Contents

---

1 Database Compatibility for Oracle Developers Built-in Packages Guide navTitle: Built-in Package Guide

introduction packages built-in_packages acknowledgements conclusion

---

2 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:

https://www.enterprisedb.com/edb-docs/

What's New

The following database compatibility for Oracle features have been added to Advanced Server 11 to create Advanced Server 12:

- Advanced Server introduces `COMPOUND TRIGGERS`, which are stored as a PL block that executes in response to a specified triggering event. For information, see the *Database Compatibility for Oracle Developer's Guide*.
- Advanced Server now supports new `DATA DICTIONARY VIEWS` that provide information compatible with the Oracle data dictionary views. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.
- Advanced Server has added the `LISTAGG` function to support string aggregation that concatenates data from multiple rows into a single row in an ordered manner. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.
- Advanced Server now supports `CAST(MULTISET)` function, allowing subquery output to be `CAST` to a nested table type. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.
- Advanced Server has added the `MEDIAN` function to calculate a median value from the set of provided values. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.

- Advanced Server has added the `SYS_GUID` function to generate and return a globally unique identifier in the form of 16-bytes of `RAW` data. For information, see the *Database Compatibility for Oracle Developer's Reference Guide.*
- Advanced Server now supports an Oracle-compatible `SELECT UNIQUE` clause in addition to an existing `SELECT DISTINCT` clause. For information, see the *Database Compatibility for Oracle Developer's Reference Guide.*
- Advanced Server has re-implemented `default_with_rowids` to create a table that includes a `ROWID` column in the newly created table. For information, see the *EDB Postgres Advanced Server Guide.*
- Advanced Server now supports logical decoding on the standby server, which allows creating a logical replication slot on a standby, independently of a primary server. For information, see the *EDB Postgres Advanced Server Guide.*
- Advanced Server introduces `INTERVAL PARTITIONING`, which allows a database to automatically create partitions of a specified interval as new data is inserted into a table. For information, see the *Database Compatibility for Oracle Developer's Guide.*

Note

Database Compatibility for Oracle Developer's Guide, Database Compatibility for Oracle Developer's Reference Guide, and EDB Postgres Advanced Server Guides are available at:

https://www.enterprisedb.com/edb-docs/

3.1 Packages

This chapter discusses the concept of packages in Advanced Server. A *package* is a named collection of functions, procedures, variables, cursors, user-defined record types, and records that are referenced using a common qualifier – the package identifier. Packages have the following characteristics:

- Packages provide a convenient means of organizing the functions and procedures that perform a related purpose. Permission to use the package functions and procedures is dependent upon one privilege granted to the entire package. All of the package programs must be referenced with a common name.
- Certain functions, procedures, variables, types, etc. in the package can be declared as *public*. Public entities are visible and can be referenced by other programs that are given `EXECUTE` privilege on the package. For public functions and procedures, only their signatures are visible - the program names, parameters if any, and return types of functions. The SPL code of these functions and procedures is not accessible to others,

therefore applications that utilize a package are dependent only upon the information available in the signature – not in the procedural logic itself.
- Other functions, procedures, variables, types, etc. in the package can be declared as *private*. Private entities can be referenced and used by function and procedures within the package, but not by other external applications. Private entities are for use only by programs within the package.
- Function and procedure names can be overloaded within a package. One or more functions/procedures can be defined with the same name, but with different signatures. This provides the capability to create identically named programs that perform the same job, but on different types of input.

package_components creating_packages referencing_a_package using_packages_with_user_defined_types dropping_a_package

---

3.2 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>` :=

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

Where <function_definition> :=

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

Where <function_body> :=

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

Where <argument_list> :=

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

Where <argument_declaration> :=

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

Where <options_list> :=

`<option>` [...]

Where `<option>` :=

```
STRICT
LEAKPROOF

PARALLEL { UNSAFE | RESTRICTED | SAFE }

COST <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.

`PRAGMA AUTONOMOUS_TRANSACTION`

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the procedure as an autonomous transaction.

`<declaration>`

A variable, type, `REF CURSOR`, or subprogram declaration. If subprogram declarations are included, they must be declared after all other variable, type, and `REF CURSOR` declarations.

`<statement>`

An SPL program statement. Note that a `DECLARE - BEGIN - END` block is considered an SPL statement unto itself. Thus, the function body may contain nested blocks.

`<exception>`

An exception condition name such as `NO_DATA_FOUND, OTHERS`, etc.

`<func_name>`

The name of the function being created.

`<rettype>`

The return data type, which may be any of the types listed for `<argtype>`. As for `<argtype>`, a length must not be specified for `<rettype>`.

`DETERMINISTIC`

Include `DETERMINISTIC` to specify that the function will always return the same result when given the same argument values. A `DETERMINISTIC` function must not modify the database.

Note

The `DETERMINISTIC` keyword is equivalent to the PostgreSQL `IMMUTABLE` option.

Note

If `DETERMINISTIC` is specified for a public function in the package body, it must also be specified for the function declaration in the package specification. (For private functions, there is no function declaration in the package specification.)

`PRAGMA AUTONOMOUS_TRANSACTION`

`PRAGMA AUTONOMOUS_TRANSACTION` is the directive that sets the function as an autonomous transaction.

`<declaration>`

A variable, type, `REF CURSOR`, or subprogram declaration. If subprogram declarations are included, they must be declared after all other variable, type, and `REF CURSOR` declarations.

`<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.

Note

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

3.3 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;
```

3.4 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`.

3.5 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
```

3.6 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;
```

4.1 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:

https://www.enterprisedb.com/edb-docs

All built-in packages are owned by the special `sys` user which must be specified when granting or revoking privileges on built-in packages:

```
GRANT EXECUTE ON PACKAGE SYS.UTL_FILE TO john;
```

dbms_alert dbms_aq dbms_aqadm dbms_crypto dbms_job dbms_lob dbms_lock dbms_mview dbms_output dbms_pipe dbms_profiler dbms_random dbms_redact dbms_rls dbms_scheduler dbms_session dbms_sql dbms_utility utl_encode utl_file utl_http utl_mail utl_raw utl_smtp utl_url

4.2 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 t
`REMOVE(<name>)`	n/a	Remove registration
`REMOVEALL`	n/a	Remove registration
`SIGNAL(<name>, <message>)`	n/a	Signals the alert nan
`WAITANY(<name> OUT, <message> OUT, <status> OUT, <timeout>)`	n/a	Wait for any register
`WAITONE(<name>, <message. OUT, <status> OUT, <timeout>)`	n/a	Wait for the specifie

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.

DBMS_ALERT_Register

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...
```

DBMS_ALERT_Remove

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_  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);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
```

DBMS_ALERT_Comprehensive_example

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
```

4.3.1 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
DBMS_AQ.BROWSE (0)	Read the message without locking.
DBMS_AQ.LOCKED (1)	This constant is defined, but will return an error if used.
DBMS_AQ.REMOVE (2)	Delete the message after reading; the default.
DBMS_AQ.REMOVE_NODATA (3)	This constant is defined, but will return an error if used.
DBMS_AQ.FIRST_MESSAGE (0)	Return the first available message that matches the search
DBMS_AQ.NEXT_MESSAGE (1)	Return the next available message that matches the search
DBMS_AQ.NEXT_TRANSACTION (2)	This constant is defined, but will return an error if used.
DBMS_AQ.FOREVER (-1)	Wait forever if a message that matches the search term is r
DBMS_AQ.NO_WAIT (0)	Do not wait if a message that matches the search term is n
DBMS_AQ.ON_COMMIT (0)	The dequeue is part of the current transaction.
DBMS_AQ.IMMEDIATE (1)	This constant is defined, but will return an error if used.
DBMS_AQ.PERSISTENT (0)	The message should be stored in a table.
DBMS_AQ.BUFFERED (1)	This constant is defined, but will return an error if used.
DBMS_AQ.READY (0)	Specifies that the message is ready to process.
DBMS_AQ.WAITING (1)	Specifies that the message is waiting to be processed.
DBMS_AQ.PROCESSED (2)	Specifies that the message has been processed.
DBMS_AQ.EXPIRED (3)	Specifies that the message is in the exception queue.
DBMS_AQ.NO_DELAY (0)	This constant is defined, but will return an error if used
DBMS_AQ.NEVER (NULL)	This constant is defined, but will return an error if used
DBMS_AQ.NAMESPACE_AQ (0)	Accept notifications from DBMS_AQ queues.
DBMS_AQ.NAMESPACE_ANONYMOUS (1)	This constant is defined, but will return an error if used

The DBMS_AQ configuration parameters listed in the following table can be defined in the `postgresql.conf` file. After the configuration parameters are defined, you can invoke the DBMS_AQ package to use and manage messages held in queues and queue tables.

Parameter	Description
`dbms_aq.max_workers`	The maximum number of workers to run.
`dbms_aq.max_idle_time`	The idle time a worker must wait before exiting.
`dbms_aq.min_work_time`	The minimum time a worker can run before exiting.
`dbms_aq.launch_delay`	The minimum time between creating workers.
`dbms_aq.batch_size`	The maximum number of messages to process in a single transaction. The
`dbms_aq.max_databases`	The size of DBMS_AQ's hash table of databases. The default value is 102
`dbms_aq.max_pending_retries`	The size of DBMS_AQ's hash table of pending retries. The default value i

enqueue dequeue register unregister

4.3.2 '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 INTEGER,
 delay INTEGER,
 expiration INTEGER,
 correlation CHARACTER VARYING(128) COLLATE pg_catalog."C",
 attempts INTEGER,
 recipient_list "AQ$_RECIPIENT_LIST_T",
 exception_queue CHARACTER VARYING(61) COLLATE pg_catalog."C",
 enqueue_time TIMESTAMP WITHOUT TIME ZONE,
 state INTEGER,
 original_msgid BYTEA,
 transaction_group CHARACTER VARYING(30) COLLATE pg_catalog."C",
 delivery_mode INTEGER
DBMS_AQ.PERSISTENT);
```

The supported values for `message_properties_t` are:

`<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;
```

4.3.3 '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 CHARACTER VARYING(30),
  dequeue_mode INTEGER,
  navigation INTEGER,
  visibility INTEGER,
  wait INTEGER,
  msgid BYTEA,
  correlation CHARACTER VARYING(128),
  deq_condition CHARACTER VARYING(4000),
  transformation CHARACTER VARYING(61),
  delivery_mode INTEGER);
```

Currently, the supported parameter values for `dequeue_options_t` are:

`<message_properties>`

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

```
message_properties_t IS RECORD(
  priority INTEGER,
  delay INTEGER,
  expiration INTEGER,
  correlation CHARACTER VARYING(128) COLLATE pg_catalog."C",
  attempts INTEGER,
  recipient_list "AQ$_RECIPIENT_LIST_T",
  exception_queue CHARACTER VARYING(61) COLLATE pg_catalog."C",
  enqueue_time TIMESTAMP WITHOUT TIME ZONE,
  state INTEGER,
  original_msgid BYTEA,
  transaction_group CHARACTER VARYING(30) COLLATE pg_catalog."C",
  delivery_mode INTEGER
DBMS_AQ.PERSISTENT);
```

The supported values for `message_properties_t` are:

`priority`	If the queue table definition includes a `sort_list` that references priority, this parame
`delay`	Specify the number of seconds that will pass before a message is available for dequeuei
`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
`attempts`	This is a system-maintained value that specifies the number of attempts to dequeue th
`recipient_list`	This parameter is not supported.
`exception_queue`	Use the `exception_queue` parameter to specify the name of an exception queue to wh

`enqueue_time`	`enqueue_time` is the time the record was added to the queue; this value is provided by
`state`	This parameter is maintained by `DBMS_AQ`; state can be: `DBMS_AQ.WAITING` – the delay
`original_msgid`	This parameter is accepted for compatibility and ignored.
`transaction_group`	This parameter is accepted for compatibility and ignored.
`delivery_mode`	This parameter is not supported; specify a value of `DBMS_AQ.PERSISTENT`.

`<payload>`

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

`<msgid>`

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

Example

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

```
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`.

4.3.4 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:

Y{0.2}

`<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`.

4.3.5 'UNREGISTER'

Use the `UNREGISTER` procedure to turn off notifications related to enqueueing and dequeueing. The signature is:

```
UNREGISTER(

    <reg_list> IN SYS.AQ$_REG_INFO_LIST,

    <count> IN NUMBER)
```

Parameter

`<reg_list>`

> `<reg_list>` is a list of type `AQ$_REG_INFO_LIST`; that provides information about each subscription that you would like to register. Each entry within the list is of the type `AQ$_REG_INFO`, and may contain:

`<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`.

4.4.1 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.

Function/Procedure	Return Type	Description
`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
`DBMS_AQADM.TRANSACTIONAL(1)`	This constant is defined, but will return an error if used.	`mes`
`DBMS_AQADM.NONE(0)`	Use to specify message grouping for a queue table.	`mes`
`DBMS_AQADM.NORMAL_QUEUE(0)`	Use with `create_queue` to `specify queue_type`.	`que`
`DBMS_AQADM.EXCEPTION_QUEUE (1)`	Use with `create_queue` to specify `queue_type`.	`que`
`DBMS_AQADM.INFINITE(-1)`	Use with `create_queue` to specify `retention_time`.	`ret`
`DBMS_AQADM.PERSISTENT (0)`	The message should be stored in a table.	`enq`
`DBMS_AQADM.BUFFERED (1)`	This constant is defined, but will return an error if used.	`enq`
`DBMS_AQADM.PERSISTENT_OR_BUFFERED (2)`	This constant is defined, but will return an error if used.	`enq`

alter_queue alter_queue_table create_queue create_queue_table drop_queue drop_queue_table purge_queue_table start_queue stop_queue

---

4.4.2 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);
```

4.4.3 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.

4.4.4 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;
```

4.4.5 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>`

`<multiple_consumers>` queue tables is not supported.

`<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`.

4.4.6 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;
```

4.4.7 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;
```

4.4.8 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
`delivery_mode`	INTEGER	`<delivery_mode>` specifies the type of message that will be purged. The only

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;
```

4.4.9 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;
```

4.4.10 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.

`<enqueu>`

> 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.

4.5.1 DBMS_CRYPTO

The `DBMS_CRYPTO` package provides functions and procedures that allow you to encrypt or decrypt `RAW`, `BLOB` or `CLOB` data. You can also use `DBMS_CRYPTO` functions to generate cryptographically strong random values.

The following table lists the `DBMS_CRYPTO` Functions and Procedures.

Function/Procedure	Return Type	Description
`DECRYPT(src, typ, key, iv)`	`RAW`	Decrypts `RAW` data.
`DECRYPT(dst INOUT, src, typ, key, iv)`	N/A	Decrypts `BLOB` data.
`DECRYPT(dst INOUT, src, typ, key, iv)`	N/A	Decrypts `CLOB` data.
`ENCRYPT(src, typ, key, iv)`	`RAW`	Encrypts `RAW` data.
`ENCRYPT(dst INOUT, src, typ, key, iv)`	N/A	Encrypts `BLOB` data.

`ENCRYPT(dst INOUT, src, typ, key, iv)`	N/A	Encrypts `CLOB` data.
`HASH(src, typ)`	`RAW`	Applies a hash algorithm to `RAW` data.
`HASH(src)`	`RAW`	Applies a hash algorithm to `CLOB` data.
`MAC(src, typ, key)`	`RAW`	Returns the hashed `MAC` value of the given `RAW`
`MAC(src, typ, key)`	`RAW`	Returns the hashed `MAC` value of the given `CLO`
`RANDOMBYTES(number_bytes)`	`RAW`	Returns a specified number of cryptographicall
`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 encrypt hash mac randombytes randominteger randomnumber

4.5.2 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.

4.5.3 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`.

4.5.4 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;
```

4.5.5 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`.

4.5.6 RANDOMBYTES

The `RANDOMBYTES` function returns a `RAW` value of the specified length, containing cryptographically random bytes. The signature is:

```
RANDOMBYTES
 (<number_bytes> IN INTEGER) RETURNS RAW
```

Parameter

`<number_bytes>`

> `<number_bytes>` specifies the number of random bytes to be returned

Examples

The following example uses `RANDOMBYTES` to return a value that is 1024 bytes long:

```
DECLARE
  result RAW(1024);
BEGIN
```

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

4.5.7 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;
```

4.5.8 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;
```

4.6.1 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**	**De**
`BROKEN(<job>, <broken> [, <next_date> ])`	n/a	Sp
`CHANGE(<job, <what>, <next_date>, <interval, instance, force>)`	n/a	Ch
`INTERVAL(<job>, <interval>)`	n/a	Set
`NEXT_DATE(<job>, <next_date>)`	n/a	Set
`REMOVE(<job>)`	n/a	De
`RUN(<job>)`	n/a	For
`SUBMIT(<job> OUT, <what> [, <next_date> [, <interval> [, <no_parse> ]]])`	n/a	Cr
`WHAT(<job>, <what>)`	n/a	Ch

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.

Before using DBMS_JOB, a database superuser must create the pgAgent extension. Use the psql client to connect to a database and invoke the command:

```
CREATE EXTENSION pgagent;
```

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 change interval next_date remove run submit what

4.6.2 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;
```

4.6.3 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;
```

4.6.4 INTERVAL

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

```
INTERVAL(<job> BINARY_INTEGER, <interval> VARCHAR2)
```

Parameters

`<job>`

Identifier of the job to modify.

`<interval>`

Date function that when evaluated, provides the next date/time the job is to be run. If `<interval>` is `NULL` and the job is complete, the job is removed from the queue.

Examples

Change the job to run once a week:

```
BEGIN

   DBMS_JOB.INTERVAL(104,'SYSDATE + 7');

END;
```

4.6.5 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;
```

4.6.6 REMOVE

The `REMOVE` procedure deletes the specified job from the database. The job must be resubmitted using the `SUBMIT` procedure in order to have it executed again. Note that the stored procedure that was associated with the job is not deleted.

```
REMOVE(<job> BINARY_INTEGER)
```

Parameter

`<job>`

Identifier of the job that is to be removed from the database.

Examples

Remove a job from the database:

```
BEGIN

   DBMS_JOB.REMOVE(104);

END;
```

4.6.7 RUN

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

```
RUN(<job> BINARY_INTEGER)
```

Parameter

`<job>`

Identifier of the job to be run.

Examples

Force a job to be run.

```
BEGIN

   DBMS_JOB.RUN(104);

END;
```

4.6.8 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)
```

4.6.9 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;
```

4.7.1 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
`APPEND(<dest_lob> IN OUT, <src_lob>)`

`COMPARE(<lob_1>, <lob_2> [, <amount> [, <offset_1> [, <offset_2> ]]])`
`CONVERTOBLOB(<dest_lob> IN OUT, <src_clob>, <amount>, <dest_offset> IN OUT, <src_offset> IN`
`CONVERTTOCLOB(<dest_lob> IN OUT, <src_blob>, <amount>, <dest_offset> IN OUT, <src_offset> I`
`COPY(<dest_lob> IN OUT, <src_lob>, <amount> [, <dest_offset> [, <src_offset> ]])`
`ERASE(lob_loc IN OUT, <amount> IN OUT [, <offset> ])`
`GET_STORAGE_LIMIT(<lob_loc>)`
`GETLENGTH(<lob_loc>)`
`INSTR(<lob_loc>, <pattern> [, <offset> [, <nth> ]])`
`READ(<lob_loc>, <amount> IN OUT, <offset>, <buffer> OUT)`
`SUBSTR(<lob_loc> [, <amount> [, <offset> ]])`
`TRIM(<lob_loc> IN OUT, <newlen>)`
`WRITE(<lob_loc> IN OUT, <amount>, <offset>, <buffer>)`
`WRITEAPPEND(<lob_loc> IN OUT, <amount>, <buffer>)`

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 compare converttoblob converttoclob copy erase get_storage_limit getlength instr read substr trim write writeappend

---

4.7.2 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>`.

4.7.3 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 is the maximum size of a large object.

`<offset_1>`

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

`<offset_2>`

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

`<status>`

Zero if both large objects are exactly the same for the specified length for the specified offsets. Non-zero, if the objects are not the same. `<NULL>` if `<amount>`, `<offset_1>`, or `<offset_2>` are less than zero.

4.7.4 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.

4.7.5 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.

4.7.6 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.

4.7.7 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.

4.7.8 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.

4.7.9 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`.

4.7.10 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>`.

4.7.11 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`.

4.7.12 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`.

4.7.13 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.

4.7.14 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`.

4.7.15 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`.

4.8 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.

4.9.1 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
`GET_MV_DEPENDENCIES(<list> VARCHAR2, <deplist> VARCHAR2);`
`REFRESH(<list> VARCHAR2, <method> VARCHAR2, <rollback_seg> VARCHAR2 , <push_deferred_rpc> BO`
`REFRESH(<tab> dbms_utility.uncl_array, <method> VARCHAR2, <rollback_seg> VARCHAR2, <push_def`
`REFRESH_ALL_MVIEWS(<number_of_failures> BINARY_INTEGER, <method> VARCHAR2, <rollback_seg> VA`
`REFRESH_DEPENDENT(<number_of_failures> BINARY_INTEGER, <list> VARCHAR2, <method< VARCHAR2, <`
`REFRESH_DEPENDENT(<number_of_failures> BINARY_INTEGER, <tab> dbms_utility.uncl_array, <metho`

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 refresh refresh_all_mviews refresh_dependent

---

4.9.2 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`.

4.9.3 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');
```

4.9.4 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.

4.9.5 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.

4.10 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 recei
ENABLE(<buffer_size>)	n/a	Enable the capability to send and recei
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 bu
NEW_LINE	n/a	Puts an end-of-line character sequence.
PUT(<item>)	n/a	Puts a partial line without an end-of-lin
PUT_LINE(<item>)	n/a	Puts a complete line with an end-of-line
SERVEROUTPUT(<stdout>)	n/a	Direct messages from PUT, PUT_LINE,

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;
```

DBMS_OUTPUT_DISABLE

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;
```

DBMS_OUTPUT_ENABLE

ENABLE

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

The destination of a message sent with PUT, PUT_LINE, or NEW_LINE depends upon the state of SERVEROUTPUT.

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

- If the last state of `SERVEROUTPUT` is `FALSE`, the message goes to the message buffer.

```
ENABLE [ (<buffer_size> INTEGER) ]
```

Parameter

`<buffer_size>`

Maximum length of the message buffer in bytes. If a `<buffer_size>` of less than 2000 is specified, the buffer size is set to 2000.

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
```

Parameter

The `NEW_LINE` procedure expects no parameters.

DBMS_OUTPUT_PUT

PUT

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

```
PUT(<item> VARCHAR2)
```

Parameter

`<item>`

Text written to the message buffer.

Examples

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

```
DECLARE
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    FOR i IN emp_cur LOOP
        DBMS_OUTPUT.PUT(i.empno);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.ename);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.job);
        DBMS_OUTPUT.PUT(',');
        DBMS_OUTPUT.PUT(i.mgr);
        DBMS_OUTPUT.PUT(',');
        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
```

DBMS_OUTPUT_PUT_LINE

PUT_LINE

The `PUT_LINE` procedure writes a single line to the message buffer including an end-of-line character sequence.

```
PUT_LINE(<item> VARCHAR2)
```

Parameter

`<item>`

Text to be written to the message buffer.

Examples

The following example uses the `PUT_LINE` procedure to display a comma-delimited list of employees from the `emp` table.

```
DECLARE
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    FOR i IN emp_cur LOOP
        v_emprec := i.empno || ',' || i.ename || ',' || i.job || ',' ||
            NVL(LTRIM(TO_CHAR(i.mgr,'9999')),'') || ',' || i.hiredate ||
            ',' || i.sal || ',' ||
            NVL(LTRIM(TO_CHAR(i.comm,'9990.99')),'') || ',' || i.deptno;
        DBMS_OUTPUT.PUT_LINE(v_emprec);
    END LOOP;
END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

SERVEROUTPUT

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

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

```
    SERVEROUTPUT(<stdout> BOOLEAN)
```

Parameter

`<stdout>`

> Set to `TRUE` if subsequent `PUT, PUT_LINE`, or `NEW_LINE` commands are to send text directly to standard output of the command line. Set to `FALSE` if text is to be sent to the message buffer.

Examples

The following anonymous block sends the first message to the command line and the second message to the message buffer.

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('This message goes to the command line');
    DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    DBMS_OUTPUT.PUT_LINE('This message goes to the message buffer');
END;

This message goes to the command line
```

If within the same session, the following anonymous block is executed, the message stored in the message buffer from the prior example is flushed and displayed on the command line as well as the new message.

```
BEGIN
    DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    DBMS_OUTPUT.PUT_LINE('Flush messages from the buffer');
END;

This message goes to the message buffer
Flush messages from the buffer
```

4.11.1 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 p
`NEXT_ITEM_TYPE`	`INTEGER`	Determine the data
`PACK_MESSAGE(<item>)`	n/a	Place `<item>` in the
`PURGE(<pipename>)`	n/a	Remove unreceived
`RECEIVE_MESSAGE(<pipename> [, <timeout> ])`	`INTEGER`	Get a message from
`REMOVE_PIPE(<pipename>)`	`INTEGER`	Delete an explicitly
`RESET_BUFFER`	n/a	Reset the local mess
`SEND_MESSAGE(<pipename> [, <timeout> ] [, <maxpipesize> ])`	`INTEGER`	Send a message on a
`UNIQUE_SESSION_NAME`	`VARCHAR2`	Obtain a unique ses
`UNPACK_MESSAGE(<item> OUT)`	n/a	Retrieve the next da

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 next_item_pipe pack_message purge receive_message remove_pipe reset_buffer send_message unique_session_name unpack_message comprehensive_example

---

4.11.2 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
```

4.11.3 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 the following table.

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

Note

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

Examples

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

```
DECLARE
    v_number        NUMBER := 123;
    v_varchar       VARCHAR2(20) := 'Character data';
    v_date          DATE := SYSDATE;
    v_raw           RAW(4) := '21222324';
    v_status        INTEGER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE(v_number);
    DBMS_PIPE.PACK_MESSAGE(v_varchar);
    DBMS_PIPE.PACK_MESSAGE(v_date);
    DBMS_PIPE.PACK_MESSAGE(v_raw);
    v_status := DBMS_PIPE.SEND_MESSAGE('datatypes');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

SEND_MESSAGE status: 0

DECLARE
    v_number        NUMBER;
    v_varchar       VARCHAR2(20);
    v_date          DATE;
    v_timestamp     TIMESTAMP;
    v_raw           RAW(4);
    v_status        INTEGER;
```

```
BEGIN
    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
```

4.11.4 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.

4.11.5 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.

Parameter

`<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
```

4.11.6 RECEIVE_MESSAGE

The `RECEIVE_MESSAGE` function obtains a message from a specified pipe.

```
<status> INTEGER RECEIVE_MESSAGE(<pipename> VARCHAR2
        [, <timeout> INTEGER ])
```

Parameters

`<pipename>`

Name of the pipe.

`<timeout>`

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

`<status>`

Status code returned by the operation.

The possible status codes are:

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

4.11.7 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
```

4.11.8 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?
```

4.11.9 SEND_MESSAGE

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

```
    <status> SEND_MESSAGE(<pipename> VARCHAR2 [, <timeout>
    INTEGER ]

        [, <maxpipesize> INTEGER ])
```

Parameters

`<pipename>`

Name of the pipe.

`<timeout>`

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

`<maxpipesize>`

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

`<status>`

Status code returned by the operation.

The possible status codes are:

Status Code	Description
0	Success
1	Time out
3	Function interrupted

4.11.10 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
```

4.11.11 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 })
```

Parameter

<item>

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

4.11.12 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, toc

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,','

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
```

4.12 DBMS_PROFILER

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

Function/Procedure	Return Type
`FLUSH_DATA`	Status Code or Exception
`GET_VERSION(<major> OUT, <minor> OUT)`	n/a
`INTERNAL_VERSION_CHECK`	Status Code
`PAUSE_PROFILER`	Status Code or Exception
`RESUME_PROFILER`	Status Code or Exception
`START_PROFILER(<run_comment>, <run_comment1> [, <run_number> OUT ])`	Status Code or Exception
`STOP_PROFILER`	Status Code or Exception

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
```

Parameter

`<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
```

Parameter

`<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
```

Parameter

`<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
```

Parameter

`<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
```

Parameter

`<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:

```
CREATE OR REPLACE FUNCTION getBalance(acctNumber INTEGER)
RETURNS NUMERIC AS $$
DECLARE
result NUMERIC;
BEGIN
SELECT INTO result balance FROM acct WHERE id = acctNumber;

IF (result IS NULL) THEN
    RAISE INFO 'Balance is null';
END IF;

RETURN result;
END;
$$ LANGUAGE 'plpgsql';
```

DBMS_PROFILER adds one PLSQL_PROFILER_DATA entry for each line of code 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).

2. Then call the `list_emp` function:

```
acctg=# SELECT list_emp();
INFO:  EMPNO    ENAME
INFO:  -----    -------
INFO:  7369     SMITH
INFO:  7499     ALLEN
INFO:  7521     WARD
INFO:  7566     JONES
INFO:  7654     MARTIN
INFO:  7698     BLAKE
INFO:  7782     CLARK
INFO:  7788     SCOTT
INFO:  7839     KING
INFO:  7844     TURNER
INFO:  7876     ADAMS
INFO:  7900     JAMES
INFO:  7902     FORD
INFO:  7934     MILLER
 list_emp
----------

(1 row)
```

3. Stop the profiling session with a call to `dbms_profiler.stop_profiler`:

```
acctg=# EXEC dbms_profiler.stop_profiler;

EDB-SPL Procedure successfully completed
```

4. Start a new session with the `dbms_profiler.start_profiler` function (followed by a new comment):

```
acctg=# EXEC dbms_profiler.start_profiler('profile get_dept_name and
emp_sal_trig');

EDB-SPL Procedure successfully completed
```

5. Invoke the `get_dept_name` function:

```
acctg=# SELECT get_dept_name(10);
 get_dept_name
---------------
 ACCOUNTING
(1 row)
```

6. Execute an `UPDATE` statement that causes a trigger to execute:

```
acctg=# UPDATE memp SET sal = 500 WHERE empno = 7902;
INFO: Updating employee 7902
INFO: ..Old salary: 3000.00
INFO: ..New salary: 500.00
INFO: ..Raise: -2500.00
INFO: User enterprisedb updated employee(s) on 04-FEB-14
UPDATE 1
```

7. Terminate the profiling session and flush the performance information to the profiling tables:

```
acctg=# EXEC dbms_profiler.stop_profiler;

EDB-SPL Procedure successfully completed
```

8. Now, query the `plsql_profiler_runs` table to view a list of the profiling sessions, arranged by `runid`:

```
acctg=# SELECT * FROM plsql_profiler_runs;
 runid | related_run |  run_owner   |         run_date          |              run_comment
|run_total_time | run_system_info | run_comment1 | spare1
-------+-------------+--------------+---------------------------+---------------------------
     1 |             | enterprisedb | 04-FEB-14 09:32:48.874315 | profile list_emp
     2 |             | enterprisedb | 04-FEB-14 09:41:30.546503 | profile get_dept_name and
(2 rows)
```

9. Query the `plsql_profiler_units` table to view the amount of time consumed by each unit (each function, procedure, or trigger):

```
acctg=# SELECT * FROM plsql_profiler_units;
 runid | unit_number | unit_type |  unit_owner  |            unit_name            | unit_ti
-------+-------------+-----------+--------------+---------------------------------+---------
     1 |       16999 | FUNCTION  | enterprisedb | list_emp()                      |
     2 |       17002 | FUNCTION  | enterprisedb | user_audit_trig()               |
     2 |       17000 | FUNCTION  | enterprisedb | get_dept_name(p_deptno numeric) |
     2 |       17004 | FUNCTION  | enterprisedb | emp_sal_trig()                  |
(4 rows)
```

10. Query the `plsql_profiler_rawdata` table to view a list of the wait event counters and wait event times:

```
acctg=# SELECT runid, sourcecode, func_oid, line_number, exec_count, tuples_returned, time_t

 runid |                            sourcecode                             | func_oid | line_n
-------+-------------------------------------------------------------------+----------+-------
     1 | DECLARE                                                           |    16999 |
     1 |     v_empno         NUMERIC(4);                                   |    16999 |
     1 |     v_ename         VARCHAR(10);                                  |    16999 |
     1 |     emp_cur CURSOR FOR                                            |    16999 |
     1 |         SELECT empno, ename FROM memp ORDER BY empno;             |    16999 |
     1 | BEGIN                                                         |     16999 |
     1 |     OPEN emp_cur;                                             |     16999 |
     1 |     RAISE INFO 'EMPNO    ENAME';                              |     16999 |
     1 |     RAISE INFO '-----    -------';                            |     16999 |
     1 |     LOOP                                                      |     16999 |
     1 |         FETCH emp_cur INTO v_empno, v_ename;                  |     16999 |
     1 |         EXIT WHEN NOT FOUND;                                  |     16999 |
     1 |         RAISE INFO '%     %', v_empno, v_ename;               |     16999 |
     1 |     END LOOP;                                                 |     16999 |
     1 |     CLOSE emp_cur;                                            |     16999 |
     1 |     RETURN;                                                   |     16999 |
     1 | END;                                                          |     16999 |
     1 |                                                               |     16999 |
     2 | DECLARE                                                       |     17002 |
     2 |     v_action        VARCHAR(24);                              |     17002 |
     2 |     v_text          TEXT;                                     |     17002 |
     2 | BEGIN                                                         |     17002 |
     2 |     IF TG_OP = 'INSERT' THEN                                  |     17002 |
     2 |         v_action := ' added employee(s) on ';                 |     17002 |
     2 |     ELSIF TG_OP = 'UPDATE' THEN                               |     17002 |
     2 |         v_action := ' updated employee(s) on ';               |     17002 |
     2 |     ELSIF TG_OP = 'DELETE' THEN                               |     17002 |
     2 |         v_action := ' deleted employee(s) on ';               |     17002 |
     2 |     END IF;                                                   |     17002 |
     2 |     v_text := 'User ' || USER || v_action || CURRENT_DATE;    |     17002 |
     2 |     RAISE INFO ' %', v_text;                                  |     17002 |
     2 |     RETURN NULL;                                              |     17002 |
     2 | END;                                                          |     17002 |
     2 |                                                               |     17002 |
     2 | DECLARE                                                       |     17000 |
     2 |     v_dname         VARCHAR(14);                              |     17000 |
     2 | BEGIN                                                         |     17000 |
     2 |     SELECT INTO v_dname dname FROM dept WHERE deptno = p_deptno; |     17000 |
     2 |     RETURN v_dname;                                           |     17000 |
     2 |     IF NOT FOUND THEN                                         |     17000 |
     2 |         RAISE INFO 'Invalid department number %', p_deptno;   |     17000 |
     2 |         RETURN '';                                            |     17000 |
     2 |     END IF;                                                   |     17000 |
     2 | END;                                                          |     17000 |
     2 |                                                               |     17000 |
     2 | DECLARE                                                       |     17004 |
     2 |     sal_diff       NUMERIC(7,2);                              |     17004 |
     2 | BEGIN                                                         |     17004 |
     2 |     IF TG_OP = 'INSERT' THEN                                  |     17004 |
     2 |         RAISE INFO 'Inserting employee %', NEW.empno;         |     17004 |
     2 |         RAISE INFO '..New salary: %', NEW.sal;                |     17004 |
     2 |         RETURN NEW;                                          |    17004 |
     2 |     END IF;                                                  |    17004 |
     2 |     IF TG_OP = 'UPDATE' THEN                                 |    17004 |
     2 |         sal_diff := NEW.sal - OLD.sal;                       |    17004 |
     2 |         RAISE INFO 'Updating employee %', OLD.empno;         |    17004 |
     2 |         RAISE INFO '..Old salary: %', OLD.sal;               |    17004 |
     2 |         RAISE INFO '..New salary: %', NEW.sal;               |    17004 |
     2 |         RAISE INFO '..Raise     : %', sal_diff;              |    17004 |
     2 |         RETURN NEW;                                          |    17004 |
     2 |     END IF;                                                  |    17004 |
     2 |     IF TG_OP = 'DELETE' THEN                                 |    17004 |
     2 |         RAISE INFO 'Deleting employee %', OLD.empno;         |    17004 |
     2 |         RAISE INFO '..Old salary: %', OLD.sal;               |    17004 |
     2 |         RETURN OLD;                                          |    17004 |
     2 |     END IF;                                                  |    17004 |
     2 | END;                                                         |    17004 |
     2 |                                                              |    17004 |
  (68 rows)
```

11. Query the `plsql_profiler_data` view to review a subset of the information found in `plsql_profiler_rawdata` table:

```
acctg=# SELECT * FROM plsql_profiler_data;
runid | unit_number | line# | total_occur | total_time | min_time | max_time | spare1 | spar
-------+-------------+-------+-------------+------------+----------+----------+--------+----
     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 statis

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 tab
`unit_type`	`TEXT`	PL/SQL function, procedure, trigger or anonymous blo
`unit_owner`	`TEXT`	The identity of the role that owns the unit.
`unit_name`	`TEXT`	The complete signature of the unit.
`unit_timestamp`	`TIMESTAMP WITHOUT TIME ZONE`	Creation date of the unit (currently NULL).
`total_time`	`BIGINT`	Time spent within the unit (in milliseconds)
`spare1`	`BIGINT`	Currently Unused
`spare2`	`BIGINT`	Currently Unused

PLSQL_PROFILER_DATA The `PLSQL_PROFILER_DATA` view contains the following columns:

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

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

Column	Data Type	Description
`runid`	`INTEGER`	The run identifier (plsql_profiler_runn
`sourcecode`	`TEXT`	The individual line of profiled code.
`func_oid`	`OID`	Object ID of the unit that contains the
`line_number`	`INTEGER`	Current line number of the profiled wor
`exec_count`	`BIGINT`	The number of times that the line was
`tuples_returned`	`BIGINT`	Currently Unused
`time_total`	`DOUBLE PRECISION`	The amount of time spent executing th
`time_shortest`	`DOUBLE PRECISION`	The minimum execution time for the li
`time_longest`	`DOUBLE PRECISION`	The maximum execution time for the li
`num_scans`	`BIGINT`	Currently Unused
`tuples_fetched`	`BIGINT`	Currently Unused
`tuples_inserted`	`BIGINT`	Currently Unused
`tuples_updated`	`BIGINT`	Currently Unused
`tuples_deleted`	`BIGINT`	Currently Unused
`blocks_fetched`	`BIGINT`	Currently Unused
`blocks_hit`	`BIGINT`	Currently Unused
`wal_write`	`BIGINT`	A server has waited for a write to the w
`wal_flush`	`BIGINT`	A server has waited for the write-ahead
`wal_file_sync`	`BIGINT`	A server has waited for the write-ahead
`db_file_read`	`BIGINT`	A server has waited for the completion
`db_file_write`	`BIGINT`	A server has waited for the completion
`db_file_sync`	`BIGINT`	A server has waited for the operating sy

Column	Data Type	Description
`db_file_extend`	BIGINT	A server has waited for the operating sy
`sql_parse`	BIGINT	Currently Unused.
`query_plan`	BIGINT	A server has generated a query plan.
`other_lwlock_acquire`	BIGINT	A server has waited for other light-weig
`shared_plan_cache_collision`	BIGINT	A server has waited for the completion
`shared_plan_cache_insert`	BIGINT	A server has waited for the completion
`shared_plan_cache_hit`	BIGINT	A server has waited for the completion
`shared_plan_cache_miss`	BIGINT	A server has waited for the completion
`shared_plan_cache_lock`	BIGINT	A server has waited for the completion
`shared_plan_cache_busy`	BIGINT	A server has waited for the completion
`shmemindexlock`	BIGINT	A server has waited to find or allocate s
`oidgenlock`	BIGINT	A server has waited to allocate or assig
`xidgenlock`	BIGINT	A server has waited to allocate or assig
`procarraylock`	BIGINT	A server has waited to get a snapshot o
`sinvalreadlock`	BIGINT	A server has waited to retrieve or remov
`sinvalwritelock`	BIGINT	A server has waited to add a message t
`walbufmappinglock`	BIGINT	A server has waited to replace a page in
`walwritelock`	BIGINT	A server has waited for WAL buffers to
`controlfilelock`	BIGINT	A server has waited to read or update t
`checkpointlock`	BIGINT	A server has waited to perform a check
`clogcontrollock`	BIGINT	A server has waited to read or update t
`subtranscontrollock`	BIGINT	A server has waited to read or update t
`multixactgenlock`	BIGINT	A server has waited to read or update t
`multixactoffsetcontrollock`	BIGINT	A server has waited to read or update r
`multixactmembercontrollock`	BIGINT	A server has waited to read or update r
`relcacheinitlock`	BIGINT	A server has waited to read or write the
`checkpointercommlock`	BIGINT	A server has waited to manage the fsyn
`twophasestatelock`	BIGINT	A server has waited to read or update t
`tablespacecreatelock`	BIGINT	A server has waited to create or drop th
`btreevacuumlock`	BIGINT	A server has waited to read or update t
`addinshmeminitlock`	BIGINT	A server has waited to manage space al
`autovacuumlock`	BIGINT	The autovacuum launcher waiting to re
`autovacuumschedulelock`	BIGINT	A server has waited to ensure that the
`syncscanlock`	BIGINT	A server has waited to get the start loc
`relationmappinglock`	BIGINT	A server has waited to update the relat
`asyncctllock`	BIGINT	A server has waited to read or update s
`asyncqueuelock`	BIGINT	A server has waited to read or update t
`serializablexacthashlock`	BIGINT	A server has waited to retrieve or store
`serializablefinishedlistlock`	BIGINT	A server has waited to access the list of
`serializablepredicatelocklistlock`	BIGINT	A server has waited to perform an oper
`oldserxidlock`	BIGINT	A server has waited to read or record th
`syncreplock`	BIGINT	A server has waited to read or update i
`backgroundworkerlock`	BIGINT	A server has waited to read or update t
`dynamicsharedmemorycontrollock`	BIGINT	A server has waited to read or update t

Column	Data Type	Description
`autofilelock`	BIGINT	A server has waited to update the `post`
`replicationslotallocationlock`	BIGINT	A server has waited to allocate or free a
`replicationslotcontrollock`	BIGINT	A server has waited to read or update r
`committscontrollock`	BIGINT	A server has waited to read or update t
`committslock`	BIGINT	A server has waited to read or update t
`replicationoriginlock`	BIGINT	A server has waited to set up, drop, or
`multixacttruncationlock`	BIGINT	A server has waited to read or truncate
`oldsnapshottimemaplock`	BIGINT	A server has waited to read or update o
`backendrandomlock`	BIGINT	A server has waited to generate a rando
`logicalrepworkerlock`	BIGINT	A server has waited for the action on lo
`clogtruncationlock`	BIGINT	A server has waited to truncate the wri
`bulkloadlock`	BIGINT	A server has waited for the `bulkloadlo`
`edbresourcemanagerlock`	BIGINT	The `edbresourcemanagerlock` provides
`wal_write_time`	BIGINT	The amount of time that the server has
`wal_flush_time`	BIGINT	The amount of time that the server has
`wal_file_sync_time`	BIGINT	The amount of time that the server has
`db_file_read_time`	BIGINT	The amount of time that the server has
`db_file_write_time`	BIGINT	The amount of time that the server has
`db_file_sync_time`	BIGINT	The amount of time that the server has
`db_file_extend_time`	BIGINT	The amount of time that the server has
`sql_parse_time`	BIGINT	The amount of time that the server has
`query_plan_time`	BIGINT	The amount of time that the server has
`other_lwlock_acquire_time`	BIGINT	The amount of time that the server has
`shared_plan_cache_collision_time`	BIGINT	The amount of time that the server has
`shared_plan_cache_insert_time`	BIGINT	The amount of time that the server has
`shared_plan_cache_hit_time`	BIGINT	The amount of time that the server has
`shared_plan_cache_miss_time`	BIGINT	The amount of time that the server has
`shared_plan_cache_lock_time`	BIGINT	The amount of time that the server has
`shared_plan_cache_busy_time`	BIGINT	The amount of time that the server has
`shmemindexlock_time`	BIGINT	The amount of time that the server has
`oidgenlock_time`	BIGINT	The amount of time that the server has
`xidgenlock_time`	BIGINT	The amount of time that the server has
`procarraylock_time`	BIGINT	The amount of time that the server has
`sinvalreadlock_time`	BIGINT	The amount of time that the server has
`sinvalwritelock_time`	BIGINT	The amount of time that the server has
`walbufmappinglock_time`	BIGINT	The amount of time that the server has
`walwritelock_time`	BIGINT	The amount of time that the server has
`controlfilelock_time`	BIGINT	The amount of time that the server has
`checkpointlock_time`	BIGINT	The amount of time that the server has
`clogcontrollock_time`	BIGINT	The amount of time that the server has
`subtranscontrollock_time`	BIGINT	The amount of time that the server has
`multixactgenlock_time`	BIGINT	The amount of time that the server has
`multixactoffsetcontrollock_time`	BIGINT	The amount of time that the server has
`multixactmembercontrollock_time`	BIGINT	The amount of time that the server has

Column	Data Type	Description
relcacheinitlock_time	BIGINT	The amount of time that the server has
checkpointercommlock_time	BIGINT	The amount of time that the server has
twophasestatelock_time	BIGINT	The amount of time that the server has
tablespacecreatelock_time	BIGINT	The amount of time that the server has
btreevacuumlock_time	BIGINT	The amount of time that the server has
addinshmeminitlock_time	BIGINT	The amount of time that the server has
autovacuumlock_time	BIGINT	The amount of time that the server has
autovacuumschedulelock_time	BIGINT	The amount of time that the server has
syncscanlock_time	BIGINT	The amount of time that the server has
relationmappinglock_time	BIGINT	The amount of time that the server has
asyncctllock_time	BIGINT	The amount of time that the server has
asyncqueuelock_time	BIGINT	The amount of time that the server has
serializablexacthashlock_time	BIGINT	The amount of time that the server has
serializablefinishedlistlock_time	BIGINT	The amount of time that the server has
serializablepredicatelocklistlock_time	BIGINT	The amount of time that the server has
oldserxidlock_time	BIGINT	The amount of time that the server has
syncreplock_time	BIGINT	The amount of time that the server has
backgroundworkerlock_time	BIGINT	The amount of time that the server has
dynamicsharedmemorycontrollock_time	BIGINT	The amount of time that the server has
autofilelock_time	BIGINT	The amount of time that the server has
replicationslotallocationlock_time	BIGINT	The amount of time that the server has
replicationslotcontrollock_time	BIGINT	The amount of time that the server has
committscontrollock_time	BIGINT	The amount of time that the server has
committslock_time	BIGINT	The amount of time that the server has
replicationoriginlock_time	BIGINT	The amount of time that the server has
multixacttruncationlock_time	BIGINT	The amount of time that the server has
oldsnapshottimemaplock_time	BIGINT	The amount of time that the server has
backendrandomlock_time	BIGINT	The amount of time that the server has
logicalrepworkerlock_time	BIGINT	The amount of time that the server has
clogtruncationlock_time	BIGINT	The amount of time that the server has
bulkloadlock_time	BIGINT	The amount of time that the server has
edbresourcemanagerlock_time	BIGINT	The amount of time that the server has
totalwaits	BIGINT	The total number of event waits.
totalwaittime	BIGINT	The total time spent waiting for an eve

4.13 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`
`NORMAL()`	`NUMBER`	Returns a random `NUMBER`.
`RANDOM`	`INTEGER`	Returns a random `INTEGER` with a value greater than or equal to -2
`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
`VALUE`	`NUMBER`	Returns a random number with a value greater than or equal to `0`
`VALUE(<low>, <high>)`	`NUMBER`	Returns a random number with a value greater than or equal to `<l`

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.

Parameter

`<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()
```

Parameter

`<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.

Parameter

`<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)
```

Parameter

`<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)
```

Parameter

`<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:

Y{0.6}|

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.

DBMS_RANDOM_VALUE_FIRST_FORM

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()
```

Parameter

`<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();
```

DBMS_RANDOM_VALUE_SECOND_FORM

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);
```

4.14 DBMS_REDACT

The `DBMS_REDACT` package enables the redacting or masking of data returned by a query. The `DBMS_REDACT` package provides a procedure to create policies, alter policies, enable policies, disable policies, and drop policies. The procedures available in the `DBMS_REDACT` package are listed in the following table.

Function/Procedure
`ADD_POLICY(<object_schema>, <object_name>, <policy_name>, <policy_description>, <column_name`
`ALTER_POLICY(<object_schema>, <object_name>, <policy_name>, <action>, <column_name>, <functi`
`DISABLE_POLICY(<object_schema>, <object_name>, <policy_name>)`
`ENABLE_POLICY(<object_schema,> <object_name>, <policy_name>)`
`DROP_POLICY(<object_schema>, <object_name>, <policy_name>)`
`UPDATE_FULL_REDACTION_VALUES(<number_val>, <binfloat_val>, <bindouble_val>, <char_val>, <var`

The data redaction feature uses the `DBMS_REDACT` package to define policies or conditions to redact data in a column based on the table column type and redaction type.

Note that you must be the owner of the table to create or change the data redaction policies. The users are exempted from all the column redaction policies, which the table owner or super-user is by default.

Using DBMS_REDACT Constants and Function Parameters

The `DBMS_REDACT` package uses the constants and redacts the column data by using any one of the data redaction types. The redaction type can be decided based on the function_type parameter of `dbms_redact.add_policy` and `dbms_redact.alter_policy` procedure. The below table highlights the values for function_type parameters of `dbms_redact.add_policy` and `dbms_redact.alter_policy`.

Constant	**Type**	**Value**	**Description**
`NONE`	`INTEGER`	`0`	No redaction, zero effect on the result of a query against table.
`FULL`	`INTEGER`	`1`	Full redaction, redacts full values of the column data.
`PARTIAL`	`INTEGER`	`2`	Partial redaction, redacts a portion of the column data.
`RANDOM`	`INTEGER`	`4`	Random redaction, each query results in a different random value depending
`REGEXP`	`INTEGER`	`5`	Regular Expression based redaction, searches for the pattern of data to reda

`CUSTOM`	`INTEGER`	`99`	Custom redaction type.

The following table shows the values for the `action` parameter of `dbms_redact.alter_policy`.

Constant	Type	Value	Description
`ADD_COLUMN`	`INTEGER`	`1`	Adds a column to the redaction policy.
`DROP_COLUMN`	`INTEGER`	`2`	Drops a column from the redaction policy.
`MODIFY_EXPRESSION`	`INTEGER`	`3`	Modifies the expression of a redaction policy. The redaction i
`MODIFY_COLUMN`	`INTEGER`	`4`	Modifies a column in the redaction policy to change the reda
`SET_POLICY_DESCRIPTION`	`INTEGER`	`5`	Sets the redaction policy description.
`SET_COLUMN_DESCRIPTION`	`INTEGER`	`6`	Sets a description for the redaction performed on the column

The partial data redaction enables you to redact only a portion of the column data. To use partial redaction, you must set the `dbms_redact.add_policy` procedure `function_type` parameter to `dbms_redact.partial` and use the `function_parameters` parameter to specify the partial redaction behavior.

The data redaction feature provides a predefined format to configure policies that use the following datatype:

- `Character`
- `Number`
- `Datetime`

The following table highlights the format descriptor for partial redaction with respect to datatype. The example described below shows how to perform a redaction for a string datatype (in this scenario, a Social Security Number (SSN)), a `Number` datatype, and a `DATE` datatype.

Datatype	Format Descriptor	Description
Character	`REDACT_PARTIAL_INPUT_FORMAT`	Specifies the input format. Enter `V` for each character from t
	`REDACT_PARTIAL_OUTPUT_FORMAT`	Specifies the output format. Enter `V` for each character from
	`REDACT_PARTIAL_MASKCHAR`	Specifies the character to be used for redaction.
	`REDACT_PARTIAL_MASKFROM`	Specifies which `V` within the input format from which to star
	`REDACT_PARTIAL_MASKTO`	Specifies which `V` within the input format at which to end th
Number	`REDACT_PARTIAL_MASKCHAR`	Specifies the character to be displayed in the range between
	`REDACT_PARTIAL_MASKFROM`	Specifies the start digit position for redaction.
	`REDACT_PARTIAL_MASKTO`	Specifies the end digit position for redaction.
Datetime	`REDACT_PARTIAL_DATE_MONTH`	`‘m’` redacts the month. To mask a specific month, specify `‘m`
	`REDACT_PARTIAL_DATE_DAY`	`‘d’` redacts the day of the month. To mask with a day of th
	`REDACT_PARTIAL_DATE_YEAR`	`‘y’` redacts the year. To mask with a year, append `1-9999` t
	`REDACT_PARTIAL_DATE_HOUR`	`‘h’` redacts the hour. To mask with an hour, append `0-23` t
	`REDACT_PARTIAL_DATE_MINUTE`	`‘m’` redacts the minute. To mask with a minute, append `0-5`

`REDACT_PARTIAL_DATE_SECOND`	`'s'` redacts the second. To mask with a second, append `0-5`

The following table represents `function_parameters` values that can be used in partial redaction.

A regular expression-based redaction searches for patterns of data to redact. The `regexp_pattern` search the values in order for the `regexp_replace_string` to change the value. The following table illustrates the `regexp_pattern` values that you can use during `REGEXP` based redaction.

Function Parameter and Description	**Data Type**
	RE_PATTERN_CC_L6_T4: Searches for the middle digit **RE_PATTERN_ANY_DIGITs**: Searches for any digit and **RE_PATTERN_US_PHONE**: Searches for the U.S phone **RE_PATTERN_EMAIL_ADDRESS**: Searches for the em **RE_PATTERN_IP_ADDRESS**: Searches for an IP addres **RE_PATTERN_AMEX_CCN**: Searches for the American **RE_PATTERN_CCN**: Searches for the credit card number **RE_PATTERN_US_SSN**: Searches the SSN number and re **RE_REDACT_CC_MIDDLE_DIGITS**: Redacts the mid **RE_REDACT_WITH_SINGLE_X**: Replaces the data wi

The below table illustrates the `regexp_replace_string` values that you can use during `REGEXP` based redaction.

The following tables show the `regexp_position` value and `regexp_occurence` values that you can use during `REGEXP` based redaction.

Function Parameter	Data Type	Value	Description
RE_BEGINNING	INTEGER	1	Specifies the position of a character where search must begin. By

Function Parameter	Data Type	Value	Description
RE_ALL	INTEGER	0	Specifies the replacement occurrence of a substring. If the value i
RE_FIRST	INTEGER	1	Specifies the replacement occurrence of a substring. If the value i

The following table shows the `regexp_match_parameter` values that you can use during `REGEXP` based redaction which lets you change the default matching behavior of a function.

Function Parameter	Data Type	Value	Description
RE_CASE_SENSITIVE	VARCHAR2	'c'	Specifies the case-sensitive matching.

RE_CASE_INSENSITIVE	VARCHAR2	'i'	Specifies the case-insensitive matching.
RE_MULTIPLE_LINES	VARCHAR2	'm'	Treats the source string as multiple lines but if you
RE_NEWLINE_WILDCARD	VARCHAR2	'n'	Specifies the period (.), but if you omit this parame
RE_IGNORE_WHITESPACE	VARCHAR2	'x'	Ignores the whitespace characters.

Note

If you create a redaction policy based on a numeric type column, then make sure that the result after redaction is a number and accordingly set the replacement string to avoid runtime errors.

Note

If you create a redaction policy based on a character type column, then make sure that a length of the result after redaction is compatible with the column type and accordingly set the replacement string to avoid runtime errors.

DBMS_REDACT_ADD_POLICY

ADD_POLICY

The `add_policy` procedure creates a new data redaction policy for a table.

```
PROCEDURE add_policy (
object_schema         IN VARCHAR2 DEFAULT NULL,
object_name           IN VARCHAR2,
policy_name           IN VARCHAR2,
policy_description        IN VARCHAR2 DEFAULT NULL,
column_name           IN VARCHAR2 DEFAULT NULL,
column_description        IN VARCHAR2 DEFAULT NULL,
function_type     IN INTEGER DEFAULT DBMS_REDACT.FULL,
function_parameters       IN VARCHAR2 DEFAULT NULL,
expression            IN VARCHAR2,
enable                IN BOOLEAN DEFAULT TRUE,
regexp_pattern            IN VARCHAR2 DEFAULT NULL,
regexp_replace_string     IN VARCHAR2 DEFAULT NULL,
regexp_position    IN INTEGER DEFAULT DBMS_REDACT.RE_BEGINNING,
regexp_occurrence IN INTEGER DEFAULT   DBMS_REDACT.RE_ALL,
regexp_match_parameter        IN VARCHAR2 DEFAULT NULL,
custom_function_expression    IN VARCHAR2 DEFAULT NULL
)
```

Parameters

`<object_schema>`

Specifies the name of the schema in which the object resides and on which the data redaction policy will be applied. If you specify

NULL then the given object is searched by the order specified by search_path setting.

<object_name>

Name of the table on which the data redaction policy is created.

<policy_name>

Name of the policy to be added. Ensure that the policy_name is unique for the table on which the policy is created.

<policy_description>

Specify the description of a redaction policy.

<column_name>

Name of the column to which the redaction policy applies. To redact more than one column, use the alter_policy procedure to add additional columns.

<column_description>

Description of the column to be redacted. The column_description is not supported, but if you specify the description for a column then, you will get a warning message.

<function_type>

The type of redaction function to be used. The possible values are NONE, FULL, PARTIAL, RANDOM, REGEXP, and CUSTOM.

<function_parameters>

Specifies the function parameters for the partition redaction and is applicable only for partial redaction.

<expression>

Specifies the Boolean expression for the table and determines how the policy is to be applied. The redaction occurs if this policy expression is evaluated to TRUE.

<enable>

When set to TRUE, the policy is enabled upon creation. The default is set as TRUE. When set to FALSE, the policy is disabled but the policy can be enabled by calling the enable_policy procedure.

<regexp_pattern>

Specifies the regular expression pattern to redact data. If the regexp_pattern does not match, then the NULL value is returned.

<regexp_replace_string>

Specifies the replacement string value.

`<regexp_position>`

Specifies the position of a character where search must begin. By default, the function parameter is `RE_BEGINNING`.

`<regexp_occurrence>`

Specifies the replacement occurrence of a substring. If the constant is `RE_ALL`, then the replacement of each matching substring occurs. If the constant is `RE_FIRST`, then the replacement of the first matching substring occurs.

`<regexp_match_parameter>`

Changes the default matching behavior of a function. The possible regexp__match__parameter constants can be `‘RE_CASE_SENSITIVE’`, `‘RE_CASE_INSENSITIVE’`, `‘RE_MULTIPLE_LINES’`, `‘RE_NEWLINE_WILDCARD’`, `‘RE_IGNORE_WHITESPACE’`.

Note

For more information on `constants`, `function_parameters`, or `regexp` (regular expressions) see, Using `DBMS_REDACT Constants` and `Function Parameters`.

`<custom_function_expression>`

The `custom_function_expression` is applicable only for the `CUSTOM` redaction type. The `custom_function_expression` is a function expression that is, schema-qualified function with a parameter such as `schema_name.function_name (argument1, …)`that allows a user to use their redaction logic to redact the column data.

Example

The following example illustrates how to create a policy and use full redaction for values in the `payment_details_tab` table `customer id` column.

```
edb=# CREATE TABLE payment_details_tab (
customer_id NUMBER       NOT NULL,
card_string VARCHAR2(19) NOT NULL);
CREATE TABLE

edb=# BEGIN
  INSERT INTO payment_details_tab VALUES (4000, '1234-1234-1234-1234');
  INSERT INTO payment_details_tab VALUES (4001, '2345-2345-2345-2345');
END;

EDB-SPL Procedure successfully completed
```

```
edb=# CREATE USER redact_user;
CREATE ROLE
edb=# GRANT SELECT ON payment_details_tab TO redact_user;
GRANT

\c edb base_user

BEGIN
  DBMS_REDACT.add_policy(
    object_schema              => 'public',
    object_name                => 'payment_details_tab',
    policy_name                => 'redactPolicy_001',
    policy_description         => 'redactPolicy_001 for payment_details_tab table',
    column_name                => 'customer_id',
    function_type          => DBMS_REDACT.full,
    expression                 => '1=1',
    enable                     => TRUE);
END;
```

Redacted Result:

```
edb=# \c edb redact_user
You are now connected to database "edb" as user "redact_user".

edb=> select customer_id from payment_details_tab order by 1;
 customer_id
-------------
           0
           0
(2 rows)
```

ALTER_POLICY

The `alter_policy` procedure alters or modifies an existing data redaction policy for a table.

```
PROCEDURE alter_policy (
object_schema                  IN VARCHAR2 DEFAULT NULL,
object_name                    IN VARCHAR2,
policy_name                    IN VARCHAR2,
action            IN INTEGER DEFAULT DBMS_REDACT.ADD_COLUMN,
column_name                    IN VARCHAR2 DEFAULT NULL,
function_type                  IN INTEGER DEFAULT DBMS_REDACT.FULL,
function_parameters          IN VARCHAR2 DEFAULT NULL,
expression                     IN VARCHAR2 DEFAULT NULL,
regexp_pattern             IN VARCHAR2 DEFAULT NULL,
regexp_replace_string          IN VARCHAR2 DEFAULT NULL,
regexp_position  IN INTEGER DEFAULT DBMS_REDACT.RE_BEGINNING,
regexp_occurrence IN INTEGER DEFAULT DBMS_REDACT.RE_ALL,
regexp_match_parameter      IN VARCHAR2 DEFAULT NULL,
policy_description            IN VARCHAR2 DEFAULT NULL,
column_description            IN VARCHAR2 DEFAULT NULL,
custom_function_expression  IN VARCHAR2 DEFAULT NULL
)
```

Parameters

`<object_schema>`

Specifies the name of the schema in which the object resides and on which the data redaction policy will be altered. If you specify `NULL` then the given object is searched by the order specified by `search_path` setting.

`<object_name>`

Name of the table to which to alter a data redaction policy.

`<policy_name>`

Name of the policy to be altered.

`<action>`

The action to perform. For more information about action parameters see, `DBMS_REDACT Constants and Function Parameters`.

`<column_name>`

Name of the column to which the redaction policy applies.

`<function_type>`

The type of redaction function to be used. The possible values are `NONE, FULL, PARTIAL, RANDOM, REGEXP`, and `CUSTOM`.

`<function_parameters>`

Specifies the function parameters for the redaction function.

`<expression>`

Specifies the Boolean expression for the table and determines how the policy is to be applied. The redaction occurs if this policy expression is evaluated to `TRUE`.

`<regexp_pattern>`

Enables the use of regular expressions to redact data. If the `regexp_pattern` does not match the data, then the `NULL` value is returned.

`<regexp_replace_string>`

Specifies the replacement string value.

`<regexp_position>`

Specifies the position of a character where search must begin. By default, the function parameter is `RE_BEGINNING`.

`<regexp_occurence>`

Specifies the replacement occurrence of a substring. If the constant is `RE_ALL`, then the replacement of each matching substring occurs. If the constant is `RE_FIRST`, then the replacement of the first matching substring occurs.

`<regexp_match_parameter>`

Changes the default matching behavior of a function. The possible regexp_match_parameter constants can be `'RE_CASE_SENSITIVE'`, `'RE_CASE_INSENSITIVE'`, `'RE_MULTIPLE_LINES'`, `'RE_NEWLINE_WILDCARD'`, `'RE_IGNORE_WHITESPACE'`.

Note

For more information on `constants, function_parameters`, or `regexp` (regular expressions) see, `Using DBMS_REDACT Constants and Function Parameters`.

`<policy_description>`

Specify the description of a redaction policy.

`<column_description>`

Description of the column to be redacted. The `column_description` is not supported, but if you specify the description for a column then, you will get a warning message.

`<custom_function_expression>`

The `custom_function_expression` is applicable only for the `CUSTOM` redaction type. The `custom_function_expression` is a function expression that is, schema-qualified function with a parameter such as `schema_name.function_name (argument1`, …)that allows a user to use their redaction logic to redact the column data.

Example

The following example illustrates to alter a policy partial redaction for values in the `payment_details_tab` table `card_string` (usually a credit card number) column.

```
\c edb base _user
```

```
BEGIN
  DBMS_REDACT.alter_policy (
    object_schema              => 'public',
    object_name                => 'payment_details_tab',
    policy_name                => 'redactPolicy_001',
    action                     => DBMS_REDACT.ADD_COLUMN,
    column_name                => 'card_string',
    function_type              => DBMS_REDACT.partial,
    function_parameters        => DBMS_REDACT.REDACT_CCN16_F12);
END;
```

Redacted Result:

```
edb=# \c - redact_user
You are now connected to database "edb" as user "redact_user".
edb=> SELECT * FROM payment_details_tab;
 customer_id |     card_string
-------------+---------------------
           0 | ****-****-****-1234
           0 | ****-****-****-2345
(2 rows)
```

DISABLE_POLICY

The `disable_policy` procedure disables an existing data redaction policy.

```
PROCEDURE disable_policy (
     <object_schema>     IN VARCHAR2 DEFAULT NULL,
     <object_name>       IN VARCHAR2,
     <policy_name>       IN VARCHAR2
     )
```

Parameters

`<object_schema>`

Specifies the name of the schema in which the object resides and on which the data redaction policy will be applied. If you specify `NULL` then the given object is searched by the order specified by `search_path` setting.

`<object_name>`

Name of the table for which to disable a data redaction policy.

`<policy_name>`

Name of the policy to be disabled.

Example

The following example illustrates how to disable a policy.

```
\c edb base_user

BEGIN
  DBMS_REDACT.disable_policy(
    object_schema => 'public',
    object_name => 'payment_details_tab',
    policy_name => 'redactPolicy_001');
END;
```

Redacted Result: Data is no longer redacted after disabling a policy.

DBMS_REDACT_ENABLE_POLICY

ENABLE_POLICY

The `enable_policy` procedure enables the previously disabled data redaction policy.

```
PROCEDURE enable_policy (
```

```
    <object_schema> IN VARCHAR2 DEFAULT NULL,

    <object_name> IN VARCHAR2,

    <policy_name> IN VARCHAR2
```

)

Parameters

`<object_schema>`

> Specifies the name of the schema in which the object resides and on which the data redaction policy will be applied. If you specify `NULL` then the given object is searched by the order specified by `search_path` setting.

`<object_name>`

> Name of the table to which to enable a data redaction policy.

`<policy_name>`

> Name of the policy to be enabled.

Example

The following example illustrates how to enable a policy.

```
\c edb base_user
```

```
BEGIN
  DBMS_REDACT.enable_policy(
    object_schema => 'public',
    object_name => 'payment_details_tab',
    policy_name => 'redactPolicy_001');
END;
```

Redacted Result: Data is redacted after enabling a policy.

DBMS_REDACT_DROP_POLICY

DROP_POLICY

The `drop_policy` procedure drops a data redaction policy by removing the masking policy from a table.

`PROCEDURE drop_policy` (

> `<object_schema IN VARCHAR2 DEFAULT NULL`,
>
> `<object_name IN VARCHAR2`,
>
> `<policy_name IN VARCHAR2`
>
>)

Parameters

`<object_schema>`

> Specifies the name of the schema in which the object resides and on which the data redaction policy will be applied. If you specify `NULL` then the given object is searched by the order specified by `search_path` setting.

`<object_name>`

> Name of the table from which to drop a data redaction policy.

`<policy_name>`

> Name of the policy to be dropped.

Example

The following example illustrates how to drop a policy.

```
\c edb base_user

BEGIN
  DBMS_REDACT.drop_policy(
    object_schema => 'public',
    object_name => 'payment_details_tab',
    policy_name => 'redactPolicy_001');
```

Redacted Result: The server drops the specified policy.

UPDATE_FULL_REDACTION_VALUES

The `update_full_redaction_values` procedure updates the default displayed values for a data redaction policy and these default values can be viewed using the `redaction_values_for_type_full` view that use the full redaction type.

```
PROCEDURE update_full_redaction_values (
number_val      IN NUMBER       DEFAULT NULL,
binfloat_val    IN FLOAT4       DEFAULT NULL,
bindouble_val   IN FLOAT8       DEFAULT NULL,
char_val        IN CHAR             DEFAULT NULL,
varchar_val         IN VARCHAR2         DEFAULT NULL,
nchar_val       IN NCHAR        DEFAULT NULL,
nvarchar_val    IN NVARCHAR2            DEFAULT NULL,
datecol_val     IN DATE             DEFAULT NULL,
ts_val      IN TIMESTAMP            DEFAULT NULL,
tswtz_val       IN TIMESTAMPTZ          DEFAULT NULL,
blob_val        IN BLOB             DEFAULT NULL,
clob_val        IN CLOB             DEFAULT NULL,
nclob_val       IN CLOB             DEFAULT NULL
)
```

Parameters

`<number_val>`

Updates the default value for columns of the `NUMBER` datatype.

`<binfloat_val>`

The `FLOAT4` datatype is a random value. The binary float datatype is not supported.

`<bindouble_val>`

The `FLOAT8` datatype is a random value. The binary double datatype is not supported.

`<char_val>`

Updates the default value for columns of the `CHAR` datatype.

`<varchar_val>`

Updates the default value for columns of the `VARCHAR2` datatype.

`<nchar_val>`

The `nchar_val` is mapped to `CHAR` datatype and returns the `CHAR` value.

`<nvarchar_val>`

> The `nvarchar_val` is mapped to `VARCHAR2` datatype and returns the `VARCHAR` value.

`<datecol_val>`

> Updates the default value for columns of the `DATE` datatype.

`<ts_val>`

> Updates the default value for columns of the `TIMESTAMP` datatype.

`<tswtz_val>`

> Updates the default value for columns of the `TIMESTAMPTZ` datatype.

`<blob_val>`

> Updates the default value for columns of the `BLOB` datatype.

`<clob_val>`

> Updates the default value for columns of the `CLOB` datatype.

`<nclob_val>`

> The `nclob_val` is mapped to `CLOB` datatype and returns the `CLOB` value.

Example

The following example illustrates how to update the full redaction values but before updating the values, you can:

1. View the default values using `redaction_values_for_type_full` view as shown below:

```
edb=# \x
Expanded display is on.
edb=# SELECT number_value, char_value, varchar_value, date_value,
       timestamp_value, timestamp_with_time_zone_value, blob_value, clob_value
FROM redaction_values_for_type_full;
-[ RECORD 1 ]------------------+--------------------------
number_value                   | 0
char_value                     |
varchar_value                  |
date_value                     | 01-JAN-01 00:00:00
timestamp_value                | 01-JAN-01 01:00:00
timestamp_with_time_zone_value | 31-DEC-00 20:00:00 -05:00
blob_value                     | \x5b72656461637465645d
clob_value                     | [redacted]
(1 row)
```

2. Now, update the default values for full redaction type. The NULL values will be ignored. c edb base_user

```
edb=# BEGIN
    DBMS_REDACT.update_full_redaction_values (
    number_val => 9999999,
    char_val => 'Z',
    varchar_val => 'V',
    datecol_val => to_date('17/10/2018', 'DD/MM/YYYY'),
    ts_val => to_timestamp('17/10/2018 11:12:13', 'DD/MM/YYYY HH24:MI:SS'),
    tswtz_val => NULL,
    blob_val => 'NEW REDACTED VALUE',
    clob_val => 'NEW REDACTED VALUE');
END;
```

3. You can now see the updated values using redaction_values_for_type_full view.

```
EDB-SPL Procedure successfully completed
edb=# SELECT number_value, char_value, varchar_value, date_value,
       timestamp_value, timestamp_with_time_zone_value, blob_value, clob_value
FROM redaction_values_for_type_full;
-[ RECORD 1 ]------------------+--------------------------------------
number_value                   | 9999999
char_value                     | Z
varchar_value                  | V
date_value                     | 17-OCT-18 00:00:00
timestamp_value                | 17-OCT-18 11:12:13
timestamp_with_time_zone_value | 31-DEC-00 20:00:00 -05:00
blob_value                     | \x4e45572052454441435445442056414c5545
clob_value                     | NEW REDACTED VALUE
(1 row)
```

Redacted Result:

```
edb=# \c edb redact_user
You are now connected to database "edb" as user "redact_user".
edb=> select * from payment_details_tab order by 1;
 customer_id | card_string
-------------+-------------
     9999999 | V
     9999999 | V
(2 rows)
```

4.15 DBMS_RLS

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

Function/Procedure
ADD_POLICY(<object_schema>, <object_name>, <policy_name>, <function_schema>, <policy_functi
DROP_POLICY(<object_schema>, <object_name>, <policy_name>)
ENABLE_POLICY(<object_schema>, <object_name>, <policy_name>, <enable>)

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:

> https://www.enterprisedb.com/edb-docs/

The `SYS_CONTEXT` function is often used with `DBMS_RLS`. The signature is:

```
SYS_CONTEXT(<namespace>, <attribute>)
```

Where:

`<namespace>` is a `VARCHAR2`; the only accepted value is `USERENV`. Any other value will return `NULL`.

`<attribute>` is a `VARCHAR2`. `<attribute>` may be:

attribute Value	Equivalent Value
SESSION_USER	pg_catalog.session_user
CURRENT_USER	pg_catalog.current_user
CURRENT_SCHEMA	pg_catalog.current_schema
HOST	pg_catalog.inet_host
IP_ADDRESS	pg_catalog.inet_client_addr
SERVER_HOST	pg_catalog.inet_server_addr

Note

The examples used to illustrate the `DBMS_RLS` package are based on a modified copy of the sample `emp` table provided with Advanced Server along with a role named `salesmgr` that is granted all privileges on the table. You can create the modified copy of the `emp` table named `vpemp` and the `salesmgr` role as shown by the following:

```
CREATE TABLE public.vpemp AS SELECT empno, ename, job, sal, comm, deptno FROM emp;
ALTER TABLE vpemp ADD authid VARCHAR2(12);
UPDATE vpemp SET authid = 'researchmgr' WHERE deptno = 20;
UPDATE vpemp SET authid = 'salesmgr' WHERE deptno = 30;
SELECT * FROM vpemp;

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;
```

DBMS_RLSADD_POLICY:

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)
```

DBMS_RLS_DROP_POLICY

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;
```

DBMS_RLS_ENABLE_POLICY

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;
```

4.16.1 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
CREATE_JOB(<job_name>, <job_type>, <job_action>, <number_of_arguments>, <start_date>, <re
CREATE_JOB(<job_name>, <program_name>, <schedule_name>, <job_class>, <enabled>, <auto_dr
CREATE_PROGRAM(<program_name>, <program_type>, <program_action>, <number_of_arguments
CREATE_SCHEDULE(<schedule_name>, <start_date>, <repeat_interval>, <end_date>, <comments>
DEFINE_PROGRAM_ARGUMENT(<program_name>, <argument_position>, <argument_name>, <ar
DEFINE_PROGRAM_ARGUMENT(<program_name>, <argument_position>, <argument_name>, <ar
DISABLE(<name>, <force>, <commit_semantics>)
DROP_JOB(<job_name>, <force>, <defer>, <commit_semantics>)
DROP_PROGRAM(<program_name>, <force>)
DROP_PROGRAM_ARGUMENT(<program_name>, <argument_position>)
DROP_PROGRAM_ARGUMENT(<program_name>, <argument_name>)
DROP_SCHEDULE(<schedule_name>, <force>)
ENABLE(<name>, <commit_semantics>)
EVALUATE_CALENDAR_STRING(<calendar_string>, <start_date>, <return_date_after>, <next_ru
RUN_JOB(<job_name>, <use_current_session>, <manually>)
SET_JOB_ARGUMENT_VALUE(<job_name>, <argument_position>, <argument_value>)

SET_JOB_ARGUMENT_VALUE(<job_name>, <argument_name>, <argument_value>)

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 create_job create_program create_schedule define_program_argument dbms_scheduler_disable drop_job drop_program drop_program_argument drop_schedule dbms_scheduler_enable evaluate_calendar_string run_job set_job_argument_value

---

4.16.2 '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:

4.16.3 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_ame>`

> `<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.

4.16.4 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.

4.16.5 CREATE_SCHEDULE

Use the `CREATE_SCHEDULE` procedure to create a job schedule. The signature of the `CREATE_SCHEDULE` procedure is:

```
create_schedule(
<schedule_name> IN VARCHAR2,
<start_date> IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
<repeat_interval> IN VARCHAR2,
<end_date> IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
<comments> IN VARCHAR2 DEFAULT NULL)
```

Parameters

`<schedule_name>`

`<schedule_name>` specifies the name of the schedule.

`<start_date>`

`<start_date>` is a `TIMESTAMP WITH TIME ZONE` value that specifies the date and time that the schedule is eligible to execute. If a `<start_date>` is not specified, the date that the job is enabled is used as the `<start_date>`. By default, `<start_date>` is `NULL`.

`<repeat_interval>`

`<repeat_interval>` is a `VARCHAR2` value that specifies how often the job will repeat. If a `<repeat_interval>` is not specified, the job will execute only once, on the date specified by `<start_date>`.

Note

You must provide a value for either `<start_date>` or `<repeat_interval>`; if both `<start_date>` and `<repeat_interval>` are `NULL`, the server will return an error.

`<end_date> 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`.

4.16.6 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.

4.16.7 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');
```

4.16.8 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');
```

4.16.9 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');
```

4.16.10 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');
```

4.16.11 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.

4.16.12 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');
```

4.16.13 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> 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.

4.16.14 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.

4.16.15 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');
```

4.17 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 s

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');
```

4.18.1 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	Functi
BIND_VARIABLE(c, name, value [, out_value_size])	Proced
BIND_VARIABLE_CHAR(c, name, value [, out_value_size])	Proced
BIND_VARIABLE_RAW(c, name, value [, out_value_size])	Proced
CLOSE_CURSOR(c IN OUT)	Proced
COLUMN_VALUE(c, position, value OUT [, column_error OUT [, actual_length OUT]])	Proced
COLUMN_VALUE_CHAR(c, position, value OUT [, column_error OUT [, actual_length OUT]])	Proced
COLUMN_VALUE_RAW(c, position, value OUT [, column_error OUT [, actual_length OUT]])	Proced
DEFINE_COLUMN(c, position, column [, column_size])	Proced
DEFINE_COLUMN_CHAR(c, position, column, column_size)	Proced
DEFINE_COLUMN_RAW(c, position, column, column_size)	Proced
DESCRIBE_COLUMNS	Proced
EXECUTE(c)	Functi
EXECUTE_AND_FETCH(c [, exact])	Functi
FETCH_ROWS(c)	Functi
IS_OPEN(c)	Functi
LAST_ROW_COUNT	Functi
OPEN_CURSOR	Functi
PARSE(c, statement, language_flag)	Proced

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.PA
V6	INTEGER	2	Provided for compatibility with Oracle syntax. See DBMS_SQL.PA
V7	INTEGER	3	Provided for compatibility with Oracle syntax. See DBMS_SQL.PA

bind_variable bind_variable_char bind_variable_raw close_cursor column_value column_value_char column_value_raw define_column define_column_char define_column_raw describe_columns execute execute_and_fetch fetch_rows is_open last_row_count open_cursor parse

---

4.18.2 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
```

4.18.3 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.

4.18.4 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.

4.18.5 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;
```

4.18.6 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;
```

4.18.7 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.

4.18.8 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.

4.18.9 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;
```

4.18.10 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.

4.18.11 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.

4.18.12 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

4.18.13 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;
```

4.18.14 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
```

4.18.15 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
```

4.18.16 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.

4.18.17 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
```

4.18.18 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;
```

4.18.19 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;
```

4.19 DBMS_UTILITY

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

Function/Procedure
ANALYZE_DATABASE(method [, estimate_rows [, estimate_percent [, method_opt]]])
ANALYZE_PART_OBJECT(schema, object_name [, object_type [, command_type [, command_opt [, sam
ANALYZE_SCHEMA(schema, method [, estimate_rows [, estimate_percent [, method_opt]]])
CANONICALIZE(name, canon_name OUT, canon_len)
COMMA_TO_TABLE(list, tablen OUT, tab OUT)
DB_VERSION(version OUT, compatibility OUT)
EXEC_DDL_STATEMENT(parse_string)
FORMAT_CALL_STACK
GET_CPU_TIME
GET_DEPENDENCY(type, schema, name)
GET_HASH_VALUE(name, base, hash_size)
GET_PARAMETER_VALUE(parnam, intval OUT, strval OUT)

GET_TIME
NAME_TOKENIZE(name, a OUT, b OUT, c OUT, dblink OUT, nextpos OUT)
TABLE_TO_COMMA(tab, tablen OUT, list OUT)

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 the LNAME_ARRAY <lname_array> section).

`<UNCL_ARRAY>`

A `DBMS_UTILITY UNCL_ARRAY` (as described in the UNCL_ARRAY <uncl_array> section).

Examples

The following procedure uses the `COMMA_TO_TABLE` procedure to convert a list of names to a table. The table entries are then displayed.

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

EXEC comma_to_table('edb.dept, edb.emp, edb.jobhist')

edb.dept
edb.emp
edb.jobhist
```

DB_VERSION

The `DB_VERSION` procedure returns the version number of the database.

```
DB_VERSION(<version> OUT VARCHAR2, <compatibility> OUT
VARCHAR2)
```

Parameters

`<version>`

Database version number.

`<compatibility>`

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

Examples

The following anonymous block displays the database version information.

```
DECLARE
    v_version       VARCHAR2(150);
    v_compat        VARCHAR2(150);
BEGIN
    DBMS_UTILITY.DB_VERSION(v_version,v_compat);
    DBMS_OUTPUT.PUT_LINE('Version: '       || v_version);
    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
Compatibility: EnterpriseDB 10.0.0 on i686-pc-linux-gnu, compiled by GCC gcc (GCC) 4.1.22008
```

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.. code-block:: text) :=
            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.. code-block:: text
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
:
a
:
b
:
c
:
dblink :
nextpos:
edb.list_emp
EDB
LIST_EMP
12
```

Tokenize the name, `"edb"."Emp_Admin".update_emp_sal :`

```
BEGIN
name_tokenize('"edb"."Emp_Admin".update_emp_sal');
END;
name
:
a
:
b
:
c
:
dblink :
nextpos:
"edb"."Emp_Admin".update_emp_sal
edb
Emp_Admin
UPDATE_EMP_SAL
32
```

Tokenize the name `edb.emp@edb_dblink` :

```
BEGIN
```

```
name_tokenize('edb.emp@edb_dblink');
END;
name
:
a
```

```
:
b
:
c
:
dblink :
nextpos:
edb.emp@edb_dblink
EDB
EMP
EDB_DBLINK
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 the LNAME ARRAY section.

`UNCL_ARRAY`

A `DBMS_UTILITY UNCL_ARRAY` (as described the UNCL_ARRAY <uncl_array> section).

`<tablen>`

Number of entries in `<list>`.

`<list>`

Comma-delimited list of names from `<tab>`.

Examples

The following example first uses the `COMMA_TO_TABLE` procedure to convert a comma-delimited list to a table. The `TABLE_TO_COMMA` procedure then converts the table back to a comma-delimited list that is displayed.

```
CREATE OR REPLACE PROCEDURE table_to_comma (
    p_list      VARCHAR2
)
IS
    r_lname     DBMS_UTILITY.LNAME_ARRAY;
    v_length    BINARY_INTEGER;
    v_listlen   BINARY_INTEGER;
    v_list      VARCHAR2(80);
BEGIN
    DBMS_UTILITY.COMMA_TO_TABLE(p_list,v_length,r_lname);
    DBMS_OUTPUT.PUT_LINE('Table Entries');
    DBMS_OUTPUT.PUT_LINE('-------------');
    FOR i IN 1..v_length LOOP
        DBMS_OUTPUT.PUT_LINE(r_lname(i));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('-------------');
    DBMS_UTILITY.TABLE_TO_COMMA(r_lname,v_listlen,v_list);
    DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END;

EXEC table_to_comma('edb.dept, edb.emp, edb.jobhist')

Table Entries
-------------
edb.dept
edb.emp
edb.jobhist
-------------
Comma-Delimited List: edb.dept, edb.emp, edb.jobhist
```

4.20.1 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 fu
BASE64_ENCODE(r)	RAW	Use the BASE64_ENCODE fu
BASE64_ENCODE(loid)	TEXT	Use the BASE64_ENCODE fu
MIMEHEADER_DECODE(buf)	VARCHAR2	Use the MIMEHEADER_DEC
MIMEHEADER_ENCODE(buf, encode_charset, encoding)	VARCHAR2	Use the MIMEHEADER_ENC
QUOTED_PRINTABLE_DECODE(r)	RAW	Use the QUOTED_PRINTAB
QUOTED_PRINTABLE_ENCODE(r)	RAW	Use the QUOTED_PRINTAB
TEXT_DECODE(buf, encode_charset, encoding)	VARCHAR2	Use the TEXT_DECODE func

TEXT_ENCODE(buf, encode_charset, encoding)	VARCHAR2	Use the TEXT_ENCODE fun
UUDECODE(r)	RAW	Use the UUDECODE function
UUENCODE(r, type, filename, permission)	RAW	Use the UUENCODE function

base64_decode base64_encode mimeheader_decode mimeheader_encode quoted_printable_decode quoted_printable_encode text_decode text_encode uudecode uuencode

---

4.20.2 BASE64_DECODE

Use the `BASE64_DECODE` function to translate a Base64 encoded string to the original value originally encoded by `BASE64_ENCODE`. The signature is:

```
BASE64_DECODE (<r> IN RAW)
```

This function returns a `RAW` value.

Parameters

`<r>`

> `<r>` is the string that contains the Base64 encoded data that will be translated to `RAW` form.

Examples

Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

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

https://www.postgresql.org/docs/12/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)
```

4.20.3 BASE64_ENCODE

Use the `BASE64_ENCODE` function to translate and encode a string in Base64 format (as described in RFC 4648). This function can be useful when composing `MIME` email that you intend to send using the `UTL_SMTP` package. The `BASE64_ENCODE` function has two signatures:

```
BASE64_ENCODE(<r> IN RAW)
```

and

```
BASE64_ENCODE(<loid> IN OID)
```

This function returns a `RAW` value or an `OID`.

Parameters

`<r>`

`<r>` specifies the `RAW` string that will be translated to Base64.

`<loid>`

`<loid>` specifies the object ID of a large object that will be translated to Base64.

Examples

Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

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

https://www.postgresql.org/docs/12/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)
```

4.20.4 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)
```

4.20.5 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)
```

4.20.6 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

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:

https://www.postgresql.org/docs/12/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)
```

4.20.7 QUOTED_PRINTABLE_ENCODE

Use the QUOTED_PRINTABLE_ENCODE function to translate and encode a string in quoted-printable format. The signature is:

```
QUOTED_PRINTABLE_ENCODE(<r> IN RAW)
```

This function returns a RAW value.

Parameters

<r>

    <r> contains the string (a RAW value) that will be encoded in a quoted-printable format.

Examples

Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

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

https://www.postgresql.org/docs/12/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)
```

4.20.8 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)
```

4.20.9 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)
```

4.20.10 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.

If you are using the Advanced Server UUDECODE function to decode uuencoded data that was created by the Oracle implementation of the UTL_ENCODE.UUENCODE function, then you must first set the Advanced Server configuration parameter utl_encode.uudecode_redwood to TRUE before invoking the Advanced Server UUDECODE function on the Oracle-created data. (For example, this situation may occur if you migrated Oracle tables containing uuencoded data to an Advanced Server database.)

The uuencoded data created by the Oracle version of the UUENCODE function results in a format that differs from the uuencoded data created by the Advanced Server UUENCODE function. As a result, attempting to use the Advanced Server UUDECODE function on the Oracle uuencoded data results in an error unless the configuration parameter utl_encode.uudecode_redwood is set to TRUE.

However, if you are using the Advanced Server UUDECODE function on uuencoded data created by the Advanced Server UUENCODE function, then utl_encode.uudecode_redwood must be set to FALSE, which is the default setting.

Parameters

`<r>`

`<r>` contains the uuencoded string that will be translated to `RAW`.

Examples

Before executing the following example, invoke the command:

```
SET bytea_output = escape;
```

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

https://www.postgresql.org/docs/12/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)
```

4.20.11 UUENCODE

Use the `UUENCODE` function to translate `RAW` data into a uuencode formatted encoded string. The signature is:

```
UUENCODE(<r> IN RAW, <type> IN INTEGER DEFAULT 1,
<filename> IN     VARCHAR2 DEFAULT NULL, <permission> IN
VARCHAR2 DEFAULT NULL)
```

This function returns a `RAW` value.

Parameters

`<r>`

`<r>` contains the RAW string that will be translated to uuencode format.

<type>

<type> is an INTEGER value or constant that specifies the type of uuencoded string that will be returned; the default value is 1. The possible values are:

Value	Constant
1	complete
2	header_piece
3	middle_piece
4	end_piece

<filename>

<filename> is a VARCHAR2 value that specifies the file name that you want to embed in the encoded form; if you do not specify a file name, UUENCODE will include a filename of uuencode.txt in the encoded form.

<permission>

<permission> is a VARCHAR2 that specifies the permission mode; the default value is NULL.

Examples

Before executing the following example, invoke the command:

SET bytea_output = escape;

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

https://www.postgresql.org/docs/12/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)
```

4.21 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 speci
FCLOSE_ALL	n/a	Closes all open
FCOPY(location, filename, dest_dir, dest_file [, start_line [, end_line]])	n/a	Copies filename
FFLUSH(file)	n/a	Forces data in t
FOPEN(location, filename, open_mode [, max_linesize])	FILE_TYPE	Opens file, filena
FREMOVE(location, filename)	n/a	Removes the sp
FRENAME(location, filename, dest_dir, dest_file [, overwrite])	n/a	Renames the sp
GET_LINE(file, buffer OUT)	n/a	Reads a line of
IS_OPEN(file)	BOOLEAN	Determines whe
NEW_LINE(file [, lines])	n/a	Writes an end-o
PUT(file, buffer)	n/a	Writes buffer to
PUT_LINE(file, buffer)	n/a	Writes buffer to
PUTF(file, format [, arg1] [, ...])	n/a	Writes a 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
```
UTL_FILE_PUT

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
```

UTL_FILE_PUT_LINE

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
```

4.22 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	Descripti
BEGIN_REQUEST(url, method, http_version)	UTL_HTTP.REQ	Initiates
END_REQUEST(r IN OUT)	n/a	Ends an
END_RESPONSE(r IN OUT)	n/a	Ends the
GET_BODY_CHARSET	VARCHAR2	Returns
GET_BODY_CHARSET(charset OUT)	n/a	Returns
GET_FOLLOW_REDIRECT(max_redirects OUT)	n/a	Current
GET_HEADER(r IN OUT, n, name OUT, value OUT)	n/a	Returns
GET_HEADER_BY_NAME(r IN OUT, name, value OUT, n)	n/a	Returns
GET_HEADER_COUNT(r IN OUT)	INTEGER	Returns
GET_RESPONSE(r IN OUT)	UTL_HTTP.RESP	Returns
GET_RESPONSE_ERROR_CHECK(enable OUT)	n/a	Returns
GET_TRANSFER_TIMEOUT(timeout OUT)	n/a	Returns
(r IN OUT, data OUT, remove_crlf)	n/a	Returns
READ_RAW(r IN OUT, data OUT, len)	n/a	Returns
READ_TEXT(r IN OUT, data OUT, len)	n/a	Returns
REQUEST(url)	VARCHAR2	Returns
REQUEST_PIECES(url, max_pieces)	UTL_HTTP. HTML_PIECES	Returns
SET_BODY_CHARSET(charset)	n/a	Sets the

SET_FOLLOW_REDIRECT(max_redirects)	n/a	Sets the
SET_FOLLOW_REDIRECT(r IN OUT, max_redirects)	n/a	Sets the
SET_HEADER(r IN OUT, name, value)	n/a	Sets the
SET_RESPONSE_ERROR_CHECK(enable)	n/a	Determin
SET_TRANSFER_TIMEOUT(timeout)	n/a	Sets the
SET_TRANSFER_TIMEOUT(r IN OUT, timeout)	n/a	Sets the
WRITE_LINE(r IN OUT, data)	n/a	Writes C
WRITE_RAW(r IN OUT, data)	n/a	Writes d
WRITE_TEXT(r IN OUT, data)	n/a	Writes d

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

Note

In Advanced Server, an `HTTP 4xx` or `HTTP 5xx` response produces a database error; in Oracle, this is configurable but `FALSE` by default.

In Advanced Server, the `UTL_HTTP` text interfaces expect the downloaded data to be in the database encoding. All currently-available interfaces are text interfaces. In Oracle, the encoding is detected from HTTP headers; in the absence of the header, the default is configurable and defaults to `ISO-8859-1`.

Advanced Server ignores all cookies it receives.

The `UTL_HTTP` exceptions that can be raised in Oracle are not recognized by Advanced Server. In addition, the error codes returned by Advanced Server are not the same as those returned by Oracle.

There are various public constants available with `UTL_HTTP`. These are listed in the following tables.

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

HTTP VERSIONS	
HTTP_VERSION_1_0	CONSTANT VARCHAR2(64) := 'HTTP/1.0';
HTTP_VERSION_1_1	CONSTANT VARCHAR2(64) := 'HTTP/1.1';
STANDARD PORT ASSIGNMENTS	
DEFAULT_HTTP_PORT	CONSTANT INTEGER := 80;
DEFAULT_HTTPS_PORT	CONSTANT INTEGER := 443;

The following table contains `UTL_HTTP` public status code constants.

1XX INFORMATIONAL	
HTTP_CONTINUE	CONSTANT INTEGER := 100;
HTTP_SWITCHING_PROTOCOLS	CONSTANT INTEGER := 101;
HTTP_PROCESSING	CONSTANT INTEGER := 102;
2XX SUCCESS	
HTTP_OK	CONSTANT INTEGER := 200;
HTTP_CREATED	CONSTANT INTEGER := 201;
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;

4XX CLIENT ERROR	
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 `nth` header of the HTTP response. The signature is:

```
GET_HEADER(<r> IN OUT UTL_HTTP.RESP, <n> INTEGER, <name>
OUT
VARCHAR2, <value> OUT VARCHAR2)
```

Parameters

`<r>`

`<r>` is the HTTP response record.

`<n>`

`<n>` is the `nth` header of the HTTP response record to retrieve.

`<name>`

`<name>` is the name of the response header.

`<value>`

<value> is the value of the response header.

Examples

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

```
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, 2
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 `nth` header of the HTTP response record to retrieve according to the values specified by `<name>`. The default is 1.

Examples

The following example retrieves the header for Content-Type.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_name          VARCHAR2(30) := 'Content-Type';
    v_value         VARCHAR2(200);
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    UTL_HTTP.GET_HEADER_BY_NAME(v_resp, v_name, v_value);
    DBMS_OUTPUT.PUT_LINE(v_name || ': ' || v_value);
    UTL_HTTP.END_RESPONSE(v_resp);
END;

Content-Type: text/html; charset=utf-8
```

GET_HEADER_COUNT

The `GET_HEADER_COUNT` function returns the number of HTTP response headers. The signature is:

```
GET_HEADER_COUNT(<r> IN OUT UTL_HTTP.RESP) RETURN
INTEGER
```

This function returns an `INTEGER` value.

Parameters

`<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,
<meta name="description" content="The leader in open source database products, services, sup
<meta name="abstract" content="The Enterprise PostgreSQL Company" />
<link rel="EditURI" type="application/rsd+xml" title="RSD" href="http://www.enterprisedb.com
<link rel="alternate" type="application/rss+xml" title="EnterpriseDB RSS" href="http://www.e
<link rel="shortcut icon" href="/sites/all/themes/edb_pixelcrayons/favicon.ico" type="image/
    <link type="text/css" rel="stylesheet" media="all" href="/sites/default/files/css/css_db
<!--[if IE 6]>
<link type="text/css" rel="stylesheet" media="all" href="/sites/all/themes/oho_basic/css/ie6
<![endif]-->
<!--[if IE 7]>
<link type="text/css" rel="stylesheet" media="all" href="/sites/all/themes/oho_basic/css/ie7
<![endif]-->
    <script type="text/javascript" src="/sites/default/files/js/js_74d97b1176812e2fd6e43d625
<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.

```
\x3c21444f43545950452068746d6c205055424c494320222d2f2f5733432f2f4454442058485
44d4c20312e30205374726963742f2f454e220d0a202022687474703a2f2f7777772e77332e6f
72672f54522f7868746d6c312f4454442f7868746d6c312d7374726963742e647464223e0d0a3
c68746d6c20786d6c6e733d22687474703a2f2f7777772e77332e6f72672f313939392f
```

READ_TEXT

The `READ_TEXT` procedure returns the data from the HTTP response body in text form. The maximum number of characters returned is specified by the `<len>` parameter. The signature is:

```
READ_TEXT(<r> IN OUT UTL_HTTP.RESP, <data> OUT VARCHAR2,
<len> INTEGER)
```

Parameters

`<r>`

`<r>` is the HTTP response record.

`<data>`

`<data>` is the response body in text form.

`<len>`

Set `<len>` to the maximum number of characters to be returned.

Examples

The following example retrieves the first 150 characters.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_data          VARCHAR2(150);
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    UTL_HTTP.READ_TEXT(v_resp, v_data, 150);
    DBMS_OUTPUT.PUT_LINE(v_data);
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

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
```

4.23 UTL_MAIL

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

Function/Procedure
SEND(<sender>, <recipients>, <cc>, <bcc>, <subject>, <message> [, <mime_type> [, <priority>]])
SEND_ATTACH_RAW(<sender>, <recipients>, <cc>, <bcc>, <subject>, <message>, <mime_type>, <
SEND_ATTACH_VARCHAR2(<sender>, <recipients>, <cc>, <bcc>, <subject>, <message>, <mime_ty

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. 21 at ' ||
             '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 ]]])
```

or

```
SEND_ATTACH_RAW(<sender> VARCHAR2, <recipients> VARCHAR2, <cc>
VARCHAR2, <bcc> VARCHAR2, <subject> VARCHAR2, <message> VARCHAR2,
<mime_type> VARCHAR2, <priority> PLS_INTEGER, <attachment> OID
[, <att_inline> BOOLEAN [, <att_mime_type> VARCHAR2  [, <att_filename>
VARCHAR2 ]]])
```

Parameters

`<sender>`

E-mail address of the sender.

`<recipients>`

Comma-separated e-mail addresses of the recipients.

`<cc>`

Comma-separated e-mail addresses of copy recipients.

`<bcc>`

Comma-separated e-mail addresses of blind copy recipients.

`<subject>`

Subject line of the e-mail.

`<message>`

Body of the e-mail.

`<mime_type>`

Mime type of the message. The default is `text/plain; charset=us-ascii`.

`<priority>`

Priority of the e-mail. The default is `3`.

`<attachment>`

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`.

4.24 UTL_RAW

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

Note

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

Function/Procedure	Function or Procedure
CAST_TO_RAW(c IN VARCHAR2)	Function
CAST_TO_VARCHAR2(r IN RAW)	Function
CONCAT(r1 IN RAW, r2 IN RAW, r3 IN RAW,…)	Function
CONVERT(r IN RAW, to_charset IN VARCHAR2, from_charset IN VARCHAR2	Function
LENGTH(r IN RAW)	Function
SUBSTR(r IN RAW, pos IN INTEGER, len IN INTEGER)	Function

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('hello'));
length
--------
```

```
     5
(1 row)
```

UTL_RAW_SUBSTR

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)
```

4.25 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	Descripti
CLOSE_DATA(c IN OUT)	Procedure	n/a	Ends an
COMMAND(c IN OUT, cmd [, arg])	Both	REPLY	Execute a
COMMAND_REPLIES(c IN OUT, cmd [, arg])	Function	REPLIES	Execute a
DATA(c IN OUT, body VARCHAR2)	Procedure	n/a	Specify th
EHLO(c IN OUT, domain)	Procedure	n/a	Perform i
HELO(c IN OUT, domain)	Procedure	n/a	Perform i
HELP(c IN OUT [, command])	Function	REPLIES	Send the
MAIL(c IN OUT, sender [, parameters])	Procedure	n/a	Start a m
NOOP(c IN OUT)	Both	REPLY	Send the
OPEN_CONNECTION(host [, port [, tx_timeout]])	Function	CONNECTION	Open a c
OPEN_DATA(c IN OUT)	Both	REPLY	Send the
QUIT(c IN OUT)	Procedure	n/a	Terminat
RCPT(c IN OUT, recipient [, parameters])	Procedure	n/a	Specify th
RSET(c IN OUT)	Procedure	n/a	Terminat
VRFY(c IN OUT, recipient)	Function	REPLY	Validate a
WRITE_DATA(c IN OUT, data)	Procedure	n/a	Write a p

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 Reply/Replies <reply/replies> for a description of `REPLY` and `REPLIES`.

COMMAND_REPLIES

The `COMMAND_REPLIES` function processes an SMTP command that returns multiple reply lines. Use `COMMAND` if only a single reply line is expected.

```
<replies> REPLIES COMMAND(<c> IN OUT CONNECTION, <cmd>
VARCHAR2

[, <arg> VARCHAR2 ])
```

Parameters

`<c>`

The SMTP connection to which the command is to be sent.

`<cmd>`

The SMTP command to be processed.

`<arg>`

An argument to the SMTP command. The default is null.

`<replies>`

SMTP reply lines to the command. See Reply/Replies <reply/replies> for a description of `REPLY` and `REPLIES`.

DATA

The `DATA` procedure provides the capability to specify the body of the e-mail message. The message is terminated with a `<CR><LF>.<CR><LF>` sequence.

```
DATA(<c> IN OUT CONNECTION, <body> VARCHAR2)
```

Parameters

`<c>`

The SMTP connection to which the command is to be sent.

`<body>`

Body of the e-mail message to be sent.

EHLO

The `EHLO` procedure performs initial handshaking with the SMTP server after establishing the connection. The `EHLO` procedure allows the client to identify itself to the SMTP server according to RFC 821. RFC 1869 specifies the format of the information returned in the server's reply. The *HELO* procedure performs the equivalent functionality, but returns less information about the server.

```
EHLO(<c> IN OUT CONNECTION, <domain> VARCHAR2)
```

Parameters

`<c>`

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

`<domain>`

Domain name of the sending host.

HELO

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

```
HELO(<c> IN OUT, <domain*> VARCHAR2)
```

Parameters

`<c>`

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

`<domain>`

Domain name of the sending host.

HELP

The `HELP` function provides the capability to send the `HELP` command to the SMTP server.

```
<replies> REPLIES HELP(<c> IN OUT CONNECTION [, <command>
VARCHAR2])
```

Parameters

`<c>`

The SMTP connection to which the command is to be sent.

`<command>`

Command on which help is requested.

`<replies>`

SMTP reply lines to the command. See Reply/Replies <reply/replies> for a description of `REPLY` and `REPLIES`.

MAIL

The `MAIL` procedure initiates a mail transaction.

```
MAIL(<c> IN OUT CONNECTION, <sender> VARCHAR2
[, <parameters> VARCHAR2 ])
```

Parameters

`<c>`

Connection to SMTP server on which to start a mail transaction.

`<sender>`

The sender's e-mail address.

`<parameters>`

Mail command parameters in the format, `key=value` as defined in RFC 1869.

NOOP

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

```
<reply> REPLY NOOP(<c> IN OUT CONNECTION)
NOOP(<c> IN OUT CONNECTION)
```

Parameters

`<c>`

The SMTP connection on which to send the command.

`<reply>`

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in `<reply>`. See Reply/Replies <reply/replies> for a description of `REPLY` and `REPLIES`.

OPEN_CONNECTION

The `OPEN_CONNECTION` functions open a connection to an SMTP server.

```
<c> CONNECTION OPEN_CONNECTION(<host> VARCHAR2 [, <port>
PLS_INTEGER [, <tx_timeout> PLS_INTEGER DEFAULT NULL]])
```

Parameters

`<host>`

Name of the SMTP server.

`<port>`

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

`<tx_timeout>`

Time out value in seconds. Do not wait is indicated by specifying 0. Wait indefinitely is indicated by setting timeout to null. The default is null.

`<c>`

Connection handle returned by the SMTP server.

OPEN_DATA

The `OPEN_DATA` procedure sends the `DATA` command to the SMTP server.

```
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 Reply/Replies <reply/replies> for a description of `REPLY` and `REPLIES`.

WRITE_DATA

The `WRITE_DATA` procedure provides the capability to add `VARCHAR2` data to an e-mail message. The `WRITE_DATA` procedure may be repetitively called to add data.

```
WRITE_DATA(<c> IN OUT CONNECTION, <data> VARCHAR2)
```

Parameters

`<c>`

The SMTP connection on which to add data.

`<data>`

Data to be added to the e-mail message. The data must conform to the RFC 822 specification.

UTL_SMTP_Comprehensive_example

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
```

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 yo
```

4.26 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 an
UNESCAPE(url, url_charset)	VARCHAR2	The UNESCAPE function to convert ar

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.
DBMS_OUTPUT.PUT_LINE(result);
END;
```

The resulting (unescaped) URL is:

```
http://www.example.com/Using the UNESCAPE function.htm
```

5 Acknowledgements

The PostgreSQL 8.3, 8.4, 9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 10, 11, and 12 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

6 Conclusion

Database Compatibility for Oracle Developers Built-in Packages Guide

EnterpriseDB Corporation 34 Crosby Drive, Suite 201, Bedford, MA 01730, USA

T +1 781 357 3390 F +1 978 467 1307 E info@enterprisedb.com
www.enterprisedb.com