. The signature is:

TERMINATE

The TERMINATE procedure should be considered deprecated; the procedure is supported for compatibility only.

### VALUE

The VALUE function returns a random NUMBER that is greater than or equal to 0, and less than 1, with 38 digit precision. The VALUE function has two forms; the signature of the first form is:

*result* NUMBER VALUE()

Parameters

*result*

*result* is a random value of type NUMBER.

Example

The following code snippet demonstrates a call to the VALUE function. The call returns a random NUMBER:

x := DBMS\_RANDOM.VALUE();

### VALUE

The VALUE function returns a random NUMBER with a value that is between user-specified boundaries. The VALUE function has two forms; the signature of the second form is:

*result* NUMBER VALUE(*low* IN NUMBER, *high* IN NUMBER)

Parameters

*low*

*low* specifies the lower boundary for the random value. The random value may be equal to *low*.

*high*

*high* specifies the upper boundary for the random value; the random value will be less than *high*.

*result*

*result* is a random value of type NUMBER.

Example

The following code snippet demonstrates a call to the VALUE function. The call returns a random NUMBER with a value that is greater than or equal to 1 and less than 100:

x := DBMS\_RANDOM.VALUE(1, 100);

## DBMS\_RLS

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

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| ADD\_POLICY(object\_schema, object\_name, policy\_name, function\_schema, policy\_function [, statement\_types [, update\_check [, enable [, static\_policy [, policy\_type [, long\_predicate [, sec\_relevant\_cols [, sec\_relevant\_cols\_opt ]]]]]]]]) | Procedure | n/a | Add a security policy to a database object. |
| DROP\_POLICY(object\_schema, object\_name, policy\_name) | Procedure | n/a | Remove a security policy from a database object. |
| ENABLE\_POLICY(object\_schema, object\_name, policy\_name, enable) | Procedure | n/a | Enable or disable a security policy. |

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

Virtual Private Database is a type of fine-grained access control using security policies. Fine-grained access control in Virtual Private Database means that access to data can be controlled down to specific rows as defined by the security policy.

The rules that encode a security policy are defined in a policy function, which is an SPL function with certain input parameters and return value. The security policy is the named association of the policy function to a particular database object, typically a table.

Note: In Advanced Server, the policy function can be written in any language supported by Advanced Server such as SQL, PL/pgSQL and SPL.

Note: The database objects currently supported by Advanced Server Virtual Private Database are tables. Policies cannot be applied to views or synonyms.

The advantages of using Virtual Private Database are the following:

* Provides a fine-grained level of security. Database object level privileges given by the GRANT command determine access privileges to the entire instance of a database object, while Virtual Private Database provides access control for the individual rows of a database object instance.
* A different security policy can be applied depending upon the type of SQL command (INSERT, UPDATE, DELETE, or SELECT).
* The security policy can vary dynamically for each applicable SQL command affecting the database object depending upon factors such as the session user of the application accessing the database object.
* Invocation of the security policy is transparent to all applications that access the database object and thus, individual applications do not have to be modified to apply the security policy.
* Once a security policy is enabled, it is not possible for any application (including new applications) to circumvent the security policy except by the system privilege noted by the following.
* Even superusers cannot circumvent the security policy except by the system privilege noted by the following.

Note: The only way security policies can be circumvented is if the EXEMPT ACCESS POLICY system privilege has been granted to a user. The EXEMPT ACCESS POLICY privilege should be granted with extreme care as a user with this privilege is exempted from all policies in the database.

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:

|  |  |
| --- | --- |
| attribute Value | Equivalent Value |
| SESSION\_USER | pg\_catalog.session\_user |
| CURRENT\_USER | pg\_catalog.current\_user |
| CURRENT\_SCHEMA | pg\_catalog.current\_schema |
| HOST | pg\_catalog.inet\_host |
| IP\_ADDRESS | pg\_catalog.inet\_client\_addr |
| SERVER\_HOST | pg\_catalog.inet\_server\_addr |

Note: The examples used to illustrate the DBMS\_RLS package are based on a modified copy of the sample emp table provided with Advanced Server along with a role named salesmgr that is granted all privileges on the table. You can create the modified copy of the emp table named vpemp and the salesmgr role as shown by the following:

CREATE TABLE public.vpemp AS SELECT empno, ename, job, sal, comm, deptno FROM emp;

ALTER TABLE vpemp ADD authid VARCHAR2(12);

UPDATE vpemp SET authid = 'researchmgr' WHERE deptno = 20;

UPDATE vpemp SET authid = 'salesmgr' WHERE deptno = 30;

SELECT \* FROM vpemp;

empno | ename | job | sal | comm | deptno | authid

-------+--------+-----------+---------+---------+--------+-------------

7782 | CLARK | MANAGER | 2450.00 | | 10 |

7839 | KING | PRESIDENT | 5000.00 | | 10 |

7934 | MILLER | CLERK | 1300.00 | | 10 |

7369 | SMITH | CLERK | 800.00 | | 20 | researchmgr

7566 | JONES | MANAGER | 2975.00 | | 20 | researchmgr

7788 | SCOTT | ANALYST | 3000.00 | | 20 | researchmgr

7876 | ADAMS | CLERK | 1100.00 | | 20 | researchmgr

7902 | FORD | ANALYST | 3000.00 | | 20 | researchmgr

7499 | ALLEN | SALESMAN | 1600.00 | 300.00 | 30 | salesmgr

7521 | WARD | SALESMAN | 1250.00 | 500.00 | 30 | salesmgr

7654 | MARTIN | SALESMAN | 1250.00 | 1400.00 | 30 | salesmgr

7698 | BLAKE | MANAGER | 2850.00 | | 30 | salesmgr

7844 | TURNER | SALESMAN | 1500.00 | 0.00 | 30 | salesmgr

7900 | JAMES | CLERK | 950.00 | | 30 | salesmgr

(14 rows)

CREATE ROLE salesmgr WITH LOGIN PASSWORD 'password';

GRANT ALL ON vpemp TO salesmgr;

### ADD\_POLICY

The ADD\_POLICY procedure creates a new policy by associating a policy function with a database object.

You must be a superuser to execute this procedure.

ADD\_POLICY(object\_schema VARCHAR2, object\_name VARCHAR2,

policy\_name VARCHAR2, function\_schema VARCHAR2,

policy\_function VARCHAR2

[, statement\_types VARCHAR2

[, update\_check BOOLEAN

[, enable BOOLEAN

[, static\_policy BOOLEAN

[, policy\_type INTEGER

[, long\_predicate BOOLEAN

[, sec\_relevant\_cols VARCHAR2

[, sec\_relevant\_cols\_opt INTEGER ]]]]]]]])

Parameters

object\_schema

Name of the schema containing the database object to which the policy is to be applied.

object\_name

Name of the database object to which the policy is to be applied. A given database object may have more than one policy applied to it.

policy\_name

Name assigned to the policy. The combination of database object (identified by object\_schema and object\_name) and policy name must be unique within the database.

function\_schema

Name of the schema containing the policy function.

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

policy\_function

Name of the SPL function that defines the rules of the security policy. The same function may be specified in more than one policy.

Note: The policy function may belong to a package in which case policy\_function must also contain the package name in dot notation (that is, package\_name.function\_name).

statement\_types

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

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

update\_check

Applies to INSERT and UPDATE SQL commands only.

When set to TRUE, the policy is applied to newly inserted rows and to the modified image of updated rows. If any of the new or modified rows do not qualify according to the policy function predicate, then the INSERT or UPDATE command throws an exception and no rows are inserted or modified by the INSERT or UPDATE command.

When set to FALSE, the policy is not applied to newly inserted rows or the modified image of updated rows. Thus, a newly inserted row may not appear in the result set of a subsequent SQL command that invokes the same policy. Similarly, rows which qualified according to the policy prior to an UPDATE command may not appear in the result set of a subsequent SQL command that invokes the same policy.

The default is FALSE.

enable

When set to TRUE, the policy is enabled and applied to the SQL commands given by the statement\_types parameter. When set to FALSE the policy is disabled and not applied to any SQL commands. The policy can be enabled using the ENABLE\_POLICY procedure. The default is TRUE.

static\_policy

In Oracle, when set to TRUE, the policy is static, which means the policy function is evaluated once per database object the first time it is invoked by a policy on that database object. The resulting policy function predicate string is saved in memory and reused for all invocations of that policy on that database object while the database server instance is running.

When set to FALSE, the policy is dynamic, which means the policy function is re-evaluated and the policy function predicate string regenerated for all invocations of the policy.

The default is FALSE.

Note: In Oracle 10g, the policy\_type parameter was introduced, which is intended to replace the static\_policy parameter. In Oracle, if the policy\_type parameter is not set to its default value of NULL, the policy\_type parameter setting overrides the 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.

DECLARE

v\_object\_schema VARCHAR2(30) := 'public';

v\_object\_name VARCHAR2(30) := 'vpemp';

v\_policy\_name VARCHAR2(30) := 'secure\_update';

v\_function\_schema VARCHAR2(30) := 'enterprisedb';

v\_policy\_function VARCHAR2(30) := 'verify\_session\_user';

v\_statement\_types VARCHAR2(30) := 'INSERT,UPDATE,DELETE';

v\_update\_check BOOLEAN := TRUE;

v\_enable BOOLEAN := TRUE;

BEGIN

DBMS\_RLS.ADD\_POLICY(

v\_object\_schema,

v\_object\_name,

v\_policy\_name,

v\_function\_schema,

v\_policy\_function,

v\_statement\_types,

v\_update\_check,

v\_enable

);

END;

After successful creation of the policy, a terminal session is started by user salesmgr. The following query shows the content of the vpemp table:

edb=# \c edb salesmgr

Password for user salesmgr:

You are now connected to database "edb" as user "salesmgr".

edb=> SELECT \* FROM vpemp;

empno | ename | job | sal | comm | deptno | authid

-------+--------+-----------+---------+---------+--------+-------------

7782 | CLARK | MANAGER | 2450.00 | | 10 |

7839 | KING | PRESIDENT | 5000.00 | | 10 |

7934 | MILLER | CLERK | 1300.00 | | 10 |

7369 | SMITH | CLERK | 800.00 | | 20 | researchmgr

7566 | JONES | MANAGER | 2975.00 | | 20 | researchmgr

7788 | SCOTT | ANALYST | 3000.00 | | 20 | researchmgr

7876 | ADAMS | CLERK | 1100.00 | | 20 | researchmgr

7902 | FORD | ANALYST | 3000.00 | | 20 | researchmgr

7499 | ALLEN | SALESMAN | 1600.00 | 300.00 | 30 | salesmgr

7521 | WARD | SALESMAN | 1250.00 | 500.00 | 30 | salesmgr

7654 | MARTIN | SALESMAN | 1250.00 | 1400.00 | 30 | salesmgr

7698 | BLAKE | MANAGER | 2850.00 | | 30 | salesmgr

7844 | TURNER | SALESMAN | 1500.00 | 0.00 | 30 | salesmgr

7900 | JAMES | CLERK | 950.00 | | 30 | salesmgr

(14 rows)

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

edb=> UPDATE vpemp SET comm = sal \* .75;

UPDATE 6

Instead of updating all rows in the table, the policy restricts the effect of the update to only those rows where the authid column contains the value salesmgr as specified by the policy function predicate authid = SYS\_CONTEXT('USERENV', 'SESSION\_USER').

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

edb=> SELECT \* FROM vpemp;

empno | ename | job | sal | comm | deptno | authid

-------+--------+-----------+---------+---------+--------+-------------

7782 | CLARK | MANAGER | 2450.00 | | 10 |

7839 | KING | PRESIDENT | 5000.00 | | 10 |

7934 | MILLER | CLERK | 1300.00 | | 10 |

7369 | SMITH | CLERK | 800.00 | | 20 | researchmgr

7566 | JONES | MANAGER | 2975.00 | | 20 | researchmgr

7788 | SCOTT | ANALYST | 3000.00 | | 20 | researchmgr

7876 | ADAMS | CLERK | 1100.00 | | 20 | researchmgr

7902 | FORD | ANALYST | 3000.00 | | 20 | researchmgr

7499 | ALLEN | SALESMAN | 1600.00 | 1200.00 | 30 | salesmgr

7521 | WARD | SALESMAN | 1250.00 | 937.50 | 30 | salesmgr

7654 | MARTIN | SALESMAN | 1250.00 | 937.50 | 30 | salesmgr

7698 | BLAKE | MANAGER | 2850.00 | 2137.50 | 30 | salesmgr

7844 | TURNER | SALESMAN | 1500.00 | 1125.00 | 30 | salesmgr

7900 | JAMES | CLERK | 950.00 | 712.50 | 30 | salesmgr

(14 rows)

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

edb=> INSERT INTO vpemp VALUES (9001,'SMITH','ANALYST',3200.00,NULL,20, 'researchmgr');

ERROR: policy with check option violation

DETAIL: Policy predicate was evaluated to FALSE with the updated values

If update\_check was set to FALSE, the preceding INSERT command would have succeeded.

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

CREATE OR REPLACE FUNCTION sal\_lt\_2000 (

p\_schema VARCHAR2,

p\_object VARCHAR2

)

RETURN VARCHAR2

IS

BEGIN

RETURN 'sal < 2000';

END;

The policy is created so that it is enforced only if a SELECT command includes columns sal or comm:

DECLARE

v\_object\_schema VARCHAR2(30) := 'public';

v\_object\_name VARCHAR2(30) := 'vpemp';

v\_policy\_name VARCHAR2(30) := 'secure\_salary';

v\_function\_schema VARCHAR2(30) := 'enterprisedb';

v\_policy\_function VARCHAR2(30) := 'sal\_lt\_2000';

v\_statement\_types VARCHAR2(30) := 'SELECT';

v\_sec\_relevant\_cols VARCHAR2(30) := 'sal,comm';

BEGIN

DBMS\_RLS.ADD\_POLICY(

v\_object\_schema,

v\_object\_name,

v\_policy\_name,

v\_function\_schema,

v\_policy\_function,

v\_statement\_types,

sec\_relevant\_cols => v\_sec\_relevant\_cols

);

END;

If a query does not reference columns sal or comm, then the policy is not applied. The following query returns all 14 rows of table vpemp:

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

empno | ename | job | deptno | authid

-------+--------+-----------+--------+-------------

7782 | CLARK | MANAGER | 10 |

7839 | KING | PRESIDENT | 10 |

7934 | MILLER | CLERK | 10 |

7369 | SMITH | CLERK | 20 | researchmgr

7566 | JONES | MANAGER | 20 | researchmgr

7788 | SCOTT | ANALYST | 20 | researchmgr

7876 | ADAMS | CLERK | 20 | researchmgr

7902 | FORD | ANALYST | 20 | researchmgr

7499 | ALLEN | SALESMAN | 30 | salesmgr

7521 | WARD | SALESMAN | 30 | salesmgr

7654 | MARTIN | SALESMAN | 30 | salesmgr

7698 | BLAKE | MANAGER | 30 | salesmgr

7844 | TURNER | SALESMAN | 30 | salesmgr

7900 | JAMES | CLERK | 30 | salesmgr

(14 rows)

If the query references the sal or comm columns, then the policy is applied to the query eliminating any rows where sal is greater than or equal to 2000 as shown by the following:

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

empno | ename | job | sal | comm | deptno | authid

-------+--------+----------+---------+---------+--------+-------------

7934 | MILLER | CLERK | 1300.00 | | 10 |

7369 | SMITH | CLERK | 800.00 | | 20 | researchmgr

7876 | ADAMS | CLERK | 1100.00 | | 20 | researchmgr

7499 | ALLEN | SALESMAN | 1600.00 | 1200.00 | 30 | salesmgr

7521 | WARD | SALESMAN | 1250.00 | 937.50 | 30 | salesmgr

7654 | MARTIN | SALESMAN | 1250.00 | 937.50 | 30 | salesmgr

7844 | TURNER | SALESMAN | 1500.00 | 1125.00 | 30 | salesmgr

7900 | JAMES | CLERK | 950.00 | 712.50 | 30 | salesmgr

(8 rows)

### DROP\_POLICY

The DROP\_POLICY procedure deletes an existing policy. The policy function and database object associated with the policy are not deleted by the DROP\_POLICY procedure.

You must be a superuser to execute this procedure.

DROP\_POLICY(object\_schema VARCHAR2, object\_name VARCHAR2,

policy\_name VARCHAR2)

Parameters

object\_schema

Name of the schema containing the database object to which the policy applies.

object\_name

Name of the database object to which the policy applies.

policy\_name

Name of the policy to be deleted.

Examples

The following example deletes policy secure\_update on table public.vpemp:

DECLARE

v\_object\_schema VARCHAR2(30) := 'public';

v\_object\_name VARCHAR2(30) := 'vpemp';

v\_policy\_name VARCHAR2(30) := 'secure\_update';

BEGIN

DBMS\_RLS.DROP\_POLICY(

v\_object\_schema,

v\_object\_name,

v\_policy\_name

);

END;

### ENABLE\_POLICY

The ENABLE\_POLICY procedure enables or disables an existing policy on the specified database object.

You must be a superuser to execute this procedure.

ENABLE\_POLICY(object\_schema VARCHAR2, object\_name VARCHAR2,

policy\_name VARCHAR2, enable BOOLEAN)

Parameters

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;

## DBMS\_SCHEDULER

The DBMS\_SCHEDULER package provides a way to create and manage Oracle-styled jobs, programs and job schedules. The DBMS\_SCHEDULER package implements the following functions and procedures:

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| CREATE\_JOB(*job\_name*, *job\_type*, *job\_action*, *number\_of\_arguments*, *start\_date*, *repeat\_interval*, *end\_date*, *job\_class*, *enabled*, *auto\_drop*, *comments*) | n/a | Use the first form of the CREATE\_JOB procedure to create a job, specifying program and schedule details by means of parameters. |
| CREATE\_JOB(*job\_name*, *program\_name*, *schedule\_name*, *job\_class*, *enabled*, *auto\_drop*, *comments*) | n/a | Use the second form of CREATE\_JOB to create a job that uses a named program and named schedule. |
| CREATE\_PROGRAM(*program\_name*, *program\_type*, *program\_action*, *number\_of\_arguments*, *enabled*, *comments*) | n/a | Use CREATE\_PROGRAM to create a program. |
| CREATE\_SCHEDULE( *schedule\_name*, *start\_date*, *repeat\_interval*, *end\_date*, *comments*) | n/a | Use the CREATE\_SCHEDULE procedure to create a schedule. |
| DEFINE\_PROGRAM\_ARGUMENT( *program\_name*, *argument\_position*, *argument\_name*, *argument\_type*, *default\_value*, *out\_argument*) | n/a | Use the first form of the DEFINE\_PROGRAM\_ARGUMENT procedure to define a program argument that has a default value. |
| DEFINE\_PROGRAM\_ARGUMENT( *program\_name*, *argument\_position*, *argument\_name*, *argument\_type*, *out\_argument*) | n/a | Use the first form of the DEFINE\_PROGRAM\_ARGUMENT procedure to define a program argument that does not have a default value. |
| DISABLE(*name*, *force*, *commit\_semantics*) | n/a | Use the DISABLE procedure to disable a job or program. |
| DROP\_JOB(*job\_name*, *force*, *defer*, *commit\_semantics*) | n/a | Use the DROP\_JOB procedure to drop a job. |
| DROP\_PROGRAM(*program\_name*, *force*) | n/a | Use the DROP\_PROGRAM procedure to drop a program. |
| DROP\_PROGRAM\_ARGUMENT( *program\_name*, *argument\_position*) | n/a | Use the first form of DROP\_PROGRAM\_ARGUMENT to drop a program argument by specifying the argument position. |
| DROP\_PROGRAM\_ARGUMENT( *program\_name*, *argument\_name*) | n/a | Use the second form of DROP\_PROGRAM\_ARGUMENT to drop a program argument by specifying the argument name. |
| DROP\_SCHEDULE(*schedule\_name*, *force*) | n/a | Use the DROP SCHEDULE procedure to drop a schedule. |
| ENABLE(*name*, *commit\_semantics*) | n/a | Use the ENABLE command to enable a program or job. |
| EVALUATE\_CALENDAR\_STRING( *calendar\_string*, *start\_date*, *return\_date\_after*, *next\_run\_date*) | n/a | Use EVALUATE\_CALENDAR\_STRING to review the execution date described by a user-defined calendar schedule. |
| RUN\_JOB(*job\_name*, *use\_current\_session*, *manually*) | n/a | Use the RUN\_JOB procedure to execute a job immediately. |
| SET\_JOB\_ARGUMENT\_VALUE( *job\_name*, *argument\_position*, *argument\_value*) | n/a | Use the first form of SET\_JOB\_ARGUMENT value to set the value of a job argument described by the argument's position. |
| SET\_JOB\_ARGUMENT\_VALUE( *job\_name*, *argument\_name*, *argument\_value*) | n/a | Use the second form of SET\_JOB\_ARGUMENT value to set the value of a job argument described by the argument's name. |

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

The DBMS\_SCHEDULER package is dependent on the pgAgent service; you must have a pgAgent service installed and running on your server before using DBMS\_SCHEDULER.

Before using DBMS\_SCHEDULER, a database superuser must create the catalog tables in which the DBMS\_SCHEDULER programs, schedules and jobs are stored. Use the psql client to connect to the database, and invoke the command:

CREATE EXTENSION dbms\_scheduler;

By default, the dbms\_scheduler extension resides in the contrib/dbms\_scheduler\_ext subdirectory (under the Advanced Server installation).

Note that after creating the DBMS\_SCHEDULER tables, only a superuser will be able to perform a dump or reload of the database.

### 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:

FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;BYMINUTE=45

Defines a schedule that is executed each weeknight at 5:45.

The token types and syntax described in the table below are supported by Advanced Server:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Token type | Syntax | Valid Values | | |
| FREQ | FREQ=*predefined\_interval* | Where *predefined\_interval* is one of the following: YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY. The SECONDLY keyword is not supported. | | |
| BYMONTH | BYMONTH=*month*(, *month*)... | Where *month* is the three-letter abbreviation of the month name: JAN | FEB | MAR | APR | MAY | JUN | JUL | AUG | SEP | OCT | NOV | DEC | | |
| BYMONTH | BYMONTH=*month*(, *month*)... | Where *month* is the numeric value representing the month: 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | | |
| BYMONTHDAY | BYMONTHDAY=*day\_of\_month* | Where *day\_of\_month* is a value from 1 through 31 | | |
| BYDAY | BYDAY=*weekday* | Where *weekday* is a three-letter abbreviation or single-digit value representing the day of the week. | | |
| Monday | MON | 1 |
| Tuesday | TUE | 2 |
| Wednesday | WED | 3 |
| Thursday | THU | 4 |
| Friday | FRI | 5 |
| Saturday | SAT | 6 |
| Sunday | SUN | 7 |
| BYDATE | BYDATE=*date*(, *date*)... | Where *date* is *YYYYMMDD*.  YYYY is a four-digit year representation of the year, MM is a two-digit representation of the month, and DD is a two-digit day representation of the day. | | |
| BYDATE | BYDATE=*date*(, *date*)... | Where *date* is *MMDD*.  MM is a two-digit representation of the month, and DD is a two-digit day representation of the day | | |
| BYHOUR | BYHOUR=*hour* | Where *hour* is a value from 0 through 23. | | |
| BYMINUTE | BYMINUTE=*minute* | Where *minute* is a value from 0 through 59. | | |

### 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:

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.

### CREATE\_PROGRAM

Use the CREATE\_PROGRAM procedure to create a DBMS\_SCHEDULER program. The signature is:

CREATE\_PROGRAM(  
 *program\_name* IN VARCHAR2,  
 *program\_type* IN VARCHAR2,  
 *program\_action* IN VARCHAR2,  
 *number\_of\_arguments* IN PLS\_INTEGER DEFAULT 0,  
 *enabled* IN BOOLEAN DEFAULT FALSE,  
 *comments* IN VARCHAR2 DEFAULT NULL)

Parameters

*program\_name*

*program\_name* specifies the name of the program that is being created.

*program\_type*

*program\_type* specifies the type of program. The current implementation of CREATE\_PROGRAM supports a *program\_type* of PLSQL\_BLOCK or PROCEDURE.

*program\_action*

If *program\_type* is PLSQL\_BLOCK, *program\_action* contains the PL/SQL block that will execute when the program is invoked. The PL/SQL block must be terminated with a semi-colon (;).

If *program\_type* is PROCEDURE, *program\_action* contains the name of the stored procedure.

*number\_of\_arguments*

If *program\_type* is PLSQL\_BLOCK, this argument is ignored.

If *program*\_*type* is PROCEDURE, *number\_of\_arguments* specifies the number of arguments required by the procedure. The default value is 0.

*enabled*

*enabled* specifies if the program is created enabled or disabled:

* If *enabled* is TRUE, the program is created enabled.
* If *enabled* is FALSE, the program is created disabled; use the DBMS\_SCHEDULER.ENABLE program to enable a disabled program.

The default value is FALSE.

*comments*

Use the *comments* parameter to specify a comment about the program; by default, this parameter is NULL.

Example

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.

### CREATE\_SCHEDULE

Use the CREATE\_SCHEDULE procedure to create a job schedule. The signature of the CREATE\_SCHEDULE procedure is:

create\_schedule(  
 *schedule\_name* IN VARCHAR2,  
 *start\_date* IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 *repeat\_interval* IN VARCHAR2,  
 *end\_date* IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,  
 *comments* IN VARCHAR2 DEFAULT NULL)

Parameters

*schedule\_name*

*schedule\_name* specifies the name of the schedule.

*start\_date*

*start\_date* is a TIMESTAMP WITH TIME ZONE value that specifies the date and time that the schedule is eligible to execute. If a *start\_date* is not specified, the date that the job is enabled is used as the *start\_date*. By default, *start\_date* is NULL.

*repeat\_interval*

*repeat\_interval* is a VARCHAR2 value that specifies how often the job will repeat. If a *repeat\_interval* is not specified, the job will execute only once, on the date specified by *start\_date*.

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* IN TIMESTAMP WITH TIME ZONE DEFAULT NULL

*end\_date* is a TIMESTAMP WITH TIME ZONE value that specifies a time after which the schedule will no longer execute. If a date is specified, the *end\_date* must be after the *start\_date*. The default value is NULL.

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.

*comments* IN VARCHAR2 DEFAULT NULL)

Use the *comments* parameter to specify a comment about the schedule; by default, this parameter is NULL.

Example

The following code fragment calls CREATE\_SCHEDULE to create a schedule named weeknights\_at\_5:

EXEC

DBMS\_SCHEDULER.CREATE\_SCHEDULE (

schedule\_name => 'weeknights\_at\_5',

start\_date => '01-JUN-13 09:00:00.000000'

repeat\_interval => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',

comments => 'This schedule executes each weeknight at 5:00');

The schedule executes each weeknight, at 5:00 pm, effective after June 1, 2013. Since no end\_date is specified, the schedule will execute indefinitely until it is disabled with DBMS\_SCHEDULER.DISABLE.

### DEFINE\_PROGRAM\_ARGUMENT

Use the DEFINE\_PROGRAM\_ARGUMENT procedure to define a program argument. The DEFINE\_PROGRAM\_ARGUMENT procedure comes in two forms; the first form defines an argument with a default value:

DEFINE\_PROGRAM\_ARGUMENT(  
 *program\_name* IN VARCHAR2,  
 *argument\_position* IN PLS\_INTEGER,  
 *argument\_name* IN VARCHAR2 DEFAULT NULL,  
 *argument\_type* IN VARCHAR2,  
 *default\_value* IN VARCHAR2,  
 *out\_argument* IN BOOLEAN DEFAULT FALSE)

The second form defines an argument without a default value:

DEFINE\_PROGRAM\_ARGUMENT(  
 *program\_name* IN VARCHAR2,  
 *argument\_position* IN PLS\_INTEGER,  
 *argument\_name* IN VARCHAR2 DEFAULT NULL,  
 *argument\_type* IN VARCHAR2,  
 *out\_argument* IN BOOLEAN DEFAULT FALSE)

Parameters

*program\_name*

*program\_name* is the name of the program to which the arguments belong.

*argument\_position*

*argument\_position* specifies the position of the argument as it is passed to the program.

*argument\_name*

*argument\_name* specifies the optional name of the argument. By default, *argument\_name* is NULL.

*argument\_type* IN VARCHAR2

*argument\_type* specifies the data type of the argument.

*default\_value*

*default\_value* specifies the default value assigned to the argument. *default\_value* will be overridden by a value specified by the job when the job executes.

*out\_argument* IN BOOLEAN DEFAULT FALSE

*out\_argument* is not currently used; if specified, the value must be FALSE.

Example

The following code fragment uses the DEFINE\_PROGRAM\_ARGUMENT procedure to define the first and second arguments in a program named add\_emp:

EXEC

DBMS\_SCHEDULER.DEFINE\_PROGRAM\_ARGUMENT(

program\_name => 'add\_emp',

argument\_position => 1,

argument\_name => 'dept\_no',

argument\_type => 'INTEGER,

default\_value => '20');

EXEC

DBMS\_SCHEDULER.DEFINE\_PROGRAM\_ARGUMENT(

program\_name => 'add\_emp',

argument\_position => 2,

argument\_name => 'emp\_name',

argument\_type => 'VARCHAR2');

The first argument is an INTEGER value named dept\_no that has a default value of 20. The second argument is a VARCHAR2 value named emp\_name; the second argument does not have a default value.

### DISABLE

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

disable(  
 *name* IN VARCHAR2,  
 *force* IN BOOLEAN DEFAULT FALSE,  
 *commit\_semantics* IN VARCHAR2 DEFAULT 'STOP\_ON\_FIRST\_ERROR')

Parameters

*name*

*name* specifies the name of the program or job that is being disabled.

*force*

*force* is accepted for compatibility, and ignored.

*commit\_semantics*

*commit\_semantics* instructs the server how to handle an error encountered while disabling a program or job. By default, *commit\_semantics* is set to STOP\_ON\_FIRST\_ERROR, instructing the server to stop when it encounters an error. Any programs or jobs that were successfully disabled prior to the error will be committed to disk.

The TRANSACTIONAL and ABSORB\_ERRORS keywords are accepted for compatibility, and ignored.

Example

The following call to the DISABLE procedure disables a program named update\_emp:

DBMS\_SCHEDULER.DISABLE('update\_emp');

### DROP\_JOB

Use the DROP\_JOB procedure to DROP a job, DROP any arguments that belong to the job, and eliminate any future job executions. The signature of the procedure is:

drop\_job(  
 *job\_name* IN VARCHAR2,  
 *force* IN BOOLEAN DEFAULT FALSE,  
 *defer* IN BOOLEAN DEFAULT FALSE,  
 *commit\_semantics* IN VARCHAR2 DEFAULT 'STOP\_ON\_FIRST\_ERROR')

Parameters

*job\_name*

*job\_name* specifies the name of the job that is being dropped.

*force*

*force* is accepted for compatibility, and ignored.

*defer*

*defer* is accepted for compatibility, and ignored.

*commit\_semantics*

*commit\_semantics* instructs the server how to handle an error encountered while dropping a program or job. By default, *commit\_semantics* is set to STOP\_ON\_FIRST\_ERROR, instructing the server to stop when it encounters an error.

The TRANSACTIONAL and ABSORB\_ERRORS keywords are accepted for compatibility, and ignored.

Example

The following call to DROP\_JOB drops a job named update\_log:

DBMS\_SCHEDULER.DROP\_JOB('update\_log');

### 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');

### 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');

### DROP\_SCHEDULE

Use the DROP\_SCHEDULE procedure to drop a schedule. The signature is:

DROP\_SCHEDULE(  
 *schedule\_name* IN VARCHAR2,  
 *force* IN BOOLEAN DEFAULT FALSE)

Parameters

*schedule\_name*

*schedule\_name* specifies the name of the schedule that is being dropped.

*force*

*force* specifies the behavior of the server if the specified schedule is referenced by any job:

* Specify FALSE to instruct the server to return an error if the specified schedule is referenced by a job. This is the default behavior.
* Specify TRUE to instruct the server to disable to any jobs that use the specified schedule before dropping the schedule. Any running jobs will be allowed to complete before the schedule is dropped.

Example

The following call to DROP\_SCHEDULE drops a schedule named weeknights\_at\_5:

DBMS\_SCHEDULER.DROP\_SCHEDULE('weeknights\_at\_5', TRUE);

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

### ENABLE

Use the ENABLE procedure to enable a disabled program or job.

The signature of the ENABLE procedure is:

ENABLE(  
 *name* IN VARCHAR2,  
 *commit\_semantics* IN VARCHAR2 DEFAULT 'STOP\_ON\_FIRST\_ERROR')

Parameters

*name*

*name* specifies the name of the program or job that is being enabled.

*commit\_semantics*

*commit\_semantics* instructs the server how to handle an error encountered while enabling a program or job. By default, *commit\_semantics* is set to STOP\_ON\_FIRST\_ERROR, instructing the server to stop when it encounters an error.

The TRANSACTIONAL and ABSORB\_ERRORS keywords are accepted for compatibility, and ignored.

Example

The following call to DBMS\_SCHEDULER.ENABLE enables the update\_emp program:

DBMS\_SCHEDULER.ENABLE('update\_emp');

### EVALUATE\_CALENDAR\_STRING

Use the EVALUATE\_CALENDAR\_STRING procedure to evaluate the *repeat\_interval* value specified when creating a schedule with the CREATE\_SCHEDULE procedure. The EVALUATE\_CALENDAR\_STRING procedure will return the date and time that a specified schedule will execute without actually scheduling the job.

The signature of the EVALUATE\_CALENDAR\_STRING procedure is:

evaluate\_calendar\_string(  
 *calendar*\_*string* IN VARCHAR2,  
 *start\_date* IN TIMESTAMP WITH TIME ZONE,  
 *return\_date\_after* IN TIMESTAMP WITH TIME ZONE,  
 *next\_run\_date* OUT TIMESTAMP WITH TIME ZONE)

Parameters

*calendar*\_*string*

*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:

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.

### RUN\_JOB

Use the RUN\_JOB procedure to execute a job immediately. The signature of the RUN\_JOB procedure is:

run\_job(  
 *job\_name* IN VARCHAR2,  
 *use\_current\_session* IN BOOLEAN DEFAULT TRUE

Parameters

*job\_name*

*job\_name* specifies the name of the job that will execute.

*use\_current\_session*

By default, the job will execute in the current session. If specified, *use\_current\_session* must be set to TRUE ; if *use\_current\_session* is set to FALSE, Advanced Server will return an error.

Example

The following call to RUN\_JOB executes a job named update\_log:

DBMS\_SCHEDULER.RUN\_JOB('update\_log', TRUE);

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

### SET\_JOB\_ARGUMENT\_VALUE

Use the SET\_JOB\_ARGUMENT\_VALUE procedure to specify a value for an argument. The SET\_JOB\_ARGUMENT\_VALUE procedure comes in two forms; the first form specifies which argument should be modified by position:

set\_job\_argument\_value(  
 *job\_name* IN VARCHAR2,  
 *argument\_position* IN PLS\_INTEGER,  
 *argument\_value* IN VARCHAR2)

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

set\_job\_argument\_value(  
 *job\_name* IN VARCHAR2,  
 *argument\_name* IN VARCHAR2,  
 *argument\_value* IN VARCHAR2)

Argument values set by the SET\_JOB\_ARGUMENT\_VALUE procedure override any values set by default.

Parameters

*job\_name*

*job\_name* specifies the name of the job to which the modified argument belongs.

*argument\_position*

Use *argument\_position* to specify the argument position for which the value will be set.

*argument\_name*

Use *argument\_name* to specify the argument by name for which the value will be set.

*argument\_value*

*argument\_value* specifies the new value of the argument.

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');

## DBMS\_SESSION

Advanced Server provides support for the following DBMS\_SESSION.SET\_ROLE procedure:

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| SET\_ROLE(role\_cmd) | n/a | Executes a SET ROLE statement followed by the string value specified in role\_cmd. |

Advanced Server's implementation of DBMS\_SESSION is a partial implementation when compared to Oracle's version. Only DBMS\_SESSION.SET\_ROLE is supported.

### SET\_ROLE

The SET\_ROLE procedure sets the current session user to the role specified in *role\_cmd*. After invoking the SET\_ROLE procedure, the current session will use the permissions assigned to the specified role. The signature of the procedure is:

SET\_ROLE(*role\_cmd*)

The SET\_ROLE procedure appends the value specified for *role\_cmd* to the SET ROLE statement, and then invokes the statement.

Parameters

*role\_cmd*

*role\_cmd* specifies a role name in the form of a string value.

Example

The following call to the SET\_ROLE procedure invokes the SET ROLE command to set the identity of the current session user to manager:

edb=# exec DBMS\_SESSION.SET\_ROLE('manager');

## 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.

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| BIND\_VARIABLE(c, name, value [, out\_value\_size ]) | Procedure | n/a | Bind a value to a variable. |
| BIND\_VARIABLE\_CHAR(c, name, value [, out\_value\_size ]) | Procedure | n/a | Bind a CHAR value to a variable. |
| BIND\_VARIABLE\_RAW(c, name, value [, out\_value\_size ]) | Procedure | n/a | Bind a RAW value to a variable. |
| CLOSE\_CURSOR(c IN OUT) | Procedure | n/a | Close a cursor. |
| COLUMN\_VALUE(c, position, value OUT [, column\_error OUT [, actual\_length OUT ]]) | Procedure | n/a | Return a column value into a variable. |
| COLUMN\_VALUE\_CHAR(c, position, value OUT [, column\_error OUT [, actual\_length OUT ]]) | Procedure | n/a | Return a CHAR column value into a variable. |
| COLUMN\_VALUE\_RAW(c, position, value OUT [, column\_error OUT [, actual\_length OUT ]]) | Procedure | n/a | Return a RAW column value into a variable. |
| DEFINE\_COLUMN(c, position, column [, column\_size ]) | Procedure | n/a | Define a column in the SELECT list. |
| DEFINE\_COLUMN\_CHAR(c, position, column, column\_size) | Procedure | n/a | Define a CHAR column in the SELECT list. |
| DEFINE\_COLUMN\_RAW(c, position, column, column\_size) | Procedure | n/a | Define a RAW column in the SELECT list. |
| DESCRIBE\_COLUMNS | Procedure | n/a | Defines columns to hold a cursor result set. |
| EXECUTE(c) | Function | INTEGER | Execute a cursor. |
| EXECUTE\_AND\_FETCH(c [, exact ]) | Function | INTEGER | Execute a cursor and fetch a single row. |
| FETCH\_ROWS(c) | Function | INTEGER | Fetch rows from the cursor. |
| IS\_OPEN(c) | Function | BOOLEAN | Check if a cursor is open. |
| LAST\_ROW\_COUNT | Function | INTEGER | Return cumulative number of rows fetched. |
| OPEN\_CURSOR | Function | INTEGER | Open a cursor. |
| PARSE(c, statement, language\_flag) | Procedure | n/a | Parse a statement. |

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

The following table lists the public variable available in the DBMS\_SQL package.

| Public Variables | Data Type | Value | Description |
| --- | --- | --- | --- |
| native | INTEGER | 1 | Provided for compatibility with Oracle syntax. See DBMS\_SQL.PARSE for more information. |
| V6 | INTEGER | 2 | Provided for compatibility with Oracle syntax. See DBMS\_SQL.PARSE for more information. |
| V7 | INTEGER | 3 | Provided for compatibility with Oracle syntax. See DBMS\_SQL.PARSE for more information |

### BIND\_VARIABLE

The BIND\_VARIABLE procedure provides the capability to associate a value with an IN or IN OUT bind variable in a SQL command.

BIND\_VARIABLE(c INTEGER, name VARCHAR2,

value { BLOB | CLOB | DATE | FLOAT | INTEGER | NUMBER |

TIMESTAMP | VARCHAR2 }

[, out\_value\_size INTEGER ])

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value to be assigned.

out\_value\_size

If name is an IN OUT variable, defines the maximum length of the output value. If not specified, the length of value is assumed.

Examples

The following anonymous block uses bind variables to insert a row into the emp table.

DECLARE

curid INTEGER;

v\_sql VARCHAR2(150) := 'INSERT INTO emp VALUES ' ||

'(:p\_empno, :p\_ename, :p\_job, :p\_mgr, ' ||

':p\_hiredate, :p\_sal, :p\_comm, :p\_deptno)';

v\_empno emp.empno%TYPE;

v\_ename emp.ename%TYPE;

v\_job emp.job%TYPE;

v\_mgr emp.mgr%TYPE;

v\_hiredate emp.hiredate%TYPE;

v\_sal emp.sal%TYPE;

v\_comm emp.comm%TYPE;

v\_deptno emp.deptno%TYPE;

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid,v\_sql,DBMS\_SQL.native);

v\_empno := 9001;

v\_ename := 'JONES';

v\_job := 'SALESMAN';

v\_mgr := 7369;

v\_hiredate := TO\_DATE('13-DEC-07','DD-MON-YY');

v\_sal := 8500.00;

v\_comm := 1500.00;

v\_deptno := 40;

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_empno',v\_empno);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_ename',v\_ename);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_job',v\_job);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_mgr',v\_mgr);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_hiredate',v\_hiredate);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_sal',v\_sal);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_comm',v\_comm);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_deptno',v\_deptno);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('Number of rows processed: ' || v\_status);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

Number of rows processed: 1

### BIND\_VARIABLE\_CHAR

The BIND\_VARIABLE\_CHAR procedure provides the capability to associate a CHAR value with an IN or IN OUT bind variable in a SQL command.

BIND\_VARIABLE\_CHAR(c INTEGER, name VARCHAR2, value CHAR

[, out\_value\_size INTEGER ])

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value of type CHAR to be assigned.

out\_value\_size

If name is an IN OUT variable, defines the maximum length of the output value. If not specified, the length of value is assumed.

### 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.

BIND\_VARIABLE\_RAW(c INTEGER, name VARCHAR2, value RAW

[, out\_value\_size INTEGER ])

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value of type RAW to be assigned.

out\_value\_size

If name is an IN OUT variable, defines the maximum length of the output value. If not specified, the length of value is assumed.

### 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;

### 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,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,5,v\_comm);

DBMS\_OUTPUT.PUT\_LINE(v\_empno || ' ' || RPAD(v\_ename,10) || ' ' ||

TO\_CHAR(v\_hiredate,'yyyy-mm-dd') || ' ' ||

TO\_CHAR(v\_sal,'9,999.99') || ' ' ||

TO\_CHAR(NVL(v\_comm,0),'9,999.99'));

END LOOP;

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

### 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.

### COLUMN VALUE RAW

The COLUMN\_VALUE\_RAW procedure defines a variable to receive a RAW value from a cursor.

COLUMN\_VALUE\_RAW(c INTEGER, position INTEGER, value OUT RAW

[, column\_error OUT NUMBER [, actual\_length OUT INTEGER ]])

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

position

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

value

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

column\_error

Error number associated with the column, if any.

actual\_length

Actual length of the data prior to any truncation.

### DEFINE\_COLUMN

The DEFINE\_COLUMN procedure defines a column or expression in the SELECT list that is to be returned and retrieved in a cursor.

DEFINE\_COLUMN(c INTEGER, position INTEGER, column { BLOB |

CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }

[, column\_size INTEGER ])

Parameters

c

Cursor id of the cursor associated with the SELECT command.

position

Position of the column or expression in the SELECT list that is being defined.

column

A variable that is of the same data type as the column or expression in position position of the SELECT list.

column\_size

The maximum length of the returned data. column\_size must be specified only if column is VARCHAR2. Returned data exceeding column\_size is truncated to column\_size characters.

Examples

The following shows how the empno, ename, hiredate, sal, and comm columns of the emp table are defined with the DEFINE\_COLUMN procedure.

DECLARE

curid INTEGER;

v\_empno NUMBER(4);

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;

### 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.

### 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.

### 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:

|  |  |
| --- | --- |
| Column Name | Type |
| col\_type | INTEGER |
| col\_max\_len | INTEGER |
| col\_name | VARCHAR2(128) |
| col\_name\_len | INTEGER |
| col\_schema\_name | VARCHAR2(128) |
| col\_schema\_name\_len | INTEGER |
| col\_precision | INTEGER |
| col\_scale | INTEGER |
| col\_charsetid | INTEGER |
| col\_charsetform | INTEGER |
| col\_null\_ok | BOOLEAN |

### 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.

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;

### EXECUTE\_AND\_FETCH

Function EXECUTE\_AND\_FETCH executes a parsed SELECT command and fetches one row.

status INTEGER EXECUTE\_AND\_FETCH(c INTEGER

[, exact BOOLEAN ])

Parameters

c

Cursor id of the cursor for the SELECT command to be executed.

exact

If set to TRUE, an exception is thrown if the number of rows in the result set is not exactly equal to 1. If set to FALSE, no exception is thrown. The default is FALSE. A NO\_DATA\_FOUND exception is thrown if exact is TRUE and there are no rows in the result set. A TOO\_MANY\_ROWS exception is thrown if exact is TRUE and there is more than one row in the result set.

status

Returns 1 if a row was successfully fetched, 0 if no rows to fetch. If an exception is thrown, no value is returned.

Examples

The following stored procedure uses the EXECUTE\_AND\_FETCH function to retrieve one employee using the employee’s name. An exception will be thrown if the employee is not found, or there is more than one employee with the same name.

CREATE OR REPLACE PROCEDURE select\_by\_name(

p\_ename emp.ename%TYPE

)

IS

curid INTEGER;

v\_empno emp.empno%TYPE;

v\_hiredate emp.hiredate%TYPE;

v\_sal emp.sal%TYPE;

v\_comm emp.comm%TYPE;

v\_dname dept.dname%TYPE;

v\_disp\_date VARCHAR2(10);

v\_sql VARCHAR2(120) := 'SELECT empno, hiredate, sal, ' ||

'NVL(comm, 0), dname ' ||

'FROM emp e, dept d ' ||

'WHERE ename = :p\_ename ' ||

'AND e.deptno = d.deptno';

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid,v\_sql,DBMS\_SQL.native);

DBMS\_SQL.BIND\_VARIABLE(curid,':p\_ename',UPPER(p\_ename));

DBMS\_SQL.DEFINE\_COLUMN(curid,1,v\_empno);

DBMS\_SQL.DEFINE\_COLUMN(curid,2,v\_hiredate);

DBMS\_SQL.DEFINE\_COLUMN(curid,3,v\_sal);

DBMS\_SQL.DEFINE\_COLUMN(curid,4,v\_comm);

DBMS\_SQL.DEFINE\_COLUMN(curid,5,v\_dname,14);

v\_status := DBMS\_SQL.EXECUTE\_AND\_FETCH(curid,TRUE);

DBMS\_SQL.COLUMN\_VALUE(curid,1,v\_empno);

DBMS\_SQL.COLUMN\_VALUE(curid,2,v\_hiredate);

DBMS\_SQL.COLUMN\_VALUE(curid,3,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_comm);

DBMS\_SQL.COLUMN\_VALUE(curid,5,v\_dname);

v\_disp\_date := TO\_CHAR(v\_hiredate, 'MM/DD/YYYY');

DBMS\_OUTPUT.PUT\_LINE('Number : ' || v\_empno);

DBMS\_OUTPUT.PUT\_LINE('Name : ' || UPPER(p\_ename));

DBMS\_OUTPUT.PUT\_LINE('Hire Date : ' || v\_disp\_date);

DBMS\_OUTPUT.PUT\_LINE('Salary : ' || v\_sal);

DBMS\_OUTPUT.PUT\_LINE('Commission: ' || v\_comm);

DBMS\_OUTPUT.PUT\_LINE('Department: ' || v\_dname);

DBMS\_SQL.CLOSE\_CURSOR(curid);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Employee ' || p\_ename || ' not found');

DBMS\_SQL.CLOSE\_CURSOR(curid);

WHEN TOO\_MANY\_ROWS THEN

DBMS\_OUTPUT.PUT\_LINE('Too many employees named, ' ||

p\_ename || ', found');

DBMS\_SQL.CLOSE\_CURSOR(curid);

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('The following is SQLERRM:');

DBMS\_OUTPUT.PUT\_LINE(SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('The following is SQLCODE:');

DBMS\_OUTPUT.PUT\_LINE(SQLCODE);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

EXEC select\_by\_name('MARTIN')

Number : 7654

Name : MARTIN

Hire Date : 09/28/1981

Salary : 1250

Commission: 1400

Department: SALES

### 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,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,5,v\_comm);

DBMS\_OUTPUT.PUT\_LINE(v\_empno || ' ' || RPAD(v\_ename,10) || ' ' ||

TO\_CHAR(v\_hiredate,'yyyy-mm-dd') || ' ' ||

TO\_CHAR(v\_sal,'9,999.99') || ' ' ||

TO\_CHAR(NVL(v\_comm,0),'9,999.99'));

END LOOP;

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

EMPNO ENAME HIREDATE SAL COMM

----- ---------- ---------- -------- --------

7369 SMITH 1980-12-17 800.00 .00

7499 ALLEN 1981-02-20 1,600.00 300.00

7521 WARD 1981-02-22 1,250.00 500.00

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

### IS\_OPEN

The IS\_OPEN function provides the capability to test if the given cursor is open.

status BOOLEAN IS\_OPEN(c INTEGER)

Parameters

c

Cursor ID of the cursor to be tested.

status

Set to TRUE if the cursor is open, set to FALSE if the cursor is not open.

### LAST\_ROW\_COUNT

The LAST\_ROW\_COUNT function returns the number of rows that have been currently fetched.

rowcnt INTEGER LAST\_ROW\_COUNT

Parameters

rowcnt

Number of row fetched thus far.

Examples

The following example uses the LAST\_ROW\_COUNT function to display the total number of rows fetched in the query.

DECLARE

curid INTEGER;

v\_empno NUMBER(4);

v\_ename VARCHAR2(10);

v\_hiredate DATE;

v\_sal NUMBER(7,2);

v\_comm NUMBER(7,2);

v\_sql VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||

'comm FROM emp';

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid,v\_sql,DBMS\_SQL.native);

DBMS\_SQL.DEFINE\_COLUMN(curid,1,v\_empno);

DBMS\_SQL.DEFINE\_COLUMN(curid,2,v\_ename,10);

DBMS\_SQL.DEFINE\_COLUMN(curid,3,v\_hiredate);

DBMS\_SQL.DEFINE\_COLUMN(curid,4,v\_sal);

DBMS\_SQL.DEFINE\_COLUMN(curid,5,v\_comm);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('EMPNO ENAME HIREDATE SAL COMM');

DBMS\_OUTPUT.PUT\_LINE('----- ---------- ---------- -------- ' ||

'--------');

LOOP

v\_status := DBMS\_SQL.FETCH\_ROWS(curid);

EXIT WHEN v\_status = 0;

DBMS\_SQL.COLUMN\_VALUE(curid,1,v\_empno);

DBMS\_SQL.COLUMN\_VALUE(curid,2,v\_ename);

DBMS\_SQL.COLUMN\_VALUE(curid,3,v\_hiredate);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,4,v\_sal);

DBMS\_SQL.COLUMN\_VALUE(curid,5,v\_comm);

DBMS\_OUTPUT.PUT\_LINE(v\_empno || ' ' || RPAD(v\_ename,10) || ' ' ||

TO\_CHAR(v\_hiredate,'yyyy-mm-dd') || ' ' ||

TO\_CHAR(v\_sal,'9,999.99') || ' ' ||

TO\_CHAR(NVL(v\_comm,0),'9,999.99'));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Number of rows: ' || DBMS\_SQL.LAST\_ROW\_COUNT);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

EMPNO ENAME HIREDATE SAL COMM

----- ---------- ---------- -------- --------

7369 SMITH 1980-12-17 800.00 .00

7499 ALLEN 1981-02-20 1,600.00 300.00

7521 WARD 1981-02-22 1,250.00 500.00

7566 JONES 1981-04-02 2,975.00 .00

7654 MARTIN 1981-09-28 1,250.00 1,400.00

7698 BLAKE 1981-05-01 2,850.00 .00

7782 CLARK 1981-06-09 2,450.00 .00

7788 SCOTT 1987-04-19 3,000.00 .00

7839 KING 1981-11-17 5,000.00 .00

7844 TURNER 1981-09-08 1,500.00 .00

7876 ADAMS 1987-05-23 1,100.00 .00

7900 JAMES 1981-12-03 950.00 .00

7902 FORD 1981-12-03 3,000.00 .00

7934 MILLER 1982-01-23 1,300.00 .00

Number of rows: 14

### 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 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;

### PARSE

The PARSE procedure parses a SQL command or SPL block. If the SQL command is a DDL command, it is immediately executed and does not require running the EXECUTE function.

PARSE(c INTEGER, statement VARCHAR2, language\_flag INTEGER)

Parameters

c

Cursor ID of an open cursor.

statement

SQL command or SPL block to be parsed. A SQL command must not end with the semi-colon terminator, however an SPL block does require the semi-colon terminator.

language\_flag

Language flag provided for compatibility with Oracle syntax. Use DBMS\_SQL.V6, DBMS\_SQL.V7 or DBMS\_SQL.native. This flag is ignored, and all syntax is assumed to be in EnterpriseDB Advanced Server form.

Examples

The following anonymous block creates a table named, job. Note that DDL statements are executed immediately by the PARSE procedure and do not require a separate EXECUTE step.

DECLARE

curid INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

DBMS\_SQL.PARSE(curid, 'CREATE TABLE job (jobno NUMBER(3), ' ||

'jname VARCHAR2(9))',DBMS\_SQL.native);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

The following inserts two rows into the job table.

DECLARE

curid INTEGER;

v\_sql VARCHAR2(50);

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

v\_sql := 'INSERT INTO job VALUES (100, ''ANALYST'')';

DBMS\_SQL.PARSE(curid, v\_sql, DBMS\_SQL.native);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('Number of rows processed: ' || v\_status);

v\_sql := 'INSERT INTO job VALUES (200, ''CLERK'')';

DBMS\_SQL.PARSE(curid, v\_sql, DBMS\_SQL.native);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_OUTPUT.PUT\_LINE('Number of rows processed: ' || v\_status);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

Number of rows processed: 1

Number of rows processed: 1

The following anonymous block uses the DBMS\_SQL package to execute a block containing two INSERT statements. Note that the end of the block contains a terminating semi-colon, while in the prior example, each individual INSERT statement does not have a terminating semi-colon.

DECLARE

curid INTEGER;

v\_sql VARCHAR2(100);

v\_status INTEGER;

BEGIN

curid := DBMS\_SQL.OPEN\_CURSOR;

v\_sql := 'BEGIN ' ||

'INSERT INTO job VALUES (300, ''MANAGER''); ' ||

'INSERT INTO job VALUES (400, ''SALESMAN''); ' ||

'END;';

DBMS\_SQL.PARSE(curid, v\_sql, DBMS\_SQL.native);

v\_status := DBMS\_SQL.EXECUTE(curid);

DBMS\_SQL.CLOSE\_CURSOR(curid);

END;

## DBMS\_UTILITY

The DBMS\_UTILITY package provides support for the following various utility programs:

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| ANALYZE\_DATABASE(method [, estimate\_rows [, estimate\_percent [, method\_opt ]]]) | Procedure | n/a | Analyze database tables. |
| ANALYZE\_PART\_OBJECT(schema, object\_name [, object\_type [, command\_type [, command\_opt [, sample\_clause ]]]]) | Procedure | n/a | Analyze a partitioned table. |
| ANALYZE\_SCHEMA(schema, method [, estimate\_rows [, estimate\_percent [, method\_opt ]]]) | Procedure | n/a | Analyze schema tables. |
| CANONICALIZE(name, canon\_name OUT, canon\_len) | Procedure | n/a | Canonicalizes a string – e.g., strips off white space. |
| COMMA\_TO\_TABLE(list, tablen OUT, tab OUT) | Procedure | n/a | Convert a comma-delimited list of names to a table of names. |
| DB\_VERSION(version OUT, compatibility OUT) | Procedure | n/a | Get the database version. |
| EXEC\_DDL\_STATEMENT(parse\_string) | Procedure | n/a | Execute a DDL statement. |
| FORMAT\_CALL\_STACK | Function | TEXT | Formats the current call stack. |
| GET\_CPU\_TIME | Function | NUMBER | Get the current CPU time. |
| GET\_DEPENDENCY(type, schema, name) | Procedure | n/a | Get objects that are dependent upon the given object.. |
| GET\_HASH\_VALUE(name, base, hash\_size) | Function | NUMBER | Compute a hash value. |
| GET\_PARAMETER\_VALUE(parnam, intval OUT, strval OUT) | Procedure | BINARY\_INTEGER | Get database initialization parameter settings. |
| GET\_TIME | Function | NUMBER | Get the current time. |
| NAME\_TOKENIZE(name, a OUT, b OUT, c OUT, dblink OUT, nextpos OUT) | Procedure | n/a | Parse the given name into its component parts. |
| TABLE\_TO\_COMMA(tab, tablen OUT, list OUT) | Procedure | n/a | Convert a table of names to a comma-delimited list. |

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

The following table lists the public variables available in the DBMS\_UTILITY package.

| Public Variables | Data Type | Value | Description |
| --- | --- | --- | --- |
| inv\_error\_on\_restrictions | PLS\_INTEGER | 1 | Used by the INVALIDATE procedure. |
| lname\_array | TABLE |  | For lists of long names. |
| uncl\_array | TABLE |  | For lists of users and names. |

### LNAME\_ARRAY

The LNAME\_ARRAY is for storing lists of long names including fully-qualified names.

TYPE lname\_array IS TABLE OF VARCHAR2(4000) INDEX BY BINARY\_INTEGER;

### UNCL\_ARRAY

The UNCL\_ARRAY is for storing lists of users and names.

TYPE uncl\_array IS TABLE OF VARCHAR2(227) INDEX BY BINARY\_INTEGER;

### ANALYZE\_DATABASE, ANALYZE SCHEMA and ANALYZE PART\_OBJECT

The ANALYZE\_DATABASE(), ANALYZE\_SCHEMA() and ANALYZE\_PART\_OBJECT() procedures provide the capability to gather statistics on tables in the database. When you execute the ANALYZE statement, Postgres samples the data in a table and records distribution statistics in the pg\_statistics system table.

ANALYZE\_DATABASE, ANALYZE\_SCHEMA, and ANALYZE\_PART\_OBJECT differ primarily in the number of tables that are processed:

* ANALYZE\_DATABASE analyzes all tables in all schemas within the current database.
* ANALYZE\_SCHEMA analyzes all tables in a given schema (within the current database).
* ANALYZE\_PART\_OBJECT analyzes a single table.

The syntax for the ANALYZE commands are:

ANALYZE\_DATABASE(method VARCHAR2 [, estimate\_rows NUMBER

[, estimate\_percent NUMBER [, method\_opt VARCHAR2 ]]])

ANALYZE\_SCHEMA(schema VARCHAR2, method VARCHAR2

[, estimate\_rows NUMBER [, estimate\_percent NUMBER

[, method\_opt VARCHAR2 ]]])

ANALYZE\_PART\_OBJECT(schema VARCHAR2, object\_name VARCHAR2

[, object\_type CHAR [, command\_type CHAR

[, command\_opt VARCHAR2 [, sample\_clause ]]]])

Parameters - ANALYZE\_DATABASE and ANALYZE\_SCHEMA

method

method determines whether the ANALYZE procedure populates the pg\_statistics table or removes entries from the pg\_statistics table. If you specify a method of DELETE, the ANALYZE procedure removes the relevant rows from pg\_statistics. If you specify a method of COMPUTE or ESTIMATE, the ANALYZE procedure analyzes a table (or multiple tables) and records the distribution information in pg\_statistics. There is no difference between COMPUTE and ESTIMATE; both methods execute the Postgres ANALYZE statement. All other parameters are validated and then ignored.

estimate\_rows

Number of rows upon which to base estimated statistics. One of estimate\_rows or estimate\_percent must be specified if method is ESTIMATE.

This argument is ignored, but is included for compatibility.

estimate\_percent

Percentage of rows upon which to base estimated statistics. One of estimate\_rows or estimate\_percent must be specified if method is ESTIMATE.

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.

### CANONICALIZE

The CANONICALIZE procedure performs the following operations on an input string:

* If the string is not double-quoted, verifies that it uses the characters of a legal identifier. If not, an exception is thrown. If the string is double-quoted, all characters are allowed.
* If the string is not double-quoted and does not contain periods, uppercases all alphabetic characters and eliminates leading and trailing spaces.
* If the string is double-quoted and does not contain periods, strips off the double quotes.
* If the string contains periods and no portion of the string is double-quoted, uppercases each portion of the string and encloses each portion in double quotes.
* If the string contains periods and portions of the string are double-quoted, returns the double-quoted portions unchanged including the double quotes and returns the non-double-quoted portions uppercased and enclosed in double quotes.

CANONICALIZE(name VARCHAR2, canon\_name OUT VARCHAR2,

canon\_len BINARY\_INTEGER)

Parameters

name

String to be canonicalized.

canon\_name

The canonicalized string.

canon\_len

Number of bytes in name to canonicalize starting from the first character.

Examples

The following procedure applies the CANONICALIZE procedure on its input parameter and displays the results.

CREATE OR REPLACE PROCEDURE canonicalize (

p\_name VARCHAR2,

p\_length BINARY\_INTEGER DEFAULT 30

)

IS

v\_canon VARCHAR2(100);

BEGIN

DBMS\_UTILITY.CANONICALIZE(p\_name,v\_canon,p\_length);

DBMS\_OUTPUT.PUT\_LINE('Canonicalized name ==>' || v\_canon || '<==');

DBMS\_OUTPUT.PUT\_LINE('Length: ' || LENGTH(v\_canon));

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

EXEC canonicalize('Identifier')

Canonicalized name ==>IDENTIFIER<==

Length: 10

EXEC canonicalize('"Identifier"')

Canonicalized name ==>Identifier<==

Length: 10

EXEC canonicalize('"\_+142%"')

Canonicalized name ==>\_+142%<==

Length: 6

EXEC canonicalize('abc.def.ghi')

Canonicalized name ==>"ABC"."DEF"."GHI"<==

Length: 17

EXEC canonicalize('"abc.def.ghi"')

Canonicalized name ==>abc.def.ghi<==

Length: 11

EXEC canonicalize('"abc".def."ghi"')

Canonicalized name ==>"abc"."DEF"."ghi"<==

Length: 17

EXEC canonicalize('"abc.def".ghi')

Canonicalized name ==>"abc.def"."GHI"<==

Length: 15

### COMMA\_TO\_TABLE

The COMMA\_TO\_TABLE procedure converts a comma-delimited list of names into a table of names. Each entry in the list becomes a table entry. The names must be formatted as valid identifiers.

COMMA\_TO\_TABLE(list VARCHAR2, tablen OUT BINARY\_INTEGER,

tab OUT { LNAME\_ARRAY | UNCL\_ARRAY })

Parameters

list

Comma-delimited list of names.

tablen

Number of entries in tab.

tab

Table containing the individual names in list.

LNAME\_ARRAY

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

UNCL\_ARRAY

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

Examples

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

CREATE OR REPLACE PROCEDURE comma\_to\_table (

p\_list VARCHAR2

)

IS

r\_lname DBMS\_UTILITY.LNAME\_ARRAY;

v\_length BINARY\_INTEGER;

BEGIN

DBMS\_UTILITY.COMMA\_TO\_TABLE(p\_list,v\_length,r\_lname);

FOR i IN 1..v\_length LOOP

DBMS\_OUTPUT.PUT\_LINE(r\_lname(i));

END LOOP;

END;

EXEC comma\_to\_table('edb.dept, edb.emp, edb.jobhist')

edb.dept

edb.emp

edb.jobhist

### DB\_VERSION

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

DB\_VERSION(version OUT VARCHAR2, compatibility OUT VARCHAR2)

Parameters

version

Database version number.

compatibility

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

Examples

The following anonymous block displays the database version information.

DECLARE

v\_version VARCHAR2(150);

v\_compat VARCHAR2(150);

BEGIN

DBMS\_UTILITY.DB\_VERSION(v\_version,v\_compat);

DBMS\_OUTPUT.PUT\_LINE('Version: ' || v\_version);

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

### EXEC\_DDL\_STATEMENT

The EXEC\_DDL\_STATEMENT provides the capability to execute a DDL command.

EXEC\_DDL\_STATEMENT(parse\_string VARCHAR2)

Parameters

parse\_string

The DDL command to be executed.

Examples

The following anonymous block creates the job table.

BEGIN

DBMS\_UTILITY.EXEC\_DDL\_STATEMENT(

'CREATE TABLE job (' ||

'jobno NUMBER(3),' ||

'jname VARCHAR2(9))'

);

END;

If the *parse\_string* does not include a valid DDL statement, Advanced Server returns the following error:

edb=#  exec dbms\_utility.exec\_ddl\_statement('select rownum from dual');

ERROR:  EDB-20001: 'parse\_string' must be a valid DDL statement

In this case, Advanced Server's behavior differs from Oracle's; Oracle accepts the invalid *parse\_string* without complaint.

### FORMAT\_CALL\_STACK

The FORMAT\_CALL\_STACK function returns the formatted contents of the current call stack.

DBMS\_UTILITY.FORMAT\_CALL\_STACK  
return VARCHAR2

This function can be used in a stored procedure, function or package to return the current call stack in a readable format. This function is useful for debugging purposes.

### GET\_CPU\_TIME

The GET\_CPU\_TIME function returns the CPU time in hundredths of a second from some arbitrary point in time.

cputime NUMBER GET\_CPU\_TIME

Parameters

cputime

Number of hundredths of a second of CPU time.

Examples

The following SELECT command retrieves the current CPU time, which is 603 hundredths of a second or .0603 seconds.

SELECT DBMS\_UTILITY.GET\_CPU\_TIME FROM DUAL;

get\_cpu\_time

--------------

603

### GET\_DEPENDENCY

The GET\_DEPENDENCY procedure provides the capability to list the objects that are dependent upon the specified object. GET\_DEPENDENCY does not show dependencies for functions or procedures.

GET\_DEPENDENCY(type VARCHAR2, schema VARCHAR2,

name VARCHAR2)

Parameters

type

The object type of name. Valid values are INDEX, PACKAGE, PACKAGE BODY, SEQUENCE, TABLE, TRIGGER, TYPE and VIEW.

schema

Name of the schema in which name exists.

name

Name of the object for which dependencies are to be obtained.

Examples

The following anonymous block finds dependencies on the EMP table.

BEGIN

DBMS\_UTILITY.GET\_DEPENDENCY('TABLE','public','EMP');

END;

DEPENDENCIES ON public.EMP

------------------------------------------------------------------

\*TABLE public.EMP()

\* CONSTRAINT c public.emp()

\* CONSTRAINT f public.emp()

\* CONSTRAINT p public.emp()

\* TYPE public.emp()

\* CONSTRAINT c public.emp()

\* CONSTRAINT f public.jobhist()

\* VIEW .empname\_view()

### GET\_HASH\_VALUE

The GET\_HASH\_VALUE function provides the capability to compute a hash value for a given string.

hash NUMBER GET\_HASH\_VALUE(name VARCHAR2, base NUMBER,

hash\_size NUMBER)

Parameters

name

The string for which a hash value is to be computed.

base

Starting value at which hash values are to be generated.

hash\_size

The number of hash values for the desired hash table.

hash

The generated hash value.

Examples

The following anonymous block creates a table of hash values using the ename column of the emp table and then displays the key along with the hash value. The hash values start at 100 with a maximum of 1024 distinct values.

DECLARE

v\_hash NUMBER;

TYPE hash\_tab IS TABLE OF NUMBER INDEX BY VARCHAR2(10);

r\_hash HASH\_TAB;

CURSOR emp\_cur IS SELECT ename FROM emp;

BEGIN

FOR r\_emp IN emp\_cur LOOP

r\_hash(r\_emp.ename) :=

DBMS\_UTILITY.GET\_HASH\_VALUE(r\_emp.ename,100,1024);

END LOOP;

FOR r\_emp IN emp\_cur LOOP

DBMS\_OUTPUT.PUT\_LINE(RPAD(r\_emp.ename,10) || ' ' ||

r\_hash(r\_emp.ename));

END LOOP;

END;

SMITH 377

ALLEN 740

WARD 718

JONES 131

MARTIN 176

BLAKE 568

CLARK 621

SCOTT 1097

KING 235

TURNER 850

ADAMS 156

JAMES 942

FORD 775

MILLER 148

### GET\_PARAMETER\_VALUE

The GET\_PARAMETER\_VALUE procedure provides the capability to retrieve database initialization parameter settings.

status BINARY\_INTEGER GET\_PARAMETER\_VALUE(parnam VARCHAR2,

intval OUT INTEGER, strval OUT VARCHAR2)

Parameters

parnam

Name of the parameter whose value is to be returned. The parameters are listed in the pg\_settings system view.

intval

Value of an integer parameter or the length of strval.

strval

Value of a string parameter.

status

Returns 0 if the parameter value is INTEGER or BOOLEAN. Returns 1 if the parameter value is a string.

Examples

The following anonymous block shows the values of two initialization parameters.

DECLARE

v\_intval INTEGER;

v\_strval VARCHAR2(80);

BEGIN

DBMS\_UTILITY.GET\_PARAMETER\_VALUE('max\_fsm\_pages', v\_intval, v\_strval);

DBMS\_OUTPUT.PUT\_LINE('max\_fsm\_pages' || ': ' || v\_intval);

DBMS\_UTILITY.GET\_PARAMETER\_VALUE('client\_encoding', v\_intval, v\_strval);

DBMS\_OUTPUT.PUT\_LINE('client\_encoding' || ': ' || v\_strval);

END;

max\_fsm\_pages: 72625

client\_encoding: SQL\_ASCII

### GET\_TIME

The GET\_TIME function provides the capability to return the current time in hundredths of a second.

time NUMBER GET\_TIME

Parameters

time

Number of hundredths of a second from the time in which the program is started.

Examples

The following example shows calls to the GET\_TIME function.

SELECT DBMS\_UTILITY.GET\_TIME FROM DUAL;

get\_time

----------

1555860

SELECT DBMS\_UTILITY.GET\_TIME FROM DUAL;

get\_time

----------

1556037

### NAME\_TOKENIZE

The NAME\_TOKENIZE procedure parses a name into its component parts. Names without double quotes are uppercased. The double quotes are stripped from names with double quotes.

NAME\_TOKENIZE(name VARCHAR2, a OUT VARCHAR2,  
 b OUT VARCHAR2,c OUT VARCHAR2, dblink OUT VARCHAR2,  
 nextpos OUT BINARY\_INTEGER)

Parameters

name

String containing a name in the following format:

a[.b[.c]][@dblink ]

a

Returns the leftmost component.

b

Returns the second component, if any.

c

Returns the third component, if any.

dblink

Returns the database link name.

nextpos

Position of the last character parsed in name.

Examples

The following stored procedure is used to display the returned parameter values of the NAME\_TOKENIZE procedure for various names.

CREATE OR REPLACE PROCEDURE name\_tokenize (

p\_name VARCHAR2

)

IS

v\_a VARCHAR2(30);

v\_b VARCHAR2(30);

v\_c VARCHAR2(30);

v\_dblink VARCHAR2(30);

v\_nextpos BINARY\_INTEGER;

BEGIN

DBMS\_UTILITY.NAME\_TOKENIZE(p\_name,v\_a,v\_b,v\_c,v\_dblink,v\_nextpos);

DBMS\_OUTPUT.PUT\_LINE('name : ' || p\_name);

DBMS\_OUTPUT.PUT\_LINE('a : ' || v\_a);

DBMS\_OUTPUT.PUT\_LINE('b : ' || v\_b);

DBMS\_OUTPUT.PUT\_LINE('c : ' || v\_c);

DBMS\_OUTPUT.PUT\_LINE('dblink : ' || v\_dblink);

DBMS\_OUTPUT.PUT\_LINE('nextpos: ' || v\_nextpos);

END;

Tokenize the name, emp:

BEGIN

name\_tokenize('emp');

END;

name : emp

a : EMP

b :

c :

dblink :

nextpos: 3

Tokenize the name, edb.list\_emp:

BEGIN

name\_tokenize('edb.list\_emp');

END;

name : edb.list\_emp

a : EDB

b : LIST\_EMP

c :

dblink :

nextpos: 12

Tokenize the name, "edb"."Emp\_Admin".update\_emp\_sal:

BEGIN

name\_tokenize('"edb"."Emp\_Admin".update\_emp\_sal');

END;

name : "edb"."Emp\_Admin".update\_emp\_sal

a : edb

b : Emp\_Admin

c : UPDATE\_EMP\_SAL

dblink :

nextpos: 32

Tokenize the name edb.emp@edb\_dblink:

BEGIN

name\_tokenize('edb.emp@edb\_dblink');

END;

name : edb.emp@edb\_dblink

a : EDB

b : EMP

c :

dblink : EDB\_DBLINK

nextpos: 18

### TABLE\_TO\_COMMA

The TABLE\_TO\_COMMA procedure converts table of names into a comma-delimited list of names. Each table entry becomes a list entry. The names must be formatted as valid identifiers.

TABLE\_TO\_COMMA(tab { LNAME\_ARRAY | UNCL\_ARRAY },

tablen OUT BINARY\_INTEGER, list OUT VARCHAR2)

Parameters

tab

Table containing names.

LNAME\_ARRAY

A DBMS\_UTILITY LNAME\_ARRAY (as described in Section [3.17.1](#_LNAME_ARRAY)).

UNCL\_ARRAY

A DBMS\_UTILITY UNCL\_ARRAY (as described in Section [3.17.2](#_UNCL_ARRAY)).

tablen

Number of entries in list.

list

Comma-delimited list of names from tab.

Examples

The following example first uses the COMMA\_TO\_TABLE procedure to convert a comma-delimited list to a table. The TABLE\_TO\_COMMA procedure then converts the table back to a comma-delimited list that is displayed.

CREATE OR REPLACE PROCEDURE table\_to\_comma (

p\_list VARCHAR2

)

IS

r\_lname DBMS\_UTILITY.LNAME\_ARRAY;

v\_length BINARY\_INTEGER;

v\_listlen BINARY\_INTEGER;

v\_list VARCHAR2(80);

BEGIN

DBMS\_UTILITY.COMMA\_TO\_TABLE(p\_list,v\_length,r\_lname);

DBMS\_OUTPUT.PUT\_LINE('Table Entries');

DBMS\_OUTPUT.PUT\_LINE('-------------');

FOR i IN 1..v\_length LOOP

DBMS\_OUTPUT.PUT\_LINE(r\_lname(i));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('-------------');

DBMS\_UTILITY.TABLE\_TO\_COMMA(r\_lname,v\_listlen,v\_list);

DBMS\_OUTPUT.PUT\_LINE('Comma-Delimited List: ' || v\_list);

END;

EXEC table\_to\_comma('edb.dept, edb.emp, edb.jobhist')

Table Entries

-------------

edb.dept

edb.emp

edb.jobhist

-------------

Comma-Delimited List: edb.dept, edb.emp, edb.jobhist

## UTL\_ENCODE

The UTL\_ENCODE package provides a way to encode and decode data. Advanced Serve supports the following functions and procedures:

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| BASE64\_DECODE(*r*) | RAW | Use the BASE64\_DECODE function to translate a Base64 encoded string to the original RAW value. |
| BASE64\_ENCODE(*r*) | RAW | Use the BASE64\_ENCODE function to translate a RAW string to an encoded Base64 value. |
| BASE64\_ENCODE(*loid*) | TEXT | Use the BASE64\_ENCODE function to translate a TEXT string to an encoded Base64 value. |
| MIMEHEADER\_DECODE(*buf*) | VARCHAR2 | Use the MIMEHEADER\_DECODE function to translate an encoded MIMEHEADER formatted string to it's original value. |
| MIMEHEADER\_ENCODE(*buf*, *encode*\_*charset*, *encoding*) | VARCHAR2 | Use the MIMEHEADER\_ENCODE function to convert and encode a string in MIMEHEADER format. |
| QUOTED\_PRINTABLE\_DECODE(*r*) | RAW | Use the QUOTED\_PRINTABLE\_DECODE function to translate an encoded string to a RAW value. |
| QUOTED\_PRINTABLE\_ENCODE(*r*) | RAW | Use the QUOTED\_PRINTABLE\_ENCODE function to translate an input string to a quoted-printable formatted RAW value. |
| TEXT\_DECODE(*buf*, *encode*\_*charset*, *encoding*) | VARCHAR2 | Use the TEXT\_DECODE function to decode a string encoded by TEXT\_ENCODE. |
| TEXT\_ENCODE(*buf*, *encode*\_*charset*, *encoding*) | VARCHAR2 | Use the TEXT\_ENCODE function to translate a string to a user-specified character set, and then encode the string. |
| UUDECODE(*r*) | RAW | Use the UUDECODE function to translate a uuencode encoded string to a RAW value. |
| UUENCODE(*r*, *type*, *filename*, *permission*) | RAW | Use the UUENCODE function to translate a RAW string to an encoded uuencode value. |

### 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:

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 (using BASE64\_ENCODE), and then decodes (using BASE64\_DECODE) a string that contains the text abc:

edb=# SELECT UTL\_ENCODE.BASE64\_ENCODE(CAST ('abc' AS RAW));

base64\_encode

---------------

YWJj

(1 row)

edb=# SELECT UTL\_ENCODE.BASE64\_DECODE(CAST ('YWJj' AS RAW));

base64\_decode

---------------

abc

(1 row)

### BASE64\_ENCODE

Use the BASE64\_ENCODE function to translate and encode a string in Base64 format (as described in RFC 4648). This function can be useful when composing MIME email that you intend to send using the UTL\_SMTP package. The BASE64\_ENCODE function has two signatures:

BASE64\_ENCODE(r IN RAW)

and

BASE64\_ENCODE(loid IN OID)

This function returns a RAW value or an OID.

Parameters

*r*

*r* specifies the RAW string that will be translated to Base64.

*loid*

*loid* specifies the object ID of a large object that will be translated to Base64.

Examples

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 (using BASE64\_ENCODE), and then decodes (using BASE64\_DECODE) a string that contains the text abc:

edb=# SELECT UTL\_ENCODE.BASE64\_ENCODE(CAST ('abc' AS RAW));

base64\_encode

---------------

YWJj

(1 row)

edb=# SELECT UTL\_ENCODE.BASE64\_DECODE(CAST ('YWJj' AS RAW));

base64\_decode

---------------

abc

(1 row)

### MIMEHEADER\_DECODE

Use the MIMEHEADER\_DECODE function to decode values that are encoded by the MIMEHEADER\_ENCODE function. The signature is:

MIMEHEADER\_DECODE(*buf* IN VARCHAR2)

This function returns a VARCHAR2 value.

Parameters

*buf*

*buf* contains the value (encoded by MIMEHEADER\_ENCODE) that will be decoded.

Examples

The following examples use the MIMEHEADER\_ENCODE and MIMEHEADER\_DECODE functions to first encode, and then decode a string:

edb=# SELECT UTL\_ENCODE.MIMEHEADER\_ENCODE('What is the date?') FROM DUAL;

mimeheader\_encode

------------------------------

=?UTF8?Q?What is the date??=

(1 row)

edb=# SELECT UTL\_ENCODE.MIMEHEADER\_DECODE('=?UTF8?Q?What is the date??=') FROM DUAL;

mimeheader\_decode

-------------------

What is the date?

(1 row)

### MIMEHEADER\_ENCODE

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

MIMEHEADER\_ENCODE(*buf* IN VARCHAR2, *encode\_charset* IN VARCHAR2 DEFAULT NULL, *encoding* IN INTEGER DEFAULT NULL)

This function returns a VARCHAR2 value.

Parameters

*buf*

*buf* contains the string that will be formatted and encoded. The string is a VARCHAR2 value.

*encode\_charset*

*encode\_charset* specifies the character set to which the string will be converted before being formatted and encoded. The default value is NULL.

*encoding*

*encoding* specifies the encoding type used when encoding the string. You can specify:

* Q to enable quoted-printable encoding. If you do not specify a value, MIMEHEADER\_ENCODE will use quoted-printable encoding.
* B to enable base-64 encoding.

Examples

The following examples use the MIMEHEADER\_ENCODE and MIMEHEADER\_DECODE functions to first encode, and then decode a string:

edb=# SELECT UTL\_ENCODE.MIMEHEADER\_ENCODE('What is the date?') FROM DUAL;

mimeheader\_encode

------------------------------

=?UTF8?Q?What is the date??=

(1 row)

edb=# SELECT UTL\_ENCODE.MIMEHEADER\_DECODE('=?UTF8?Q?What is the date??=') FROM DUAL;

mimeheader\_decode

-------------------

What is the date?

(1 row)

### 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:

<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)

### 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)

### TEXT\_DECODE

Use the TEXT\_DECODE function to translate and decode an encoded string to the VARCHAR2 value that was originally encoded by the TEXT\_ENCODE function. The signature is:

TEXT\_DECODE(*buf* IN VARCHAR2, encode\_charset IN VARCHAR2 DEFAULT NULL, encoding IN PLS\_INTEGER DEFAULT NULL)

This function returns a VARCHAR2 value.

Parameters

*buf*

*buf* contains the encoded string that will be translated to the original value encoded by TEXT\_ENCODE.

*encode\_charset*

*encode\_charset* specifies the character set to which the string will be translated before encoding. The default value is NULL.

*encoding*

*encoding* specifies the encoding type used by TEXT\_DECODE. Specify:

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

Examples

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

edb=# SELECT UTL\_ENCODE.TEXT\_ENCODE('What is the date?', 'BIG5', UTL\_ENCODE.BASE64) FROM DUAL;

text\_encode

--------------------------

V2hhdCBpcyB0aGUgZGF0ZT8=

(1 row)

edb=# SELECT UTL\_ENCODE.TEXT\_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5', UTL\_ENCODE.BASE64) FROM DUAL;

text\_decode

-------------------

What is the date?

(1 row)

### TEXT\_ENCODE

Use the TEXT\_ENCODE function to translate a string to a user-specified character set, and then encode the string. The signature is:

TEXT\_DECODE(*buf* IN VARCHAR2, *encode\_charset* IN VARCHAR2 DEFAULT NULL, *encoding* IN PLS\_INTEGER DEFAULT NULL)

This function returns a VARCHAR2 value.

Parameters

*buf*

*buf* contains the encoded string that will be translated to the specified character set and encoded by TEXT\_ENCODE.

*encode\_charset*

*encode\_charset* specifies the character set to which the value will be translated before encoding. The default value is NULL.

*encoding*

*encoding* specifies the encoding type used by TEXT\_ENCODE. Specify:

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

Examples

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

edb=# SELECT UTL\_ENCODE.TEXT\_ENCODE('What is the date?', 'BIG5', UTL\_ENCODE.BASE64) FROM DUAL;

text\_encode

--------------------------

V2hhdCBpcyB0aGUgZGF0ZT8=

(1 row)

edb=# SELECT UTL\_ENCODE.TEXT\_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5', UTL\_ENCODE.BASE64) FROM DUAL;

text\_decode

-------------------

What is the date?

(1 row)

### UUDECODE

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

UUDECODE(r IN RAW)

This function returns a RAW value.

Note: If you are using the Advanced Server UUDECODE function to decode uuencoded data that was created by the Oracle implementation of the UTL\_ENCODE.UUENCODE function, then you must first set the Advanced Server configuration parameter utl\_encode.uudecode\_redwood to TRUE before invoking the Advanced Server UUDECODE function on the Oracle-created data. (For example, this situation may occur if you migrated Oracle tables containing uuencoded data to an Advanced Server database.)

The uuencoded data created by the Oracle version of the UUENCODE function results in a format that differs from the uuencoded data created by the Advanced Server UUENCODE function. As a result, attempting to use the Advanced Server UUDECODE function on the Oracle uuencoded data results in an error unless the configuration parameter utl\_encode.uudecode\_redwood is set to TRUE.

However, if you are using the Advanced Server UUDECODE function on uuencoded data created by the Advanced Server UUENCODE function, then utl\_encode.uudecode\_redwood must be set to FALSE, which is the default setting.

Parameters

*r*

*r* contains the uuencoded string that will be translated to RAW.

Examples

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 uses UUENCODE and UUDECODE to first encode and then decode a string:

edb=# SET bytea\_output = escape;

SET

edb=# SELECT UTL\_ENCODE.UUENCODE('What is the date?') FROM DUAL;

uuencode

--------------------------------------------------------------------

begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012

(1 row)

edb=# SELECT UTL\_ENCODE.UUDECODE

edb-# ('begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012')

edb-# FROM DUAL;

uudecode

-------------------

What is the date?

(1 row)

### UUENCODE

Use the UUENCODE function to translate RAW data into a uuencode formatted encoded string. The signature is:

UUENCODE(*r* IN RAW, *type* IN INTEGER DEFAULT 1, *filename* IN VARCHAR2 DEFAULT NULL, *permission* IN VARCHAR2 DEFAULT NULL)

This function returns a RAW value.

Parameters

*r*

*r* contains the RAW string that will be translated to uuencode format.

*type*

*type* is an INTEGER value or constant that specifies the type of uuencoded string that will be returned; the default value is 1. The possible values are:

|  |  |
| --- | --- |
| Value | Constant |
| 1 | complete |
| 2 | header\_piece |
| 3 | middle\_piece |
| 4 | end\_piece |

*filename*

*filename* is a VARCHAR2 value that specifies the file name that you want to embed in the encoded form; if you do not specify a file name, UUENCODE will include a filename of uuencode.txt in the encoded form.

*permission*

*permission* is a VARCHAR2 that specifies the permission mode; the default value is NULL.

Examples

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 uses UUENCODE and UUDECODE to first encode and then decode a string:

edb=# SET bytea\_output = escape;

SET

edb=# SELECT UTL\_ENCODE.UUENCODE('What is the date?') FROM DUAL;

uuencode

--------------------------------------------------------------------

begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012

(1 row)

edb=# SELECT UTL\_ENCODE.UUDECODE

edb-# ('begin 0 uuencode.txt\01215VAA="!I<R!T:&4@9&%T93\\`\012`\012end\012')

edb-# FROM DUAL;

uudecode

-------------------

What is the date?

(1 row)

## UTL\_FILE

The UTL\_FILE package provides the capability to read from, and write to files on the operating system’s file system. Non-superusers must be granted EXECUTE privilege on the UTL\_FILE package by a superuser before using any of the functions or procedures in the package. For example the following command grants the privilege to user mary:

GRANT EXECUTE ON PACKAGE SYS.UTL\_FILE TO mary;

Also, the operating system username, enterprisedb, must have the appropriate read and/or write permissions on the directories and files to be accessed using the UTL\_FILE functions and procedures. If the required file permissions are not in place, an exception is thrown in the UTL\_FILE function or procedure.

A handle to the file to be written to, or read from is used to reference the file. The file handle is defined by a public variable in the UTL\_FILE package named, UTL\_FILE.FILE\_TYPE. A variable of type FILE\_TYPE must be declared to receive the file handle returned by calling the FOPEN function. The file handle is then used for all subsequent operations on the file.

References to directories on the file system are done using the directory name or alias that is assigned to the directory using the CREATE DIRECTORY command. The procedures and functions available in the UTL\_FILE package are listed in the following table:

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| FCLOSE(file IN OUT) | n/a | Closes the specified file identified by file. |
| FCLOSE\_ALL | n/a | Closes all open files. |
| FCOPY(location, filename, dest\_dir, dest\_file [, start\_line [, end\_line ] ]) | n/a | Copies filename in the directory identified by location to file, dest\_file, in directory, dest\_dir, starting from line, start\_line, to line, end\_line. |
| FFLUSH(file) | n/a | Forces data in the buffer to be written to disk in the file identified by file. |
| FOPEN(location, filename, open\_mode [, max\_linesize ]) | FILE\_TYPE | Opens file, filename, in the directory identified by location. |
| FREMOVE(location, filename) | n/a | Removes the specified file from the file system. |
| FRENAME(location, filename, dest\_dir, dest\_file [, overwrite ]) | n/a | Renames the specified file. |
| GET\_LINE(file, buffer OUT) | n/a | Reads a line of text into variable, buffer, from the file identified by file. |
| IS\_OPEN(file) | BOOLEAN | Determines whether or not the given file is open. |
| NEW\_LINE(file [, lines ]) | n/a | Writes an end-of-line character sequence into the file. |
| PUT(file, buffer) | n/a | Writes buffer to the given file. PUT does not write an end-of-line character sequence. |
| PUT\_LINE(file, buffer) | n/a | Writes buffer to the given file. An end-of-line character sequence is added by the PUT\_LINE procedure. |
| PUTF(file, format [, arg1 ] [, ...]) | n/a | Writes a formatted string to the given file. Up to five substitution parameters, arg1,...arg5 may be specified for replacement in format. |

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

**UTL\_FILE Exception Codes**

If a call to a UTL\_FILE procedure or function raises an exception, you can use the condition name to catch the exception. The UTL\_FILE package reports the following exception codes compatible with Oracle databases:

|  |  |
| --- | --- |
| **Exception Code** | **Condition name** |
| -29283 | invalid\_operation |
| -29285 | write\_error |
| -29284 | read\_error |
| -29282 | invalid\_filehandle |
| -29287 | invalid\_maxlinesize |
| -29281 | invalid\_mode |
| -29280 | invalid\_path |

### Setting File Permissions with utl\_file.umask

When a UTL\_FILE function or procedure creates a file, there are default file permissions as shown by the following.

-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile

Note that all permissions are denied on users belonging to the enterprisedb group as well as all other users. Only the enterprisedb user has read and write permissions on the created file.

If you wish to have a different set of file permissions on files created by the UTL\_FILE functions and procedures, you can accomplish this by setting the utl\_file.umask configuration parameter.

The utl\_file.umask parameter sets the file mode creation mask or simply, the mask, in a manner similar to the Linux umask command. This is for usage only within the Advanced Server UTL\_FILE package.

Note: The utl\_file.umask parameter is not supported on Windows systems.

The value specified for utl\_file.umask is a 3 or 4-character octal string that would be valid for the Linux umask command. The setting determines the permissions on files created by the UTL\_FILE functions and procedures. (Refer to any information source regarding Linux or Unix systems for information on file permissions and the usage of the umask command.)

The following is an example of setting the file permissions with utl\_file.umask.

First, set up the directory in the file system to be used by the UTL\_FILE package. Be sure the operating system account, enterprisedb or postgres, whichever is applicable, can read and write in the directory.

mkdir /tmp/utldir

chmod 777 /tmp/utldir

The CREATE DIRECTORY command is issued in psql to create the directory database object using the file system directory created in the preceding step.

CREATE DIRECTORY utldir AS '/tmp/utldir';

Set the utl\_file.umask configuration parameter. The following setting allows the file owner any permission. Group users and other users are permitted any permission except for the execute permission.

SET utl\_file.umask TO '0011';

In the same session during which the utl\_file.umask parameter is set to the desired value, run the UTL\_FILE functions and procedures.

DECLARE

v\_utlfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'utldir';

v\_filename VARCHAR2(20) := 'utlfile';

BEGIN

v\_utlfile := UTL\_FILE.FOPEN(v\_directory, v\_filename, 'w');

UTL\_FILE.PUT\_LINE(v\_utlfile, 'Simple one-line file');

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_utlfile);

END;

The permission settings on the resulting file show that group users and other users have read and write permissions on the file as well as the file owner.

$ pwd

/tmp/utldir

$ ls -l

total 4

-rw-rw-rw- 1 enterprisedb enterprisedb 21 Jul 24 16:04 utlfile

This parameter can also be set on a per role basis with the ALTER ROLE command, on a per database basis with the ALTER DATABASE command, or for the entire database server instance by setting it in the postgresql.conf file.

### FCLOSE

The FCLOSE procedure closes an open file.

FCLOSE(file IN OUT FILE\_TYPE)

Parameters

file

Variable of type FILE\_TYPE containing a file handle of the file to be closed.

### FCLOSE\_ALL

The FLCLOSE\_ALL procedures closes all open files. The procedure executes successfully even if there are no open files to close.

FCLOSE\_ALL

### FCOPY

The FCOPY procedure copies text from one file to another.

FCOPY(location VARCHAR2, filename VARCHAR2,

dest\_dir VARCHAR2, dest\_file VARCHAR2

[, start\_line PLS\_INTEGER [, end\_line PLS\_INTEGER ] ])

Parameters

location

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory containing the file to be copied.

filename

Name of the source file to be copied.

dest\_dir

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory to which the file is to be copied.

dest\_file

Name of the destination file.

start\_line

Line number in the source file from which copying will begin. The default is 1.

end\_line

Line number of the last line in the source file to be copied. If omitted or null, copying will go to the last line of the file.

Examples

The following makes a copy of a file, C:\TEMP\EMPDIR\empfile.csv, containing a comma-delimited list of employees from the emp table. The copy, empcopy.csv, is then listed.

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

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_src\_dir VARCHAR2(50) := 'empdir';

v\_src\_file VARCHAR2(20) := 'empfile.csv';

v\_dest\_dir VARCHAR2(50) := 'empdir';

v\_dest\_file VARCHAR2(20) := 'empcopy.csv';

v\_emprec VARCHAR2(120);

v\_count INTEGER := 0;

BEGIN

UTL\_FILE.FCOPY(v\_src\_dir,v\_src\_file,v\_dest\_dir,v\_dest\_file);

v\_empfile := UTL\_FILE.FOPEN(v\_dest\_dir,v\_dest\_file,'r');

DBMS\_OUTPUT.PUT\_LINE('The following is the destination file, ''' ||

v\_dest\_file || '''');

LOOP

UTL\_FILE.GET\_LINE(v\_empfile,v\_emprec);

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

v\_count := v\_count + 1;

END LOOP;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

UTL\_FILE.FCLOSE(v\_empfile);

DBMS\_OUTPUT.PUT\_LINE(v\_count || ' records retrieved');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

The following is the destination file, 'empcopy.csv'

7369,SMITH,CLERK,7902,17-DEC-80,800,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81,1600,300,30

7521,WARD,SALESMAN,7698,22-FEB-81,1250,500,30

7566,JONES,MANAGER,7839,02-APR-81,2975,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81,1250,1400,30

7698,BLAKE,MANAGER,7839,01-MAY-81,2850,,30

7782,CLARK,MANAGER,7839,09-JUN-81,2450,,10

7788,SCOTT,ANALYST,7566,19-APR-87,3000,,20

7839,KING,PRESIDENT,,17-NOV-81,5000,,10

7844,TURNER,SALESMAN,7698,08-SEP-81,1500,0,30

7876,ADAMS,CLERK,7788,23-MAY-87,1100,,20

7900,JAMES,CLERK,7698,03-DEC-81,950,,30

7902,FORD,ANALYST,7566,03-DEC-81,3000,,20

7934,MILLER,CLERK,7782,23-JAN-82,1300,,10

14 records retrieved

### FFLUSH

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

FFLUSH(file FILE\_TYPE)

Parameters

file

Variable of type FILE\_TYPE containing a file handle.

Examples

Each line is flushed after the NEW\_LINE procedure is called.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

UTL\_FILE.PUT(v\_empfile,i.empno);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.ename);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.job);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.mgr);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.hiredate);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.sal);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.comm);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.deptno);

UTL\_FILE.NEW\_LINE(v\_empfile);

UTL\_FILE.FFLUSH(v\_empfile);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

END;

### FOPEN

The FOPEN function opens a file for I/O.

filetype FILE\_TYPE FOPEN(location VARCHAR2,   
 filename VARCHAR2,open\_mode VARCHAR2   
 [, max\_linesize BINARY\_INTEGER ])

Parameters

location

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory containing the file to be opened.

filename

Name of the file to be opened.

open\_mode

Mode in which the file will be opened. Modes are: a - append to file; r - read from file; w - write to file.

max\_linesize

Maximum size of a line in characters. In read mode, an exception is thrown if an attempt is made to read a line exceeding max\_linesize. In write and append modes, an exception is thrown if an attempt is made to write a line exceeding max\_linesize. The end-of-line character(s) are not included in determining if the maximum line size is exceeded. This behavior is not compatible with Oracle databases; Oracle does count the end-of-line character(s).

filetype

Variable of type FILE\_TYPE containing the file handle of the opened file.

### FREMOVE

The FREMOVE procedure removes a file from the system.

FREMOVE(location VARCHAR2, filename VARCHAR2)

An exception is thrown if the file to be removed does not exist.

Parameters

location

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory containing the file to be removed.

filename

Name of the file to be removed.

Examples

The following removes file empfile.csv.

DECLARE

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

BEGIN

UTL\_FILE.FREMOVE(v\_directory,v\_filename);

DBMS\_OUTPUT.PUT\_LINE('Removed file: ' || v\_filename);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

Removed file: empfile.csv

### FRENAME

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

FRENAME(location VARCHAR2, filename VARCHAR2,

dest\_dir VARCHAR2, dest\_file VARCHAR2,   
 [ overwrite BOOLEAN ])

Parameters

location

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory containing the file to be renamed.

filename

Name of the source file to be renamed.

dest\_dir

Directory name, as stored in pg\_catalog.edb\_dir.dirname, of the directory to which the renamed file is to exist.

dest\_file

New name of the original file.

overwrite

Replaces any existing file named dest\_file in dest\_dir if set to TRUE, otherwise an exception is thrown if set to FALSE. This is the default.

Examples

The following renames a file, C:\TEMP\EMPDIR\empfile.csv, containing a comma-delimited list of employees from the emp table. The renamed file, C:\TEMP\NEWDIR\newemp.csv, is then listed.

CREATE DIRECTORY "newdir" AS 'C:/TEMP/NEWDIR';

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_src\_dir VARCHAR2(50) := 'empdir';

v\_src\_file VARCHAR2(20) := 'empfile.csv';

v\_dest\_dir VARCHAR2(50) := 'newdir';

v\_dest\_file VARCHAR2(50) := 'newemp.csv';

v\_replace BOOLEAN := FALSE;

v\_emprec VARCHAR2(120);

v\_count INTEGER := 0;

BEGIN

UTL\_FILE.FRENAME(v\_src\_dir,v\_src\_file,v\_dest\_dir,

v\_dest\_file,v\_replace);

v\_empfile := UTL\_FILE.FOPEN(v\_dest\_dir,v\_dest\_file,'r');

DBMS\_OUTPUT.PUT\_LINE('The following is the renamed file, ''' ||

v\_dest\_file || '''');

LOOP

UTL\_FILE.GET\_LINE(v\_empfile,v\_emprec);

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

v\_count := v\_count + 1;

END LOOP;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

UTL\_FILE.FCLOSE(v\_empfile);

DBMS\_OUTPUT.PUT\_LINE(v\_count || ' records retrieved');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

The following is the renamed file, 'newemp.csv'

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

14 records retrieved

### GET\_LINE

The GET\_LINE procedure reads a line of text from a given file up to, but not including the end-of-line terminator. A NO\_DATA\_FOUND exception is thrown when there are no more lines to read.

GET\_LINE(file FILE\_TYPE, buffer OUT VARCHAR2)

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the opened file.

buffer

Variable to receive a line from the file.

Examples

The following anonymous block reads through and displays the records in file empfile.csv.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

v\_emprec VARCHAR2(120);

v\_count INTEGER := 0;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'r');

LOOP

UTL\_FILE.GET\_LINE(v\_empfile,v\_emprec);

DBMS\_OUTPUT.PUT\_LINE(v\_emprec);

v\_count := v\_count + 1;

END LOOP;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

UTL\_FILE.FCLOSE(v\_empfile);

DBMS\_OUTPUT.PUT\_LINE('End of file ' || v\_filename || ' - ' ||

v\_count || ' records retrieved');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

End of file empfile.csv - 14 records retrieved

### IS\_OPEN

The IS\_OPEN function determines whether or not the given file is open.

status BOOLEAN IS\_OPEN(file FILE\_TYPE)

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to be tested.

status

TRUE if the given file is open, FALSE otherwise.

### NEW\_LINE

The NEW\_LINE procedure writes an end-of-line character sequence in the file.

NEW\_LINE(file FILE\_TYPE [, lines INTEGER ])

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to which end-of-line character sequences are to be written.

lines

Number of end-of-line character sequences to be written. The default is one.

Examples

A file containing a double-spaced list of employee records is written.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

UTL\_FILE.PUT(v\_empfile,i.empno);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.ename);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.job);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.mgr);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.hiredate);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.sal);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.comm);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.deptno);

UTL\_FILE.NEW\_LINE(v\_empfile,2);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

END;

Created file: empfile.csv

This file is then displayed:

C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### PUT

The PUT procedure writes a string to the given file. No end-of-line character sequence is written at the end of the string. Use the NEW\_LINE procedure to add an end-of-line character sequence.

PUT(file FILE\_TYPE, buffer { DATE | NUMBER | TIMESTAMP |

VARCHAR2 })

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to which the given string is to be written.

buffer

Text to be written to the specified file.

Examples

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

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

UTL\_FILE.PUT(v\_empfile,i.empno);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.ename);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.job);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.mgr);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.hiredate);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.sal);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.comm);

UTL\_FILE.PUT(v\_empfile,',');

UTL\_FILE.PUT(v\_empfile,i.deptno);

UTL\_FILE.NEW\_LINE(v\_empfile);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

END;

Created file: empfile.csv

The following is the contents of empfile.csv created above:

C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### PUT\_LINE

The PUT\_LINE procedure writes a single line to the given file including an end-of-line character sequence.

PUT\_LINE(file FILE\_TYPE,   
 buffer {DATE|NUMBER|TIMESTAMP|VARCHAR2})

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to which the given line is to be written.

buffer

Text to be written to the specified file.

Examples

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

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

v\_emprec VARCHAR2(120);

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

v\_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||

NVL(LTRIM(TO\_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||

',' || i.sal || ',' ||

NVL(LTRIM(TO\_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;

UTL\_FILE.PUT\_LINE(v\_empfile,v\_emprec);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

END;

The following is the contents of empfile.csv created above:

C:\TEMP\EMPDIR>TYPE empfile.csv

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20

7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30

7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30

7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20

7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30

7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30

7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10

7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20

7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10

7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30

7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20

7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30

7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20

7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10

### PUTF

The PUTF procedure writes a formatted string to the given file.

PUTF(file FILE\_TYPE, format VARCHAR2 [, arg1 VARCHAR2]

[, ...])

Parameters

file

Variable of type FILE\_TYPE containing the file handle of the file to which the formatted line is to be written.

format

String to format the text written to the file. The special character sequence, %s, is substituted by the value of arg. The special character sequence, \n, indicates a new line. Note, however, in Advanced Server, a new line character must be specified with two consecutive backslashes instead of one - \\n. This characteristic is not compatible with Oracle databases.

arg1

Up to five arguments, arg1,...arg5, to be substituted in the format string for each occurrence of %s. The first arg is substituted for the first occurrence of %s, the second arg is substituted for the second occurrence of %s, etc.

Examples

The following anonymous block produces formatted output containing data from the emp table. Note the use of the E literal syntax and double backslashes for the new line character sequence in the format string which are not compatible with Oracle databases.

DECLARE

v\_empfile UTL\_FILE.FILE\_TYPE;

v\_directory VARCHAR2(50) := 'empdir';

v\_filename VARCHAR2(20) := 'empfile.csv';

v\_format VARCHAR2(200);

CURSOR emp\_cur IS SELECT \* FROM emp ORDER BY empno;

BEGIN

v\_format := E'%s %s, %s\\nSalary: $%s Commission: $%s\\n\\n';

v\_empfile := UTL\_FILE.FOPEN(v\_directory,v\_filename,'w');

FOR i IN emp\_cur LOOP

UTL\_FILE.PUTF(v\_empfile,v\_format,i.empno,i.ename,i.job,i.sal,

NVL(i.comm,0));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('Created file: ' || v\_filename);

UTL\_FILE.FCLOSE(v\_empfile);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('SQLERRM: ' || SQLERRM);

DBMS\_OUTPUT.PUT\_LINE('SQLCODE: ' || SQLCODE);

END;

Created file: empfile.csv

The following is the contents of empfile.csv created above:

C:\TEMP\EMPDIR>TYPE empfile.csv

7369 SMITH, CLERK

Salary: $800.00 Commission: $0

7499 ALLEN, SALESMAN

Salary: $1600.00 Commission: $300.00

7521 WARD, SALESMAN

Salary: $1250.00 Commission: $500.00

7566 JONES, MANAGER

Salary: $2975.00 Commission: $0

7654 MARTIN, SALESMAN

Salary: $1250.00 Commission: $1400.00

7698 BLAKE, MANAGER

Salary: $2850.00 Commission: $0

7782 CLARK, MANAGER

Salary: $2450.00 Commission: $0

7788 SCOTT, ANALYST

Salary: $3000.00 Commission: $0

7839 KING, PRESIDENT

Salary: $5000.00 Commission: $0

7844 TURNER, SALESMAN

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

## UTL\_HTTP

The UTL\_HTTP package provides a way to use the HTTP or HTTPS protocol to retrieve information found at an URL. Advanced Server supports the following functions and procedures:

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| BEGIN\_REQUEST(url, method, http\_version) | UTL\_HTTP.REQ | Initiates a new HTTP request. |
| END\_REQUEST(r IN OUT) | n/a | Ends an HTTP request before allowing it to complete. |
| END\_RESPONSE(r IN OUT) | n/a | Ends the HTTP response. |
| GET\_BODY\_CHARSET | VARCHAR2 | Returns the default character set of the body of future HTTP requests. |
| GET\_BODY\_CHARSET(charset OUT) | n/a | Returns the default character set of the body of future HTTP requests. |
| GET\_FOLLOW\_REDIRECT(max\_redirects OUT) | n/a | Current setting for the maximum number of redirections allowed. |
| GET\_HEADER(r IN OUT, n, name OUT, value OUT) | n/a | Returns the nth header of the HTTP response. |
| GET\_HEADER\_BY\_NAME(r IN OUT, name, value OUT, n) | n/a | Returns the HTTP response header for the specified name. |
| GET\_HEADER\_COUNT(r IN OUT) | INTEGER | Returns the number of HTTP response headers. |
| GET\_RESPONSE(r IN OUT) | UTL\_HTTP.RESP | Returns the HTTP response. |
| GET\_RESPONSE\_ERROR\_CHECK(enable OUT) | n/a | Returns whether or not response error check is set. |
| GET\_TRANSFER\_TIMEOUT(timeout OUT) | n/a | Returns the transfer timeout setting for HTTP requests. |
| READ\_LINE(r IN OUT, data OUT, remove\_crlf) | n/a | Returns the HTTP response body in text form until the end of line. |
| READ\_RAW(r IN OUT, data OUT, len) | n/a | Returns the HTTP response body in binary form for a specified number of bytes. |
| READ\_TEXT(r IN OUT, data OUT, len) | n/a | Returns the HTTP response body in text form for a specified number of characters. |
| REQUEST(url) | VARCHAR2 | Returns the content of a web page. |
| REQUEST\_PIECES(url, max\_pieces) | UTL\_HTTP. HTML\_PIECES | Returns a table of 2000-byte segments retrieved from an URL. |
| SET\_BODY\_CHARSET(charset) | n/a | Sets the default character set of the body of future HTTP requests. |
| SET\_FOLLOW\_REDIRECT(max\_redirects) | n/a | Sets the maximum number of times to follow the redirect instruction. |
| SET\_FOLLOW\_REDIRECT(r IN OUT, max\_redirects) | n/a | Sets the maximum number of times to follow the redirect instruction for an individual request. |
| SET\_HEADER(r IN OUT, name, value) | n/a | Sets the HTTP request header. |
| SET\_RESPONSE\_ERROR\_CHECK(enable) | n/a | Determines whether or not HTTP 4xx and 5xx status codes are to be treated as errors. |
| SET\_TRANSFER\_TIMEOUT(timeout) | n/a | Sets the default, transfer timeout value for HTTP requests. |
| SET\_TRANSFER\_TIMEOUT(r IN OUT, timeout) | n/a | Sets the transfer timeout value for an individual HTTP request. |
| WRITE\_LINE(*r* IN OUT, *data*) | n/a | Writes CRLF terminated data to the HTTP request body in TEXT form. |
| WRITE\_RAW(r IN OUT, data) | n/a | Writes data to the HTTP request body in BINARY form. |
| WRITE\_TEXT(*r* IN OUT, *data*) | n/a | Writes data to the HTTP request body in TEXT form. |

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

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.

|  |  |
| --- | --- |
| HTTP VERSIONS | |
| HTTP\_VERSION\_1\_0 | CONSTANT VARCHAR2(64) := 'HTTP/1.0'; |
| HTTP\_VERSION\_1\_1 | CONSTANT VARCHAR2(64) := 'HTTP/1.1'; |
| STANDARD PORT ASSIGNMENTS | |
| DEFAULT\_HTTP\_PORT | CONSTANT INTEGER := 80; |
| DEFAULT\_HTTPS\_PORT | CONSTANT INTEGER := 443; |

The following table contains UTL\_HTTP public status code constants.

|  |  |
| --- | --- |
| 1XX INFORMATIONAL | |
| HTTP\_CONTINUE | CONSTANT INTEGER := 100; |
| HTTP\_SWITCHING\_PROTOCOLS | CONSTANT INTEGER := 101; |
| HTTP\_PROCESSING | CONSTANT INTEGER := 102; |
| 2XX SUCCESS | |
| HTTP\_OK | CONSTANT INTEGER := 200; |
| HTTP\_CREATED | CONSTANT INTEGER := 201; |
| HTTP\_ACCEPTED | CONSTANT INTEGER := 202; |
| HTTP\_NON\_AUTHORITATIVE\_INFO | CONSTANT INTEGER := 203; |
| HTTP\_NO\_CONTENT | CONSTANT INTEGER := 204; |
| HTTP\_RESET\_CONTENT | CONSTANT INTEGER := 205; |
| HTTP\_PARTIAL\_CONTENT | CONSTANT INTEGER := 206; |
| HTTP\_MULTI\_STATUS | CONSTANT INTEGER := 207; |
| HTTP\_ALREADY\_REPORTED | CONSTANT INTEGER := 208; |
| HTTP\_IM\_USED | CONSTANT INTEGER := 226; |
| 3XX REDIRECTION | |
| HTTP\_MULTIPLE\_CHOICES | CONSTANT INTEGER := 300; |
| HTTP\_MOVED\_PERMANENTLY | CONSTANT INTEGER := 301; |
| HTTP\_FOUND | CONSTANT INTEGER := 302; |
| HTTP\_SEE\_OTHER | CONSTANT INTEGER := 303; |
| HTTP\_NOT\_MODIFIED | CONSTANT INTEGER := 304; |
| HTTP\_USE\_PROXY | CONSTANT INTEGER := 305; |
| HTTP\_SWITCH\_PROXY | CONSTANT INTEGER := 306; |
| HTTP\_TEMPORARY\_REDIRECT | CONSTANT INTEGER := 307; |
| HTTP\_PERMANENT\_REDIRECT | CONSTANT INTEGER := 308; |
| 4XX CLIENT ERROR | |
| HTTP\_BAD\_REQUEST | CONSTANT INTEGER := 400; |
| HTTP\_UNAUTHORIZED | CONSTANT INTEGER := 401; |
| HTTP\_PAYMENT\_REQUIRED | CONSTANT INTEGER := 402; |
| HTTP\_FORBIDDEN | CONSTANT INTEGER := 403; |
| HTTP\_NOT\_FOUND | CONSTANT INTEGER := 404; |
| HTTP\_METHOD\_NOT\_ALLOWED | CONSTANT INTEGER := 405; |
| HTTP\_NOT\_ACCEPTABLE | CONSTANT INTEGER := 406; |
| HTTP\_PROXY\_AUTH\_REQUIRED | CONSTANT INTEGER := 407; |
| HTTP\_REQUEST\_TIME\_OUT | CONSTANT INTEGER := 408; |
| HTTP\_CONFLICT | CONSTANT INTEGER := 409; |
| HTTP\_GONE | CONSTANT INTEGER := 410; |
| HTTP\_LENGTH\_REQUIRED | CONSTANT INTEGER := 411; |
| HTTP\_PRECONDITION\_FAILED | CONSTANT INTEGER := 412; |
| HTTP\_REQUEST\_ENTITY\_TOO\_LARGE | CONSTANT INTEGER := 413; |
| HTTP\_REQUEST\_URI\_TOO\_LARGE | CONSTANT INTEGER := 414; |
| HTTP\_UNSUPPORTED\_MEDIA\_TYPE | CONSTANT INTEGER := 415; |
| HTTP\_REQ\_RANGE\_NOT\_SATISFIABLE | CONSTANT INTEGER := 416; |
| HTTP\_EXPECTATION\_FAILED | CONSTANT INTEGER := 417; |
| HTTP\_I\_AM\_A\_TEAPOT | CONSTANT INTEGER := 418; |
| HTTP\_AUTHENTICATION\_TIME\_OUT | CONSTANT INTEGER := 419; |
| HTTP\_ENHANCE\_YOUR\_CALM | CONSTANT INTEGER := 420; |
| HTTP\_UNPROCESSABLE\_ENTITY | CONSTANT INTEGER := 422; |
| HTTP\_LOCKED | CONSTANT INTEGER := 423; |
| HTTP\_FAILED\_DEPENDENCY | CONSTANT INTEGER := 424; |
| HTTP\_UNORDERED\_COLLECTION | CONSTANT INTEGER := 425; |
| HTTP\_UPGRADE\_REQUIRED | CONSTANT INTEGER := 426; |
| HTTP\_PRECONDITION\_REQUIRED | CONSTANT INTEGER := 428; |
| HTTP\_TOO\_MANY\_REQUESTS | CONSTANT INTEGER := 429; |
| HTTP\_REQUEST\_HEADER\_FIELDS\_TOO\_LARGE | CONSTANT INTEGER := 431; |
| HTTP\_NO\_RESPONSE | CONSTANT INTEGER := 444; |
| HTTP\_RETRY\_WITH | CONSTANT INTEGER := 449; |
| HTTP\_BLOCKED\_BY\_WINDOWS\_PARENTAL\_CONTROLS | CONSTANT INTEGER := 450; |
| HTTP\_REDIRECT | CONSTANT INTEGER := 451; |
| HTTP\_REQUEST\_HEADER\_TOO\_LARGE | CONSTANT INTEGER := 494; |
| HTTP\_CERT\_ERROR | CONSTANT INTEGER := 495; |
| HTTP\_NO\_CERT | CONSTANT INTEGER := 496; |
| HTTP\_HTTP\_TO\_HTTPS | CONSTANT INTEGER := 497; |
| HTTP\_CLIENT\_CLOSED\_REQUEST | CONSTANT INTEGER := 499; |
| 5XX SERVER ERROR | |
| HTTP\_INTERNAL\_SERVER\_ERROR | CONSTANT INTEGER := 500; |
| HTTP\_NOT\_IMPLEMENTED | CONSTANT INTEGER := 501; |
| HTTP\_BAD\_GATEWAY | CONSTANT INTEGER := 502; |
| HTTP\_SERVICE\_UNAVAILABLE | CONSTANT INTEGER := 503; |
| HTTP\_GATEWAY\_TIME\_OUT | CONSTANT INTEGER := 504; |
| HTTP\_VERSION\_NOT\_SUPPORTED | CONSTANT INTEGER := 505; |
| HTTP\_VARIANT\_ALSO\_NEGOTIATES | CONSTANT INTEGER := 506; |
| HTTP\_INSUFFICIENT\_STORAGE | CONSTANT INTEGER := 507; |
| HTTP\_LOOP\_DETECTED | CONSTANT INTEGER := 508; |
| HTTP\_BANDWIDTH\_LIMIT\_EXCEEDED | CONSTANT INTEGER := 509; |
| HTTP\_NOT\_EXTENDED | CONSTANT INTEGER := 510; |
| HTTP\_NETWORK\_AUTHENTICATION\_REQUIRED | CONSTANT INTEGER := 511; |
| HTTP\_NETWORK\_READ\_TIME\_OUT\_ERROR | CONSTANT INTEGER := 598; |
| HTTP\_NETWORK\_CONNECT\_TIME\_OUT\_ERROR | CONSTANT INTEGER := 599; |

### HTML\_PIECES

The UTL\_HTTP package declares a type named HTML\_PIECES, which is a table of type VARCHAR2 (2000) indexed by BINARY INTEGER. A value of this type is returned by the REQUEST\_PIECES function.

TYPE html\_pieces IS TABLE OF VARCHAR2(2000) INDEX BY BINARY\_INTEGER;

### REQ

The REQ record type holds information about each HTTP request.

TYPE req IS RECORD (

url VARCHAR2(32767), -- URL to be accessed

method VARCHAR2(64), -- HTTP method

http\_version VARCHAR2(64), -- HTTP version

private\_hndl INTEGER -- Holds handle for this request

);

### RESP

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

TYPE resp IS RECORD (

status\_code INTEGER, -- HTTP status code

reason\_phrase VARCHAR2(256), -- HTTP response reason phrase

http\_version VARCHAR2(64), -- HTTP version

private\_hndl INTEGER -- Holds handle for this response

);

### BEGIN\_REQUEST

The BEGIN\_REQUEST function initiates a new HTTP request. A network connection is established to the web server with the specified URL. The signature is:

BEGIN\_REQUEST(*url* IN VARCHAR2, *method* IN VARCHAR2 DEFAULT 'GET ', *http\_version* IN VARCHAR2 DEFAULT NULL) RETURN UTL\_HTTP.REQ

The BEGIN\_REQUEST function returns a record of type UTL\_HTTP.REQ.

Parameters

*url*

*url* is the Uniform Resource Locator from which UTL\_HTTP will return content.

*method*

*method* is the HTTP method to be used. The default is GET.

*http\_version*

*http\_version* is the HTTP protocol version sending the request. The specified values should be either HTTP/1.0 or HTTP/1.1. The default is null in which case the latest HTTP protocol version supported by the UTL\_HTTP package is used which is 1.1.

### END\_REQUEST

The END\_REQUEST procedure terminates an HTTP request. Use the END\_REQUEST procedure to terminate an HTTP request without completing it and waiting for the response. The normal process is to begin the request, get the response, then close the response. The signature is:

END\_REQUEST(*r* IN OUT UTL\_HTTP.REQ)

Parameters

*r*

*r* is the HTTP request record.

### END\_RESPONSE

The END\_RESPONSE procedure terminates the HTTP response. The END\_RESPONSE procedure completes the HTTP request and response. This is the normal method to end the request and response process. The signature is:

END\_RESPONSE(*r* IN OUT UTL\_HTTP.RESP)

Parameters

*r*

*r* is the HTTP response record.

### GET\_BODY\_CHARSET

The GET\_BODY\_CHARSET program is available in the form of both a procedure and a function. A call to GET\_BODY\_CHARSET returns the default character set of the body of future HTTP requests.

The procedure signature is:

GET\_BODY\_CHARSET(*charset* OUT VARCHAR2)

The function signature is:

GET\_BODY\_CHARSET() RETURN VARCHAR2

This function returns a VARCHAR2 value.

Parameters

*charset*

*charset* is the character set of the body.

Examples

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

edb=# SELECT UTL\_HTTP.GET\_BODY\_CHARSET() FROM DUAL;

get\_body\_charset

------------------

ISO-8859-1

(1 row)

### GET\_FOLLOW\_REDIRECT

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

GET\_FOLLOW\_REDIRECT(*max\_redirects* OUT INTEGER)

Parameters

*max\_redirects*

*max\_redirects* is maximum number of redirections allowed.

### GET\_HEADER

The GET\_HEADER procedure returns the *n*th header of the HTTP response. The signature is:

GET\_HEADER(*r* IN OUT UTL\_HTTP.RESP, *n* INTEGER, *name* OUT VARCHAR2, *value* OUT VARCHAR2)

Parameters

*r*

*r* is the HTTP response record.

*n*

*n* is the *n*th header of the HTTP response record to retrieve.

*name*

*name* is the name of the response header.

*value*

*value* is the value of the response header.

Examples

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

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

v\_name VARCHAR2(30);

v\_value VARCHAR2(200);

v\_header\_cnt INTEGER;

BEGIN

-- Initiate request and get response

v\_req := UTL\_HTTP.BEGIN\_REQUEST('www.enterprisedb.com');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

-- Get header count

v\_header\_cnt := UTL\_HTTP.GET\_HEADER\_COUNT(v\_resp);

DBMS\_OUTPUT.PUT\_LINE('Header Count: ' || v\_header\_cnt);

-- Get all headers

FOR i IN 1 .. v\_header\_cnt LOOP

UTL\_HTTP.GET\_HEADER(v\_resp, i, v\_name, v\_value);

DBMS\_OUTPUT.PUT\_LINE(v\_name || ': ' || v\_value);

END LOOP;

-- Terminate request

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

The following is the output from the example.

Header Count: 23

Age: 570

Cache-Control: must-revalidate

Content-Type: text/html; charset=utf-8

Date: Wed, 30 Apr 2015 14:57:52 GMT

ETag: "aab02f2bd2d696eed817ca89ef411dda"

Expires: Sun, 19 Nov 1978 05:00:00 GMT

Last-Modified: Wed, 30 Apr 2015 14:15:49 GMT

RTSS: 1-1307-3

Server: Apache/2.2.3 (Red Hat)

Set-Cookie: SESS2771d0952de2a1a84d322a262e0c173c=jn1u1j1etmdi5gg4lh8hakvs01; expires=Fri, 23-May-2015 18:21:43 GMT; path=/; domain=.enterprisedb.com

Vary: Accept-Encoding

Via: 1.1 varnish

X-EDB-Backend: ec

X-EDB-Cache: HIT

X-EDB-Cache-Address: 10.31.162.212

X-EDB-Cache-Server: ip-10-31-162-212

X-EDB-Cache-TTL: 600.000

X-EDB-Cacheable: MAYBE: The user has a cookie of some sort. Maybe it's double choc-chip!

X-EDB-Do-GZIP: false

X-Powered-By: PHP/5.2.17

X-Varnish: 484508634 484506789

transfer-encoding: chunked

Connection: keep-alive

### GET\_HEADER\_BY\_NAME

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

### 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.

### GET\_RESPONSE

The GET\_RESPONSE function sends the network request and returns any HTTP response. The signature is:

GET\_RESPONSE(*r* IN OUT UTL\_HTTP.REQ) RETURN UTL\_HTTP.RESP

This function returns a UTL\_HTTP.RESP record.

Parameters

*r*

*r* is the HTTP request record.

### GET\_RESPONSE\_ERROR\_CHECK

The GET\_RESPONSE\_ERROR\_CHECK procedure returns whether or not response error check is set. The signature is:

GET\_RESPONSE\_ERROR\_CHECK(*enable* OUT BOOLEAN)

Parameters

*enable*

*enable* returns TRUE if response error check is set, otherwise it returns FALSE.

### GET\_TRANSFER\_TIMEOUT

The GET\_TRANSFER\_TIMEOUT procedure returns the current, default transfer timeout setting for HTTP requests. The signature is:

GET\_TRANSFER\_TIMEOUT(*timeout* OUT INTEGER)

Parameters

*timeout*

*timeout* is the transfer timeout setting in seconds.

### READ\_LINE

The READ\_LINE procedure returns the data from the HTTP response body in text form until the end of line is reached. A CR character, a LF character, a CR LF sequence, or the end of the response body constitutes the end of line. The signature is:

READ\_LINE(*r* IN OUT UTL\_HTTP.RESP, *data* OUT VARCHAR2, *remove\_crlf* BOOLEAN DEFAULT FALSE)

Parameters

*r*

*r* is the HTTP response record.

*data*

*data* is the response body in text form.

*remove\_crlf*

Set *remove\_crlf* to TRUE to remove new line characters, otherwise set to FALSE. The default is FALSE.

Examples

The following example retrieves and displays the body of the specified website.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

v\_value VARCHAR2(1024);

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.enterprisedb.com');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

LOOP

UTL\_HTTP.READ\_LINE(v\_resp, v\_value, TRUE);

DBMS\_OUTPUT.PUT\_LINE(v\_value);

END LOOP;

EXCEPTION

WHEN OTHERS THEN

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

The following is the output.

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">

<!-- \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ HEAD \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ -->

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>EnterpriseDB | The Postgres Database Company</title>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<meta name="keywords" content="postgres, postgresql, postgresql installer, mysql migration, open source database, training, replication" />

<meta name="description" content="The leader in open source database products, services, support, training and expertise based on PostgreSQL. Free downloads, documentation, and tutorials." />

<meta name="abstract" content="The Enterprise PostgreSQL Company" />

<link rel="EditURI" type="application/rsd+xml" title="RSD" href="http://www.enterprisedb.com/blogapi/rsd" />

<link rel="alternate" type="application/rss+xml" title="EnterpriseDB RSS" href="http://www.enterprisedb.com/rss.xml" />

<link rel="shortcut icon" href="/sites/all/themes/edb\_pixelcrayons/favicon.ico" type="image/x-icon" />

<link type="text/css" rel="stylesheet" media="all" href="/sites/default/files/css/css\_db11adabae0aed6b79a2c3c52def4754.css" />

<!--[if IE 6]>

<link type="text/css" rel="stylesheet" media="all" href="/sites/all/themes/oho\_basic/css/ie6.css?g" />

<![endif]-->

<!--[if IE 7]>

<link type="text/css" rel="stylesheet" media="all" href="/sites/all/themes/oho\_basic/css/ie7.css?g" />

<![endif]-->

<script type="text/javascript" src="/sites/default/files/js/js\_74d97b1176812e2fd6e43d62503a5204.js"></script>

<script type="text/javascript">

<!--//--><![CDATA[//><!--

### READ\_RAW

The READ\_RAW procedure returns the data from the HTTP response body in binary form. The number of bytes returned is specified by the *len* parameter. The signature is:

READ\_RAW(*r* IN OUT UTL\_HTTP.RESP, *data* OUT RAW, *len* INTEGER)

Parameters

*r*

*r* is the HTTP response record.

*data*

*data* is the response body in binary form.

*len*

Set *len* to the number of bytes of data to be returned.

Examples

The following example retrieves and displays the first 150 bytes in binary form.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

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.

\x3c21444f43545950452068746d6c205055424c494320222d2f2f5733432f2f445444205848544d4c20312e30205374726963742f2f454e220d0a202022687474703a2f2f7777772e77332e6f72672f54522f7868746d6c312f4454442f7868746d6c312d7374726963742e647464223e0d0a3c68746d6c20786d6c6e733d22687474703a2f2f7777772e77332e6f72672f313939392f

### 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/

### REQUEST

The REQUEST function returns the first 2000 bytes retrieved from a user-specified URL. The signature is:

REQUEST(url IN VARCHAR2) RETURN VARCHAR2

If the data found at the given URL is longer than 2000 bytes, the remainder will be discarded. If the data found at the given URL is shorter than 2000 bytes, the result will be shorter than 2000 bytes.

Parameters

*url*

*url* is the Uniform Resource Locator from which UTL\_HTTP will return content.

Example

The following command returns the first 2000 bytes retrieved from the EnterpriseDB website:

SELECT UTL\_HTTP.REQUEST('http://www.enterprisedb.com/') FROM DUAL;

### REQUEST\_PIECES

The REQUEST\_PIECES function returns a table of 2000-byte segments retrieved from an URL. The signature is:

REQUEST\_PIECES(*url* IN VARCHAR2, *max\_pieces* NUMBER IN DEFAULT 32767) RETURN UTL\_HTTP.HTML\_PIECES

Parameters

*url*

*url* is the Uniform Resource Locator from which UTL\_HTTP will return content.

*max\_pieces*

*max\_pieces* specifies the maximum number of 2000-byte segments that the REQUEST\_PIECES function will return. If *max\_pieces* specifies more units than are available at the specified *url*, the final unit will contain fewer bytes.

Example

The following example returns the first four 2000 byte segments retrieved from the EnterpriseDB website:

DECLARE

result UTL\_HTTP.HTML\_PIECES;

BEGIN

result := UTL\_HTTP.REQUEST\_PIECES('http://www.enterprisedb.com/', 4);

END;

### SET\_BODY\_CHARSET

The SET\_BODY\_CHARSET procedure sets the default character set of the body of future HTTP requests. The signature is:

SET\_BODY\_CHARSET(*charset* VARCHAR2 DEFAULT NULL)

Parameters

*charset*

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

### SET\_FOLLOW\_REDIRECT

The SET\_FOLLOW\_REDIRECT procedure sets the maximum number of times the HTTP redirect instruction is to be followed in the response to this request or future requests. This procedures has two signatures:

SET\_FOLLOW\_REDIRECT(*max\_redirects* IN INTEGER DEFAULT 3)

and

SET\_FOLLOW\_REDIRECT(*r* IN OUT UTL\_HTTP.REQ, *max\_redirects* IN INTEGER DEFAULT 3)

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

Parameters

*r*

*r* is the HTTP request record.

*max\_redirects*

*max\_redirects* is maximum number of redirections allowed. Set to 0 to disable redirections. The default is 3.

### SET\_HEADER

The SET\_HEADER procedure sets the HTTP request header. The signature is:

SET\_HEADER(*r* IN OUT UTL\_HTTP.REQ, *name* IN VARCHAR2, *value* IN VARCHAR2 DEFAULT NULL)

Parameters

*r*

*r* is the HTTP request record.

*name*

*name* is the name of the request header.

*value*

*value* is the value of the request header. The default is null.

### SET\_RESPONSE\_ERROR\_CHECK

The SET\_RESPONSE\_ERROR\_CHECK procedure determines whether or not HTTP 4xx and 5xx status codes returned by the GET\_RESPONSE function should be interpreted as errors. The signature is:

SET\_RESPONSE\_ERROR\_CHECK(*enable* IN BOOLEAN DEFAULT FALSE)

Parameters

*enable*

Set *enable* to TRUE if HTTP 4xx and 5xx status codes are to be treated as errors, otherwise set to FALSE. The default is FALSE.

### SET\_TRANSFER\_TIMEOUT

The SET\_TRANSFER\_TIMEOUT procedure sets the default, transfer timeout setting for waiting for a response from an HTTP request. This procedure has two signatures:

SET\_TRANSFER\_TIMEOUT(*timeout* IN INTEGER DEFAULT 60)

and

SET\_TRANSFER\_TIMEOUT(*r* IN OUT UTL\_HTTP.REQ, *timeout* IN INTEGER DEFAULT 60)

Use the second form to change the transfer timeout setting for an individual request that a request inherits from the session default settings.

Parameters

*r*

*r* is the HTTP request record.

*timeout*

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

### WRITE\_LINE

The WRITE\_LINE procedure writes data to the HTTP request body in text form; the text is terminated with a CRLF character pair. The signature is:

WRITE\_LINE(*r* IN OUT UTL\_HTTP.REQ, *data* IN VARCHAR2)

Parameters

*r*

*r* is the HTTP request record.

*data*

*data* is the request body in TEXT form.

Example

The following example writes data (Account balance $500.00) in text form to the request body to be sent using the HTTP POST method. The data is sent to a hypothetical web application (post.php) that accepts and processes data.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.example.com/post.php',

'POST');

UTL\_HTTP.SET\_HEADER(v\_req, 'Content-Length', '23');

UTL\_HTTP.WRITE\_LINE(v\_req, 'Account balance $500.00');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

DBMS\_OUTPUT.PUT\_LINE('Status Code: ' || v\_resp.status\_code);

DBMS\_OUTPUT.PUT\_LINE('Reason Phrase: ' || v\_resp.reason\_phrase);

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

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

Status Code: 200

Reason Phrase: OK

### WRITE\_RAW

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

WRITE\_RAW(*r* IN OUT UTL\_HTTP.REQ, *data* IN RAW)

Parameters

*r*

*r* is the HTTP request record.

*data*

*data* is the request body in binary form.

Example

The following example writes data in binary form to the request body to be sent using the HTTP POST method to a hypothetical web application that accepts and processes such data.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.example.com/post.php',

'POST');

UTL\_HTTP.SET\_HEADER(v\_req, 'Content-Length', '23');

UTL\_HTTP.WRITE\_RAW(v\_req, HEXTORAW

('54657374696e6720504f5354206d6574686f6420696e20485454502072657175657374'));

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

DBMS\_OUTPUT.PUT\_LINE('Status Code: ' || v\_resp.status\_code);

DBMS\_OUTPUT.PUT\_LINE('Reason Phrase: ' || v\_resp.reason\_phrase);

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

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

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

Status Code: 200

Reason Phrase: OK

### WRITE\_TEXT

The WRITE\_TEXT procedure writes data to the HTTP request body in text form. The signature is:

WRITE\_TEXT(*r* IN OUT UTL\_HTTP.REQ, *data* IN VARCHAR2)

Parameters

*r*

*r* is the HTTP request record.

*data*

*data* is the request body in text form.

Example

The following example writes data (Account balance $500.00) in text form to the request body to be sent using the HTTP POST method. The data is sent to a hypothetical web application (post.php) that accepts and processes data.

DECLARE

v\_req UTL\_HTTP.REQ;

v\_resp UTL\_HTTP.RESP;

BEGIN

v\_req := UTL\_HTTP.BEGIN\_REQUEST('http://www.example.com/post.php',

'POST');

UTL\_HTTP.SET\_HEADER(v\_req, 'Content-Length', '23');

UTL\_HTTP.WRITE\_TEXT(v\_req, 'Account balance $500.00');

v\_resp := UTL\_HTTP.GET\_RESPONSE(v\_req);

DBMS\_OUTPUT.PUT\_LINE('Status Code: ' || v\_resp.status\_code);

DBMS\_OUTPUT.PUT\_LINE('Reason Phrase: ' || v\_resp.reason\_phrase);

UTL\_HTTP.END\_RESPONSE(v\_resp);

END;

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

Status Code: 200

Reason Phrase: OK

## UTL\_MAIL

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

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| SEND(sender, recipients, cc, bcc, subject, message [, mime\_type [, priority ]]) | n/a | Packages and sends an e-mail to an SMTP server. |
| SEND\_ATTACH\_RAW(sender, recipients, cc, bcc, subject, message, mime\_type, priority, attachment [, att\_inline [, att\_mime\_type [, att\_filename ]]]) | n/a | Same as the SEND procedure, but with BYTEA or large object attachments. |
| SEND\_ATTACH\_VARCHAR2(sender, recipients, cc, bcc, subject, message, mime\_type, priority, attachment [, att\_inline [, att\_mime\_type [, att\_filename ]]]) | n/a | Same as the SEND procedure, but with VARCHAR2 attachments. |

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

### SEND

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

SEND(sender VARCHAR2, recipients VARCHAR2, cc VARCHAR2,

bcc VARCHAR2, subject VARCHAR2, message VARCHAR2

[, mime\_type VARCHAR2 [, priority PLS\_INTEGER ]])

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

Comma-separated e-mail addresses of blind copy recipients.

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime\_type

Mime type of the message. The default is text/plain; charset=us-ascii.

priority

Priority of the e-mail The default is 3.

Examples

The following anonymous block sends a simple e-mail message.

DECLARE

v\_sender VARCHAR2(30);

v\_recipients VARCHAR2(60);

v\_subj VARCHAR2(20);

v\_msg VARCHAR2(200);

BEGIN

v\_sender := 'jsmith@enterprisedb.com';

v\_recipients := 'ajones@enterprisedb.com,rrogers@enterprisedb.com';

v\_subj := 'Holiday Party';

v\_msg := 'This year''s party is scheduled for Friday, Dec. 21at ' ||

'6:00 PM. Please RSVP by Dec. 15th.';

UTL\_MAIL.SEND(v\_sender,v\_recipients,NULL,NULL,v\_subj,v\_msg);

END;

### 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.

### 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.

## UTL\_RAW

The UTL\_RAW package allows you to manipulate or retrieve the length of raw data types.

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

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| CAST\_TO\_RAW(*c* IN VARCHAR2) | Function | RAW | Converts a VARCHAR2 string to a RAW value. |
| CAST\_TO\_VARCHAR2(*r* IN RAW) | Function | VARCHAR2 | Converts a RAW value to a VARCHAR2 string. |
| CONCAT(*r1* IN RAW, *r2* IN RAW, *r3* IN RAW,…) | Function | RAW | Concatenate multiple RAW values into a single RAW value. |
| CONVERT(*r* IN RAW, *to*\_*charset* IN VARCHAR2, *from*\_*charset* IN VARCHAR2 | Function | RAW | Converts encoded data from one encoding to another, and returns the result as a RAW value. |
| LENGTH(*r* IN RAW) | Function | NUMBER | Returns the length of a RAW value. |
| SUBSTR(*r* IN RAW, *pos* IN INTEGER, *len* IN INTEGER) | Function | RAW | Returns a portion of a RAW value. |

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

### CAST\_TO\_RAW

The CAST\_TO\_RAW function converts a VARCHAR2 string to a RAW value. The signature is:

CAST\_TO\_RAW(*c* VARCHAR2)

The function returns a RAW value if you pass a non-NULL value; if you pass a NULL value, the function will return NULL.

Parameters

c

The VARCHAR2 value that will be converted to RAW.

Example

The following example uses the CAST\_TO\_RAW function to convert a VARCHAR2 string to a RAW value:

DECLARE

v VARCHAR2;

r RAW;

BEGIN

v := 'Accounts';

dbms\_output.put\_line(v);

r := UTL\_RAW.CAST\_TO\_RAW(v);

dbms\_output.put\_line(r);

END;

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

Accounts

\x4163636f756e7473

### CAST\_TO\_VARCHAR2

The CAST\_TO\_VARCHAR2 function converts RAW data to VARCHAR2 data. The signature is:

CAST\_TO\_VARCHAR2(*r* RAW)

The function returns a VARCHAR2 value if you pass a non-NULL value; if you pass a NULL value, the function will return NULL.

Parameters

r

The RAW value that will be converted to a VARCHAR2 value.

Example

The following example uses the CAST\_TO\_VARCHAR2 function to convert a RAW value to a VARCHAR2 string:

DECLARE

r RAW;

v VARCHAR2;

BEGIN

r := '\x4163636f756e7473'

dbms\_output.put\_line(v);

v := UTL\_RAW.CAST\_TO\_VARCHAR2(r);

dbms\_output.put\_line(r);

END;

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

\x4163636f756e7473

Accounts

### CONCAT

The CONCAT function concatenates multiple RAW values into a single RAW value. The signature is:

CONCAT(*r1* RAW, *r2* RAW, *r3* RAW,…)

The function returns a RAW value. Unlike the Oracle implementation, the Advanced Server implementation is a variadic function, and does not place a restriction on the number of values that can be concatenated.

Parameters

r1, r2, r3,…

The RAW values that CONCAT will concatenate.

Example

The following example uses the CONCAT function to concatenate multiple RAW values into a single RAW value:

SELECT UTL\_RAW.CAST\_TO\_VARCHAR2(UTL\_RAW.CONCAT('\x61', '\x62', '\x63')) FROM DUAL;  
 concat

--------  
 abc  
(1 row)

The result (the concatenated values) is then converted to VARCHAR2 format by the CAST\_TO\_VARCHAR2 function.

### CONVERT

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

CONVERT(r RAW, to\_charset VARCHAR2, from\_charset VARCHAR2)

The function returns a RAW value.

Parameters

r

The RAW value that will be converted.

*to\_charset*

The name of the encoding to which *r* will be converted.

*from\_charset*

The name of the encoding from which *r* will be converted.

Example

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

DECLARE  
 r RAW;  
 v VARCHAR2;  
BEGIN  
 v:= 'Accounts';  
 dbms\_output.put\_line(v);

r:= UTL\_RAW.CAST\_TO\_RAW(v);  
 dbms\_output.put\_line(r);  
 r:= UTL\_RAW.CONVERT(r, 'UTF8', 'LATIN7');  
 dbms\_output.put\_line(r);  
 r:= UTL\_RAW.CONVERT(r, 'LATIN7', 'UTF8');  
 dbms\_output.put\_line(r);

The example returns the VARCHAR2 value, the RAW value, and the converted values:

Accounts

\x4163636f756e7473

\x4163636f756e7473

\x4163636f756e7473

### LENGTH

The LENGTH function returns the length of a RAW value. The signature is:

LENGTH(*r* RAW)

The function returns a RAW value.

Parameters

r

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)

### SUBSTR

The SUBSTR function returns a substring of a RAW value. The signature is:

SUBSTR (*r* RAW, *pos* INTEGER, *len* INTEGER)

This function returns a RAW value.

Parameters

r

The RAW value from which the substring will be returned.

*pos*

The position within the RAW value of the first byte of the returned substring.

* If *pos* is 0 or 1, the substring begins at the first byte of the RAW value.
* If *pos* is greater than one, the substring begins at the first byte specified by *pos*. For example, if *pos* is 3, the substring begins at the third byte of the value.
* If *pos* is negative, the substring begins at *pos* bytes from the end of the source value. For example, if *pos* is -3, the substring begins at the third byte from the end of the value.

*len*

The maximum number of bytes that will be returned.

Example

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

SELECT UTL\_RAW.SUBSTR(UTL\_RAW.CAST\_TO\_RAW('Accounts'), 3, 5) FROM DUAL;

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

The following example uses the SUBSTR function to select a substring that starts 5 bytes from the end of a RAW value:

SELECT UTL\_RAW.SUBSTR(UTL\_RAW.CAST\_TO\_RAW('Accounts'), -5 , 3) FROM DUAL;

substr

--------

oun

(1 row)

## UTL\_SMTP

The UTL\_SMTP package provides the capability to send e-mails over the Simple Mail Transfer Protocol (SMTP).

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

| Function/Procedure | Function or Procedure | Return Type | Description |
| --- | --- | --- | --- |
| CLOSE\_DATA(c IN OUT) | Procedure | n/a | Ends an e-mail message. |
| COMMAND(c IN OUT, cmd [, arg ]) | Both | REPLY | Execute an SMTP command. |
| COMMAND\_REPLIES(c IN OUT, cmd [, arg ]) | Function | REPLIES | Execute an SMTP command where multiple reply lines are expected. |
| DATA(c IN OUT, body VARCHAR2) | Procedure | n/a | Specify the body of an e-mail message. |
| EHLO(c IN OUT, domain) | Procedure | n/a | Perform initial handshaking with an SMTP server and return extended information. |
| HELO(c IN OUT, domain) | Procedure | n/a | Perform initial handshaking with an SMTP server |
| HELP(c IN OUT [, command ]) | Function | REPLIES | Send the HELP command. |
| MAIL(c IN OUT, sender [, parameters ]) | Procedure | n/a | Start a mail transaction. |
| NOOP(c IN OUT) | Both | REPLY | Send the null command. |
| OPEN\_CONNECTION(host [*,* port [, tx\_timeout ]]) | Function | CONNECTION | Open a connection. |
| OPEN\_DATA(c IN OUT) | Both | REPLY | Send the DATA command. |
| QUIT(c IN OUT) | Procedure | n/a | Terminate the SMTP session and disconnect. |
| RCPT(c IN OUT, recipient [, parameters ]) | Procedure | n/a | Specify the recipient of an e-mail message. |
| RSET(c IN OUT) | Procedure | n/a | Terminate the current mail transaction. |
| VRFY(c IN OUT, recipient) | Function | REPLY | Validate an e-mail address. |
| WRITE\_DATA(c IN OUT, data) | Procedure | n/a | Write a portion of the e-mail message. |

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

The following table lists the public variables available in the UTL\_SMTP package.

| Public Variables | Data Type | Value | Description |
| --- | --- | --- | --- |
| connection | RECORD |  | Description of an SMTP connection. |
| reply | RECORD |  | SMTP reply line. |

### CONNECTION

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

TYPE connection IS RECORD (

host VARCHAR2(255),

port PLS\_INTEGER,

tx\_timeout PLS\_INTEGER

);

### REPLY/REPLIES

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

TYPE reply IS RECORD (

code INTEGER,

text VARCHAR2(508)

);

TYPE replies IS TABLE OF reply INDEX BY BINARY\_INTEGER;

### CLOSE\_DATA

The CLOSE\_DATA procedure terminates an e-mail message by sending the following sequence:

<CR><LF>.<CR><LF>

This is a single period at the beginning of a line.

CLOSE\_DATA(c IN OUT CONNECTION)

Parameters

c

The SMTP connection to be closed.

### 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.

### 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.

### DATA

The DATA procedure provides the capability to specify the body of the e-mail message. The message is terminated with a <CR><LF>.<CR><LF> sequence.

DATA(c IN OUT CONNECTION, body VARCHAR2)

Parameters

c

The SMTP connection to which the command is to be sent.

body

Body of the e-mail message to be sent.

### EHLO

The EHLO procedure performs initial handshaking with the SMTP server after establishing the connection. The EHLO procedure allows the client to identify itself to the SMTP server according to RFC 821. RFC 1869 specifies the format of the information returned in the server’s reply. The HELO procedure performs the equivalent functionality, but returns less information about the server.

EHLO(c IN OUT CONNECTION, domain VARCHAR2)

Parameters

c

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

domain

Domain name of the sending host.

### HELO

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

HELO(c IN OUT, domain VARCHAR2)

Parameters

c

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

domain

Domain name of the sending host.

### HELP

The HELP function provides the capability to send the HELP command to the SMTP server.

replies REPLIES HELP(c IN OUT CONNECTION [, command VARCHAR2 ])

Parameters

c

The SMTP connection to which the command is to be sent.

command

Command on which help is requested.

replies

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

### MAIL

The MAIL procedure initiates a mail transaction.

MAIL(c IN OUT CONNECTION, sender VARCHAR2

[, parameters VARCHAR2 ])

Parameters

c

Connection to SMTP server on which to start a mail transaction.

sender

The sender’s e-mail address.

parameters

Mail command parameters in the format, key=value as defined in RFC 1869.

### NOOP

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

reply REPLY NOOP(c IN OUT CONNECTION)

NOOP(c IN OUT CONNECTION)

Parameters

c

The SMTP connection on which to send the command.

reply

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

### OPEN\_CONNECTION

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

c CONNECTION OPEN\_CONNECTION(host VARCHAR2 [, port PLS\_INTEGER [, tx\_timeout PLS\_INTEGER DEFAULT NULL]])

Parameters

host

Name of the SMTP server.

port

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

tx\_timeout

Time out value in seconds. Do not wait is indicated by specifying 0. Wait indefinitely is indicated by setting timeout to null. The default is null.

c

Connection handle returned by the SMTP server.

### OPEN\_DATA

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

OPEN\_DATA(c IN OUT CONNECTION)

Parameters

c

SMTP connection on which to send the command.

### QUIT

The QUIT procedure closes the session with an SMTP server.

QUIT(c IN OUT CONNECTION)

Parameters

c

SMTP connection to be terminated.

### RCPT

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

RCPT(c IN OUT CONNECTION, recipient VARCHAR2

[, parameters VARCHAR2 ])

Parameters

c

Connection to SMTP server on which to add a recipient.

recipient

The recipient’s e-mail address.

parameters

Mail command parameters in the format, key=value as defined in RFC 1869.

### RSET

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

RSET(c IN OUT CONNECTION)

Parameters

c

SMTP connection on which to cancel the mail transaction.

### VRFY

The VRFY function provides the capability to validate and verify the recipient’s e-mail address. If valid, the recipient’s full name and fully qualified mailbox is returned.

reply REPLY VRFY(c IN OUT CONNECTION, recipient VARCHAR2)

Parameters

c

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

recipient

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

reply

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

### WRITE\_DATA

The WRITE\_DATA procedure provides the capability to add VARCHAR2 data to an e-mail message. The WRITE\_DATA procedure may be repetitively called to add data.

WRITE\_DATA(c IN OUT CONNECTION, data VARCHAR2)

Parameters

c

The SMTP connection on which to add data.

data

Data to be added to the e-mail message. The data must conform to the RFC 822 specification.

### Comprehensive Example

The following procedure constructs and sends a text e-mail message using the UTL\_SMTP package.

CREATE OR REPLACE PROCEDURE send\_mail (

p\_sender VARCHAR2,

p\_recipient VARCHAR2,

p\_subj VARCHAR2,

p\_msg VARCHAR2,

p\_mailhost VARCHAR2

)

IS

v\_conn UTL\_SMTP.CONNECTION;

v\_crlf CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);

v\_port CONSTANT PLS\_INTEGER := 25;

BEGIN

v\_conn := UTL\_SMTP.OPEN\_CONNECTION(p\_mailhost,v\_port);

UTL\_SMTP.HELO(v\_conn,p\_mailhost);

UTL\_SMTP.MAIL(v\_conn,p\_sender);

UTL\_SMTP.RCPT(v\_conn,p\_recipient);

UTL\_SMTP.DATA(v\_conn, SUBSTR(

'Date: ' || TO\_CHAR(SYSDATE,

'Dy, DD Mon YYYY HH24:MI:SS') || v\_crlf

|| 'From: ' || p\_sender || v\_crlf

|| 'To: ' || p\_recipient || v\_crlf

|| 'Subject: ' || p\_subj || v\_crlf

|| p\_msg

, 1, 32767));

UTL\_SMTP.QUIT(v\_conn);

END;

EXEC send\_mail('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday Party','Are you planning to attend?','smtp.enterprisedb.com');

The following example uses the OPEN\_DATA, WRITE\_DATA, and CLOSE\_DATA procedures instead of the DATA procedure.

CREATE OR REPLACE PROCEDURE send\_mail\_2 (

p\_sender VARCHAR2,

p\_recipient VARCHAR2,

p\_subj VARCHAR2,

p\_msg VARCHAR2,

p\_mailhost VARCHAR2

)

IS

v\_conn UTL\_SMTP.CONNECTION;

v\_crlf CONSTANT VARCHAR2(2) := CHR(13) || CHR(10);

v\_port CONSTANT PLS\_INTEGER := 25;

BEGIN

v\_conn := UTL\_SMTP.OPEN\_CONNECTION(p\_mailhost,v\_port);

UTL\_SMTP.HELO(v\_conn,p\_mailhost);

UTL\_SMTP.MAIL(v\_conn,p\_sender);

UTL\_SMTP.RCPT(v\_conn,p\_recipient);

UTL\_SMTP.OPEN\_DATA(v\_conn);

UTL\_SMTP.WRITE\_DATA(v\_conn,'From: ' || p\_sender || v\_crlf);

UTL\_SMTP.WRITE\_DATA(v\_conn,'To: ' || p\_recipient || v\_crlf);

UTL\_SMTP.WRITE\_DATA(v\_conn,'Subject: ' || p\_subj || v\_crlf);

UTL\_SMTP.WRITE\_DATA(v\_conn,v\_crlf || p\_msg);

UTL\_SMTP.CLOSE\_DATA(v\_conn);

UTL\_SMTP.QUIT(v\_conn);

END;

EXEC send\_mail\_2('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday Party','Are you planning to attend?','smtp.enterprisedb.com');

## UTL\_URL

The UTL\_URL package provides a way to escape illegal and reserved characters within an URL.

| Function/Procedure | Return Type | Description |
| --- | --- | --- |
| ESCAPE(url, escape\_reserved\_chars, url\_charset) | VARCHAR2 | Use the ESCAPE function to escape any illegal and reserved characters in a URL. |
| UNESCAPE(url, url\_charset) | VARCHAR2 | The UNESCAPE function to convert an URL to it's original form. |

The UTL\_URL package will return the BAD\_URL exception if the call to a function includes an incorrectly-formed URL.

### ESCAPE

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

ESCAPE(url VARCHAR2, escape\_reserved\_chars BOOLEAN, url\_charset VARCHAR2)

Reserved characters are replaced with a percent sign, followed by the two-digit hex code of the ascii value for the escaped character.

Parameters

*url*

*url* specifies the Uniform Resource Locator that UTL\_URL will escape.

*escape\_reserved\_chars*

*escape\_reserved\_chars* is a BOOLEAN value that instructs the ESCAPE function to escape reserved characters as well as illegal characters:

* If *escaped*\_*reserved*\_chars is FALSE, ESCAPE will escape only the illegal characters in the specified URL.
* If *escape*\_*reserved*\_*chars* is TRUE, ESCAPE will escape both the illegal characters and the reserved characters in the specified URL.

By default, *escape*\_*reserved*\_*chars* is FALSE.

Within an URL, legal characters are:

|  |  |  |
| --- | --- | --- |
| Uppercase A through Z | Lowercase a through z | 0 through 9 |
| asterisk (\*) | exclamation point (!) | hyphen (-) |
| left parenthesis (() | period (.) | right parenthesis ()) |
| single-quote (') | tilde (~) | underscore (\_) |

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

|  |  |
| --- | --- |
| Illegal Character | Escape Sequence |
| a blank space ( ) | %20 |
| curly braces ({ or }) | %7b and %7d |
| hash mark (#) | %23 |

The ESCAPE function considers the following characters to be reserved, and will escape them if escape\_reserved\_chars is set to TRUE:

|  |  |
| --- | --- |
| Reserved Character | Escape Sequence |
| ampersand (&) | %5C |
| at sign (@) | %25 |
| colon (:) | %3a |
| comma (,) | %2c |
| dollar sign ($) | %24 |
| equal sign (=) | %3d |
| plus sign (+) | %2b |
| question mark (?) | %3f |
| semi-colon (;) | %3b |
| slash (/) | %2f |

*url\_charset*

*url\_charset* specifies a character set to which a given character will be converted before it is escaped. If url\_*charset* is NULL, the character will not be converted. The default value of *url*\_*charset* is ISO-8859-1.

Examples

The following anonymous block uses the ESCAPE function to escape the blank spaces in the URL:

DECLARE  
 result varchar2(400);  
BEGIN  
 result := UTL\_URL.ESCAPE('http://www.example.com/Using the ESCAPE function.html');  
 DBMS\_OUTPUT.PUT\_LINE(result);  
END;

The resulting (escaped) URL is:

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

If you include a value of TRUE for the *escape*\_*reserved*\_*chars* parameter when invoking the function:

DECLARE  
 result varchar2(400);  
BEGIN  
 result := UTL\_URL.ESCAPE('http://www.example.com/Using the ESCAPE function.html', TRUE);  
 DBMS\_OUTPUT.PUT\_LINE(result);  
END;

The ESCAPE function escapes the reserved characters as well as the illegal characters in the URL:

http%3A%2F%2Fwww.example.com%2FUsing%20the%20ESCAPE%20function.html

### UNESCAPE

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

The signature is:

UNESCAPE(*url* VARCHAR2, *url*\_*charset* VARCHAR2)

Parameters

*url*

*url* specifies the Uniform Resource Locator that UTL\_URL will unescape.

*url\_charset*

After unescaping a character, the character is assumed to be in *url\_charset* encoding, and will be converted from that encoding to database encoding before being returned. If url\_*charset* is NULL, the character will not be converted. The default value of *url*\_*charset* is ISO-8859-1.

Examples

The following anonymous block uses the ESCAPE function to escape the blank spaces in the URL:

DECLARE  
 result varchar2(400);  
BEGIN  
 result := UTL\_URL.UNESCAPE('http://www.example.com/Using%20the%20UNESCAPE%20function.html');  
 DBMS\_OUTPUT.PUT\_LINE(result);  
END;

The resulting (unescaped) URL is:

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

# 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.**