Postgres Plus® Enterprise Edition Guide

Postgres Plus Enterprise Edition v 9.5

February 16, 2016

Postgres Plus Enterprise Edition Guide, Version 2.0
by EnterpriseDB Corporation
Copyright © 2014 - 2016 EnterpriseDB Corporation

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

Table of Contents

1 Introduction

This guide describes the features of EnterpriseDB's *Postgres Plus Enterprise Edition* product. The core of Postgres Plus Enterprise Edition is EnterpriseDB's database server, *Postgres Plus Advanced Server*.

Enterprise Edition provides a wide range of additional functionality in various areas including database administration, enhanced SQL capabilities, database and application security, performance monitoring and analysis, and application development utilities.

This guide is arranged as follows:

- **Database Administration.** Chapter 2 contains the features related to database administration.

  Configuration parameters described in Section 2.1 control the basic characteristics and performance of an Advanced Server instance.

  Audit logging described in Section 2.2 provides enhanced database auditing capabilities.

  Unicode Collation Algorithm described in Section 2.3 provides the capability to create a collation specific to your particular needs on a UTF-8 encoded database.

- **Enhanced SQL Features.** Chapter 3 contains the SQL enhancements provided for an Advanced Server database.

  Synonyms described in Section 3.1 provide for easy-to-use abbreviations for the fully qualified path names of tables and views.

  Hierarchical queries described in Section 3.2 provide for a logical display of tables related by foreign key constraints.

  Extended functions and operators described in Section 3.3 provides for additional functionality of SQL.

  Partitioned tables described in Section 3.4 provide for the implementation of table partitioning using the SQL `CREATE TABLE` statement.

- **Security.** Chapter 4 contains various security features.

  SQL/Protect described in Section 4.1 provides protection against SQL injection attacks.

*EDB*Wrap* described in Section 4.2 provides obfuscation of program source code to prevent unwanted scrutiny.

Virtual Private Database described in Section 4.3 provides fine-grained, row level access.

- **EDB Resource Manager.** Chapter 5 contains information on the EDB Resource Manager feature, which provides the capability to control system resource usage by Advanced Server processes.

  Resource Groups described in Section 5.1 shows how to create and maintain the groups on which resource limits can be defined and to which Advanced Server processes can be assigned.

  CPU Usage Throttling described in Section 5.2 provides a method to control CPU usage by Advanced Server processes.

  Dirty Buffer Throttling described in Section 5.3 provides a method to control the dirty rate of shared buffers by Advanced Server processes.

- **Database Utilities.** Chapter 6 contains database utility programs and interfaces.

  *EDB*Loader* described in Section 6.1 provides a quick and easy method for loading Advanced Server tables.

  *EDB*Plus* described in Section 6.2 is a command line utility program for running SQL statements.

  The *libpq C library* described in Section 6.3 is the C application programming interface (API) language for Advanced Server.

  ECPGPlus described in Section 6.4 is a C precompiler for Advanced Server.

- **Open Client Library.** Chapter 7 provides information about the Open Client Library, an application programming interface for Advanced Server.

  The PL Debugger described in Section 7.5 is a graphically oriented debugging tool for PL/pgSQL.

- **Performance Analysis and Tuning.** Chapter 8 contains the various tools for analyzing and improving application and database server performance.

  Dynatune described in Section 8.1 provides a quick and easy means for configuring Advanced Server depending upon the type of application usage.

  Infinite Cache described in Section 8.2 provides for performance improvement using memory caching. **Note:** Infinite Cache has been deprecated and may be removed in a future release. Please contact your EnterpriseDB Account Manager or mailto:sales@enterprisedb.com for more information.

Index Advisor described in Section 8.3 helps to determine the additional indexes needed on tables to improve application performance.

SQL Profiler described in Section 8.4 locates and diagnoses poorly running SQL queries in applications.

Query optimization hints described in Section 8.5 allows you to influence the manner in which the query planner processes SQL statements.

DBMS_PROFILER described in Section 8.6 is a built-in package that can be used to gather performance statistics for PL/pgSQL programs.

Dynamic Runtime Instrumentation Tools Architecture (DRITA) described in Section 8.7 provides the capability to capture and view statistics pertaining to wait events that affect system performance.

- **Built-In Utility Packages.** Chapter 9 contains an extensive set of *built-in packages* that provide functions to quicken and ease development of PL/pgSQL applications.

- **Expanded Catalog Views.** Chapter 10 contains additional *catalog views* added to Advanced Server to simplify the querying of database object information.

- **System Catalog Tables.** Chapter 11 contains additional *system catalog tables* added for Advanced Server specific database objects.

- **Appendix.** Chapter 12 contains various miscellaneous topics such as Advanced Server database limits and keywords.

1.1 What's New

The following features have been added to Postgres Plus Advanced Server 9.4 to create Postgres Plus Advanced Server 9.5:

- Advanced Server now provides support for Profile Management. For more information, see Section 2.4.
- Advanced Server now includes support for `DBMS_SESSION.SET_ROLE`. For more information, see Section 9.13.1.
- Advanced Server now includes support for the `UTL_RAW` package. For more information, see Section 9.20.
- Advanced Server now includes the `edb_audit_tag` parameter; the parameter can be used to add a tag to an audit log. For more information, see sections 2.1.3.7.10 and 2.2.1.
- Advanced Server supports the use of `EDBLDR_ENV_STYLE` to specify the style of environment variables recognized by EDB*Loader. For more information, see Section 6.1.3.
- EDB*Loader now accepts the `ZONED [(precision[,scale])]` field type specification. For more information, see Section 6.1.3.
- Advanced Server now supports the `UTL_HTTP.WRITE_LINE` and `UTL_HTTP.WRITE_TEXT` procedures. For more information, see sections 9.18.25 and 9.18.27, respectively.
- Advanced Server now supports the `DBA_PROFILES` view. For more information, see Section 10.36.
- Advanced Server now supports the `FREEZE` keyword in the EDB*Loader control file and on the command line. For more information, see sections 6.1.3 and 6.1.4, respectively.
- Advanced Server now supports XA functions `xaoEnv` and `xaoSvcCtx` in the Open Client Library. For more information, See Section 7.4.6.
- Advanced Server now supports the `EDB_ATTR_EMPTY_STRINGS` environment attribute in the Open Client Library. For more information, See Section 7.4.2.1.

1.2 Typographical Conventions Used in this Guide

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

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

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

1.3 Other Conventions Used in this Guide

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

Throughout this guide, the directory path of Postgres Plus Advanced Server is referred to as *POSTGRES_PLUS_HOME*.

For Linux installations, the default directory path is

```
/opt/PostgresPlus/version_no
```

For Windows installations, the default directory path is

```
C:\Program Files\PostgresPlus\version_no
```

The product version number is represented by *version_no*.

1.4 About the Examples Used in this Guide

The examples in this guide are shown in the type and background illustrated below.

```
Examples and output from examples are shown in fixed-width, blue font on a
light blue background.
```

The examples use the sample tables, `dept`, `emp`, and `jobhist`, created and loaded when Postgres Plus Advanced Server is installed.

The tables and programs in the sample database can be re-created at any time by executing the following script:

POSTGRES_PLUS_HOME`/installer/server/pg-sample.sql`.

The script:

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

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

```
SHOW SEARCH_PATH;
```

You can use PSQL commands to modify the search path.

1.4.1.1 Sample Database Description

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

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

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

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

The following is the `pg-sample.sql` script:

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

--
-- Script that creates the 'sample' tables, views
-- functions, triggers, etc.
--
-- Start new transaction - commit all or nothing
--
BEGIN;
--
-- Create and load tables used in the documentation examples.
--
-- Create the 'dept' table
--
CREATE TABLE dept (
    deptno          NUMERIC(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR(13)
);
--
-- Create the 'emp' table
--
CREATE TABLE emp (
    empno           NUMERIC(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR(10),
    job             VARCHAR(9),
    mgr             NUMERIC(4),
    hiredate        DATE,
    sal             NUMERIC(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm            NUMERIC(7,2),
    deptno          NUMERIC(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
--
-- Create the 'jobhist' table
--
CREATE TABLE jobhist (
    empno           NUMERIC(4) NOT NULL,
    startdate       TIMESTAMP(0) NOT NULL,
    enddate         TIMESTAMP(0),
    job             VARCHAR(9),
    sal             NUMERIC(7,2),
    comm            NUMERIC(7,2),
    deptno          NUMERIC(2),
    chgdesc         VARCHAR(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
    CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
        REFERENCES emp(empno) ON DELETE CASCADE,
    CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
        REFERENCES dept (deptno) ON DELETE SET NULL,
    CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);
--
-- Create the 'salesemp' view
--
CREATE OR REPLACE VIEW salesemp AS
```



```
    SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
--
--  Sequence to generate values for function 'new_empno'.
--
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
--
--  Issue PUBLIC grants
--
--GRANT ALL ON emp TO PUBLIC;
--GRANT ALL ON dept TO PUBLIC;
--GRANT ALL ON jobhist TO PUBLIC;
--GRANT ALL ON salesemp TO PUBLIC;
--GRANT ALL ON next_empno TO PUBLIC;
--
--  Load the 'dept' table
--
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
--
--  Load the 'emp' table
--
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-
81',1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-
81',2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-
81',1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-
81',2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-
81',2450,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-
87',3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-
81',5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
--
--  Load the 'jobhist' table
--
INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',800,NULL,20,'New
Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,'New
Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,'New
Hire');
INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,'New
Hire');
INSERT INTO jobhist VALUES (7654,'28-SEP-
81',NULL,'SALESMAN',1250,1400,30,'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,'New
Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,'New
Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-
88','CLERK',1000,NULL,20,'New Hire');
```

```
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-
89','CLERK',1040,NULL,20,'Raise');
INSERT INTO jobhist VALUES (7788,'05-MAY-
90',NULL,'ANALYST',3000,NULL,20,'Promoted to Analyst');
INSERT INTO jobhist VALUES (7839,'17-NOV-
81',NULL,'PRESIDENT',5000,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7844,'08-SEP-81',NULL,'SALESMAN',1500,0,30,'New
Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,'New
Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-
83','CLERK',950,NULL,10,'New Hire');
INSERT INTO jobhist VALUES (7900,'15-JAN-
83',NULL,'CLERK',950,NULL,30,'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,'New
Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,'New
Hire');
--
--  Populate statistics table and view (pg_statistic/pg_stats)
--
ANALYZE dept;
ANALYZE emp;
ANALYZE jobhist;
--
--  Function that lists all employees' numbers and names
--  from the 'emp' table using a cursor.
--
CREATE OR REPLACE FUNCTION list_emp() RETURNS VOID
AS $$
DECLARE
    v_empno         NUMERIC(4);
    v_ename         VARCHAR(10);
    emp_cur CURSOR FOR
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    RAISE INFO 'EMPNO    ENAME';
    RAISE INFO '-----    -------';
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN NOT FOUND;
        RAISE INFO '%     %', v_empno, v_ename;
    END LOOP;
    CLOSE emp_cur;
    RETURN;
END;
$$ LANGUAGE 'plpgsql';
--
--  Function that selects an employee row given the employee
--  number and displays certain columns.
--
CREATE OR REPLACE FUNCTION select_emp (
    p_empno         NUMERIC
) RETURNS VOID
AS $$
DECLARE
    v_ename         emp.ename%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR(10);
```

```
BEGIN
    SELECT INTO
        v_ename, v_hiredate, v_sal, v_comm, v_dname
        ename, hiredate, sal, COALESCE(comm, 0), dname
        FROM emp e, dept d
        WHERE empno = p_empno
          AND e.deptno = d.deptno;
    IF NOT FOUND THEN
        RAISE INFO 'Employee % not found', p_empno;
        RETURN;
    END IF;
    v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
    RAISE INFO 'Number     : %', p_empno;
    RAISE INFO 'Name       : %', v_ename;
    RAISE INFO 'Hire Date  : %', v_disp_date;
    RAISE INFO 'Salary     : %', v_sal;
    RAISE INFO 'Commission: %', v_comm;
    RAISE INFO 'Department: %', v_dname;
    RETURN;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO 'The following is SQLERRM : %', SQLERRM;
        RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
        RETURN;
END;
$$ LANGUAGE 'plpgsql';
--
--  A RECORD type used to format the return value of
--  function, 'emp_query'.
--
CREATE TYPE emp_query_type AS (
    empno           NUMERIC,
    ename           VARCHAR(10),
    job             VARCHAR(9),
    hiredate        DATE,
    sal             NUMERIC
);
--
--  Function that queries the 'emp' table based on
--  department number and employee number or name.  Returns
--  employee number and name as INOUT parameters and job,
--  hire date, and salary as OUT parameters.  These are
--  returned in the form of a record defined by
--  RECORD type, 'emp_query_type'.
--
CREATE OR REPLACE FUNCTION emp_query (
    IN    p_deptno       NUMERIC,
    INOUT p_empno         NUMERIC,
    INOUT p_ename         VARCHAR,
    OUT   p_job          VARCHAR,
    OUT   p_hiredate     DATE,
    OUT   p_sal          NUMERIC
)
AS $$
BEGIN
    SELECT INTO
        p_empno, p_ename, p_job, p_hiredate, p_sal
        empno, ename, job, hiredate, sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
```

```
$$ LANGUAGE 'plpgsql';
--
--  Function to call 'emp_query_caller' with IN and INOUT
--  parameters.  Displays the results received from INOUT and
--  OUT parameters.
--
CREATE OR REPLACE FUNCTION emp_query_caller() RETURNS VOID
AS $$
DECLARE
    v_deptno        NUMERIC;
    v_empno         NUMERIC;
    v_ename         VARCHAR;
    v_rows          INTEGER;
    r_emp_query     EMP_QUERY_TYPE;
BEGIN
    v_deptno := 30;
    v_empno  := 0;
    v_ename  := 'Martin';
    r_emp_query := emp_query(v_deptno, v_empno, v_ename);
    RAISE INFO 'Department : %', v_deptno;
    RAISE INFO 'Employee No: %', (r_emp_query).empno;
    RAISE INFO 'Name       : %', (r_emp_query).ename;
    RAISE INFO 'Job        : %', (r_emp_query).job;
    RAISE INFO 'Hire Date  : %', (r_emp_query).hiredate;
    RAISE INFO 'Salary     : %', (r_emp_query).sal;
    RETURN;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO 'The following is SQLERRM : %', SQLERRM;
        RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
        RETURN;
END;
$$ LANGUAGE 'plpgsql';
--
--  Function to compute yearly compensation based on semimonthly
--  salary.
--
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMERIC,
    p_comm          NUMERIC
) RETURNS NUMERIC
AS $$
BEGIN
    RETURN (p_sal + COALESCE(p_comm, 0)) * 24;
END;
$$ LANGUAGE 'plpgsql';
--
--  Function that gets the next number from sequence, 'next_empno',
--  and ensures it is not already in use as an employee number.
--
CREATE OR REPLACE FUNCTION new_empno() RETURNS INTEGER
AS $$
DECLARE
    v_cnt           INTEGER := 1;
    v_new_empno     INTEGER;
BEGIN
    WHILE v_cnt > 0 LOOP
        SELECT INTO v_new_empno nextval('next_empno');
        SELECT INTO v_cnt COUNT(*) FROM emp WHERE empno = v_new_empno;
    END LOOP;
    RETURN v_new_empno;
END;
$$ LANGUAGE 'plpgsql';
```

```
--
--  Function that adds a new clerk to table 'emp'.
--
CREATE OR REPLACE FUNCTION hire_clerk (
    p_ename         VARCHAR,
    p_deptno        NUMERIC
) RETURNS NUMERIC
AS $$
DECLARE
    v_empno         NUMERIC(4);
    v_ename         VARCHAR(10);
    v_job           VARCHAR(9);
    v_mgr           NUMERIC(4);
    v_hiredate      DATE;
    v_sal           NUMERIC(7,2);
    v_comm          NUMERIC(7,2);
    v_deptno        NUMERIC(2);
BEGIN
    v_empno := new_empno();
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        CURRENT_DATE, 950.00, NULL, p_deptno);
    SELECT  INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        empno, ename, job, mgr, hiredate, sal, comm, deptno
        FROM emp WHERE empno = v_empno;
    RAISE INFO 'Department : %', v_deptno;
    RAISE INFO 'Employee No: %', v_empno;
    RAISE INFO 'Name       : %', v_ename;
    RAISE INFO 'Job        : %', v_job;
    RAISE INFO 'Manager    : %', v_mgr;
    RAISE INFO 'Hire Date  : %', v_hiredate;
    RAISE INFO 'Salary     : %', v_sal;
    RAISE INFO 'Commission : %', v_comm;
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO 'The following is SQLERRM : %', SQLERRM;
        RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
        RETURN -1;
END;
$$ LANGUAGE 'plpgsql';
--
--  Function that adds a new salesman to table 'emp'.
--
CREATE OR REPLACE FUNCTION hire_salesman (
    p_ename         VARCHAR,
    p_sal           NUMERIC,
    p_comm          NUMERIC
) RETURNS NUMERIC
AS $$
DECLARE
    v_empno         NUMERIC(4);
    v_ename         VARCHAR(10);
    v_job           VARCHAR(9);
    v_mgr           NUMERIC(4);
    v_hiredate      DATE;
    v_sal           NUMERIC(7,2);
    v_comm          NUMERIC(7,2);
    v_deptno        NUMERIC(2);
BEGIN
    v_empno := new_empno();
    INSERT INTO emp VALUES (v_empno, p_ename, 'SALESMAN', 7698,
        CURRENT_DATE, p_sal, p_comm, 30);
```



```
    SELECT INTO
        v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
        empno, ename, job, mgr, hiredate, sal, comm, deptno
        FROM emp WHERE empno = v_empno;
    RAISE INFO 'Department : %', v_deptno;
    RAISE INFO 'Employee No: %', v_empno;
    RAISE INFO 'Name       : %', v_ename;
    RAISE INFO 'Job        : %', v_job;
    RAISE INFO 'Manager    : %', v_mgr;
    RAISE INFO 'Hire Date  : %', v_hiredate;
    RAISE INFO 'Salary     : %', v_sal;
    RAISE INFO 'Commission : %', v_comm;
    RETURN v_empno;
EXCEPTION
    WHEN OTHERS THEN
        RAISE INFO 'The following is SQLERRM : %', SQLERRM;
        RAISE INFO 'The following is SQLSTATE: %', SQLSTATE;
        RETURN -1;
END;
$$ LANGUAGE 'plpgsql';
--
--  Rule to INSERT into view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_i AS ON INSERT TO salesemp
DO INSTEAD
    INSERT INTO emp VALUES (NEW.empno, NEW.ename, 'SALESMAN', 7698,
        NEW.hiredate, NEW.sal, NEW.comm, 30);
--
--  Rule to UPDATE view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_u AS ON UPDATE TO salesemp
DO INSTEAD
    UPDATE emp SET empno    = NEW.empno,
                   ename    = NEW.ename,
                   hiredate = NEW.hiredate,
                   sal      = NEW.sal,
                   comm     = NEW.comm
        WHERE empno = OLD.empno;
--
--  Rule to DELETE from view 'salesemp'
--
CREATE OR REPLACE RULE salesemp_d AS ON DELETE TO salesemp
DO INSTEAD
    DELETE FROM emp WHERE empno = OLD.empno;
--
--  After statement-level trigger that displays a message after
--  an insert, update, or deletion to the 'emp' table.  One message
--  per SQL command is displayed.
--
CREATE OR REPLACE FUNCTION user_audit_trig() RETURNS TRIGGER
AS $$
DECLARE
    v_action        VARCHAR(24);
    v_text          TEXT;
BEGIN
    IF TG_OP = 'INSERT' THEN
        v_action := ' added employee(s) on ';
    ELSIF TG_OP = 'UPDATE' THEN
        v_action := ' updated employee(s) on ';
    ELSIF TG_OP = 'DELETE' THEN
        v_action := ' deleted employee(s) on ';
    END IF;
    v_text := 'User ' || USER || v_action || CURRENT_DATE;
```



```
    RAISE INFO ' %', v_text;
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';
CREATE TRIGGER user_audit_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH STATEMENT EXECUTE PROCEDURE user_audit_trig();
--
--  Before row-level trigger that displays employee number and
--  salary of an employee that is about to be added, updated,
--  or deleted in the 'emp' table.
--
CREATE OR REPLACE FUNCTION emp_sal_trig() RETURNS TRIGGER
AS $$
DECLARE
    sal_diff       NUMERIC(7,2);
BEGIN
    IF TG_OP = 'INSERT' THEN
        RAISE INFO 'Inserting employee %', NEW.empno;
        RAISE INFO '..New salary: %', NEW.sal;
        RETURN NEW;
    END IF;
    IF TG_OP = 'UPDATE' THEN
        sal_diff := NEW.sal - OLD.sal;
        RAISE INFO 'Updating employee %', OLD.empno;
        RAISE INFO '..Old salary: %', OLD.sal;
        RAISE INFO '..New salary: %', NEW.sal;
        RAISE INFO '..Raise     : %', sal_diff;
        RETURN NEW;
    END IF;
    IF TG_OP = 'DELETE' THEN
        RAISE INFO 'Deleting employee %', OLD.empno;
        RAISE INFO '..Old salary: %', OLD.sal;
        RETURN OLD;
    END IF;
END;
$$ LANGUAGE 'plpgsql';
CREATE TRIGGER emp_sal_trig
    BEFORE DELETE OR INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_sal_trig();
COMMIT;
```

2 Database Administration

This chapter describes the features that aid in the management and administration of Postgres Plus Advanced Server databases.

2.1 Configuration Parameters

This section describes the database server configuration parameters of Postgres Plus Advanced Server. These parameters control various aspects of the database server's behavior and environment such as data file and log file locations, connection, authentication, and security settings, resource allocation and consumption, archiving and replication settings, error logging and statistics gathering, optimization and performance tuning, locale and formatting settings, and so on.

Most of these configuration parameters apply to PostgreSQL as well. Configuration parameters that apply only to Advanced Server are noted in Section 2.1.2.

Additional information about configuration parameters can be found in the *PostgreSQL Core Documentation*, available at the EnterpriseDB website at:

http://www.enterprisedb.com/docs/en/9.4/pg/index.html

2.1.1 Setting Configuration Parameters

This section provides an overview of how configuration parameters are specified and set.

Each configuration parameter is set using a name/value pair. Parameter names are case-insensitive. The parameter name is typically separated from its value by an optional equals sign (=).

The following is an example of some configuration parameter settings in the `postgresql.conf` file:

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

Parameter values are specified as one of five types:

- **Boolean.** Acceptable values can be written as `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0`, or any unambiguous prefix of these.
- **Integer.** Number without a fractional part.
- **Floating Point.** Number with an optional fractional part separated by a decimal point.
- **String.** Text value. Enclose in single quotes if the value is not a simple identifier or number (that is, the value contains special characters such as spaces or other punctuation marks).
- **Enum.** Specific set of string values. The allowed values can be found in the system view `pg_settings.enumvals`. Enum values are case-insensitive.

Some settings specify a memory or time value. Each of these has an implicit unit, which is kilobytes, blocks (typically 8 kilobytes), milliseconds, seconds, or minutes. Default units can be found by referencing the system view `pg_settings.unit`. A different unit can be specified explicitly.

Valid memory units are `kB` (kilobytes), `MB` (megabytes), and `GB` (gigabytes). Valid time units are `ms` (milliseconds), `s` (seconds), `min` (minutes), `h` (hours), and `d` (days). The multiplier for memory units is 1024.

The configuration parameter settings can be established in a number of different ways:

- There is a number of parameter settings that are established when the Advanced Server database product is built. These are read-only parameters, and their values cannot be changed. There are also a couple of parameters that are permanently set for each database when the database is created. These parameters are read-only as well and cannot be subsequently changed for the database.

- The initial settings for almost all configurable parameters across the entire database cluster are listed in the configuration file, `postgresql.conf`. These settings are put into effect upon database server start or restart. Some of these initial parameter settings can be overridden as discussed in the following bullet points. All configuration parameters have built-in default settings that are in effect if not explicitly overridden.
- Parameter settings can be modified in the configuration file while the database server is running. If the configuration file is then reloaded (meaning a SIGHUP signal is issued), for certain parameter types, the changed parameters settings immediately take effect. For some of these parameter types, the new settings are available in a currently running session immediately after the reload. For other of these parameter types, a new session must be started to use the new settings. And yet for other parameter types, modified settings do not take effect until the database server is stopped and restarted. See Section 18.1, "Setting Parameters" in the *PostgreSQL Core Documentation* for information on how to reload the configuration file.
- The SQL commands `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` can be used to modify certain parameter settings. The modified parameter settings take effect for new sessions after the command is executed. `ALTER DATABASE` affects new sessions connecting to the specified database. `ALTER ROLE` affects new sessions started by the specified role. `ALTER ROLE IN DATABASE` affects new sessions started by the specified role connecting to the specified database. Parameter settings established by these SQL commands remain in effect indefinitely, across database server restarts, overriding settings established by the methods discussed in the second and third bullet points. Parameter settings established using the `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` commands can only be changed by: a) re-issuing these commands with a different parameter value, or b) issuing these commands using either of the `SET` *`parameter`* `TO DEFAULT` clause or the `RESET` *`parameter`* clause. These clauses change the parameter back to using the setting established by the methods set forth in the prior bullet points. See Section I, "SQL Commands" of Chapter VI "Reference" in the *PostgreSQL Core Documentation* for the exact syntax of these SQL commands.
- Changes can be made for certain parameter settings for the duration of individual sessions using the `PGOPTIONS` environment variable or by using the `SET` command within the EDB-PSQL or PSQL command line terminal programs. Parameter settings made in this manner override settings established using any of the methods described by the second, third, and fourth bullet points, but only for the duration of the session.

2.1.2 Summary of Configuration Parameters

This section contains a summary table listing all Advanced Server configuration parameters along with a number of key attributes of the parameters.

These attributes are described by the following columns of the summary table:

- **Parameter.** Configuration parameter name.
- **Scope of Effect.** Scope of effect of the configuration parameter setting. 'Cluster' – Setting affects the entire database cluster (that is, all databases managed by the database server instance). 'Database' – Setting can vary by database and is established when the database is created. Applies to a small number of parameters related to locale settings. 'Session' – Setting can vary down to the granularity of individual sessions. In other words, different settings can be made for the following entities whereby the latter settings in this list override prior ones: a) the entire database cluster, b) specific databases in the database cluster, c) specific roles, d) specific roles when connected to specific databases, e) a specific session.
- **When Takes Effect.** When a changed parameter setting takes effect. 'Preset' – Established when the Advanced Server product is built or a particular database is created. This is a read-only parameter and cannot be changed. 'Restart' – Database server must be restarted. 'Reload' – Configuration file must be reloaded (or the database server can be restarted). 'Immediate' – Immediately effective in a session if the `PGOPTIONS` environment variable or the `SET` command is used to change the setting in the current session. Effective in new sessions if `ALTER DATABASE`, `ALTER ROLE`, or `ALTER ROLE IN DATABASE` commands are used to change the setting.
- **Authorized User.** Type of operating system account or database role that must be used to put the parameter setting into effect. 'PPAS service account' – Postgres Plus Advanced Server service account (`enterprisedb` for an Oracle compatible mode installation, `postgres` for a PostgreSQL compatible mode installation). 'Superuser' – Database role with superuser privileges. 'User' – Any database role with permissions on the affected database object (the database or role to be altered with the `ALTER` command). 'n/a' – Parameter setting cannot be changed by any user.
- **Description.** Brief description of the configuration parameter.
- **PPAS Only.** 'X' – Configuration parameter is applicable to Postgres Plus Advanced Server only. No entry in this column indicates the configuration parameter applies to PostgreSQL as well.

Note: There are a number of parameters that should never be altered. These are designated as "**Note: For internal use only**" in the Description column.

Table 2-1 - Summary of Configuration Parameters

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
`allow_system_table_mods`	Cluster	Restart	PPAS service account	Allows modifications of the structure of system tables.	
`application_name`	Session	Immediate	User	Sets the application name to be reported in statistics and logs.	
`archive_command`	Cluster	Reload	PPAS service account	Sets the shell command that will be called to archive a WAL file.	
`archive_mode`	Cluster	Restart	PPAS service account	Allows archiving of WAL files using `archive_command`.	
`archive_timeout`	Cluster	Reload	PPAS service account	Forces a switch to the next xlog file if a new file has not been started within N seconds.	
`array_nulls`	Session	Immediate	User	Enable input of NULL elements in arrays.	
`authentication_timeout`	Cluster	Reload	PPAS service account	Sets the maximum allowed time to complete client authentication.	
`autovacuum`	Cluster	Reload	PPAS service account	Starts the autovacuum subprocess.	
`autovacuum_analyze_scale_factor`	Cluster	Reload	PPAS service account	Number of tuple inserts, updates or deletes prior to analyze as a fraction of `reltuples`.	
`autovacuum_analyze_threshold`	Cluster	Reload	PPAS service account	Minimum number of tuple inserts, updates or deletes prior to analyze.	
`autovacuum_freeze_max_age`	Cluster	Restart	PPAS service account	Age at which to autovacuum a table to prevent transaction ID wraparound.	
`autovacuum_max_workers`	Cluster	Restart	PPAS service account	Sets the maximum number of simultaneously running autovacuum worker processes.	
`autovacuum_multixact_freeze_max_age`	Cluster	Restart	PPAS service account	Multixact age at which to autovacuum a table to prevent multixact wraparound.	
`autovacuum_naptime`	Cluster	Reload	PPAS service account	Time to sleep between autovacuum runs.	
`autovacuum_vacuum_cost_delay`	Cluster	Reload	PPAS service account	Vacuum cost delay in milliseconds, for autovacuum.	
`autovacuum_vacuum_cost_limit`	Cluster	Reload	PPAS service account	Vacuum cost amount available before napping, for autovacuum.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
autovacuum_vacuum_scale_factor	Cluster	Reload	PPAS service account	Number of tuple updates or deletes prior to vacuum as a fraction of reltuples.	
autovacuum_vacuum_threshold	Cluster	Reload	PPAS service account	Minimum number of tuple updates or deletes prior to vacuum.	
autovacuum_work_mem	Cluster	Reload	PPAS service account	Sets the maximum memory to be used by each autovacuum worker process.	
backslash_quote	Session	Immediate	User	Sets whether "\'" is allowed in string literals.	
bgwriter_delay	Cluster	Reload	PPAS service account	Background writer sleep time between rounds.	
bgwriter_lru_maxpages	Cluster	Reload	PPAS service account	Background writer maximum number of LRU pages to flush per round.	
bgwriter_lru_multiplier	Cluster	Reload	PPAS service account	Multiple of the average buffer usage to free per round.	
block_size	Cluster	Preset	n/a	Shows the size of a disk block.	
bonjour	Cluster	Restart	PPAS service account	Enables advertising the server via Bonjour.	
bonjour_name	Cluster	Restart	PPAS service account	Sets the Bonjour service name.	
bytea_output	Session	Immediate	User	Sets the output format for bytea.	
check_function_bodies	Session	Immediate	User	Check function bodies during CREATE FUNCTION.	
checkpoint_completion_target	Cluster	Reload	PPAS service account	Time spent flushing dirty buffers during checkpoint, as fraction of checkpoint interval.	
checkpoint_segments	Deprecated in 9.5	Deprecated in 9.5	Deprecated in 9.5	This parameter is not supported by server version 9.5 or later. Specifying a value for the parameter will prevent the server from starting.	
checkpoint_timeout	Cluster	Reload	PPAS service account	Sets the maximum time between automatic WAL checkpoints.	
checkpoint_warning	Cluster	Reload	PPAS service account	Enables warnings if checkpoint segments are filled more frequently than this.	
client_encoding	Session	Immediate	User	Sets the client's character set encoding.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
client_min_messages	Session	Immediate	User	Sets the message levels that are sent to the client.	
commit_delay	Session	Immediate	Superuser	Sets the delay in microseconds between transaction commit and flushing WAL to disk.	
commit_siblings	Session	Immediate	User	Sets the minimum concurrent open transactions before performing commit_delay.	
config_file	Cluster	Restart	PPAS service account	Sets the server's main configuration file.	
constraint_exclusion	Session	Immediate	User	Enables the planner to use constraints to optimize queries.	
cpu_index_tuple_cost	Session	Immediate	User	Sets the planner's estimate of the cost of processing each index entry during an index scan.	
cpu_operator_cost	Session	Immediate	User	Sets the planner's estimate of the cost of processing each operator or function call.	
cpu_tuple_cost	Session	Immediate	User	Sets the planner's estimate of the cost of processing each tuple (row).	
cursor_tuple_fraction	Session	Immediate	User	Sets the planner's estimate of the fraction of a cursor's rows that will be retrieved.	
custom_variable_classes	Cluster	Reload	PPAS service account	Deprecated in Advanced Server 9.2.	X
data_checksums	Cluster	Preset	n/a	Shows whether data checksums are turned on for this cluster.	
data_directory	Cluster	Restart	PPAS service account	Sets the server's data directory.	
DateStyle	Session	Immediate	User	Sets the display format for date and time values.	
db_dialect	Session	Immediate	User	Sets the precedence of built-in namespaces.	X
dbms_alert.max_alerts	Cluster	Restart	PPAS service account	Sets maximum number of alerts.	X
dbms_pipe.total_message_buffer	Cluster	Restart	PPAS service account	Specifies the total size of the buffer used for the DBMS_PIPE package.	X
db_user_namespace	Cluster	Reload	PPAS service account	Enables per-database user names.	
deadlock_timeout	Session	Immediate	Superuser	Sets the time to wait on a lock	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
				before checking for deadlock.	
debug_assertions	Cluster	Preset	n/a	Turns on various assertion checks. (Not supported in PPAS builds.)	
debug_pretty_print	Session	Immediate	User	Indents parse and plan tree displays.	
debug_print_parse	Session	Immediate	User	Logs each query's parse tree.	
debug_print_plan	Session	Immediate	User	Logs each query's execution plan.	
debug_print_rewritten	Session	Immediate	User	Logs each query's rewritten parse tree.	
default_heap_fillfactor	Session	Immediate	User	Create new tables with this heap fillfactor by default.	X
default_statistics_target	Session	Immediate	User	Sets the default statistics target.	
default_tablespace	Session	Immediate	User	Sets the default tablespace to create tables and indexes in.	
default_text_search_config	Session	Immediate	User	Sets default text search configuration.	
default_transaction_deferrable	Session	Immediate	User	Sets the default deferrable status of new transactions.	
default_transaction_isolation	Session	Immediate	User	Sets the transaction isolation level of each new transaction.	
default_transaction_read_only	Session	Immediate	User	Sets the default read-only status of new transactions.	
default_with_oids	Session	Immediate	User	Create new tables with OIDs by default.	
default_with_rowids	Session	Immediate	User	Create new tables with ROWID support (OIDs with indexes) by default.	X
dynamic_library_path	Session	Immediate	Superuser	Sets the path for dynamically loadable modules.	
dynamic_shared_memory_type	Cluster	Restart	PPAS service account	Selects the dynamic shared memory implementation used.	
edb_audit	Cluster	Reload	PPAS service account	Enable EDB Auditing to create audit reports in XML or CSV format.	X
edb_audit_connect	Cluster	Reload	PPAS service account	Audits each successful connection.	X
edb_audit_directory	Cluster	Reload	PPAS service account	Sets the destination directory for audit files.	X
edb_audit_disconnect	Cluster	Reload	PPAS service account	Audits end of a session.	X

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
edb_audit_filename	Cluster	Reload	PPAS service account	Sets the file name pattern for audit files.	X
edb_audit_rotation_day	Cluster	Reload	PPAS service account	Automatic rotation of logfiles based on day of week.	X
edb_audit_rotation_seconds	Cluster	Reload	PPAS service account	Automatic log file rotation will occur after N seconds.	X
edb_audit_rotation_size	Cluster	Reload	PPAS service account	Automatic log file rotation will occur after N Megabytes.	X
edb_audit_statement	Cluster	Reload	PPAS service account	Sets the type of statements to audit.	X
edb_audit_tag	Session	Immediate	User	Specify a tag to be included in the audit log.	X
edb_connectby_order	Session	Immediate	User	Sort results of `CONNECT BY` queries with no `ORDER BY` to depth-first order. Note: For internal use only.	X
edb_custom_plan_tries	Session	Immediate	User	Specifies the number of custom execution plans considered by the planner before the planner selects a generic execution plan.	X
edb_dynatune	Cluster	Restart	PPAS service account	Sets the edb utilization percentage.	X
edb_dynatune_profile	Cluster	Restart	PPAS service account	Sets the workload profile for dynatune.	X
edb_enable_icache	Cluster	Restart	PPAS service account	Enable external shared buffer infinitecache mechanism.	X
edb_enable_pruning	Session	Immediate	User	Enables the planner to early-prune partitioned tables.	X
edb_icache_compression_level	Session	Immediate	Superuser	Sets compression level of infinitecache buffers.	X
edb_icache_servers	Cluster	Reload	PPAS service account	A list of comma separated *hostname*:*portnumber* icache servers.	X
edb_max_resource_groups	Cluster	Restart	PPAS service account	Specifies the maximum number of resource groups for simultaneous use.	X
edb_max_spins_per_delay	Cluster	Restart	PPAS service account	Specifies the number of times a session will spin while waiting for a lock.	X

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
edb_redwood_date	Session	Immediate	User	Determines whether DATE should behave like a TIMESTAMP or not.	X
edb_redwood_greatest_least	Session	Immediate	User	Determines how GREATEST and LEAST functions should handle NULL parameters.	X
edb_redwood_raw_names	Session	Immediate	User	Return the unmodified name stored in the PostgreSQL system catalogs from Redwood interfaces.	X
edb_redwood_strings	Session	Immediate	User	Treat NULL as an empty string when concatenated with a text value.	X
edb_resource_group	Session	Immediate	User	Specifies the resource group to be used by the current process.	X
edb_sql_protect.enabled	Cluster	Reload	PPAS service account	Defines whether SQL/Protect should track queries or not.	X
edb_sql_protect.level	Cluster	Reload	PPAS service account	Defines the behavior of SQL/Protect when an event is found.	X
edb_sql_protect.max_protected_relations	Cluster	Restart	PPAS service account	Sets the maximum number of relations protected by SQL/Protect per role.	X
edb_sql_protect.max_protected_roles	Cluster	Restart	PPAS service account	Sets the maximum number of roles protected by SQL/Protect.	X
edb_sql_protect.max_queries_to_save	Cluster	Restart	PPAS service account	Sets the maximum number of offending queries to save by SQL/Protect.	X
edb_stmt_level_tx	Session	Immediate	User	Allows continuing on errors instead of requiring a transaction abort.	X
edbldr.empty_csv_field	Session	Immediate	Superuser	Specifies how EDB*Loader handles empty strings.	X
effective_cache_size	Session	Immediate	User	Sets the planner's assumption about the size of the disk cache.	
effective_io_concurrency	Session	Immediate	User	Number of simultaneous requests that can be handled efficiently by the disk subsystem.	
enable_bitmapscan	Session	Immediate	User	Enables the planner's use of bitmap-scan plans.	
enable_hashagg	Session	Immediate	User	Enables the planner's use of hashed aggregation plans.	
enable_hashjoin	Session	Immediate	User	Enables the planner's use of hash join plans.	
enable_hints	Session	Immediate	User	Enable optimizer hints in SQL	X

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
				statements.	
enable_indexonlyscan	Session	Immediate	User	Enables the planner’s use of index-only-scan plans.	
enable_indexscan	Session	Immediate	User	Enables the planner's use of index-scan plans.	
enable_material	Session	Immediate	User	Enables the planner's use of materialization.	
enable_mergejoin	Session	Immediate	User	Enables the planner's use of merge join plans.	
enable_nestloop	Session	Immediate	User	Enables the planner's use of nested-loop join plans.	
enable_seqscan	Session	Immediate	User	Enables the planner's use of sequential-scan plans.	
enable_sort	Session	Immediate	User	Enables the planner's use of explicit sort steps.	
enable_tidscan	Session	Immediate	User	Enables the planner's use of TID scan plans.	
escape_string_warning	Session	Immediate	User	Warn about backslash escapes in ordinary string literals.	
event_source	Cluster	Restart	PPAS service account	Sets the application name used to identify PostgreSQL messages in the event log.	
exit_on_error	Session	Immediate	User	Terminate session on any error.	
external_pid_file	Cluster	Restart	PPAS service account	Writes the postmaster PID to the specified file.	
extra_float_digits	Session	Immediate	User	Sets the number of digits displayed for floating-point values.	
from_collapse_limit	Session	Immediate	User	Sets the FROM-list size beyond which subqueries are not collapsed.	
fsync	Cluster	Reload	PPAS service account	Forces synchronization of updates to disk.	
full_page_writes	Cluster	Reload	PPAS service account	Writes full pages to WAL when first modified after a checkpoint.	
geqo	Session	Immediate	User	Enables genetic query optimization.	
geqo_effort	Session	Immediate	User	GEQO: effort is used to set the default for other GEQO parameters.	
geqo_generations	Session	Immediate	User	GEQO: number of iterations of the algorithm.	
geqo_pool_size	Session	Immediate	User	GEQO: number of individuals in the population.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
geqo_seed	Session	Immediate	User	GEQO: seed for random path selection.	
geqo_selection_bias	Session	Immediate	User	GEQO: selective pressure within the population.	
geqo_threshold	Session	Immediate	User	Sets the threshold of FROM items beyond which GEQO is used.	
gin_fuzzy_search_limit	Session	Immediate	User	Sets the maximum allowed result for exact search by GIN.	
hba_file	Cluster	Restart	PPAS service account	Sets the server's "hba" configuration file.	
hot_standby	Cluster	Restart	PPAS service account	Allows connections and queries during recovery.	
hot_standby_feedback	Cluster	Reload	PPAS service account	Allows feedback from a hot standby to the primary that will avoid query conflicts.	
huge_pages	Cluster	Restart	PPAS service account	Use of huge pages on Linux.	
icu_short_form	Database	Preset	n/a	Shows the ICU collation order configuration.	X
ident_file	Cluster	Restart	PPAS service account	Sets the server's "ident" configuration file.	
ignore_checksum_failure	Session	Immediate	Superuser	Continues processing after a checksum failure.	
ignore_system_indexes	Cluster/ Session	Reload/ Immediate	PPAS service account/ User	Disables reading from system indexes. (Can also be set with PGOPTIONS at session start.)	
index_advisor.enabled	Session	Immediate	User	Enable Index Advisor plugin.	X
integer_datetimes	Cluster	Preset	n/a	Datetimes are integer based.	
IntervalStyle	Session	Immediate	User	Sets the display format for interval values.	
join_collapse_limit	Session	Immediate	User	Sets the FROM-list size beyond which JOIN constructs are not flattened.	
krb_caseins_users	Cluster	Reload	PPAS service account	Sets whether Kerberos and GSSAPI user names should be treated as case-insensitive.	
krb_server_keyfile	Cluster	Reload	PPAS service account	Sets the location of the Kerberos server key file.	
lc_collate	Database	Preset	n/a	Shows the collation order locale.	
lc_ctype	Database	Preset	n/a	Shows the character	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
				classification and case conversion locale.	
lc_messages	Session	Immediate	Superuser	Sets the language in which messages are displayed.	
lc_monetary	Session	Immediate	User	Sets the locale for formatting monetary amounts.	
lc_numeric	Session	Immediate	User	Sets the locale for formatting numbers.	
lc_time	Session	Immediate	User	Sets the locale for formatting date and time values.	
listen_addresses	Cluster	Restart	PPAS service account	Sets the host name or IP address(es) to listen to.	
local_preload_libraries	Cluster/ Session	Reload/ Immediate	PPAS service account/ User	Lists shared libraries to preload into each backend. (Can also be set with PGOPTIONS at session start.)	
lock_timeout	Session	Immediate	User	Sets the maximum time allowed that a statement may wait for a lock.	
lo_compat_privileges	Session	Immediate	Superuser	Enables backward compatibility mode for privilege checks on large objects.	
log_autovacuum_min_duration	Cluster	Reload	PPAS service account	Sets the minimum execution time above which autovacuum actions will be logged.	
log_checkpoints	Cluster	Reload	PPAS service account	Logs each checkpoint.	
log_connections	Cluster/ Session	Reload/ Immediate	PPAS service account/ User	Logs each successful connection. (Can also be set with PGOPTIONS at session start.)	
log_destination	Cluster	Reload	PPAS service account	Sets the destination for server log output.	
log_directory	Cluster	Reload	PPAS service account	Sets the destination directory for log files.	
log_disconnections	Cluster/ Session	Reload/ Immediate	PPAS service account/ User	Logs end of a session, including duration. (Can also be set with PGOPTIONS at session start.)	
log_duration	Session	Immediate	Superuser	Logs the duration of each completed SQL statement.	
log_error_verbosity	Session	Immediate	Superuser	Sets the verbosity of logged messages.	
log_executor_stats	Session	Immediate	Superuser	Writes executor performance	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
				statistics to the server log.	
log_file_mode	Cluster	Reload	PPAS service account	Sets the file permissions for log files.	
log_filename	Cluster	Reload	PPAS service account	Sets the file name pattern for log files.	
log_hostname	Cluster	Reload	PPAS service account	Logs the host name in the connection logs.	
log_line_prefix	Cluster	Reload	PPAS service account	Controls information prefixed to each log line.	
log_lock_waits	Session	Immediate	Superuser	Logs long lock waits.	
log_min_duration_statement	Session	Immediate	Superuser	Sets the minimum execution time above which statements will be logged.	
log_min_error_statement	Session	Immediate	Superuser	Causes all statements generating error at or above this level to be logged.	
log_min_messages	Session	Immediate	Superuser	Sets the message levels that are logged.	
log_parser_stats	Session	Immediate	Superuser	Writes parser performance statistics to the server log.	
log_planner_stats	Session	Immediate	Superuser	Writes planner performance statistics to the server log.	
log_rotation_age	Cluster	Reload	PPAS service account	Automatic log file rotation will occur after N minutes.	
log_rotation_size	Cluster	Reload	PPAS service account	Automatic log file rotation will occur after N kilobytes.	
log_statement	Session	Immediate	Superuser	Sets the type of statements logged.	
log_statement_stats	Session	Immediate	Superuser	Writes cumulative performance statistics to the server log.	
log_temp_files	Session	Immediate	Superuser	Log the use of temporary files larger than this number of kilobytes.	
log_timezone	Cluster	Reload	PPAS service account	Sets the time zone to use in log messages.	
log_truncate_on_rotation	Cluster	Reload	PPAS service account	Truncate existing log files of same name during log rotation.	
logging_collector	Cluster	Restart	PPAS service account	Start a subprocess to capture stderr output and/or csvlogs into log files.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
maintenance_work_mem	Session	Immediate	User	Sets the maximum memory to be used for maintenance operations.	
max_connections	Cluster	Restart	PPAS service account	Sets the maximum number of concurrent connections.	
max_files_per_process	Cluster	Restart	PPAS service account	Sets the maximum number of simultaneously open files for each server process.	
max_function_args	Cluster	Preset	n/a	Shows the maximum number of function arguments.	
max_identifier_length	Cluster	Preset	n/a	Shows the maximum identifier length.	
max_index_keys	Cluster	Preset	n/a	Shows the maximum number of index keys.	
max_locks_per_transaction	Cluster	Restart	PPAS service account	Sets the maximum number of locks per transaction.	
max_pred_locks_per_transaction	Cluster	Restart	PPAS service account	Sets the maximum number of predicate locks per transaction.	
max_prepared_transactions	Cluster	Restart	PPAS service account	Sets the maximum number of simultaneously prepared transactions.	
max_replication_slots	Cluster	Restart	PPAS service account	Sets the maximum number of simultaneously defined replication slots.	
max_stack_depth	Session	Immediate	Superuser	Sets the maximum stack depth, in kilobytes.	
max_standby_archive_delay	Cluster	Reload	PPAS service account	Sets the maximum delay before canceling queries when a hot standby server is processing archived WAL data.	
max_standby_streaming_delay	Cluster	Reload	PPAS service account	Sets the maximum delay before canceling queries when a hot standby server is processing streamed WAL data.	
max_wal_senders	Cluster	Restart	PPAS service account	Sets the maximum number of simultaneously running WAL sender processes.	
max_wal_size	Cluster	Reload	PPAS service account	Sets the maximum size to which the WAL will grow between automatic WAL checkpoints. The default is 1GB.	
max_worker_processes	Cluster	Restart	PPAS service account	Maximum number of concurrent worker processes.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
min_wal_size	Cluster	Reload	PPAS service account	Sets thethreshold at which WAL logs will be recycled rather than removed. The default is 80 MB.	
nls_length_semantics	Session	Immediate	Superuser	Sets the semantics to use for char, varchar, varchar2 and long columns.	X
odbc_lib_path	Cluster	Restart	PPAS service account	Sets the path for ODBC library.	X
optimizer_mode	Session	Immediate	User	Default optimizer mode.	X
oracle_home	Cluster	Restart	PPAS service account	Sets the path for the Oracle home directory.	X
password_encryption	Session	Immediate	User	Encrypt passwords.	
port	Cluster	Restart	PPAS service account	Sets the TCP port on which the server listens.	
post_auth_delay	Cluster/ Session	Reload/ Immediate	PPAS service account/ User	Waits N seconds on connection startup after authentication. (Can also be set with `PGOPTIONS` at session start.)	
pre_auth_delay	Cluster	Reload	PPAS service account	Waits N seconds on connection startup before authentication.	
qreplace_function	Session	Immediate	Superuser	The function to be used by Query Replace feature. Note: For internal use only.	X
query_rewrite_enabled	Session	Immediate	User	Child table scans will be skipped if their constraints guarantee that no rows match the query.	X
query_rewrite_integrity	Session	Immediate	Superuser	Sets the degree to which query rewriting must be enforced.	X
quote_all_identifiers	Session	Immediate	User	When generating SQL fragments, quote all identifiers.	
random_page_cost	Session	Immediate	User	Sets the planner's estimate of the cost of a nonsequentially fetched disk page.	
restart_after_crash	Cluster	Reload	PPAS service account	Reinitialize server after backend crash.	
search_path	Session	Immediate	User	Sets the schema search order for names that are not schema-qualified.	
segment_size	Cluster	Preset	n/a	Shows the number of pages per disk file.	
seq_page_cost	Session	Immediate	User	Sets the planner's estimate of	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
				the cost of a sequentially fetched disk page.	
server_encoding	Database	Preset	n/a	Sets the server (database) character set encoding.	
server_version	Cluster	Preset	n/a	Shows the server version.	
server_version_num	Cluster	Preset	n/a	Shows the server version as an integer.	
session_preload_libraries	Session	Immediate but only at connection start	Superuser	Lists shared libraries to preload into each backend.	
session_replication_role	Session	Immediate	Superuser	Sets the session's behavior for triggers and rewrite rules.	
shared_buffers	Cluster	Restart	PPAS service account	Sets the number of shared memory buffers used by the server.	
shared_preload_libraries	Cluster	Restart	PPAS service account	Lists shared libraries to preload into server.	
sql_inheritance	Session	Immediate	User	Causes subtables to be included by default in various commands.	
ssl	Cluster	Restart	PPAS service account	Enables SSL connections.	
ssl_ca_file	Cluster	Restart	PPAS service account	Location of the SSL certificate authority file.	
ssl_cert_file	Cluster	Restart	PPAS service account	Location of the SSL server certificate file.	
ssl_ciphers	Cluster	Restart	PPAS service account	Sets the list of allowed SSL ciphers.	
ssl_crl_file	Cluster	Restart	PPAS service account	Location of the SSL certificate revocation list file.	
ssl_ecdh_curve	Cluster	Restart	PPAS service account	Sets the curve to use for ECDH.	
ssl_key_file	Cluster	Restart	PPAS service account	Location of the SSL server private key file.	
ssl_prefer_server_ciphers	Cluster	Restart	PPAS service account	Give priority to server ciphersuite order.	
ssl_renegotiation_limit	Session	Immediate	User	Set the amount of traffic to send and receive before	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
				renegotiating the encryption keys.	
standard_conforming_strings	Session	Immediate	User	Causes '...' strings to treat backslashes literally.	
statement_timeout	Session	Immediate	User	Sets the maximum allowed duration of any statement.	
stats_temp_directory	Cluster	Reload	PPAS service account	Writes temporary statistics files to the specified directory.	
superuser_reserved_connections	Cluster	Restart	PPAS service account	Sets the number of connection slots reserved for superusers.	
synchronize_seqscans	Session	Immediate	User	Enable synchronized sequential scans.	
synchronous_commit	Session	Immediate	User	Sets immediate fsync at commit.	
synchronous_standby_names	Cluster	Reload	PPAS service account	List of names of potential synchronous standbys.	
syslog_facility	Cluster	Reload	PPAS service account	Sets the syslog "facility" to be used when syslog enabled.	
syslog_ident	Cluster	Reload	PPAS service account	Sets the program name used to identify PostgreSQL messages in syslog.	
tcp_keepalives_count	Session	Immediate	User	Maximum number of TCP keepalive retransmits.	
tcp_keepalives_idle	Session	Immediate	User	Time between issuing TCP keepalives.	
tcp_keepalives_interval	Session	Immediate	User	Time between TCP keepalive retransmits.	
temp_buffers	Session	Immediate	User	Sets the maximum number of temporary buffers used by each session.	
temp_file_limit	Session	Immediate	Superuser	Limits the total size of all temporary files used by each session.	
temp_tablespaces	Session	Immediate	User	Sets the tablespace(s) to use for temporary tables and sort files.	
timed_statistics	Session	Immediate	User	Enables the recording of timed statistics.	X
timezone	Session	Immediate	User	Sets the time zone for displaying and interpreting time stamps.	
timezone_abbreviations	Session	Immediate	User	Selects a file of time zone abbreviations.	
trace_hints	Session	Immediate	User	Emit debug info about hints being honored.	X

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
trace_notify	Session	Immediate	User	Generates debugging output for LISTEN and NOTIFY.	
trace_recovery_messages	Cluster	Reload	PPAS service account	Enables logging of recovery-related debugging information.	
trace_sort	Session	Immediate	User	Emit information about resource usage in sorting.	
track_activities	Session	Immediate	Superuser	Collects information about executing commands.	
track_activity_query_size	Cluster	Restart	PPAS service account	Sets the size reserved for pg_stat_activity.current_query, in bytes.	
track_counts	Session	Immediate	Superuser	Collects statistics on database activity.	
track_functions	Session	Immediate	Superuser	Collects function-level statistics on database activity.	
track_io_timing	Session	Immediate	Superuser	Collects timing statistics for database I/O activity.	
transaction_deferrable	Session	Immediate	User	Whether to defer a read-only serializable transaction until it can be executed with no possible serialization failures.	
transaction_isolation	Session	Immediate	User	Sets the current transaction's isolation level.	
transaction_read_only	Session	Immediate	User	Sets the current transaction's read-only status.	
transform_null_equals	Session	Immediate	User	Treats "expr=NULL" as "expr IS NULL".	
unix_socket_directories	Cluster	Restart	PPAS service account	Sets the directory where the Unix-domain socket will be created.	
unix_socket_group	Cluster	Restart	PPAS service account	Sets the owning group of the Unix-domain socket.	
unix_socket_permissions	Cluster	Restart	PPAS service account	Sets the access permissions of the Unix-domain socket.	
update_process_title	Session	Immediate	Superuser	Updates the process title to show the active SQL command.	
utl_encode.uudecode_redwood	Session	Immediate	User	Allows decoding of Oracle-created uuencoded data.	X
utl_file.umask	Session	Immediate	User	Umask used for files created through the UTL_FILE package.	X
vacuum_cost_delay	Session	Immediate	User	Vacuum cost delay in milliseconds.	
vacuum_cost_limit	Session	Immediate	User	Vacuum cost amount available before napping.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
vacuum_cost_page_dirty	Session	Immediate	User	Vacuum cost for a page dirtied by vacuum.	
vacuum_cost_page_hit	Session	Immediate	User	Vacuum cost for a page found in the buffer cache.	
vacuum_cost_page_miss	Session	Immediate	User	Vacuum cost for a page not found in the buffer cache.	
vacuum_defer_cleanup_age	Cluster	Reload	PPAS service account	Number of transactions by which VACUUM and HOT cleanup should be deferred, if any.	
vacuum_freeze_min_age	Session	Immediate	User	Minimum age at which VACUUM should freeze a table row.	
vacuum_freeze_table_age	Session	Immediate	User	Age at which VACUUM should scan whole table to freeze tuples.	
vacuum_multixact_freeze_min_age	Session	Immediate	User	Minimum age at which VACUUM should freeze a MultiXactId in a table row.	
vacuum_multixact_freeze_table_age	Session	Immediate	User	Multixact age at which VACUUM should scan whole table to freeze tuples.	
wal_block_size	Cluster	Preset	n/a	Shows the block size in the write ahead log.	
wal_buffers	Cluster	Restart	PPAS service account	Sets the number of disk-page buffers in shared memory for WAL.	
wal_keep_segments	Cluster	Reload	PPAS service account	Sets the number of WAL files held for standby servers.	
wal_level	Cluster	Restart	PPAS service account	Set the level of information written to the WAL.	
wal_log_hints	Cluster	Restart	PPAS service account	Writes full pages to WAL when first modified after a checkpoint, even for non-critical modifications.	
wal_receiver_status_interval	Cluster	Reload	PPAS service account	Sets the maximum interval between WAL receiver status reports to the primary.	
wal_receiver_timeout	Cluster	Reload	PPAS service account	Sets the maximum wait time to receive data from the primary.	
wal_segment_size	Cluster	Preset	n/a	Shows the number of pages per write ahead log segment.	
wal_sender_timeout	Cluster	Reload	PPAS service account	Sets the maximum time to wait for WAL replication.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	PPAS Only
wal_sync_method	Cluster	Reload	PPAS service account	Selects the method used for forcing WAL updates to disk.	
wal_writer_delay	Cluster	Reload	PPAS service account	WAL writer sleep time between WAL flushes.	
work_mem	Session	Immediate	User	Sets the maximum memory to be used for query workspaces.	
xloginsert_locks	Cluster	Restart	PPAS service account	Sets the number of locks used for concurrent xlog insertions.	
xmlbinary	Session	Immediate	User	Sets how binary values are to be encoded in XML.	
Xmloption	Session	Immediate	User	Sets whether XML data in implicit parsing and serialization operations is to be considered as documents or content fragments.	
zero_damaged_pages	Session	Immediate	Superuser	Continues processing past damaged page headers.	

2.1.3 Configuration Parameters by Functionality

This section provides more detail for certain groups of configuration parameters.

The section heading for each parameter is followed by a list of attributes:

- **Parameter Type.** Type of values the parameter can accept. See Section 2.1.1 for a discussion of parameter type values.
- **Default Value.** Default setting if a value is not explicitly set in the configuration file.
- **Range.** Permitted range of values.
- **Minimum Scope of Effect.** Smallest scope for which a distinct setting can be made. Generally, the minimal scope of a distinct setting is either the entire **cluster** (the setting is the same for all databases and sessions thereof, in the cluster), or **per session** (the setting may vary by role, database, or individual session). (This attribute has the same meaning as the "Scope of Effect" column in the table of Section 2.1.2.)
- **When Value Changes Take Effect.** Least invasive action required to activate a change to a parameter's value. All parameter setting changes made in the configuration file can be put into effect with a restart of the database server; however certain parameters require a database server **restart**. Some parameter setting changes can be put into effect with a **reload** of the configuration file without stopping the database server. Finally, other parameter setting changes can be put into effect with some client side action whose result is **immediate**. (This attribute has the same meaning as the "When Takes Effect" column in the table of Section 2.1.2.)
- **Required Authorization to Activate.** The type of user authorization to activate a change to a parameter's setting. If a database server restart or a configuration file reload is required, then the user must be a PPAS service account (`enterprisedb` or `postgres` depending upon the Advanced Server compatibility installation mode). This attribute has the same meaning as the "Authorized User" column in the table of Section 2.1.2.

2.1.3.1 Top Performance Related Parameters

This section discusses the configuration parameters that have the most immediate impact on performance.

2.1.3.1.1 shared_buffers

Parameter Type: Integer

Default Value: 32MB

Range: 128kB to system dependent

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Sets the amount of memory the database server uses for shared memory buffers. The default is typically 32 megabytes (`32MB`), but might be less if your kernel settings will not support it (as determined during `initdb`). This setting must be at least 128 kilobytes. (Non-default values of `BLCKSZ` change the minimum.) However, settings significantly higher than the minimum are usually needed for good performance.

If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for `shared_buffers` is 25% of the memory in your system. There are some workloads where even large settings for `shared_buffers` are effective, but because Postgres Plus also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to `shared_buffers` will work better than a smaller amount.

On systems with less than 1GB of RAM, a smaller percentage of RAM is appropriate, so as to leave adequate space for the operating system (15% of memory is more typical in these situations). Also, on Windows, large values for `shared_buffers` aren't as effective. You may find better results keeping the setting relatively low and using the operating system cache more instead. The useful range for `shared_buffers` on Windows systems is generally from 64MB to 512MB.

Increasing this parameter might cause Postgres Plus to request more System V shared memory than your operating system's default configuration allows. See Section 17.4.1, “Shared Memory and Semaphores” in the *PostgreSQL Core Documentation* for information on how to adjust those parameters, if necessary.

2.1.3.1.2 work_mem

Parameter Type: Integer

Default Value: 1MB

Range: 64kB to 2097151kB

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. The value defaults to one megabyte (`1MB`). Note that for a complex query, several sort or hash operations might be running in parallel; each operation will be allowed to use as much memory as this value specifies before it starts to write data into temporary files. Also, several running sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of `work_mem`; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for `ORDER BY`, `DISTINCT`, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of `IN` subqueries.

Reasonable values are typically between 4MB and 64MB, depending on the size of your machine, how many concurrent connections you expect (determined by `max_connections`), and the complexity of your queries.

2.1.3.1.3 maintenance_work_mem

Parameter Type: Integer

Default Value: 16MB

Range: 1024kB to 2097151kB

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Specifies the maximum amount of memory to be used by maintenance operations, such as `VACUUM`, `CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY`. It defaults to 16 megabytes (`16MB`). Since only one of these operations can be executed at a time by a database session, and an installation normally doesn't have many of them running concurrently, it's safe to set this value significantly larger than `work_mem`. Larger settings might improve performance for vacuuming and for restoring database dumps.

Note that when autovacuum runs, up to `autovacuum_max_workers` times this memory may be allocated, so be careful not to set the default value too high.

A good rule of thumb is to set this to about 5% of system memory, but not more than about 512MB. Larger values won't necessarily improve performance.

2.1.3.1.4 wal_buffers

Parameter Type: Integer

Default Value: 64kB

Range: 32kB to system dependent

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

The amount of memory used in shared memory for WAL data. The default is 64 kilobytes (`64kB`). The setting need only be large enough to hold the amount of WAL data generated by one typical transaction, since the data is written out to disk at every transaction commit.

Increasing this parameter might cause Postgres Plus to request more System V shared memory than your operating system's default configuration allows. See Section 17.4.1, “Shared Memory and Semaphores” in the *PostgreSQL Core Documentation* for information on how to adjust those parameters, if necessary.

Although even this very small setting does not always cause a problem, there are situations where it can result in extra `fsync` calls, and degrade overall system throughput. Increasing this value to 1MB or so can alleviate this problem. On very busy systems, an even higher value may be needed, up to a maximum of about 16MB. Like `shared_buffers`, this parameter increases Postgres Plus’s initial shared memory allocation, so if increasing it causes a Postgres Plus start failure, you will need to increase the operating system limit.

2.1.3.1.5 checkpoint_segments

Now deprecated; this parameter is not supported by server versions 9.5 or later.

2.1.3.1.6 checkpoint_completion_target

Parameter Type: Floating point

Default Value: 0.5

Range: 0 to 1

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the target of checkpoint completion as a fraction of total time between checkpoints. This spreads out the checkpoint writes while the system starts working towards the next checkpoint.

The default of 0.5 means aim to finish the checkpoint writes when 50% of the next checkpoint is ready. A value of 0.9 means aim to finish the checkpoint writes when 90% of the next checkpoint is done, thus throttling the checkpoint writes over a larger amount of time and avoiding spikes of performance bottlenecking.

2.1.3.1.7 checkpoint_timeout

Parameter Type: Integer

Default Value: 5min

Range: 30s to 3600s

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Maximum time between automatic WAL checkpoints, in seconds. The default is five minutes (`5min`). Increasing this parameter can increase the amount of time needed for crash recovery.

Increasing `checkpoint_timeout` to a larger value, such as 15 minutes, can reduce the I/O load on your system, especially when using large values for `shared_buffers`.

The downside of making the aforementioned adjustments to the checkpoint parameters is that your system will use a modest amount of additional disk space, and will take longer to recover in the event of a crash. However, for most users, this is a small price to pay for a significant performance improvement.

2.1.3.1.8 max_wal_size

Parameter Type: Integer

Default Value: 1 GB

Range: 2 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

`max_wal_size` specifies the maximum size that the WAL will reach between automatic WAL checkpoints. This is a soft limit; WAL size can exceed `max_wal_size` under special circumstances (when under a heavy load, a failing archive_command, or a high wal_keep_segments setting).

Increasing this parameter can increase the amount of time needed for crash recovery. This parameter can only be set in the `postgresql.conf` file or on the server command line.

2.1.3.1.9 min_wal_size

Parameter Type: Integer

Default Value: 80 MB

Range: 2 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

If WAL disk usage stays below the value specified by `min_wal_size`, old WAL files are recycled for future use at a checkpoint, rather than removed. This ensures that enough WAL space is reserved to handle spikes in WAL usage (like when running large batch jobs). This parameter can only be set in the postgresql.conf file or on the server command line.

2.1.3.1.10 bgwriter_delay

Parameter Type: Integer

Default Value: 200ms

Range: 10ms to 10000ms

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the delay between activity rounds for the background writer. In each round the writer issues writes for some number of dirty buffers (controllable by the `bgwriter_lru_maxpages` and `bgwriter_lru_multiplier` parameters). It then sleeps for `bgwriter_delay` milliseconds, and repeats.

The default value is 200 milliseconds (`200ms`). Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting `bgwriter_delay` to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10.

Typically, when tuning `bgwriter_delay`, it should be reduced from its default value. This parameter is rarely increased, except perhaps to save on power consumption on a system with very low utilization.

2.1.3.1.11 seq_page_cost

Parameter Type: Floating point

Default Value: 1

Range: 0 to 1.79769e+308

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for a particular tablespace by setting the tablespace parameter of the same name. (Refer to the `ALTER TABLESPACE` command in the *PostgreSQL Core Documentation*.)

The default value assumes very little caching, so it's frequently a good idea to reduce it. Even if your database is significantly larger than physical memory, you might want to try setting this parameter to less than 1 (rather than its default value of 1) to see whether you get better query plans that way. If your database fits entirely within memory, you can lower this value much more, perhaps to 0.1.

2.1.3.1.12 random_page_cost

Parameter Type: Floating point

Default Value: 4

Range: 0 to 1.79769e+308

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Sets the planner's estimate of the cost of a non-sequentially-fetched disk page. The default is 4.0. This value can be overridden for a particular tablespace by setting the tablespace parameter of the same name. (Refer to the `ALTER TABLESPACE` command in the *PostgreSQL Core Documentation*.)

Reducing this value relative to `seq_page_cost` will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. You can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs, which are described by the `cpu_tuple_cost` and `cpu_index_tuple_cost` parameters.

The default value assumes very little caching, so it's frequently a good idea to reduce it. Even if your database is significantly larger than physical memory, you might want to try setting this parameter to 2 (rather than its default of 4) to see whether you get better query plans that way. If your database fits entirely within memory, you can lower this value much more, perhaps to 0.1.

Although the system will let you do so, never set `random_page_cost` less than `seq_page_cost`. However, setting them equal (or very close to equal) makes sense if the database fits mostly or entirely within memory, since in that case there is no penalty for touching pages out of sequence. Also, in a heavily-cached database you should lower both values relative to the CPU parameters, since the cost of fetching a page already in RAM is much smaller than it would normally be.

2.1.3.1.13 effective_cache_size

Parameter Type: Integer

Default Value: 128MB

Range: 8kB to 17179869176kB

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Sets the planner's assumption about the effective size of the disk cache that is available to a single query. This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. When setting this parameter you should consider both Postgres Plus's shared buffers and the portion of the kernel's disk cache that will be used for Postgres Plus data files. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by Postgres Plus, nor does it reserve kernel disk cache; it is used only for estimation purposes. The default is 128 megabytes (`128MB`).

If this parameter is set too low, the planner may decide not to use an index even when it would be beneficial to do so. Setting `effective_cache_size` to 50% of physical memory is a normal, conservative setting. A more aggressive setting would be approximately 75% of physical memory.

2.1.3.1.14 synchronous_commit

Parameter Type: Boolean

Default Value: `true`

Range: {`true | false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Specifies whether transaction commit will wait for WAL records to be written to disk before the command returns a "success" indication to the client. The default, and safe, setting is on. When off, there can be a delay between when success is reported to the client and when the transaction is really guaranteed to be safe against a server crash. (The maximum delay is three times `wal_writer_delay`.)

Unlike `fsync`, setting this parameter to off does not create any risk of database inconsistency: an operating system or database crash might result in some recent allegedly-committed transactions being lost, but the database state will be just the same as if those transactions had been aborted cleanly.

So, turning `synchronous_commit` off can be a useful alternative when performance is more important than exact certainty about the durability of a transaction. See Section 29.3, *Asynchronous Commit* in the *PostgreSQL Core Documentation* for information.

This parameter can be changed at any time; the behavior for any one transaction is determined by the setting in effect when it commits. It is therefore possible, and useful, to have some transactions commit synchronously and others asynchronously. For example, to make a single multistatement transaction commit asynchronously when the default is the opposite, issue `SET LOCAL synchronous_commit TO OFF` within the transaction.

2.1.3.1.15 edb_max_spins_per_delay

Parameter Type: Integer

Default Value: `1000`

Range: {`10 | 1000`}

Minimum Scope of Effect: Per cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Use `edb_max_spins_per_delay` to specify the maximum number of times that a session will 'spin' while waiting for a spin-lock. If a lock is not acquired, the session will sleep. If you do not specify an alternative value for `edb_max_spins_per_delay`, the server will enforce the default value of `1000`.

This may be useful for sytems that use NUMA (non-uniform memory access) architecture.

2.1.3.2 Resource Usage / Memory

The configuration parameters in this section control resource usage pertaining to memory.

2.1.3.2.1 edb_dynatune

Parameter Type: Integer

Default Value: 0

Range: 0 to 100

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Determines how much of the host system's resources are to be used by the database server based upon the host machine's total available resources and the intended usage of the host machine.

When Postgres Plus Advanced Server is initially installed, the `edb_dynatune` parameter is set in accordance with the selected usage of the host machine on which it was installed (i.e., development machine, mixed use machine, or dedicated server). For most purposes, there is no need for the database administrator to adjust the various configuration parameters in the `postgresql.conf` file in order to improve performance.

The `edb_dynatune` parameter can be set to any integer value between 0 and 100, inclusive. A value of 0, turns off the dynamic tuning feature thereby leaving the database server resource usage totally under the control of the other configuration parameters in the `postgresql.conf` file.

A low non-zero, value (e.g., 1 - 33) dedicates the least amount of the host machine's resources to the database server. This setting would be used for a development machine where many other applications are being used.

A value in the range of 34 - 66 dedicates a moderate amount of resources to the database server. This setting might be used for a dedicated application server that may have a fixed number of other applications running on the same machine as Postgres Plus Advanced Server.

The highest values (e.g., 67 - 100) dedicate most of the server's resources to the database server. This setting would be used for a host machine that is totally dedicated to running Postgres Plus Advanced Server.

Once a value of `edb_dynatune` is selected, database server performance can be further fine-tuned by adjusting the other configuration parameters in the `postgresql.conf` file. Any adjusted setting overrides the corresponding value chosen by `edb_dynatune`. You can change the value of a parameter by un-commenting the configuration parameter, specifying the desired value, and restarting the database server.

2.1.3.2.2 edb_dynatune_profile

Parameter Type: Enum

Default Value: `oltp`

Range: {`oltp` | `reporting` | `mixed`}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

This parameter is used to control tuning aspects based upon the expected workload profile on the database server.

The following are the possible values:

- **oltp.** Recommended when the database server is processing heavy online transaction processing workloads.
- **reporting.** Recommended for database servers used for heavy data reporting.
- **mixed.** Recommended for servers that provide a mix of transaction processing and data reporting.

2.1.3.2.3 edb_enable_icache

Parameter Type: Boolean

Default Value: `false`

Range: {`true` | `false`}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Enables or disables Infinite Cache. If `edb_enable_icache` is set to on, Infinite Cache is enabled; if the parameter is set to off, Infinite Cache is disabled.

If you set `edb_enable_icache` to on, you must also specify a list of cache servers by setting the `edb_icache_servers` parameter.

2.1.3.2.4 edb_icache_servers

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

The `edb_icache_servers` parameter specifies a list of one or more servers with active edb-icache daemons. `edb_icache_servers` is a string value that takes the form of a comma-separated list of *`hostname`*`:`*`port`* pairs. You can specify each pair in any of the following forms:

- *`hostname`*
- *`IP_address`*
- *`hostname`*`:`*`portnumber`*
- *`IP_address`*`:`*`portnumber`*

If you do not specify a port number, Infinite Cache assumes that the cache server is listening at port 11211. This configuration parameter will take effect only if `edb_enable_icache` is set to on. Use the `edb_icache_servers` parameter to specify a maximum of 128 cache nodes.

You can dynamically modify the Infinite Cache server nodes. To change the Infinite Cache server configuration, use the `edb_icache_servers` parameter in the `postgresql.conf` file to perform the following:

- Specify additional cache information to add server(s).
- Delete server information to remove server(s).

- Specify additional server information and delete existing server information to both add and delete servers during the same reload operation.

After updating the `edb_icache_servers` parameter in the `postgresql.conf` file, you must reload the configuration parameters for the changes to take effect.

2.1.3.2.5 edb_icache_compression_level

Parameter Type: Integer

Default Value: 6

Range: 0 to 9

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Superuser

The `edb_icache_compression_level` parameter controls the compression level that is applied to each page before storing it in the distributed Infinite Cache.

When Advanced Server reads data from disk, it typically reads the data in 8kB increments. If `edb_icache_compression_level` is set to 0, each time Advanced Server sends an 8kB page to the Infinite Cache server that page is stored (uncompressed) in 8kB of cache memory. If the `edb_icache_compression_level` parameter is set to 9, Advanced Server applies the maximum compression possible before sending it to the Infinite Cache server, so a page that previously took 8kB of cached memory might take 2kB of cached memory. Exact compression numbers are difficult to predict, as they are dependent on the nature of the data on each page.

This parameter must be an integer in the range 0 to 9.

- A compression level of 0 disables compression; it uses no CPU time for compression, but requires more storage space and network resources to process.
- A compression level of 9 invokes the maximum amount of compression; it increases the load on the CPU, but less data flows across the network, so network demand is reduced. Each page takes less room on the Infinite Cache server, so memory requirements are reduced.
- A compression level of 5 or 6 is a reasonable compromise between the amount of compression received and the amount of CPU time invested.

The compression level must be set by the superuser and can be changed for the current session while the server is running. The following command disables the compression mechanism for the currently active session:

```
SET edb_icache_compression_level TO 0;
```

2.1.3.3 Resource Usage / EDB Resource Manager

The configuration parameters in this section control resource usage through EDB Resource Manager.

2.1.3.3.1 edb_max_resource_groups

Parameter Type: Integer

Default Value: 16

Range: 0 to 65536

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

This parameter controls the maximum number of resource groups that can be used simultaneously by EDB Resource Manager. More resource groups can be created than the value specified by `edb_max_resource_groups`, however, the number of resource groups in active use by processes in these groups cannot exceed this value.

Parameter `edb_max_resource_groups` should be set comfortably larger than the number of groups you expect to maintain so as not to run out.

2.1.3.3.2 edb_resource_group

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Set the `edb_resource_group` parameter to the name of the resource group to which the current session is to be controlled by EDB Resource Manager according to the group's resource type settings.

If the parameter is not set, then the current session does not utilize EDB Resource Manager.

2.1.3.4 Query Tuning

This section describes the configuration parameters used for optimizer hints.

2.1.3.4.1 enable_hints

Parameter Type: Boolean

Default Value: `true`

Range: {`true | false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Optimizer hints embedded in SQL commands are utilized when `enable_hints` is on. Optimizer hints are ignored when this parameter is off.

2.1.3.5 Query Tuning / Planner Method Configuration

This section describes the configuration parameters used for planner method configuration.

2.1.3.5.1 edb_custom_plan_tries

Parameter Type: Numeric

Default Value: `5`

Range: {`0 | 100`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session User

This configuration parameter controls the number of custom execution plans considered by the planner before the planner settles on a generic execution plan.

When a client application repeatedly executes a prepared statement, the server may decide to evaluate several execution plans before deciding to choose a *custom* plan or a *generic* plan.

- A custom plan is a plan built for a specific set of parameter values.
- A generic plan is a plan that will work with any set of parameter values supplied by the client application.

By default, the optimizer will generate five custom plans before evaluating a generic plan. That means that if you execute a prepared statement six times, the optimizer will generate five custom plans, then one generic plan, and then decide whether to stick with the generic plan.

In certain workloads, this extra planning can have a negative impact on performance. You can adjust the `edb_custom_plan_tries` configuration parameter to decrease the number of custom plans considered before evaluating a generic plan. Setting `edb_custom_plan_tries` to `0` will effectively disable custom plan generation.

Consider the following query:

```
PREPARE custQuery AS SELECT * FROM customer WHERE salesman >= $1
```

The `$1` token in this query is a parameter marker - the client application must provide a value for each parameter marker each time the statement executes.

If an index has been defined on `customer.salesman`, the optimizer may choose to execute this query using a sequential scan, or using an index scan. In some cases, an index is faster than a sequential scan; in other cases, the sequential scan will win. The optimal plan will depend on the distribution of salesman values in the table and on the search value (the value provided for the $1 parameter).

When the client application repeatedly executes the `custQuery` prepared statement, the optimizer will generate some number of parameter-value-specific execution plans (custom plans), followed by a generic plan (a plan that ignores the parameter values), and then decide whether to stick with the generic plan or to continue to generate custom plans for each execution. The decision process takes into account not only the cost of executing the plans, but the cost of generating custom plans as well.

2.1.3.5.2 edb_enable_pruning

Parameter Type: Boolean

Default Value: `true`

Range: {`true | false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When set to `TRUE`, `edb_enable_pruning` allows the query planner to early-prune partitioned tables. *Early-pruning* means that the query planner can "prune" (i.e., ignore) partitions that would not be searched in a query *before* generating query plans. This helps improve performance time as it eliminates the generation of query plans of partitions that would not be searched.

Conversely, *late-pruning* means that the query planner prunes partitions *after* generating query plans for each partition. (The `constraint_exclusion` configuration parameter controls late-pruning.)

The ability to early-prune depends upon the nature of the query in the `WHERE` clause. Early-pruning can be utilized in only simple queries with constraints of the type `WHERE` *`column`* `=` *`literal`* (e.g., `WHERE deptno = 10`).

Early-pruning is not used for more complex queries such as `WHERE` *`column`* `=` *`expression`* (e.g., `WHERE deptno = 10 + 5`).

2.1.3.6 Reporting and Logging / What to Log

The configuration parameters in this section control reporting and logging.

2.1.3.6.1 trace_hints

Parameter Type: Boolean

Default Value: `false`

Range: {`true | false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Use with the optimizer hints feature to provide more detailed information regarding whether or not a hint was used by the planner. Set the `client_min_messages` and `trace_hints` configuration parameters as follows:

```
SET client_min_messages TO info;
SET trace_hints TO true;
```

The `SELECT` command with the `NO_INDEX` hint shown below illustrates the additional information produced when the aforementioned configuration parameters are set.

```
EXPLAIN SELECT /*+ NO_INDEX(accounts accounts_pkey) */ * FROM accounts WHERE
aid = 100;

INFO:  [HINTS] Index Scan of [accounts].[accounts_pkey] rejected because of
NO_INDEX hint.

INFO:  [HINTS] Bitmap Heap Scan of [accounts].[accounts_pkey] rejected
because of NO_INDEX hint.
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on accounts  (cost=0.00..14461.10 rows=1 width=97)
   Filter: (aid = 100)
(2 rows)
```

2.1.3.7 EnterpriseDB Auditing Settings

This section describes configuration parameters used by the Postgres Plus Advanced Server database auditing feature.

2.1.3.7.1 edb_audit

Parameter Type: Enum

Default Value: `none`

Range: {`none` | `csv` | `xml`}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Enables or disables database auditing. The values `xml` or `csv` will enable database auditing. These values represent the file format in which auditing information will be captured. `none` will disable database auditing and is also the default.

2.1.3.7.2 edb_audit_directory

Parameter Type: String

Default Value: `edb_audit`

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the directory where the audit log files will be created. The path of the directory can be absolute or relative to the *`POSTGRES_PLUS_HOME`*`/data` directory.

2.1.3.7.3 edb_audit_filename

Parameter Type: String

Default Value: `audit-%Y%m%d_%H%M%S`

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the file name of the audit file where the auditing information will be stored. The default file name will be `audit-%Y%m%d_%H%M%S`. The escape sequences, `%Y`, `%m` etc., will be replaced by the appropriate current values according to the system date and time.

2.1.3.7.4 edb_audit_rotation_day

Parameter Type: String

Default Value: `every`

Range: {`none` | `every` | `sun` | `mon` | `tue` | `wed` | `thu` | `fri` | `sat`} ...

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the day of the week on which to rotate the audit files. Valid values are `sun`, `mon`, `tue`, `wed`, `thu`, `fri`, `sat`, `every`, and `none`. To disable rotation, set the value to `none`. To rotate the file every day, set the `edb_audit_rotation_day` value to `every`. To rotate the file on a specific day of the week, set the value to the desired day of the week.

2.1.3.7.5 edb_audit_rotation_size

Parameter Type: Integer

Default Value: 0MB

Range: 0MB to 5000MB

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies a file size threshold in megabytes when file rotation will be forced to occur. The default value is 0MB. If the parameter is commented out or set to 0, rotation of the file on a size basis will not occur.

2.1.3.7.6 edb_audit_rotation_seconds

Parameter Type: Integer

Default Value: 0s

Range: 0s to 2147483647s

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Specifies the rotation time in seconds when a new log file should be created. To disable this feature, set this parameter to 0.

2.1.3.7.7 edb_audit_connect

Parameter Type: Enum

Default Value: `failed`

Range: {`none` | `failed` | `all`}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Enables auditing of database connection attempts by users. To disable auditing of all connection attempts, set `edb_audit_connect` to `none`. To audit all failed connection attempts, set the value to `failed`. To audit all connection attempts, set the value to `all`.

2.1.3.7.8 edb_audit_disconnect

Parameter Type: Enum

Default Value: `none`

Range: {none | all}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Enables auditing of database disconnections by connected users. To enable auditing of disconnections, set the value to `all`. To disable, set the value to `none`.

2.1.3.7.9 edb_audit_statement

Parameter Type: String

Default Value: `ddl, error`

Range: {none | ddl | dml | select | error | rollback | all} ...

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

This configuration parameter is used to specify auditing of different categories of SQL statements. To audit statements resulting in error, set the parameter value to `error`. To audit DDL statements such as `CREATE TABLE`, `ALTER TABLE`, etc., set the parameter value to `ddl`. Modification statements such as `INSERT`, `UPDATE`, `DELETE` or `TRUNCATE` can be audited by setting `edb_audit_statement` to `dml`. Setting the value to `all` will audit every statement while `none` disables this feature.

2.1.3.7.10 edb_audit_tag

Parameter Type: String

Default Value: none

Minimum Scope of Effect: Session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: User

Use edb_audit_tag to specify a string value that will be included in the audit log when the edb_audit parameter is set to csv or xml.

2.1.3.8 Client Connection Defaults / Locale and Formatting

This section describes configuration parameters affecting locale and formatting.

2.1.3.8.1 icu_short_form

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Database

When Value Changes Take Effect: n/a

Required Authorization to Activate: n/a

The configuration parameter `icu_short_form` is a parameter containing the default ICU short form name assigned to a database or to the Advanced Server instance. See Section 2.3 for general information about the ICU short form and the Unicode Collation Algorithm.

This configuration parameter is set either when the `CREATE DATABASE` command is used with the `ICU_SHORT_FORM` parameter (see Section 2.3.3.2) in which case the specified short form name is set and appears in the `icu_short_form` configuration parameter when connected to this database, or when an Advanced Server instance is created with the `initdb` command used with the `--icu_short_form option` (see Section 2.3.3.3) in which case the specified short form name is set and appears in the `icu_short_form` configuration parameter when connected to a database in that Advanced Server instance, and the database does not override it with its own `ICU_SHORT_FORM` parameter with a different short form.

Once established in the manner described, the `icu_short_form` configuration parameter cannot be changed.

2.1.3.9 Client Connection Defaults / Statement Behavior

This section describes configuration parameters affecting statement behavior.

2.1.3.9.1 default_heap_fillfactor

Parameter Type: Integer

Default Value: 100

Range: 10 to 100

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Sets the fillfactor for a table when the `FILLFACTOR` storage parameter is omitted from a `CREATE TABLE` command.

The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, `INSERT` operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate.

2.1.3.10 Client Connection Defaults / Other Defaults

The parameters in this section set other miscellaneous client connection defaults.

2.1.3.10.1 oracle_home

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Before creating an Oracle Call Interface (OCI) database link to an Oracle server, you must direct Advanced Server to the correct Oracle home directory. Set the `LD_LIBRARY_PATH` environment variable on Linux (or `PATH` on Windows) to the `lib` directory of the Oracle client installation directory.

For Windows only, you can instead set the value of the `oracle_home` configuration parameter in the `postgresql.conf` file. The value specified in the `oracle_home` configuration parameter will override the Windows `PATH` environment variable.

The `LD_LIBRARY_PATH` environment variable on Linux (`PATH` environment variable or `oracle_home` configuration parameter on Windows) must be set properly each time you start Advanced Server.

For Windows only: To set the `oracle_home` configuration parameter in the `postgresql.conf` file, edit the file, adding the following line:

```
oracle_home = 'lib_directory'
```

Substitute the name of the Windows directory that contains `oci.dll` for *`lib_directory`*.

After setting the `oracle_home` configuration parameter, you must restart the server for the changes to take effect. Restart the server from the Windows Services console.

2.1.3.10.2 odbc_lib_path

Parameter Type: String

Default Value: none

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

If you will be using an ODBC driver manager, and if it is installed in a non-standard location, you specify the location by setting the `odbc_lib_path` configuration parameter in the `postgresql.conf` file:

```
odbc_lib_path = 'complete_path_to_libodbc.so'
```

The configuration file must include the complete pathname to the driver manager shared library (typically `libodbc.so`).

2.1.3.11 Compatibility Options

The configuration parameters described in this section control various database compatibility features.

2.1.3.11.1 edb_redwood_date

Parameter Type: Boolean

Default Value: `false`

Range: {`true` | `false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When `DATE` appears as the data type of a column in the commands, it is translated to `TIMESTAMP(0)` at the time the table definition is stored in the database if the configuration parameter `edb_redwood_date` is set to `TRUE`. Thus, a time component will also be stored in the column along with the date.

If edb_redwood_date is set to `FALSE` the column's data type in a `CREATE TABLE` or `ALTER TABLE` command remains as a native PostgreSQL `DATE` data type and is stored as such in the database. The PostgreSQL `DATE` data type stores only the date without a time component in the column.

Regardless of the setting of `edb_redwood_date`, when `DATE` appears as a data type in any other context such as the data type of a variable in an SPL declaration section, or the data type of a formal parameter in an SPL procedure or SPL function, or the return type of an SPL function, it is always internally translated to a `TIMESTAMP(0)` and thus, can handle a time component if present.

2.1.3.11.2 edb_redwood_greatest_least

Parameter Type: Boolean

Default Value: `true`

Range: {`true` | `false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

The GREATEST function returns the parameter with the greatest value from its list of parameters. The LEAST function returns the parameter with the least value from its list of parameters.

When edb_redwood_greatest_least is set to TRUE, the GREATEST and LEAST functions return null when at least one of the parameters is null.

```
SET edb_redwood_greatest_least TO on;

SELECT GREATEST(1, 2, NULL, 3);

greatest
----------

(1 row)
```

When edb_redwood_greatest_least is set to FALSE, null parameters are ignored except when all parameters are null in which case null is returned by the functions.

```
SET edb_redwood_greatest_least TO off;

SELECT GREATEST(1, 2, NULL, 3);

greatest
----------
        3
(1 row)

SELECT GREATEST(NULL, NULL, NULL);

greatest
----------

(1 row)
```

2.1.3.11.3 edb_redwood_raw_names

Parameter Type: Boolean

Default Value: false

Range: {true | false}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When `edb_redwood_raw_names` is set to its default value of `FALSE`, database object names such as table names, column names, trigger names, program names, user names, etc. appear in uppercase letters when viewed from Redwood catalogs (that is, system catalogs prefixed by `ALL_`, `DBA_`, or `USER_`. See Chapter 10 for a list of such catalogs). In addition, quotation marks enclose names that were created with enclosing quotation marks.

When `edb_redwood_raw_names` is set to `TRUE`, the database object names are displayed exactly as they are stored in the PostgreSQL system catalogs when viewed from the Redwood catalogs. Thus, names created without enclosing quotation marks appear in lowercase as expected in PostgreSQL. Names created with enclosing quotation marks appear exactly as they were created, but without the quotation marks.

For example, the following user name is created, and then a session is started with that user.

```
CREATE USER reduser IDENTIFIED BY password;
edb=# \c - reduser
Password for user reduser:
You are now connected to database "edb" as user "reduser".
```

When connected to the database as `reduser`, the following tables are created.

```
CREATE TABLE all_lower (col INTEGER);
CREATE TABLE ALL_UPPER (COL INTEGER);
CREATE TABLE "Mixed_Case" ("Col" INTEGER);
```

When viewed from the Redwood catalog, `USER_TABLES`, with `edb_redwood_raw_names` set to the default value `FALSE`, the names appear in uppercase except for the `Mixed_Case` name, which appears as created and also with enclosing quotation marks.

```
edb=> SELECT * FROM USER_TABLES;
 schema_name |  table_name   | tablespace_name | status | temporary
-------------+--------------+-----------------+--------+-----------
 REDUSER     | ALL_LOWER    |                 | VALID  | N
 REDUSER     | ALL_UPPER    |                 | VALID  | N
 REDUSER     | "Mixed_Case" |                 | VALID  | N
(3 rows)
```

When viewed with `edb_redwood_raw_names` set to `TRUE`, the names appear in lowercase except for the `Mixed_Case` name, which appears as created, but now without the enclosing quotation marks.

```
edb=> SET edb_redwood_raw_names TO true;
SET
edb=> SELECT * FROM USER_TABLES;
 schema_name | table_name | tablespace_name | status | temporary
```

```
-------------+------------+----------------+--------+-----------
 reduser     | all_lower  |                | VALID  | N
 reduser     | all_upper  |                | VALID  | N
 reduser     | Mixed_Case |                | VALID  | N
(3 rows)
```

These names now match the case when viewed from the PostgreSQL `pg_tables` catalog.

```
edb=> SELECT schemaname, tablename, tableowner FROM pg_tables WHERE
tableowner = 'reduser';
 schemaname | tablename  | tableowner
------------+------------+------------
 reduser    | all_lower  | reduser
 reduser    | all_upper  | reduser
 reduser    | Mixed_Case | reduser
(3 rows)
```

2.1.3.11.4 edb_redwood_strings

Parameter Type: Boolean

Default Value: `false`

Range: {`true` | `false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

If the `edb_redwood_strings` parameter is set to `TRUE`, when a string is concatenated with a null variable or null column, the result is the original string. If `edb_redwood_strings` is set to `FALSE`, the native PostgreSQL behavior is maintained, which is the concatenation of a string with a null variable or null column gives a null result.

The following example illustrates the difference.

The sample application contains a table of employees. This table has a column named `comm` that is null for most employees. The following query is run with `edb_redwood_string` set to `FALSE`. The concatenation of a null column with non-empty strings produces a final result of null, so only employees that have a commission appear in the query result. The output line for all other employees is null.

```
SET edb_redwood_strings TO off;

SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;
```

```
      EMPLOYEE COMPENSATION
----------------------------------

 ALLEN        1,600.00     300.00
 WARD         1,250.00     500.00

 MARTIN       1,250.00   1,400.00




 TURNER       1,500.00        .00




(14 rows)
```

The following is the same query executed when `edb_redwood_strings` is set to `TRUE`. Here, the value of a null column is treated as an empty string. The concatenation of an empty string with a non-empty string produces the non-empty string.

```
SET edb_redwood_strings TO on;

SELECT RPAD(ename,10) || ' ' || TO_CHAR(sal,'99,999.99') || ' ' ||
TO_CHAR(comm,'99,999.99') "EMPLOYEE COMPENSATION" FROM emp;

      EMPLOYEE COMPENSATION
----------------------------------
 SMITH          800.00
 ALLEN        1,600.00     300.00
 WARD         1,250.00     500.00
 JONES        2,975.00
 MARTIN       1,250.00   1,400.00
 BLAKE        2,850.00
 CLARK        2,450.00
 SCOTT        3,000.00
 KING         5,000.00
 TURNER       1,500.00        .00
 ADAMS        1,100.00
 JAMES          950.00
 FORD         3,000.00
 MILLER       1,300.00
(14 rows)
```

2.1.3.11.5 edb_stmt_level_tx

Parameter Type: Boolean

Default Value: `false`

Range: {`true` | `false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

The term *statement level transaction isolation* describes the behavior whereby when a runtime error occurs in a SQL command, all the updates on the database caused by that single command are rolled back. For example, if a single `UPDATE` command successfully updates five rows, but an attempt to update a sixth row results in an exception, the updates to all six rows made by this `UPDATE` command are rolled back. The effects of prior SQL commands that have not yet been committed or rolled back are pending until a `COMMIT` or `ROLLBACK` command is executed.

In Postgres Plus, if an exception occurs while executing a SQL command, all the updates on the database since the start of the transaction are rolled back. In addition, the transaction is left in an aborted state and either a `COMMIT` or `ROLLBACK` command must be issued before another transaction can be started.

If `edb_stmt_level_tx` is set to `TRUE`, then an exception will not automatically roll back prior uncommitted database updates. If `edb_stmt_level_tx` is set to `FALSE`, then an exception will roll back uncommitted database updates.

Note: Use `edb_stmt_level_tx` set to `TRUE` only when absolutely necessary, as this may cause a negative performance impact.

The following example run in PSQL shows that when `edb_stmt_level_tx` is `FALSE`, the abort of the second `INSERT` command also rolls back the first `INSERT` command. Note that in PSQL, the command `\set AUTOCOMMIT off` must be issued, otherwise every statement commits automatically defeating the purpose of this demonstration of the effect of `edb_stmt_level_tx`.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO off;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
ERROR:  insert or update on table "emp" violates foreign key constraint
"emp_ref_dept_fk"
DETAIL:  Key (deptno)=(0) is not present in table "dept".

COMMIT;
SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno
-------+-------+--------
(0 rows)
```

In the following example, with `edb_stmt_level_tx` set to `TRUE`, the first `INSERT` command has not been rolled back after the error on the second `INSERT` command. At this point, the first `INSERT` command can either be committed or rolled back.

```
\set AUTOCOMMIT off
SET edb_stmt_level_tx TO on;

INSERT INTO emp (empno,ename,deptno) VALUES (9001, 'JONES', 40);
INSERT INTO emp (empno,ename,deptno) VALUES (9002, 'JONES', 00);
ERROR:  insert or update on table "emp" violates foreign key constraint
"emp_ref_dept_fk"
DETAIL:  Key (deptno)=(0) is not present in table "dept"

SELECT empno, ename, deptno FROM emp WHERE empno > 9000;

empno | ename | deptno
-------+-------+--------
  9001 | JONES |     40
(1 row)

COMMIT;
```

A `ROLLBACK` command could have been issued instead of the `COMMIT` command in which case the insert of employee number `9001` would have been rolled back as well.

2.1.3.11.6 db_dialect

Parameter Type: Enum

Default Value: `postgres`

Range: {`postgres` | `redwood`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

In addition to the native PostgreSQL system catalog, `pg_catalog`, Advanced Server contains extended catalog views (see Chapter 10) as well as system catalogs compatible with Microsoft® SQL Server®. These are `sys` for the expanded catalog views and `dbo` for SQL Server. The `db_dialect` parameter controls the order in which these catalogs are searched for name resolution.

When set to `postgres`, the namespace precedence is `pg_catalog`, `sys`, then `dbo`, giving the PostgreSQL catalog the highest precedence. When set to `redwood`, the namespace precedence is `sys`, `dbo`, then `pg_catalog`, giving the expanded catalog views the highest precedence.

2.1.3.11.7 default_with_rowids

Parameter Type: Boolean

Default Value: `false`

Range: {`true` | `false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When set to on, `CREATE TABLE` includes a `ROWID` column in newly created tables, which can then be referenced in SQL commands.

2.1.3.11.8 optimizer_mode

Parameter Type: Enum

Default Value: `choose`

Range: {`choose` | `ALL_ROWS` | `FIRST_ROWS` | `FIRST_ROWS_10` | `FIRST_ROWS_100` | `FIRST_ROWS_1000`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Sets the default optimization mode for analyzing optimizer hints.

The following table shows the possible values:

Table 2-2 - Optimizer Modes

Hint	Description
`ALL_ROWS`	Optimizes for retrieval of all rows of the result set.
`CHOOSE`	Does no default optimization based on assumed number of rows to be retrieved from the result set. This is the default.
`FIRST_ROWS`	Optimizes for retrieval of only the first row of the result set.
`FIRST_ROWS_10`	Optimizes for retrieval of the first 10 rows of the results set.
`FIRST_ROWS_100`	Optimizes for retrieval of the first 100 rows of the result set.
`FIRST_ROWS_1000`	Optimizes for retrieval of the first 1000 rows of the result set.

These optimization modes are based upon the assumption that the client submitting the SQL command is interested in viewing only the first "n" rows of the result set and will then abandon the remainder of the result set. Resources allocated to the query are adjusted as such.

2.1.3.12 Customized Options

In previous releases of Advanced Server, the `custom_variable_classes` was required by those parameters not normally known to be added by add-on modules (such as procedural languages).

2.1.3.12.1 custom_variable_classes

The `custom_variable_classes` parameter is deprecated in Advanced Server 9.2; parameters that previously depended on this parameter no longer require its support.

2.1.3.12.2 dbms_alert.max_alerts

Parameter Type: Integer

Default Value: 100

Range: 0 to 500

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Specifies the maximum number of concurrent alerts allowed on a system using the `DBMS_ALERTS` package.

2.1.3.12.3 dbms_pipe.total_message_buffer

Parameter Type: Integer

Default Value: 30 Kb

Range: 30 Kb to 256 Kb

Minimum Scope of Effect: Postmaster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Specifies the total size of the buffer used for the DBMS_PIPE package.

2.1.3.12.4 index_advisor.enabled

Parameter Type: Boolean

Default Value: `true`

Range: {`true` | `false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Provides the capability to temporarily suspend Index Advisor in an EDB-PSQL or PSQL session. The Index Advisor plugin, `index_advisor`, must be loaded in the EDB-PSQL or PSQL session in order to use the `index_advisor.enabled` configuration parameter.

The Index Advisor plugin can be loaded as shown by the following example:

```
$ psql -d edb -U enterprisedb
Password for user enterprisedb:
psql (9.4.0.0)
Type "help" for help.

edb=# LOAD 'index_advisor';
LOAD
```

Use the `SET` command to change the parameter setting to control whether or not Index Advisor generates an alternative query plan as shown by the following example:

```
edb=# SET index_advisor.enabled TO off;
SET
edb=# EXPLAIN SELECT * FROM t WHERE a < 10000;
                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on t  (cost=0.00..1693.00 rows=9864 width=8)
   Filter: (a < 10000)
(2 rows)

edb=# SET index_advisor.enabled TO on;
SET
edb=# EXPLAIN SELECT * FROM t WHERE a < 10000;
                                  QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..1693.00 rows=9864 width=8)
   Filter: (a < 10000)
 Result  (cost=0.00..327.88 rows=9864 width=8)
   One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text
   ->  Index Scan using "<hypothetical-index>:1" on t  (cost=0.00..327.88
rows=9864 width=8)
         Index Cond: (a < 10000)
```



```
(6 rows)
```

2.1.3.12.5 edb_sql_protect.enabled

Parameter Type: Boolean

Default Value: `false`

Range: {`true` | `false`}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Controls whether or not SQL/Protect is actively monitoring protected roles by analyzing SQL statements issued by those roles and reacting according to the setting of `edb_sql_protect.level`. When you are ready to begin monitoring with SQL/Protect set this parameter to on.

2.1.3.12.6 edb_sql_protect.level

Parameter Type: Enum

Default Value: `passive`

Range: {`learn` | `passive` | `active`}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: PPAS service account

Sets the action taken by SQL/Protect when a SQL statement is issued by a protected role.

The `edb_sql_protect.level` configuration parameter can be set to one of the following values to use either learn mode, passive mode, or active mode:

- **learn.** Tracks the activities of protected roles and records the relations used by the roles. This is used when initially configuring SQL/Protect so the expected behaviors of the protected applications are learned.
- **passive.** Issues warnings if protected roles are breaking the defined rules, but does not stop any SQL statements from executing. This is the next step after SQL/Protect has learned the expected behavior of the protected roles. This

essentially behaves in intrusion detection mode and can be run in production when properly monitored.

- **active.** Stops all invalid statements for a protected role. This behaves as a SQL firewall preventing dangerous queries from running. This is particularly effective against early penetration testing when the attacker is trying to determine the vulnerability point and the type of database behind the application. Not only does SQL/Protect close those vulnerability points, but it tracks the blocked queries allowing administrators to be alerted before the attacker finds an alternate method of penetrating the system.

If you are using SQL/Protect for the first time, set `edb_sql_protect.level` to `learn`.

2.1.3.12.7 edb_sql_protect.max_protected_relations

Parameter Type: Integer

Default Value: 1024

Range: 1 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Sets the maximum number of relations that can be protected per role. Please note the total number of protected relations for the server will be the number of protected relations times the number of protected roles. Every protected relation consumes space in shared memory. The space for the maximum possible protected relations is reserved during database server startup.

If the server is started when `edb_sql_protect.max_protected_relations` is set to a value outside of the valid range (for example, a value of 2,147,483,648), then a warning message is logged in the database server log file:

```
2014-07-18 16:04:12 EDT WARNING:  invalid value for parameter
"edb_sql_protect.max_protected_relations": "2147483648"
2014-07-18 16:04:12 EDT HINT:  Value exceeds integer range.
```

The database server starts successfully, but with `edb_sql_protect.max_protected_relations` set to the default value of 1024.

Though the upper range for the parameter is listed as the maximum value for an integer data type, the practical setting depends on how much shared memory is available and the parameter value used during database server startup.

As long as the space required can be reserved in shared memory, the value will be acceptable. If the value is such that the space in shared memory cannot be reserved, the database server startup fails with an error message such as the following:

```
2014-07-18 15:22:17 EDT FATAL:  could not map anonymous shared memory: Cannot
allocate memory
2014-07-18 15:22:17 EDT HINT:  This error usually means that PostgreSQL's
request for a shared memory segment exceeded available memory, swap space or
huge pages. To reduce the request size (currently 2070118400 bytes), reduce
PostgreSQL's shared memory usage, perhaps by reducing shared_buffers or
max_connections.
```

In such cases, reduce the parameter value until the database server can be started successfully.

2.1.3.12.8 edb_sql_protect.max_protected_roles

Parameter Type: Integer

Default Value: 64

Range: 1 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Sets the maximum number of roles that can be protected.

Every protected role consumes space in shared memory. Please note that the server will reserve space for the number of protected roles times the number of protected relations (`edb_sql_protect.max_protected_relations`). The space for the maximum possible protected roles is reserved during database server startup.

If the database server is started when `edb_sql_protect.max_protected_roles` is set to a value outside of the valid range (for example, a value of 2,147,483,648), then a warning message is logged in the database server log file:

```
2014-07-18 16:04:12 EDT WARNING:  invalid value for parameter
"edb_sql_protect.max_protected_roles": "2147483648"
2014-07-18 16:04:12 EDT HINT:  Value exceeds integer range.
```


The database server starts successfully, but with `edb_sql_protect.max_protected_roles` set to the default value of 64.

Though the upper range for the parameter is listed as the maximum value for an integer data type, the practical setting depends on how much shared memory is available and the parameter value used during database server startup.

As long as the space required can be reserved in shared memory, the value will be acceptable. If the value is such that the space in shared memory cannot be reserved, the database server startup fails with an error message such as the following:

```
2014-07-18 15:22:17 EDT FATAL:  could not map anonymous shared memory: Cannot
allocate memory
2014-07-18 15:22:17 EDT HINT:  This error usually means that PostgreSQL's
request for a shared memory segment exceeded available memory, swap space or
huge pages. To reduce the request size (currently  2070118400 bytes), reduce
PostgreSQL's shared memory usage, perhaps by reducing shared_buffers or
max_connections.
```

In such cases, reduce the parameter value until the database server can be started successfully.

2.1.3.12.9 edb_sql_protect.max_queries_to_save

Parameter Type: Integer

Default Value: 5000

Range: 100 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: PPAS service account

Sets the maximum number of offending queries to save in view `edb_sql_protect_queries`.

Every query that is saved consumes space in shared memory. The space for the maximum possible queries that can be saved is reserved during database server startup.

If the database server is started when `edb_sql_protect.max_queries_to_save` is set to a value outside of the valid range (for example, a value of 10), then a warning message is logged in the database server log file:

```
2014-07-18 13:05:31 EDT WARNING:  10 is outside the valid range for parameter
"edb_sql_protect.max_queries_to_save" (100 .. 2147483647)
```

The database server starts successfully, but with `edb_sql_protect.max_queries_to_save` set to the default value of 5000.

Though the upper range for the parameter is listed as the maximum value for an integer data type, the practical setting depends on how much shared memory is available and the parameter value used during database server startup.

As long as the space required can be reserved in shared memory, the value will be acceptable. If the value is such that the space in shared memory cannot be reserved, the database server startup fails with an error message such as the following:

```
2014-07-18 15:22:17 EDT FATAL:  could not map anonymous shared memory: Cannot
allocate memory
2014-07-18 15:22:17 EDT HINT:  This error usually means that PostgreSQL's
request for a shared memory segment exceeded available memory, swap space or
huge pages. To reduce the request size (currently  2070118400 bytes), reduce
PostgreSQL's shared memory usage, perhaps by reducing shared_buffers or
max_connections.
```

In such cases, reduce the parameter value until the database server can be started successfully.

2.1.3.12.10 edbldr.empty_csv_field

Parameter Type: Enum

Default Value: `NULL`

Range: {`NULL` | `empty_string | pgsql`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Use the `edbldr.empty_csv_field` parameter to specify how EDB*Loader will treat an empty string. The valid values for the `edbldr.empty_csv_field` parameter are:

Parameter Setting	EDB*Loader Behavior
`NULL`	An empty field is treated as `NULL`.
`empty_string`	An empty field is treated as a string of length zero.
`pgsql`	An empty field is treated as a `NULL` if it does not contain quotes and as an empty string if it contains quotes.

For more information about the `edbldr.empty_csv_field` parameter in EDB*Loader, see Section 6.1.9.

2.1.3.12.11 utl_encode.uudecode_redwood

Parameter Type: Boolean

Default Value: `false`

Range: {`true` | `false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

When set to `TRUE`, Advanced Server's `UTL_ENCODE.UUDECODE` function can decode uuencoded data that was created by the Oracle implementation of the `UTL_ENCODE.UUENCODE` function.

When set to the default setting of `FALSE`, Advanced Server's `UTL_ENCODE.UUDECODE` function can decode uuencoded data created by Advanced Server's `UTL_ENCODE.UUENCODE` function.

2.1.3.12.12 utl_file.umask

Parameter Type: String

Default Value: 0077

Range: Octal digits for umask settings

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

The `utl_file.umask` parameter sets the *file mode creation mask* or simply, the *mask*, in a manner similar to the Linux `umask` command. This is for usage only within the Advanced Server `UTL_FILE` package.

Note: The `utl_file.umask` parameter is not supported on Windows systems.

The value specified for `utl_file.umask` is a 3 or 4-character octal string that would be valid for the Linux `umask` command. The setting determines the permissions on files created by the `UTL_FILE` functions and procedures. (Refer to any information source regarding Linux or Unix systems for information on file permissions and the usage of the `umask` command.)

The following shows the results of the default `utl_file.umask` setting of 0077. Note that all permissions are denied on users belonging to the `enterprisedb` group as well as all other users. Only user `enterprisedb` has read and write permissions on the file.

```
-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile
```

For an example of using `utl_file.umask`, see Section 9.17.1.

2.1.3.13 Ungrouped

Configuration parameters in this section apply to Postgres Plus Advanced Server only and are for a specific, limited purpose.

2.1.3.13.1 nls_length_semantics

Parameter Type: Enum

Default Value: `byte`

Range: {`byte` | `char`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Superuser

This parameter has no effect in Postgres Plus Advanced Server.

For example, the form of the `ALTER SESSION` command is accepted in Advanced Server without throwing a syntax error, but does not alter the session environment:

```
ALTER SESSION SET nls_length_semantics = char;
```

Note: Since the setting of this parameter has no effect on the server environment, it does not appear in the system view `pg_settings`.

2.1.3.13.2 query_rewrite_enabled

Parameter Type: Enum

Default Value: `false`

Range: {`true` | `false` | `force`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

This parameter has no effect in Postgres Plus Advanced Server.

For example, the following form of the `ALTER SESSION` command is accepted in Advanced Server without throwing a syntax error, but does not alter the session environment:

```
ALTER SESSION SET query_rewrite_enabled = force;
```

Note: Since the setting of this parameter has no effect on the server environment, it does not appear in the system view `pg_settings`.

2.1.3.13.3 query_rewrite_integrity

Parameter Type: Enum

Default Value: `enforced`

Range: {`enforced` | `trusted` | `stale_tolerated`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Superuser

This parameter has no effect in Postgres Plus Advanced Server.

For example, the following form of the `ALTER SESSION` command is accepted in Advanced Server without throwing a syntax error, but does not alter the session environment:

```
ALTER SESSION SET query_rewrite_integrity = stale_tolerated;
```

Note: Since the setting of this parameter has no effect on the server environment, it does not appear in the system view `pg_settings`.

2.1.3.13.4 timed_statistics

Parameter Type: Boolean

Default Value: `true`

Range: {`true` | `false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Controls the collection of timing data for the Dynamic Runtime Instrumentation Tools Architecture (DRITA) feature. When set to on, timing data is collected.

Note: When Advanced Server is installed, the `postgresql.conf` file contains an explicit entry setting `timed_statistics` to off. If this entry is commented out letting `timed_statistics` to default, and the configuration file is reloaded, timed statistics collection would be turned on.

2.2 Controlling the Audit Logs

Postgres Plus Advanced Server allows database and security administrators, auditors, and operators to track and analyze database activities using the audit logs. The audit logs can be configured to information such as:

- When a role establishes a connection to an Advanced Server database
- What database objects a role creates, modifies, or deletes when connected to Advanced Server
- When any failed authentication attempts occur

You can use parameters specified in the `postgresql.conf` file to control the information included in the audit logs.

2.2.1 Auditing Configuration Parameters

Use the following `postgresql.conf` configuration parameters to control database auditing:

`edb_audit`

> Enables or disables database auditing. The values `xml` or `csv` will enable database auditing. These values represent the file format in which auditing information will be captured. `none` will disable database auditing and is also the default. This option can only be set at server start or in the `postgresql.conf` file.

`edb_audit_directory`

> Specifies the directory where the log files will be created. The path of the directory can be relative or absolute to the data folder. This option can only be set at server start or in the `postgresql.conf` configuration file.

`edb_audit_filename`

> Specifies the file name of the audit file where the auditing information will be stored. The default file name will be `audit-%Y%m%d_%H%M%S`. The escape sequences, `%Y`, `%m` etc., will be replaced by the appropriate current values according to the system date and time. This option can only be set at server start or in the `postgresql.conf` configuration file.

`edb_audit_rotation_day`

Specifies the day of the week on which to rotate the audit files. Valid values are sun, mon, tue, wed, thu, fri, sat, every, and none. To disable rotation, set the value to none. To rotate the file every day, set the edb_audit_rotation_day value to every. To rotate the file on a specific day of the week, set the value to the desired day of the week. every is the default value. This option can only be set at server start or in the postgresql.conf configuration file.

edb_audit_rotation_size

Specifies a file size threshold in megabytes when file rotation will be forced to occur. The default value is 0 MB. If the parameter is commented out or set to 0, rotation of the file on a size basis will not occur. This option can only be set at server start or in the postgresql.conf configuration file.

edb_audit_rotation_seconds

Specifies the rotation time in seconds when a new log file should be created. To disable this feature, set this parameter to 0. This option can only be set at server start or in the postgresql.conf configuration file.

edb_audit_connect

Enables auditing of database connection attempts by users. To disable auditing of all connection attempts, set edb_audit_connect to none. To audit all failed connection attempts, set the value to failed. To audit all connection attempts, set the value to all. This option can only be set at server start or in the postgresql.conf configuration file.

edb_audit_disconnect

Enables auditing of database disconnections by connected users. To enable auditing of disconnections, set the value to all. To disable, set the value to none. This option can only be set at server start or in the postgresql.conf configuration file.

edb_audit_statement

This configuration parameter is used to specify auditing of different categories of SQL statements. To audit statements resulting in error, set the parameter value to error. To audit DDL statements such as CREATE TABLE, ALTER TABLE, etc., set the parameter value to ddl. Modification statements such as INSERT, UPDATE, DELETE or TRUNCATE can be audited by setting edb_audit_statement to dml. To audit ROLLBACK statements, set the parameter value to rollback. Setting the value to all will audit every statement while `none` disables this feature. This option can only be set at server start or in the `postgresql.conf` configuration file.

`edb_audit_tag`

Use this configuration parameter is used to specify a string value that will be included in the audit log when the `edb_audit` parameter is set to `csv` or `xml`.

The following steps describe how to configure Advanced Server to log all connections, disconnections, DDL statements, and any statements resulting in an error. The resulting audit file will rotate every Sunday.

1. Enable auditing by the setting the `edb_audit` parameter to `xml` or `csv`.
2. Set the file rotation day when the new file will be created by setting the parameter `edb_audit_rotation_day` to `sun`.
3. To audit all connections, set the parameter, `edb_audit_connect`, to `all`.
4. To audit all disconnections, set the parameter, `edb_audit_disconnect`, to `all`.
5. To audit all DDL statements and error statements, set the parameter, `edb_audit_statement`, to `ddl, error`.

The following is the CSV and XML output when auditing is enabled:

CSV Audit Log File

```
,,,1452,,,2008-03-13 12:40:02 ,startup,"AUDIT:  database system is ready"
enterprisedb,mgmtsvr,127.0.0.1(1266),1620,47d9595b.654,0,2008-03-13 12:42:03 ,connect,"AUDIT:
connection authorized: user=enterprisedb database=mgmtsvr"
enterprisedb,mgmtsvr,127.0.0.1(1266),1620,47d9595b.654,1588,2008-03-13 12:42:08 ,ddl,"AUDIT:
statement: drop table HILOSEQUENCES
     "
enterprisedb,mgmtsvr,127.0.0.1(1266),1620,47d9595b.654,1590,2008-03-13 12:42:09 ,ddl,"AUDIT:
statement: create table HILOSEQUENCES (
                SEQUENCENAME varchar(50) not null,
                HIGHVALUES integer not null,
                constraint hilo_pk primary key (SEQUENCENAME)
            )
     "
enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,0,2008-03-13 12:42:53 ,connect,"AUDIT:
connection authorized: user=enterprisedb database=edb"
enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,1618,2008-03-13 12:43:02 ,ddl,"AUDIT:
statement: CREATE TABLE test (f1 INTEGER);"
enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,1620,2008-03-13 12:43:02 ,sql
statement,"AUDIT:  statement: SELECT * FROM testx;"
enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,1620,2008-03-13 12:43:02 ,error,"ERROR:
relation "testx" does not exist"
enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,1621,2008-03-13 12:43:04 ,ddl,"AUDIT:
statement: DROP TABLE test;"
enterprisedb,edb,127.0.0.1(1269),904,47d9598d.388,0,2008-03-13 12:43:20 ,disconnect,"AUDIT:
disconnection: session time: 0:00:26.953 user=enterprisedb database=edb host=127.0.0.1
port=1269"
enterprisedb,mgmtsvr,127.0.0.1(1266),1620,47d9595b.654,0,2008-03-13 12:43:29
,disconnect,"AUDIT:  disconnection: session time: 0:01:26.594 user=enterprisedb
database=mgmtsvr host=127.0.0.1 port=1266"
,,,3148,,,2008-03-13 12:43:35 ,shutdown,"AUDIT:  database system is shut down"
```

XML Audit Log File

```
<event process_id="2516" time="2008-03-13 13:22:42 " type="startup">
       <message>AUDIT:  database system is ready</message>
</event>
<event user="enterprisedb" database="mgmtsvr" remote_host_and_port="127.0.0.1(1281)"
       process_id="352" session_id="47d96338.160" transaction="0"
       time="2008-03-13 13:24:08 " type="connect">
       <message>AUDIT:  connection authorized: user=enterprisedb
               database=mgmtsvr</message>
</event>
<event user="enterprisedb" database="mgmtsvr" remote_host_and_port="127.0.0.1(1281)"
       process_id="352" session_id="47d96338.160" transaction="1635"
       time="2008-03-13 13:24:10 " type="ddl">
       <command>AUDIT:  statement: drop table HILOSEQUENCES</command>
</event>
<event user="enterprisedb" database="mgmtsvr" remote_host_and_port="127.0.0.1(1281)"
       process_id="352" session_id="47d96338.160" transaction="1637"
       time="2008-03-13 13:24:10 " type="ddl">
       <command>AUDIT:  statement: create table HILOSEQUENCES (
          SEQUENCENAME varchar(50) not null,
          HIGHVALUES integer not null,
          constraint hilo_pk primary key (SEQUENCENAME)
       )</command>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
       process_id="3776" session_id="47d96378.ec0" transaction="0"
       time="2008-03-13 13:25:12 " type="connect">
       <message>AUDIT:  connection authorized: user=enterprisedb database=edb</message>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
       process_id="3776" session_id="47d96378.ec0" transaction="1667"
       time="2008-03-13 13:25:17 " type="ddl">
       <command>AUDIT:  statement: CREATE TABLE test (f1 INTEGER);</command>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
       process_id="3776" session_id="47d96378.ec0" transaction="1669"
       time="2008-03-13 13:25:17 " type="sql statement">
       <command>AUDIT:  statement: SELECT * FROM testx;</command>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
       process_id="3776" session_id="47d96378.ec0" transaction="1669"
       time="2008-03-13 13:25:17 " type="error">
       <message>ERROR:  relation &quot;testx&quot; does not exist</message>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
       process_id="3776" session_id="47d96378.ec0" transaction="1670"
       time="2008-03-13 13:25:18 " type="ddl">
       <command>AUDIT:  statement: DROP TABLE test;</command>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
       process_id="3776" session_id="47d96378.ec0" transaction="0"
       time="2008-03-13 13:25:22 " type="disconnect">
       <message>AUDIT:  disconnection: session time: 0:00:10.094 user=enterprisedb
               database=edb host=127.0.0.1 port=1283</message>
</event>
<event user="enterprisedb" database="mgmtsvr" remote_host_and_port="127.0.0.1(1281)"
       process_id="352" session_id="47d96338.160" transaction="0"
       time="2008-03-13 13:25:31 " type="disconnect">
       <message>AUDIT:  disconnection: session time: 0:01:23.046 user=enterprisedb
               database=mgmtsvr host=127.0.0.1 port=1281</message>
</event>
<event process_id="2768" time="2008-03-13 13:25:36 " type="shutdown">
       <message>AUDIT:  database system is shut down</message>
</event>
```


2.3 Unicode Collation Algorithm

The *Unicode Collation Algorithm* (UCA) is a specification (*Unicode Technical Report #10*) that defines a customizable method of collating and comparing Unicode data. *Collation* means how data is sorted as with a `SELECT … ORDER BY` clause. *Comparison* is relevant for searches that use ranges with less than, greater than, or equal to operators.

Customizability is an important factor for various reasons such as the following.

- Unicode supports a vast number of languages. Letters that may be common to several languages may be expected to collate in different orders depending upon the language.
- Characters that appear with letters in certain languages such as accents or umlauts have an impact on the expected collation depending upon the language.
- In some languages, combinations of several consecutive characters should be treated as a single character with regards to its collation sequence.
- There may be certain preferences as to the collation of letters according to case. For example, should the lowercase form of a letter collate before the uppercase form of the same letter or vice versa.
- There may be preferences as to whether punctuation marks such as underscore characters, hyphens, or space characters should be considered in the collating sequence or should they simply be ignored as if they did not exist in the string.

Given all of these variations with the vast number of languages supported by Unicode, there is a necessity for a method to select the specific criteria for determining a collating sequence. This is what the Unicode Collation Algorithm defines.

Note: In addition, another advantage for using ICU collations (the implementation of the Unicode Collation Algorithm) is for performance. Sorting tasks, including B-tree index creation, can complete in less than half the time it takes with a non-ICU collation. The exact performance gain depends on your operating system version, the language of your text data, and other factors.

The following sections provide a brief, simplified explanation of the Unified Collation Algorithm concepts. As the algorithm and its usage are quite complex with numerous variations, refer to the official documents cited in these sections for complete details.

2.3.1 Basic Unicode Collation Algorithm Concepts

The official information for the Unicode Collation Algorithm is specified in *Unicode Technical Report #10*, which can be found on The Unicode Consortium website at:

http://www.unicode.org/reports/tr10/

The ICU – International Components for Unicode also provides much useful information. An explanation of the collation concepts can be found on their website located at:

http://userguide.icu-project.org/collation/concepts

The basic concept behind the Unicode Collation Algorithm is the use of multilevel comparison. This means that a number of levels are defined, which are listed as level 1 through level 5 in the following bullet points. Each level defines a type of comparison. Strings are first compared using the primary level, also called level 1.

If the order can be determined based on the primary level, then the algorithm is done. If the order cannot be determined based on the primary level, then the secondary level, level 2, is applied. If the order can be determined based on the secondary level, then the algorithm is done, otherwise the tertiary level is applied, and so on. There is typically, a final tie-breaking level to determine the order if it cannot be resolved by the prior levels.

- **Level 1 – Primary Level for Base Characters.** The order of basic characters such as letters and digits determines the difference such as `A < B`.
- **Level 2 – Secondary Level for Accents.** If there are no primary level differences, then the presence or absence of accents and other such characters determine the order such as `a < á`.
- **Level 3 – Tertiary Level for Case.** If there are no primary level or secondary level differences, then a difference in case determines the order such as `a < A`.
- **Level 4 – Quaternary Level for Punctuation.** If there are no primary, secondary, or tertiary level differences, then the presence or absence of white space characters, control characters, and punctuation determine the order such as `-A < A`.
- **Level 5 – Identical Level for Tie-Breaking.** If there are no primary, secondary, tertiary, or quaternary level differences, then some other difference such as the code point values determines the order.

2.3.2 International Components for Unicode

The Unicode Collation Algorithm is implemented by open source software provided by the *International Components for Unicode* (ICU). The software is a set of C/C++ and Java libraries.

When Advanced Server is used to create a collation that invokes the ICU components to produce the collation, the result is referred to as an *ICU collation*.

2.3.2.1 Locale Collations

When creating a collation for a locale, a predefined ICU short form name for the given locale is typically provided.

An *ICU short form* is a method of specifying *collation attributes*, which are the properties of a collation. Section 2.3.2.2 provides additional information on collation attributes.

There are predefined ICU short forms for locales. The ICU short form for a locale incorporates the collation attribute settings typically used for the given locale. This simplifies the collation creation process by eliminating the need to specify the entire list of collation attributes for that locale.

The system catalog `pg_catalog.pg_icu_collate_names` contains a list of the names of the ICU short forms for locales. The ICU short form name is listed in column `icu_short_form`.

```
edb=# SELECT icu_short_form, valid_locale FROM pg_icu_collate_names ORDER BY
valid_locale;
 icu_short_form | valid_locale
----------------+--------------
 LAF            | af
 LAR            | ar
 LAS            | as
 LAZ            | az
 LBE            | be
 LBG            | bg
 LBN            | bn
 LBS            | bs
 LBS_ZCYRL      | bs_Cyrl
 LROOT          | ca
 LROOT          | chr
 LCS            | cs
 LCY            | cy
 LDA            | da
 LROOT          | de
 LROOT          | dz
 LEE            | ee
 LEL            | el
```



```
LROOT           | en
LROOT           | en_US
LEN_RUS_VPOSIX | en_US_POSIX
LEO             | eo
LES             | es
LET             | et
LFA             | fa
LFA_RAF         | fa_AF
   .
   .
   .
```

If needed, the default characteristics of an ICU short form for a given locale can be overridden by specifying the collation attributes to override that property. This is discussed in the next section.

2.3.2.2 Collation Attributes

When creating an ICU collation, the desired characteristics of the collation must be specified. As discussed in Section 2.3.2.1, this can typically be done with an ICU short form for the desired locale. However, if more specific information is required, the specification of the collation properties can be done by using *collation attributes*.

Collation attributes define the rules of how characters are to be compared for determining the collation sequence of text strings. As Unicode covers a vast set of languages in numerous variations according to country, territory and culture, these collation attributes are quite complex.

For the complete, precise meaning and usage of collation attributes, see Section 13 "Collator Naming Scheme" on the ICU – International Components for Unicode website at:

http://userguide.icu-project.org/collation/concepts

The following is a brief summary of the collation attributes and how they are specified using the ICU short form method

Each collation attribute is represented by an uppercase letter, which are listed in the following bullet points. The possible valid values for each attribute are given by codes shown within the parentheses. Some codes have general meanings for all attributes. **X** means to set the attribute off. **O** means to set the attribute on. **D** means to set the attribute to its default value.

- **A – Alternate (N, S, D).** Handles treatment of *variable* characters such as white spaces, punctuation marks, and symbols. When set to non-ignorable (N), differences in variable characters are treated with the same importance as differences in letters. When set to shifted (S), then differences in variable characters are of minor importance (that is, the variable character is ignored when comparing base characters).

- **C – Case First (X, L, U, D).** Controls whether a lowercase letter sorts before the same uppercase letter (L), or the uppercase letter sorts before the same lowercase letter (U). Off (X) is typically specified when lowercase first (L) is desired.
- **E – Case Level (X, O, D).** Set in combination with the Strength attribute, the Case Level attribute is used when accents are to be ignored, but not case.
- **F – French Collation (X, O, D).** When set to on, secondary differences (presence of accents) are sorted from the back of the string as done in the French Canadian locale.
- **H – Hiragana Quaternary (X, O, D).** Introduces an additional level to distinguish between the Hiragana and Katakana characters for compatibility with the JIS X 4061 collation of Japanese character strings.
- **N – Normalization Checking (X, O, D).** Controls whether or not text is thoroughly normalized for comparison. Normalization deals with the issue of canonical equivalence of text whereby different code point sequences represent the same character, which then present issues when sorting or comparing such characters. Languages such as Arabic, ancient Greek, Hebrew, Hindi, Thai, or Vietnamese should be used with Normalization Checking set to on.
- **S – Strength (1, 2, 3, 4, I, D).** Maximum collation level used for comparison. Influences whether accents or case are taken into account when collating or comparing strings. Each number represents a level. A setting of I represents identical strength (that is, level 5).
- **T – Variable Top (hexadecimal digits).** Applicable only when the Alternate attribute is not set to non-ignorable (N). The hexadecimal digits specify the highest character sequence that is to be considered ignorable. For example, if white space is to be ignorable, but visible variable characters are not to be ignorable, then Variable Top set to 0020 would be specified along with the Alternate attribute set to S and the Strength attribute set to 3. (The space character is hexadecimal 0020. Other non-visible variable characters such as backspace, tab, line feed, carriage return, etc. have values less than 0020. All visible punctuation marks have values greater than 0020.)

A set of collation attributes and their values is represented by a text string consisting of the collation attribute letter concatenated with the desired attribute value. Each attribute/value pair is joined to the next pair with an underscore character as shown by the following example.

```
AN_CX_EX_FX_HX_NO_S3
```

Collation attributes can be specified along with a locale's ICU short form name to override those default attribute settings of the locale.

The following is an example where the ICU short form named `LROOT` is modified with a number of other collation attribute/value pairs.

```
AN_CX_EX_LROOT_NO_S3
```


In the preceding example, the Alternate attribute (`A`) is set to non-ignorable (`N`). The Case First attribute (`C`) is set to off (`X`). The Case Level attribute (`E`) is set to off (`X`). The Normalization attribute (`N`) is set to on (`O`). The Strength attribute (`S`) is set to the tertiary level `3`. `LROOT` is the ICU short form to which these other attributes are applying modifications.

2.3.3 Creating an ICU Collation

Creating an ICU collation can be done a number of different ways.

- When creating a new database cluster with the `initdb` command, the `--icu-short-form` option can be specified to define the ICU collation to be used by default by all databases in the cluster.
- When creating a new database with the `CREATE DATABASE` command, the `ICU_SHORT_FORM` parameter can be specified to define the ICU collation to be used by default in that database.
- In an existing database, the `CREATE COLLATION` command can be used with the `ICU_SHORT_FORM` parameter to define an ICU collation to be used under specific circumstances such as when assigned with the `COLLATE` clause onto selected columns of certain tables or when appended with the `COLLATE` clause onto an expression such as `ORDER BY` *`expr`* `COLLATE "`*`collation_name`*`"`.

The following describes the various ways of creating an ICU collation.

2.3.3.1 CREATE COLLATION

Use the `ICU_SHORT_FORM` parameter with the `CREATE COLLATION` command to create an ICU collation:

```
CREATE COLLATION collation_name (
  [ LOCALE = locale, ]
  [ LC_COLLATE = lc_collate, ]
  [ LC_CTYPE = lc_ctype, ]
  [ ICU_SHORT_FORM = icu_short_form ]
);
```

To be able to create a collation, you must have `CREATE` privilege on the destination schema where the collation is to reside.

For information about the general usage of the `CREATE COLLATION` command, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-createcollation.html

UTF-8 character encoding of the database is required. Any `LOCALE`, or `LC_COLLATE` and `LC_CTYPE` settings that are accepted with UTF-8 encoding can be used.

Parameters

`collation_name`

The name of the collation to be added. The collation_name may be schema-qualified.

locale

The locale to be used. Short cut for setting `LC_COLLATE` and `LC_TYPE`. If `LOCALE` is specified, then `LC_COLLATE` and `LC_TYPE` must be omitted.

lc_collate

The collation to be used. If `LC_CTYPE` is specified, then `LC_COLLATE` must also be specified and `LOCALE` must be omitted.

lc_ctype

The character classification to be used. If `LC_COLLATE` is specified, then `LC_CTYPE` must also be specified and `LOCALE` must be omitted.

icu_short_form

The text string specifying the collation attributes and their settings. This typically consists of an ICU short form name, possibly appended with additional collation attribute/value pairs. A list of ICU short form names is available from column `icu_short_form` in system catalog `pg_catalog.pg_icu_collate_names`.

Example

The following creates a collation using the `LROOT` ICU short form.

```
edb=# CREATE COLLATION icu_collate_a (LOCALE = 'en_US.UTF8', ICU_SHORT_FORM =
'LROOT');
CREATE COLLATION
```

The definition of the new collation can be seen with the following `psql` command.

```
edb=# \dO
                       List of collations
    Schema     |      Name      |  Collate    |   Ctype     |  ICU
--------------+---------------+------------+------------+-------
 enterprisedb | icu_collate_a | en_US.UTF8 | en_US.UTF8 | LROOT
(1 row)
```

2.3.3.2 CREATE DATABASE

The following is the syntax for creating a database with an ICU collation:

```
CREATE DATABASE database_name
  [ [ WITH ] [ OWNER [=] user_name ]
    [ TEMPLATE [=] template ]
    [ ENCODING [=] encoding ]
    [ LC_COLLATE [=] lc_collate ]
    [ LC_CTYPE [=] lc_ctype ]
    [ TABLESPACE [=] tablespace_name ]
    [ CONNECTION LIMIT [=] connlimit ]
    [ ICU_SHORT_FORM [=] icu_short_form ]];
```

For complete information about the general usage, syntax, and parameters of the `CREATE DATABASE` command, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-createdatabase.html

When using the `CREATE DATABASE` command to create a database using an ICU collation, the `TEMPLATE template0` clause must be specified and the database encoding must be UTF-8.

The following is an example of creating a database using the `LROOT` ICU short form collation, but sorts an uppercase form of a letter before its lowercase counterpart (`CU`) and treats variable characters as non-ignorable (`AN`).

```
CREATE DATABASE collation_db
  TEMPLATE template0
  ENCODING 'UTF8'
  ICU_SHORT_FORM = 'AN_CU_EX_NX_LROOT';
```

The following `psql` command shows the newly created database.

```
edb=# \l collation_db
                                        List of databases
     Name     |    Owner     | Encoding |   Collate   |    Ctype    |        ICU        |
Access privileges
--------------+--------------+----------+-------------+-------------+-------------------+-----
--------------
 collation_db | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT |
(1 row)
```

The following table is created and populated with rows in the database.

```
CREATE TABLE collate_tbl (
    id              INTEGER,
    c2              VARCHAR(2)
);

INSERT INTO collate_tbl VALUES (1, 'A');
INSERT INTO collate_tbl VALUES (2, 'B');
INSERT INTO collate_tbl VALUES (3, 'C');
INSERT INTO collate_tbl VALUES (4, 'a');
INSERT INTO collate_tbl VALUES (5, 'b');
INSERT INTO collate_tbl VALUES (6, 'c');
INSERT INTO collate_tbl VALUES (7, '1');
INSERT INTO collate_tbl VALUES (8, '2');
```



```
INSERT INTO collate_tbl VALUES (9, '.B');
INSERT INTO collate_tbl VALUES (10, '-B');
INSERT INTO collate_tbl VALUES (11, ' B');
```

The following query shows that the uppercase form of a letter sorts before the lowercase form of the same base letter, and in addition, variable characters are taken into account when sorted as they appear at the beginning of the sort list. (The default behavior for `en_US.UTF-8` is to sort the lowercase form of a letter before the uppercase form of the same base letter, and to ignore variable characters.)

```
collation_db=# SELECT id, c2 FROM collate_tbl ORDER BY c2;
 id | c2
----+----
 11 |  B
 10 | -B
  9 | .B
  7 | 1
  8 | 2
  1 | A
  4 | a
  2 | B
  5 | b
  3 | C
  6 | c
(11 rows)
```

2.3.3.3 initdb

A database cluster can be created with a default ICU collation for all databases in the cluster by using the `--icu-short-form` option with the `initdb` command.

For complete information about the general usage, syntax, and parameters of the `initdb` command, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/app-initdb.html

The following illustrates this process.

```
$ su enterprisedb
Password:
$ /opt/PostgresPlus/9.4AS/bin/initdb -U enterprisedb -D /tmp/collation_data -
-encoding UTF8 --icu-short-form 'AN_CU_EX_NX_LROOT'
The files belonging to this database system will be owned by user
"enterprisedb".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.UTF-8".
The default text search configuration will be set to "english".

Data page checksums are disabled.

creating directory /tmp/collation_data ... ok
```



```
creating subdirectories ... ok
    .
    .
    .
Success. You can now start the database server using:

    /opt/PostgresPlus/9.4AS/bin/edb-postgres -D /tmp/collation_data
or
    /opt/PostgresPlus/9.4AS/bin/pg_ctl -D /tmp/collation_data -l logfile
start
```

The following shows the databases created in the cluster which all have an ICU collation of `AN_CU_EX_NX_LROOT`.

```
edb=# \l
                                              List of databases
   Name    |    Owner     | Encoding |   Collate   |    Ctype    |        ICU        |
Access privileges
-----------+--------------+----------+-------------+-------------+-------------------+--------
-----------------------
 edb       | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT |
 postgres  | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT |
 template0 | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT |
=c/enterprisedb              +
           |              |          |             |             |                   |
enterprisedb=CTc/enterprisedb
 template1 | enterprisedb | UTF8     | en_US.UTF-8 | en_US.UTF-8 | AN_CU_EX_NX_LROOT |
=c/enterprisedb              +
           |              |          |             |             |                   |
enterprisedb=CTc/enterprisedb
(4 rows)
```

2.3.4 Using a Collation

A newly defined ICU collation can be used anywhere the `COLLATION "collation_name"` clause can be used in a SQL command such as in the column specifications of the `CREATE TABLE` command or appended to an expression in the `ORDER BY` clause of a `SELECT` command.

The following are some examples of the creation and usage of ICU collations based on the English language in the United States (`en_US.UTF8`).

In these examples, ICU collations are created with the following characteristics.

Collation `icu_collate_lowercase` forces the lowercase form of a letter to sort before its uppercase counterpart (`CL`).

Collation `icu_collate_uppercase` forces the uppercase form of a letter to sort before its lowercase counterpart (`CU`).

Collation `icu_collate_ignore_punct` causes variable characters (white space and punctuation marks) to be ignored during sorting (`AS`).

Collation `icu_collate_ignore_white_sp` causes white space and other non-visible variable characters to be ignored during sorting, but visible variable characters (punctuation marks) are not ignored (`AS`, `T0020`).

```
CREATE COLLATION icu_collate_lowercase (
    LOCALE = 'en_US.UTF8',
    ICU_SHORT_FORM = 'AN_CL_EX_NX_LROOT'
);

CREATE COLLATION icu_collate_uppercase (
    LOCALE = 'en_US.UTF8',
    ICU_SHORT_FORM = 'AN_CU_EX_NX_LROOT'
);

CREATE COLLATION icu_collate_ignore_punct (
    LOCALE = 'en_US.UTF8',
    ICU_SHORT_FORM = 'AS_CX_EX_NX_LROOT_L3'
);

CREATE COLLATION icu_collate_ignore_white_sp (
    LOCALE = 'en_US.UTF8',
    ICU_SHORT_FORM = 'AS_CX_EX_NX_LROOT_L3_T0020'
);
```

Note: When creating collations, ICU may generate notice and warning messages when attributes are given to modify the `LROOT` collation.

The following `psql` command lists the collations.

```
edb=# \dO
                                         List of collations
    Schema     |             Name             |  Collate   |   Ctype    |             ICU
---------------+------------------------------+------------+------------+----------------------
-----
 enterprisedb | icu_collate_ignore_punct     | en_US.UTF8 | en_US.UTF8 | AS_CX_EX_NX_LROOT_L3
 enterprisedb | icu_collate_ignore_white_sp | en_US.UTF8 | en_US.UTF8 |
AS_CX_EX_NX_LROOT_L3_T0020
 enterprisedb | icu_collate_lowercase        | en_US.UTF8 | en_US.UTF8 | AN_CL_EX_NX_LROOT
 enterprisedb | icu_collate_uppercase        | en_US.UTF8 | en_US.UTF8 | AN_CU_EX_NX_LROOT
(4 rows)
```

The following table is created and populated.

```
CREATE TABLE collate_tbl (
    id              INTEGER,
    c2              VARCHAR(2)
);

INSERT INTO collate_tbl VALUES (1, 'A');
INSERT INTO collate_tbl VALUES (2, 'B');
INSERT INTO collate_tbl VALUES (3, 'C');
INSERT INTO collate_tbl VALUES (4, 'a');
INSERT INTO collate_tbl VALUES (5, 'b');
INSERT INTO collate_tbl VALUES (6, 'c');
INSERT INTO collate_tbl VALUES (7, '1');
INSERT INTO collate_tbl VALUES (8, '2');
INSERT INTO collate_tbl VALUES (9, '.B');
INSERT INTO collate_tbl VALUES (10, '-B');
INSERT INTO collate_tbl VALUES (11, ' B');
```

The following query sorts on column `c2` using the default collation. Note that variable characters (white space and punctuation marks) with `id` column values of `9`, `10`, and `11` are ignored and sort with the letter `B`.

```
edb=# SELECT * FROM collate_tbl ORDER BY c2;
 id | c2
----+----
  7 | 1
  8 | 2
  4 | a
  1 | A
  5 | b
  2 | B
 11 |  B
 10 | -B
  9 | .B
  6 | c
  3 | C
(11 rows)
```

The following query sorts on column `c2` using collation `icu_collate_lowercase`, which forces the lowercase form of a letter to sort before the uppercase form of the same base letter. Also note that the `AN` attribute forces variable characters to be included in the sort order at the same level when comparing base characters so rows with `id` values of `9`, `10`, and `11` appear at the beginning of the sort list before all letters and numbers.

```
edb=# SELECT * FROM collate_tbl ORDER BY c2 COLLATE "icu_collate_lowercase";
 id | c2
----+----
 11 |  B
 10 | -B
  9 | .B
  7 | 1
  8 | 2
  4 | a
  1 | A
  5 | b
  2 | B
  6 | c
  3 | C
(11 rows)
```

The following query sorts on column `c2` using collation `icu_collate_uppercase`, which forces the uppercase form of a letter to sort before the lowercase form of the same base letter.

```
edb=# SELECT * FROM collate_tbl ORDER BY c2 COLLATE "icu_collate_uppercase";
 id | c2
----+----
 11 |  B
 10 | -B
  9 | .B
  7 | 1
  8 | 2
  1 | A
  4 | a
  2 | B
  5 | b
  3 | C
  6 | c
(11 rows)
```

The following query sorts on column `c2` using collation `icu_collate_ignore_punct`, which causes variable characters to be ignored so rows with `id` values of `9`, `10`, and `11` sort with the letter `B` as that is the character immediately following the ignored variable character.

```
edb=# SELECT * FROM collate_tbl ORDER BY c2 COLLATE
"icu_collate_ignore_punct";
 id | c2
----+----
  7 | 1
  8 | 2
  4 | a
  1 | A
  5 | b
 11 |  B
  2 | B
  9 | .B
 10 | -B
```

```
  6 | c
  3 | C
(11 rows)
```

The following query sorts on column `c2` using collation `icu_collate_ignore_white_sp`. The `AS` and `T0020` attributes of the collation cause variable characters with code points less than or equal to hexadecimal `0020` to be ignored while variable characters with code points greater than hexadecimal `0020` are included in the sort.

The row with `id` value of `11`, which starts with a space character (hexadecimal `0020`) sorts with the letter `B`. The rows with `id` values of `9` and `10`, which start with visible punctuation marks greater than hexadecimal `0020`, appear at the beginning of the sort list as these particular variable characters are included in the sort order at the same level when comparing base characters.

```
edb=# SELECT * FROM collate_tbl ORDER BY c2 COLLATE
"icu_collate_ignore_white_sp";
 id | c2
----+----
 10 | -B
  9 | .B
  7 | 1
  8 | 2
  4 | a
  1 | A
  5 | b
 11 |  B
  2 | B
  6 | c
  3 | C
(11 rows)
```


2.4 Profile Management

Advanced Server 9.5 allows a database superuser to create named *profiles*. Each profile defines rules for password management that augment `password` and `md5` authentication. The rules in a profile can:

- count failed login attempts
- lock an account due to excessive failed login attempts
- mark a password for expiration
- define a grace period after a password expiration
- define rules for password complexity
- define rules that limit password re-use

A profile is a named set of password attributes that allow you to easily manage a group of roles that share comparable authentication requirements. If the password requirements change, you can modify the profile to have the new requirements applied to each user that is associated with that profile.

After creating the profile, you can associate the profile with one or more users. When a user connects to the server, the server enforces the profile that is associated with their login role. Profiles are shared by all databases within a cluster, but each cluster may have multiple profiles. A single user with access to multiple databases will use the same profile when connecting to each database within the cluster.

Advanced Server 9.5 creates a profile named `default` that is associated with a new role when the role is created unless an alternate profile is specified. If you upgrade to Advanced Server 9.5 from a previous server version, existing roles will automatically be assigned to the `default` profile. You cannot delete the `default` profile.

The `default` profile specifies the following attributes:

```
FAILED_LOGIN_ATTEMPTS      UNLIMITED
PASSWORD_LOCK_TIME         UNLIMITED
PASSWORD_LIFE_TIME         UNLIMITED
PASSWORD_GRACE_TIME        UNLIMITED
PASSWORD_REUSE_TIME        UNLIMITED
PASSWORD_REUSE_MAX         UNLIMITED
PASSWORD_VERIFY_FUNCTION   NULL
```

A database superuser can use the `ALTER PROFILE` command to modify the values specified by the `default` profile. For more information about modifying a profile, see Section 2.4.2.

2.4.1 Creating a New Profile

Use the `CREATE PROFILE` command to create a new profile. The syntax is:

```
CREATE PROFILE profile_name
       [LIMIT {parameter value} ... ];
```

Include the `LIMIT` clause and one or more space-delimited *`parameter/value`* pairs to specify the rules enforced by Advanced Server.

Parameters:

`profile_name` specifies the name of the profile.

`parameter` specifies the attribute limited by the profile.

`value` specifies the parameter limit.

Advanced Server supports the *`values`* shown in the following table for each *`parameter`*:

Parameter Name **---------------** **Description**	**Supported Values**
`FAILED_LOGIN_ATTEMPTS` Specifies the number of failed login attempts that a user may make before the server locks the user out of their account for the length of time specified by `PASSWORD_LOCK_TIME`.	• An `INTEGER` value greater than `0`. • `DEFAULT` - the value of `FAILED_LOGIN_ATTEMPTS` specified in the `DEFAULT` profile. • `UNLIMITED` – The connecting user may make an unlimited number of failed login attempts.
`PASSWORD_LOCK_TIME` Specifies the length of time that must pass before the server unlocks an account that has been locked because of	• A `NUMERIC` value greater than or equal to `0`. To specify a fractional portion of a day, specify a decimal value. For example, use the value `4.5` to specify `4` days, `12` hours. • `DEFAULT` - the value of `PASSWORD_LOCK_TIME` specified in the

FAILED_LOGIN_ATTEMPTS.	DEFAULT profile. • UNLIMITED – The account is locked until it is manually unlocked by a database superuser.
PASSWORD_LIFE_TIME Specifies the number of days that the current password may be used before the user is prompted to provide a new password. Include the PASSWORD_GRACE_TIME clause when using the PASSWORD_LIFE_TIME clause to specify the number of days that will pass after the password expires before connections by the role are rejected. If PASSWORD_GRACE_TIME is not specified, the password will expire on the day specified by the default value of PASSWORD_GRACE_TIME, and the user will not be allowed to execute any command until a new password is provided.	• A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours. • DEFAULT - the value of PASSWORD_LIFE_TIME specified in the DEFAULT profile. • UNLIMITED – The password does not have an expiration date.
PASSWORD_GRACE_TIME Specifies the length of the grace period after a password expires until the user is forced to change their password. When the grace period expires, a user will be allowed to connect, but will not be allowed to execute any command until they update their expired password.	• A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours. • DEFAULT - the value of PASSWORD_GRACE_TIME specified in the DEFAULT profile. • UNLIMITED – The grace period is infinite.
PASSWORD_REUSE_TIME Use PASSWORD_REUSE_TIME to specify the number of days a user must wait before re-using a password. The PASSWORD_REUSE_TIME and	• A NUMERIC value greater than or equal to 0. To specify a fractional portion of a day, specify a decimal value. For example, use the value 4.5 to specify 4 days, 12 hours. • DEFAULT - the value of PASSWORD_REUSE_TIME specified in the DEFAULT profile.

PASSWORD_REUSE_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse.	UNLIMITED – The password can be re-used without restrictions.
PASSWORD_REUSE_MAX Use PASSWORD_REUSE_MAX to specify the number of password changes that must occur before a password can be reused. The PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX parameters are intended to be used together. If you specify a finite value for one of these parameters while the other is UNLIMITED, old passwords can never be reused. If both parameters are set to UNLIMITED there are no restrictions on password reuse.	• An INTEGER value greater than or equal to 0. • DEFAULT - the value of PASSWORD_REUSE_MAX specified in the DEFAULT profile. • UNLIMITED – The password can be re-used without restrictions.
PASSWORD_VERIFY_FUNCTION Use PASSWORD_VERIFY_FUNCTION to specify password complexity.	• The name of a PL/SQL function. • DEFAULT - the value of PASSWORD_VERIFY_FUNCTION specified in the DEFAULT profile. • NULL

Notes

Use DROP PROFILE command to remove the profile.

Examples

The following command creates a profile named acctg. The profile specifies that if a user has not authenticated with the correct password in five attempts, the account will be locked for one day:

```
CREATE PROFILE acctg LIMIT
       FAILED_LOGIN_ATTEMPTS 5
       PASSWORD_LOCK_TIME 1;
```

The following command creates a profile named `sales`. The profile specifies that a user must change their password every 90 days:

```
CREATE PROFILE sales LIMIT
       PASSWORD_LIFE_TIME 90
       PASSWORD_GRACE_TIME 3;
```

If the user has not changed their password before the 90 days specified in the profile has passed, they will be issued a warning at login. After a grace period of 3 days, their account will not be allowed to invoke any commands until they change their password.

The following command creates a profile named `accts`. The profile specifies that a user cannot re-use a password within 180 days of the last use of the password, and must change their password at least 5 times before re-using the password:

```
CREATE PROFILE accts LIMIT
       PASSWORD_REUSE_TIME 180
       PASSWORD_REUSE_MAX 5;
```

The following command creates a profile named `resources`; the profile calls a user-defined function named `password_rules` that will verify that the password provided meets their standards for complexity:

```
CREATE PROFILE resources LIMIT
       PASSWORD_VERIFY_FUNCTION password_rules;
```

2.4.1.1 Creating a Password Function

When specifying `PASSWORD_VERIFY_FUNCTION,` you can provide a customized function that specifies the security rules that will be applied when your users change their password. For example, you can specify rules that stipulate that the new password must be at least *n* characters long, and may not contain a specific value.

The password function has the following signature:

```
function_name (user_name VARCHAR2,
               new_password VARCHAR2,
               old_password VARCHAR2) RETURN boolean
```

Where:

user_name is the name of the user.

new_password is the new password.

old_password is the user's previous password. If you reference this parameter within your function:

When a database superuser changes their password, the third parameter will always be NULL.

When a user with the CREATEROLE attribute changes their password, the parameter will pass the previous password if the statement includes the REPLACE clause. Note that the REPLACE clause is optional syntax for a user with the CREATEROLE privilege.

When a user that is not a database superuser and does not have the CREATEROLE attribute changes their password, the third parameter will contain the previous password for the role.

The function returns a Boolean value. If the function returns true and does not raise an exception, the password is accepted; if the function returns false or raises an exception, the password is rejected. If the function raises an exception, the specified error message is displayed to the user. If the function does not raise an exception, but returns false, the following error message is displayed:

```
ERROR:  password verification for the specified password failed
```

The function must be owned by a database superuser, and reside in the sys schema.

Example:

The following example creates a profile and a custom function; then, the function is associated with the profile. The following CREATE PROFILE command creates a profile named acctg_pwd_profile:

```
CREATE PROFILE acctg_pwd_profile;
```

The following commands create a (schema-qualified) function named verify_password:

```
CREATE OR REPLACE FUNCTION sys.verify_password(user_name varchar2,
new_password varchar2, old_password varchar2)
RETURN boolean IMMUTABLE
IS
BEGIN
  IF (length(new_password) < 5)
  THEN
    raise_application_error(-20001, 'too short');
  END IF;

  IF substring(new_password FROM old_password) IS NOT NULL
```

```
  THEN
    raise_application_error(-20002, 'includes old password');
  END IF;

  RETURN true;
END;
```

The function first ensures that the password is at least 5 characters long, and then compares the new password to the old password. If the new password contains fewer than 5 characters, or contains the old password, the function raises an error.

The following statement sets the ownership of the `verify_password` function to the `enterprisedb` database superuser:

```
ALTER FUNCTION verify_password(varchar2, varchar2, varchar2) OWNER TO
enterprisedb;
```

Then, the `verify_password` function is associated with the profile:

```
ALTER PROFILE acctg_pwd_profile LIMIT PASSWORD_VERIFY_FUNCTION
verify_password;
```

The following statements confirm that the function is working by first creating a test user (`alice`), and then attempting to associate invalid and valid passwords with her role:

```
CREATE ROLE alice WITH LOGIN PASSWORD 'temp_password' PROFILE
acctg_pwd_profile;
```

Then, when `alice` connects to the database and attempts to change her password, she must adhere to the rules established by the profile function. A non-superuser without `CREATEROLE` must include the `REPLACE` clause when changing a password:

```
edb=> ALTER ROLE alice PASSWORD 'hey';
ERROR:  missing REPLACE clause
```

The new password must be at least 5 characters long:

```
edb=> ALTER USER alice PASSWORD 'hey' REPLACE 'temp_password';
ERROR:  EDB-20001: too short
CONTEXT:  edb-spl function verify_password(character varying,character
varying,character varying) line 5 at procedure/function invocation statement
```

If the new password is acceptable, the command completes without error:

```
edb=> ALTER USER alice PASSWORD 'hello' REPLACE 'temp_password';
ALTER ROLE
```

If `alice` decides to change her password, the new password must not contain the old password:

```
edb=> ALTER USER alice PASSWORD 'helloworld' REPLACE 'hello';
```

```
ERROR:  EDB-20002: includes old password
CONTEXT:  edb-spl function verify_password(character varying,character
varying,character varying) line 10 at procedure/function invocation statement
```

To remove the verify function, set `password_verify_function` to `NULL`:

```
ALTER PROFILE acctg_pwd_profile LIMIT password_verify_function NULL;
```

Then, all password constraints will be lifted:

```
edb=# ALTER ROLE alice PASSWORD 'hey';
ALTER ROLE
```

2.4.2 Altering a Profile

Use the `ALTER PROFILE` command to modify a user-defined profile; Advanced Server supports two forms of the command:

```
ALTER PROFILE profile_name RENAME TO new_name;

ALTER PROFILE profile_name
      LIMIT {parameter value}[...];
```

Include the `LIMIT` clause and one or more space-delimited *parameter/value* pairs to specify the rules enforced by Advanced Server, or use `ALTER PROFILE…RENAME TO` to change the name of a profile.

Parameters:

profile_name specifies the name of the profile.

new_name specifies the new name of the profile.

parameter specifies the attribute limited by the profile.

value specifies the parameter limit.

See the table in Section 2.4.1 for a complete list of accepted parameter/value pairs.

Examples

The following example modifies a profile named `acctg_profile`:

```
ALTER PROFILE acctg_profile
       LIMIT FAILED_LOGIN_ATTEMPTS 3 PASSWORD_LOCK_TIME 1;
```

`acctg_profile` will count failed connection attempts when a login role attempts to connect to the server. The profile specifies that if a user has not authenticated with the correct password in three attempts, the account will be locked for one day.

The following example changes the name of `acctg_profile` to `payables_profile`:

```
ALTER PROFILE acctg_profile RENAME TO payables_profile;
```

2.4.3 Dropping a Profile

Use the DROP PROFILE command to drop a profile. The syntax is:

```
DROP PROFILE [IF EXISTS] profile_name [CASCADE|RESTRICT];
```

Include the IF EXISTS clause to instruct the server to not throw an error if the specified profile does not exist. The server will issue a notice if the profile does not exist.

Include the optional CASCADE clause to reassign any users that are currently associated with the profile to the default profile, and then drop the profile. Include the optional RESTRICT clause to instruct the server to not drop any profile that is associated with a role. This is the default behavior.

Parameters

profile_name

The name of the profile being dropped.

Examples

The following example drops a profile named acctg_profile:

```
DROP PROFILE acctg_profile CASCADE;
```

The command first re-associates any roles associated with the acctg_profile profile with the default profile, and then drops the acctg_profile profile.

The following example drops a profile named acctg_profile:

```
DROP PROFILE acctg_profile RESTRICT;
```

The RESTRICT clause in the command instructs the server to not drop acctg_profile if there are any roles associated with the profile.

2.4.4 Associating a Profile with an Existing Role

After creating a profile, you can use the `ALTER USER… PROFILE` or `ALTER ROLE… PROFILE` command to associate the profile with a role. The command syntax related to profile management functionality is:

```
ALTER USER|ROLE name [[WITH] option[…]
```

where `option` can be the following Oracle-compatible clauses:

```
PROFILE profile_name

    | ACCOUNT {LOCK|UNLOCK}
    | PASSWORD EXPIRE [AT 'timestamp']
```

or `option` can be the following non-compatible clauses:

```
    | PASSWORD SET AT 'timestamp'
    | LOCK TIME 'timestamp'
    | STORE PRIOR PASSWORD {'password' 'timestamp} [, ...]
```

For information about the administrative clauses of the `ALTER USER` or `ALTER ROLE` command that are supported by Advanced Server, please see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-commands.html

Only a database superuser can use the `ALTER USER|ROLE` clauses that enforce profile management. The clauses enforce the following behaviors:

Include the `PROFILE` clause and a *`profile_name`* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.

Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to specify that the user account should be placed in a locked or unlocked state.

Include the `LOCK TIME` *`'timestamp'`* clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the `PASSWORD_LOCK_TIME` parameter of the profile assigned to this role. If `LOCK TIME` is used with the `ACCOUNT LOCK` clause, the role can only be unlocked by a database superuser with the `ACCOUNT UNLOCK` clause.

Include the `PASSWORD EXPIRE` clause with the `AT 'timestamp'` keywords to specify a date/time when the password associated with the role will expire. If you omit the `AT 'timestamp'` keywords, the password will expire immediately.

Include the `PASSWORD SET AT 'timestamp'` keywords to set the password modification date to the time specified.

Include the `STORE PRIOR PASSWORD {'password' 'timestamp} [, ...]` clause to modify the password history, adding the new password and the time the password was set.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the `profile` column of the `DBA_USERS` view.

Parameters

name

The name of the role with which the specified profile will be associated.

password

The password associated with the role.

profile_name

The name of the profile that will be associated with the role.

timestamp

The date and time at which the clause will be enforced. When specifying a value for *timestamp*, enclose the value in single-quotes.

Examples

The following command uses the `ALTER USER… PROFILE` command to associate a profile named `acctg` with a user named `john`:

```
ALTER USER john PROFILE acctg_profile;
```

The following command uses the `ALTER ROLE… PROFILE` command to associate a profile named `acctg` with a user named `john`:

```
ALTER ROLE john PROFILE acctg_profile;
```

2.4.5 Unlocking a Locked Account

A database superuser can use clauses of the `ALTER USER|ROLE…` command to lock or unlock a role. The syntax is:

```
ALTER USER|ROLE name
      ACCOUNT {LOCK|UNLOCK}
      LOCK TIME 'timestamp'
```

Include the `ACCOUNT LOCK` clause to lock a role immediately; when locked, a role's `LOGIN` functionality is disabled. When you specify the `ACCOUNT LOCK` clause without the `LOCK TIME` clause, the state of the role will not change until a superuser uses the `ACCOUNT UNLOCK` clause to unlock the role.

Use the `ACCOUNT UNLOCK` clause to unlock a role.

Use the `LOCK TIME 'timestamp'` clause to instruct the server to lock the account at the time specified by the given timestamp for the length of time specified by the `PASSWORD_LOCK_TIME` parameter of the profile associated with this role.

Combine the `LOCK TIME 'timestamp'` clause and the `ACCOUNT LOCK` clause to lock an account at a specified time until the account is unlocked by a superuser invoking the `ACCOUNT UNLOCK` clause.

Parameters

name

> The name of the role that is being locked or unlocked.

timestamp

> The date and time at which the role will be locked. When specifying a value for *timestamp*, enclose the value in single-quotes.

Note

This command (available only in Advanced Server) is implemented to support Oracle-styled profile management.

Examples

The following example uses the `ACCOUNT LOCK` clause to lock the role named `john`. The account will remain locked until the account is unlocked with the `ACCOUNT UNLOCK` clause:

```
ALTER ROLE john ACCOUNT LOCK;
```

The following example uses the `ACCOUNT UNLOCK` clause to unlock the role named `john`:

```
ALTER USER john ACCOUNT UNLOCK;
```

The following example uses the `LOCK TIME 'timestamp'` clause to lock the role named `john` on September 4, 2015:

```
ALTER ROLE john LOCK TIME ‘September 4 12:00:00 2015’;
```

The role will remain locked for the length of time specified by the `PASSWORD_LOCK_TIME` parameter.

The following example combines the `LOCK TIME 'timestamp'` clause and the `ACCOUNT LOCK` clause to lock the role named `john` on September 4, 2015:

```
ALTER ROLE john LOCK TIME ‘September 4 12:00:00 2015’ ACCOUNT LOCK;
```

The role will remain locked until a database superuser uses the `ACCOUNT UNLOCK` command to unlock the role.

2.4.6 Creating a New Role Associated with a Profile

A database superuser can use clauses of the `CREATE USER|ROLE` command to assign a named profile to a role when creating the role, or to specify profile management details for a role. The command syntax related to profile management functionality is:

```
CREATE USER|ROLE name [[WITH] option […]]
```

where `option` can be the following Oracle-compatible clauses:

```
PROFILE profile_name
    |  ACCOUNT {LOCK|UNLOCK}
    |  PASSWORD EXPIRE [AT 'timestamp']
```

or `option` can be the following non-compatible clauses:

```
    |  LOCK TIME 'timestamp'
```

For information about the administrative clauses of the `CREATE USER` or `CREATE ROLE` command that are supported by Advanced Server, please see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-commands.html

`CREATE ROLE|USER… PROFILE` adds a new role with an associated profile to an Advanced Server database cluster.

Roles created with the `CREATE USER` *command are (by default) login roles. Roles created with the* `CREATE ROLE` *command are (by default) not login roles. To create a login account with the* `CREATE ROLE` *command, you must include the* `LOGIN` *keyword.*

Only a database superuser can use the `CREATE USER|ROLE` clauses that enforce profile management; these clauses enforce the following behaviors:

Include the `PROFILE` clause and a *`profile_name`* to associate a pre-defined profile with a role, or to change which pre-defined profile is associated with a user.

Include the `ACCOUNT` clause and the `LOCK` or `UNLOCK` keyword to specify that the user account should be placed in a locked or unlocked state.

Include the `LOCK TIME` *`'timestamp'`* clause and a date/time value to lock the role at the specified time, and unlock the role at the time indicated by the `PASSWORD_LOCK_TIME` parameter of the profile assigned to this role. If `LOCK`

TIME is used with the ACCOUNT LOCK clause, the role can only be unlocked by a database superuser with the ACCOUNT UNLOCK clause.

Include the PASSWORD EXPIRE clause with the optional AT '*timestamp*' keywords to specify a date/time when the password associated with the role will expire. If you omit the AT '*timestamp*' keywords, the password will expire immediately.

Each login role may only have one profile. To discover the profile that is currently associated with a login role, query the profile column of the DBA_USERS view.

Parameters

name

The name of the role.

profile_name

The name of the profile associated with the role.

timestamp

The date and time at which the clause will be enforced. When specifying a value for *timestamp*, enclose the value in single-quotes.

Examples

The following example uses CREATE USER to create a login role named john who is associated with the acctg_profile profile:

```
CREATE USER john PROFILE acctg_profile IDENTIFIED BY "1safepwd";
```

john can log in to the server, using the password 1safepwd.

The following example uses CREATE ROLE to create a login role named john who is associated with the acctg_profile profile:

```
CREATE ROLE john PROFILE acctg_profile LOGIN PASSWORD "1safepwd";
```

john can log in to the server, using the password 1safepwd.

2.4.7 Backing up Profile Management Functions

A profile may include a PASSWORD_VERIFY_FUNCTION clause that refers to a user-defined function that specifies the behavior enforced by Advanced Server. Profiles are global objects; they are shared by all of the databases within a cluster. While profiles are global objects, user-defined functions are database objects.

Invoking pg_dumpall with the -g or -r option will create a script that recreates the definition of any existing profiles, but that does not recreate the user-defined functions that are referred to by the PASSWORD_VERIFY_FUNCTION clause. You should use the pg_dump utility to explicitly dump (and later restore) the database in which those functions reside.

The script created by pg_dump will contain a command that includes the clause and function name:

```
ALTER PROFILE… LIMIT PASSWORD_VERIFY_FUNCTION function_name
```

to associate the restored function with the profile with which it was previously associated.

If the PASSWORD_VERIFY_FUNCTION clause is set to DEFAULT or NULL, the behavior will be replicated by the script generated by the pg_dumpall -g or pg_dumpall -r command.

3 Enhanced SQL Features

Advanced Server includes enhanced SQL functionality that provides additional flexibility and convenience.

This chapter describes these additions.

3.1 Synonyms

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

Postgres Plus Advanced Server supports synonyms for:

- Tables
- Views
- Sequences
- Procedures
- Functions
- Types
- Other synonyms

Neither the referenced schema or referenced object must exist at the time that you create the synonym; a synonym may refer to a non-existent object or schema. A synonym will become invalid if you drop the referenced object or schema. You must explicitly drop a synonym to remove it.

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

When Advanced Server executes an SQL command, the privileges of the current user are checked against the synonym's underlying database object; if the user does not have the proper permissions for that object, the SQL command will fail.

Creating a Synonym

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

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

Parameters:

syn_name

> *syn_name* is the name of the synonym. A synonym name must be unique within a schema.

schema

> *schema* specifies the name of the schema that the synonym resides in. If you do not specify a schema name, the synonym is created in the first existing schema in your search path.

object_name

> *object_name* specifies the name of the object.

object_schema

> *object_schema* specifies the name of the schema that the object resides in.

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

Include the `PUBLIC` clause to create the synonym in the `public` schema. The `CREATE PUBLIC SYNONYM` command creates a synonym that resides in the `public` schema:

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

This just a shorthand way to write:

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

The following example creates a synonym named `personnel` that refers to the `enterprisedb.emp` table.

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

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

```
SHOW SEARCH_PATH;

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

In our example, if a schema named `development` does not exist, the synonym will be created in the schema named `accounting`.

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

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

SELECT * FROM personnel;

 empno |  ename   |    job    | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+----------+-----------+------+--------------------+---------+---------+--------
  7369 | SMITH    | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |         |     20
  7499 | ALLEN    | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 |  300.00 |     30
  7521 | WARD     | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250.00 |  500.00 |     30
  7566 | JONES    | MANAGER   | 7839 | 02-APR-81 00:00:00 | 2975.00 |         |     20
  7654 | MARTIN   | SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1250.00 | 1400.00 |     30
  7698 | BLAKE    | MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850.00 |         |     30
  7782 | CLARK    | MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450.00 |         |     10
  7788 | SCOTT    | ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000.00 |         |     20
  7839 | KING     | PRESIDENT |      | 17-NOV-81 00:00:00 | 5000.00 |         |     10
  7844 | TURNER   | SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500.00 |    0.00 |     30
  7876 | ADAMS    | CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |         |     20
  7900 | JAMES    | CLERK     | 7698 | 03-DEC-81 00:00:00 |  950.00 |         |     30
  7902 | FORD     | ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |         |     20
  7934 | MILLER   | CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |         |     10
  8142 | ANDERSON | CLERK     | 7902 | 17-DEC-06 00:00:00 | 1300.00 |         |     20
(15 rows)
```

Deleting a Synonym

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

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

Parameters:

syn_name

> *syn_name* is the name of the synonym. A synonym name must be unique within a schema.

schema

> *schema* specifies the name of the schema in which the synonym resides.

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

You can optionally include the `PUBLIC` clause to drop a synonym that resides in the `public` schema. The `DROP PUBLIC SYNONYM` command drops a synonym that resides in the `public` schema:

```
DROP PUBLIC SYNONYM syn_name;
```

The following example drops the synonym, `personnel`:

```
DROP SYNONYM personnel;
```

3.2 Hierarchical Queries

A *hierarchical query* is a type of query that returns the rows of the result set in a hierarchical order based upon data forming a parent-child relationship. A hierarchy is typically represented by an inverted tree structure. The tree is comprised of interconnected *nodes*. Each node may be connected to none, one, or multiple *child* nodes. Each node is connected to one *parent* node except for the top node which has no parent. This node is the *root* node. Each tree has exactly one root node. Nodes that don't have any children are called *leaf* nodes. A tree always has at least one leaf node - e.g., the trivial case where the tree is comprised of a single node. In this case it is both the root and the leaf.

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

Note: It is possible that a single, given row may appear in more than one tree and thus appear more than once in the result set.

The hierarchical relationship in a query is described by the `CONNECT BY` clause which forms the basis of the order in which rows are returned in the result set. The context of where the `CONNECT BY` clause and its associated optional clauses appear in the `SELECT` command is shown below.

```
SELECT select_list FROM table_expression [ WHERE ...]
  [ START WITH start_expression ]
    CONNECT BY { PRIOR parent_expr = child_expr |
      child_expr = PRIOR parent_expr }
  [ ORDER SIBLINGS BY column1 [ ASC | DESC ]
      [, column2 [ ASC | DESC ] ] ...
  [ GROUP BY ...]
  [ HAVING ...]
  [ other ...]
```

`select_list` is one or more expressions that comprise the fields of the result set. *`table_expression`* is one or more tables or views from which the rows of the result set originate. *`other`* is any additional legal `SELECT` command clauses. The clauses pertinent to hierarchical queries, `START WITH`, `CONNECT BY`, and `ORDER SIBLINGS BY` are described in the following sections.

Note: At this time, Advanced Server does not support the use of `AND` (or other operators) in the `CONNECT BY` clause.

3.2.1 Defining the Parent/Child Relationship

For any given row, its parent and its children are determined by the `CONNECT BY` clause. The `CONNECT BY` clause must consist of two expressions compared with the equals (=) operator. In addition, one of these two expressions must be preceded by the keyword, `PRIOR`.

For any given row, to determine its children:

- Evaluate *parent_expr* on the given row
- Evaluate *child_expr* on any other row resulting from the evaluation of *table_expression*
- If *parent_expr = child_expr*, then this row is a child node of the given parent row
- Repeat the process for all remaining rows in *table_expression*. All rows that satisfy the equation in step 3 are the children nodes of the given parent row.

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

By iteratively repeating this process treating each child node found in the prior steps as a parent, an inverted tree of nodes is constructed. The process is complete when the final set of child nodes has no children of their own - these are the leaf nodes.

A `SELECT` command that includes a `CONNECT BY` clause typically includes the `START WITH` clause. The `START WITH` clause determines the rows that are to be the root nodes - i.e., the rows that are the initial parent nodes upon which the algorithm described previously is to be applied. This is further explained in the following section.

3.2.2 Selecting the Root Nodes

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

3.2.3 Organization Tree in the Sample Application

Consider the `emp` table of the sample application. The rows of the `emp` table form a hierarchy based upon the `mgr` column which contains the employee number of the employee's manager. Each employee has at most, one manager. `KING` is the president of the company so he has no manager, therefore KING's `mgr` column is null. Also, it is possible for an employee to act as a manager for more than one employee. This relationship forms a typical, tree-structured, hierarchical organization chart as illustrated below.

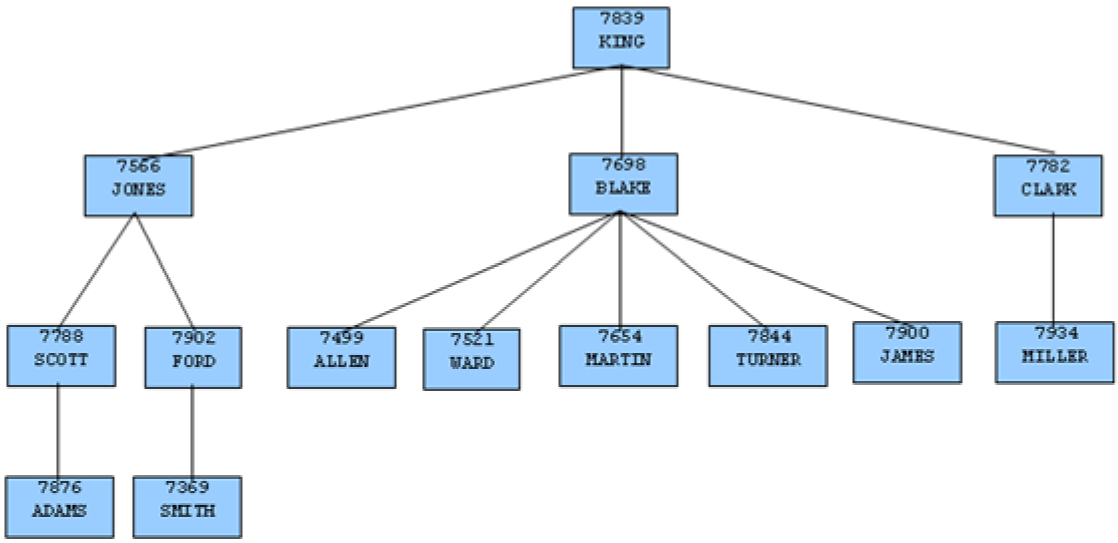


Figure 1 Employee Organization Hierarchy

To form a hierarchical query based upon this relationship, the `SELECT` command includes the clause, `CONNECT BY PRIOR empno = mgr`. For example, given the company president, `KING`, with employee number `7839`, any employee whose `mgr` column is `7839` reports directly to `KING` which is true for `JONES`, `BLAKE`, and `CLARK` (these are the child nodes of `KING`). Similarly, for employee, `JONES`, any other employee with `mgr` column equal to `7566` is a child node of `JONES` - these are `SCOTT` and `FORD` in this example.

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

The complete `SELECT` command is shown below.

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

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

```
 ename  | empno | mgr
--------+-------+------
 KING   |  7839 |
 JONES  |  7566 | 7839
```



```
 SCOTT  |  7788 | 7566
 ADAMS  |  7876 | 7788
 FORD   |  7902 | 7566
 SMITH  |  7369 | 7902
 BLAKE  |  7698 | 7839
 ALLEN  |  7499 | 7698
 WARD   |  7521 | 7698
 MARTIN |  7654 | 7698
 TURNER |  7844 | 7698
 JAMES  |  7900 | 7698
 CLARK  |  7782 | 7839
 MILLER |  7934 | 7782
(14 rows)
```

3.2.4 Node Level

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

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

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

The output from this query follows.

```
 level |  employee    | empno | mgr
-------+-------------+-------+------
     1 | KING        |  7839 |
     2 |   JONES     |  7566 | 7839
     3 |     SCOTT   |  7788 | 7566
     4 |       ADAMS |  7876 | 7788
     3 |     FORD    |  7902 | 7566
     4 |       SMITH |  7369 | 7902
     2 |   BLAKE     |  7698 | 7839
     3 |     ALLEN   |  7499 | 7698
     3 |     WARD    |  7521 | 7698
     3 |     MARTIN  |  7654 | 7698
     3 |     TURNER  |  7844 | 7698
     3 |     JAMES   |  7900 | 7698
     2 |   CLARK     |  7782 | 7839
     3 |     MILLER  |  7934 | 7782
(14 rows)
```

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

3.2.5 Ordering the Siblings

The result set can be ordered so the siblings appear in ascending or descending order by selected column value(s) using the ORDER SIBLINGS BY clause. This is a special case of the ORDER BY clause that can be used only with hierarchical queries.

The previous query is further modified with the addition of ORDER SIBLINGS BY ename ASC.

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

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

```
 level |   employee    | empno | mgr
-------+---------------+-------+------
     1 | KING          |  7839 |
     2 |   BLAKE       |  7698 | 7839
     3 |     ALLEN     |  7499 | 7698
     3 |     JAMES     |  7900 | 7698
     3 |     MARTIN    |  7654 | 7698
     3 |     TURNER    |  7844 | 7698
     3 |     WARD      |  7521 | 7698
     2 |   CLARK       |  7782 | 7839
     3 |     MILLER    |  7934 | 7782
     2 |   JONES       |  7566 | 7839
     3 |     FORD      |  7902 | 7566
     4 |       SMITH   |  7369 | 7902
     3 |     SCOTT     |  7788 | 7566
     4 |       ADAMS   |  7876 | 7788
(14 rows)
```

This final example adds the WHERE clause and starts with three root nodes. After the node tree is constructed, the WHERE clause filters out rows in the tree to form the result set.

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

The output from the query shows three root nodes (level one) - BLAKE, CLARK, and JONES. In addition, rows that do not satisfy the WHERE clause have been eliminated from the output.

```
 level | employee   | empno | mgr
-------+------------+-------+------
     1 | BLAKE      |  7698 | 7839
```



```
     1 | CLARK      |  7782 | 7839
     2 |   MILLER   |  7934 | 7782
     1 | JONES      |  7566 | 7839
     3 |     SMITH  |  7369 | 7902
     3 |     ADAMS  |  7876 | 7788
(6 rows)
```

3.2.6 Retrieving the Root Node with CONNECT_BY_ROOT

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

Note: A *unary operator* operates on a single operand, which in the case of `CONNECT_BY_ROOT`, is the column name following the `CONNECT_BY_ROOT` keyword.

In the context of the `SELECT` list, the `CONNECT_BY_ROOT` operator is shown by the following.

```
SELECT [... ,] CONNECT_BY_ROOT column [, ...]
  FROM table_expression ...
```

The following are some points to note about the `CONNECT_BY_ROOT` operator.

- The `CONNECT_BY_ROOT` operator can be used in the `SELECT` list, the `WHERE` clause, the `GROUP BY` clause, the `HAVING` clause, the `ORDER BY` clause, and the `ORDER SIBLINGS BY` clause as long as the `SELECT` command is for a hierarchical query.
- The `CONNECT_BY_ROOT` operator cannot be used in the `CONNECT BY` clause or the `START WITH` clause of the hierarchical query.
- It is possible to apply `CONNECT_BY_ROOT` to an expression involving a column, but to do so, the expression must be enclosed within parentheses.

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

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

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

```
 level | employee   | empno | mgr  | mgr empno | mgr ename
-------+-----------+-------+------+-----------+-----------
     1 | BLAKE     |  7698 | 7839 |      7698 | BLAKE
     2 |   ALLEN   |  7499 | 7698 |      7698 | BLAKE
     2 |   JAMES   |  7900 | 7698 |      7698 | BLAKE
     2 |   MARTIN  |  7654 | 7698 |      7698 | BLAKE
     2 |   TURNER  |  7844 | 7698 |      7698 | BLAKE
     2 |   WARD    |  7521 | 7698 |      7698 | BLAKE
     1 | CLARK     |  7782 | 7839 |      7782 | CLARK
     2 |   MILLER  |  7934 | 7782 |      7782 | CLARK
     1 | JONES     |  7566 | 7839 |      7566 | JONES
     2 |   FORD    |  7902 | 7566 |      7566 | JONES
     3 |     SMITH |  7369 | 7902 |      7566 | JONES
     2 |   SCOTT   |  7788 | 7566 |      7566 | JONES
     3 |     ADAMS |  7876 | 7788 |      7566 | JONES
(13 rows)
```

The following is a similar query, but producing only one tree starting with the single, top-level, employee where the `mgr` column is null.

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

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

```
 level |   employee    | empno | mgr  | mgr empno | mgr ename
-------+-------------+-------+------+-----------+-----------
     1 | KING        |  7839 |      |      7839 | KING
     2 |   BLAKE     |  7698 | 7839 |      7839 | KING
     3 |     ALLEN   |  7499 | 7698 |      7839 | KING
     3 |     JAMES   |  7900 | 7698 |      7839 | KING
     3 |     MARTIN  |  7654 | 7698 |      7839 | KING
     3 |     TURNER  |  7844 | 7698 |      7839 | KING
     3 |     WARD    |  7521 | 7698 |      7839 | KING
     2 |   CLARK     |  7782 | 7839 |      7839 | KING
     3 |     MILLER  |  7934 | 7782 |      7839 | KING
     2 |   JONES     |  7566 | 7839 |      7839 | KING
     3 |     FORD    |  7902 | 7566 |      7839 | KING
     4 |       SMITH |  7369 | 7902 |      7839 | KING
     3 |     SCOTT   |  7788 | 7566 |      7839 | KING
     4 |       ADAMS |  7876 | 7788 |      7839 | KING
(14 rows)
```

By contrast, the following example omits the `START WITH` clause thereby resulting in fourteen trees.

```
SELECT LEVEL, LPAD (' ', 2 * (LEVEL - 1)) || ename "employee", empno, mgr,
```

```
CONNECT_BY_ROOT empno "mgr empno",
CONNECT_BY_ROOT ename "mgr ename"
FROM emp
CONNECT BY PRIOR empno = mgr
ORDER SIBLINGS BY ename ASC;
```

The following is the output from the query. Each node appears at least once as a root node under the `mgr empno` and `mgr ename` columns since even the leaf nodes form the top of their own trees.

```
 level |   employee    | empno | mgr  | mgr empno | mgr ename
-------+---------------+-------+------+-----------+-----------
     1 | ADAMS         |  7876 | 7788 |      7876 | ADAMS
     1 | ALLEN         |  7499 | 7698 |      7499 | ALLEN
     1 | BLAKE         |  7698 | 7839 |      7698 | BLAKE
     2 |   ALLEN       |  7499 | 7698 |      7698 | BLAKE
     2 |   JAMES       |  7900 | 7698 |      7698 | BLAKE
     2 |   MARTIN      |  7654 | 7698 |      7698 | BLAKE
     2 |   TURNER      |  7844 | 7698 |      7698 | BLAKE
     2 |   WARD        |  7521 | 7698 |      7698 | BLAKE
     1 | CLARK         |  7782 | 7839 |      7782 | CLARK
     2 |   MILLER      |  7934 | 7782 |      7782 | CLARK
     1 | FORD          |  7902 | 7566 |      7902 | FORD
     2 |   SMITH       |  7369 | 7902 |      7902 | FORD
     1 | JAMES         |  7900 | 7698 |      7900 | JAMES
     1 | JONES         |  7566 | 7839 |      7566 | JONES
     2 |   FORD        |  7902 | 7566 |      7566 | JONES
     3 |     SMITH     |  7369 | 7902 |      7566 | JONES
     2 |   SCOTT       |  7788 | 7566 |      7566 | JONES
     3 |     ADAMS     |  7876 | 7788 |      7566 | JONES
     1 | KING          |  7839 |      |      7839 | KING
     2 |   BLAKE       |  7698 | 7839 |      7839 | KING
     3 |     ALLEN     |  7499 | 7698 |      7839 | KING
     3 |     JAMES     |  7900 | 7698 |      7839 | KING
     3 |     MARTIN    |  7654 | 7698 |      7839 | KING
     3 |     TURNER    |  7844 | 7698 |      7839 | KING
     3 |     WARD      |  7521 | 7698 |      7839 | KING
     2 |   CLARK       |  7782 | 7839 |      7839 | KING
     3 |     MILLER    |  7934 | 7782 |      7839 | KING
     2 |   JONES       |  7566 | 7839 |      7839 | KING
     3 |     FORD      |  7902 | 7566 |      7839 | KING
     4 |       SMITH   |  7369 | 7902 |      7839 | KING
     3 |     SCOTT     |  7788 | 7566 |      7839 | KING
     4 |       ADAMS   |  7876 | 7788 |      7839 | KING
     1 | MARTIN        |  7654 | 7698 |      7654 | MARTIN
     1 | MILLER        |  7934 | 7782 |      7934 | MILLER
     1 | SCOTT         |  7788 | 7566 |      7788 | SCOTT
     2 |   ADAMS       |  7876 | 7788 |      7788 | SCOTT
     1 | SMITH         |  7369 | 7902 |      7369 | SMITH
     1 | TURNER        |  7844 | 7698 |      7844 | TURNER
     1 | WARD          |  7521 | 7698 |      7521 | WARD
(39 rows)
```

The following illustrates the unary operator effect of `CONNECT_BY_ROOT`. As shown in this example, when applied to an expression that is not enclosed in parentheses, the `CONNECT_BY_ROOT` operator affects only the term, `ename`, immediately following it. The subsequent concatenation of `|| ' manages ' || ename` is not part of the `CONNECT_BY_ROOT` operation, hence the second occurrence of `ename` results in the value of the currently processed row while the first occurrence of `ename` results in the value from the root node.

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

The following is the output from the query. Note the values produced under the `top mgr/employee` column.

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

The following example uses the `CONNECT_BY_ROOT` operator on an expression enclosed in parentheses.

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

The following is the output of the query. Note that the values of both `ename` and `empno` are affected by the `CONNECT_BY_ROOT` operator and as a result, return the values from the root node as shown under the `top mgr/empno` column.

```
 level | employee   | empno | mgr  |         top mgr/empno
-------+-----------+-------+------+----------------------------
     1 | BLAKE      |  7698 | 7839 | Manager BLAKE is emp # 7698
     2 |   ALLEN    |  7499 | 7698 | Manager BLAKE is emp # 7698
     2 |   JAMES    |  7900 | 7698 | Manager BLAKE is emp # 7698
     2 |   MARTIN   |  7654 | 7698 | Manager BLAKE is emp # 7698
     2 |   TURNER   |  7844 | 7698 | Manager BLAKE is emp # 7698
     2 |   WARD     |  7521 | 7698 | Manager BLAKE is emp # 7698
     1 | CLARK      |  7782 | 7839 | Manager CLARK is emp # 7782
     2 |   MILLER   |  7934 | 7782 | Manager CLARK is emp # 7782
     1 | JONES      |  7566 | 7839 | Manager JONES is emp # 7566
```

```
     2 |   FORD    |  7902 | 7566 | Manager JONES is emp # 7566
     3 |     SMITH |  7369 | 7902 | Manager JONES is emp # 7566
     2 |   SCOTT   |  7788 | 7566 | Manager JONES is emp # 7566
     3 |     ADAMS |  7876 | 7788 | Manager JONES is emp # 7566
(13 rows)
```

3.2.7 Retrieving a Path with SYS_CONNECT_BY_PATH

`SYS_CONNECT_BY_PATH` is a function that works within a hierarchical query to retrieve the column values of a specified column that occur between the current node and the root node. The signature of the function is:

`SYS_CONNECT_BY_PATH (`*`column`*`, `*`delimiter`*`)`

The function takes two arguments:

> *`column`* is the name of a column that resides within a table specified in the hierarchical query that is calling the function.
>
> *`delimiter`* is the `varchar` value that separates each entry in the specified column.

The following example returns a list of employee names, and their managers; if the manager has a manager, that name is appended to the result:

```
edb=# SELECT level, ename , SYS_CONNECT_BY_PATH(ename, '/') managers
      FROM emp
      CONNECT BY PRIOR empno = mgr
      START WITH mgr IS NULL
      ORDER BY level, ename, managers;

 level | ename  |        managers
-------+--------+-------------------------
     1 | KING   | /KING
     2 | BLAKE  | /KING/BLAKE
     2 | CLARK  | /KING/CLARK
     2 | JONES  | /KING/JONES
     3 | ALLEN  | /KING/BLAKE/ALLEN
     3 | FORD   | /KING/JONES/FORD
     3 | JAMES  | /KING/BLAKE/JAMES
     3 | MARTIN | /KING/BLAKE/MARTIN
     3 | MILLER | /KING/CLARK/MILLER
     3 | SCOTT  | /KING/JONES/SCOTT
     3 | TURNER | /KING/BLAKE/TURNER
     3 | WARD   | /KING/BLAKE/WARD
     4 | ADAMS  | /KING/JONES/SCOTT/ADAMS
     4 | SMITH  | /KING/JONES/FORD/SMITH
(14 rows)
```

Within the result set:

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

The Advanced Server implementation of SYS_CONNECT_BY_PATH does not support use of:

- SYS_CONNECT_BY_PATH inside CONNECT_BY_PATH
- SYS_CONNECT_BY_PATH inside SYS_CONNECT_BY_PATH

3.3 Extended Functions and Operators

This section describes the extended functions and operators provided in Advanced Server.

3.3.1 Logical Operators

The usual logical operators are available: `AND`, `OR`, `NOT`

SQL uses a three-valued Boolean logic where the null value represents "unknown". Observe the following truth tables:

Table 3-3-1 AND/OR Truth Table

a	b	a AND b	a OR b
True	True	True	True
True	False	False	True
True	Null	Null	True
False	False	False	False
False	Null	False	Null
Null	Null	Null	Null

Table 3-3-2 NOT Truth Table

a	NOT a
True	False
False	True
Null	Null

The operators `AND` and `OR` are commutative, that is, you can switch the left and right operand without affecting the result.

3.3.2 Comparison Operators

The usual comparison operators are shown in the following table.

Table 3-3-3 Comparison Operators

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal
<>	Not equal
!=	Not equal

Comparison operators are available for all data types where this makes sense. All comparison operators are binary operators that return values of type `BOOLEAN`; expressions like `1 < 2 < 3` are not valid (because there is no < operator to compare a Boolean value with `3`).

In addition to the comparison operators, the special `BETWEEN` construct is available.

```
a BETWEEN x AND y
```

is equivalent to

```
a >= x AND a <= y
```

Similarly,

```
a NOT BETWEEN x AND y
```

is equivalent to

```
a < x OR a > y
```

There is no difference between the two respective forms apart from the CPU cycles required to rewrite the first one into the second one internally.

To check whether a value is or is not null, use the constructs

```
expression IS NULL
expression IS NOT NULL
```

Do not write `expression = NULL` because `NULL` is not "equal to" `NULL`. (The null value represents an unknown value, and it is not known whether two unknown values are equal.) This behavior conforms to the SQL standard.

Some applications may expect that `expression = NULL` returns true if `expression` evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard.

3.3.3 Mathematical Functions and Operators

Mathematical operators are provided for many Postgres Plus Advanced Server types. For types without common mathematical conventions for all possible permutations (e.g., date/time types) the actual behavior is described in subsequent sections.

The following table shows the available mathematical operators.

Table 3-3-4 Mathematical Operators

Operator	Description	Example	Result
`+`	Addition	`2 + 3`	`5`
`-`	Subtraction	`2 - 3`	`-1`
`*`	Multiplication	`2 * 3`	`6`
`/`	Division (integer division truncates results)	`4 / 2`	`2`
`**`	Exponentiation Operator	`2 ** 3`	`8`

The following table shows the available mathematical functions. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with `DOUBLE PRECISION` data are mostly implemented on top of the host system's C library; accuracy and behavior in boundary cases may therefore vary depending on the host system.

Table 3-3-5 Mathematical Functions

Function	Return Type	Description	Example	Result
ABS(*x*)	Same as *x*	Absolute value	ABS(-17.4)	17.4
CEIL(DOUBLE PRECISION or NUMBER)	Same as input	Smallest integer not less than argument	CEIL(-42.8)	-42
EXP(DOUBLE PRECISION or NUMBER)	Same as input	Exponential	EXP(1.0)	2.7182818284590452
FLOOR(DOUBLE PRECISION or NUMBER)	Same as input	Largest integer not greater than argument	FLOOR(-42.8)	43
LN(DOUBLE PRECISION or NUMBER)	Same as input	Natural logarithm	LN(2.0)	0.6931471805599453
LOG(*b* NUMBER, *x* NUMBER)	NUMBER	Logarithm to base *b*	LOG(2.0, 64.0)	6.0000000000000000
MOD(*y*, *x*)	Same as argument types	Remainder of *y*/*x*	MOD(9, 4)	1
NVL(*x*, *y*)	Same as argument types; where both arguments are of the same data type	If *x* is null, then NVL returns *y*	NVL(9, 0)	9
POWER(*a* DOUBLE PRECISION, *b* DOUBLE PRECISION)	DOUBLE PRECISION	*a* raised to the power of *b*	POWER(9.0, 3.0)	729.0000000000000000

Function	Return Type	Description	Example	Result
POWER(*a* NUMBER, *b* NUMBER)	NUMBER	*a* raised to the power of *b*	POWER(9.0, 3.0)	729.0000000000000000
ROUND(DOUBLE PRECISION or NUMBER)	Same as input	Round to nearest integer	ROUND(42.4)	42
ROUND(*v* NUMBER, *s* INTEGER)	NUMBER	Round to *s* decimal places	ROUND(42.4382, 2)	42.44
SIGN(DOUBLE PRECISION or NUMBER)	Same as input	Sign of the argument (-1, 0, +1)	SIGN(-8.4)	-1
SQRT(DOUBLE PRECISION or NUMBER)	Same as input	Square root	SQRT(2.0)	1.414213562373095
TRUNC(DOUBLE PRECISION or NUMBER)	Same as input	Truncate toward zero	TRUNC(42.8)	42
TRUNC(*v* NUMBER, *s* INTEGER)	NUMBER	Truncate to s decimal places	TRUNC(42.4382, 2)	42.43
WIDTH_BUCKET(*op* NUMBER, *b1* NUMBER, *b2* NUMBER, *count* INTEGER)	INTEGER	Return the bucket to which *op* would be assigned in an equidepth histogram with *count* buckets, in the range *b1* to *b2*	WIDTH_BUCKET(5.35, 0.024, 10.06, 5)	3

The following table shows the available trigonometric functions. All trigonometric functions take arguments and return values of type DOUBLE PRECISION.

Table 3-3-6 Trigonometric Functions

Function	Description
ACOS(*x*)	Inverse cosine
ASIN(*x*)	Inverse sine
ATAN(*x*)	Inverse tangent
ATAN2(*x*, *y*)	Inverse tangent of *x*/*y*
COS(*x*)	Cosine
SIN(*x*)	Sine
TAN(*x*)	Tangent

3.3.4 String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types `CHAR`, `VARCHAR2`, and `CLOB`. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of automatic padding when using the `CHAR` type. Generally, the functions described here also work on data of non-string types by converting that data to a string representation first.

Table 3-3-7 SQL String Functions and Operators

Function	Return Type	Description	Example	Result
string || *string*	CLOB	String concatenation	'Enterprise' || 'DB'	EnterpriseDB
CONCAT(*string*, *string*)	CLOB	String concatenation	'a' || 'b'	ab
HEXTORAW(*varchar2*)	RAW	Converts a VARCHAR2 value to a RAW value	HEXTORAW('303132')	'012'
RAWTOHEX(*raw*)	VARCHAR2	Converts a RAW value to a HEXADECIMAL value	RAWTOHEX('012')	'303132'
INSTR(*string*, *set*, [*start* [, *occurrence*]])	INTEGER	Finds the location of a set of characters in a string, starting at position *start* in the string, *string*, and looking for the first, second, third and so on occurrences of the set. Returns 0 if the set is not found.	INSTR('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PI',1,3)	30
INSTRB(*string*, *set*)	INTEGER	Returns the position of the *set* within the *string*. Returns 0 if *set* is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS', 'PICK')	13
INSTRB(*string*, *set*, *start*)	INTEGER	Returns the position of the *set* within the *string*, beginning at *start*. Returns 0 if *set* is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 14)	30
INSTRB(*string*, *set*, *start*, *occurrence*)	INTEGER	Returns the position of the specified *occurrence* of *set* within the *string*, beginning at *start*. Returns 0 if *set* is not found.	INSTRB('PETER PIPER PICKED a PECK of PICKLED PEPPERS','PICK', 1, 2)	30
LOWER(*string*)	CLOB	Convert *string* to lower case	LOWER('TOM')	tom
SUBSTR(*string*, *start* [, *count*])	CLOB	Extract substring starting from *start* and going for *count* characters. If *count* is not specified, the string is clipped from the start till the end.	SUBSTR('This is a test',6,2)	is
SUBSTRB(*string*, *start* [, *count*])	CLOB	Same as SUBSTR except	SUBSTRB('abc',3) (assuming a double-byte	c

Function	Return Type	Description	Example	Result
		start and *count* are in number of bytes.	character set)	
SUBSTR2(*string*, *start* [, *count*])	CLOB	Alias for SUBSTR.	SUBSTR2('This is a test',6,2)	is
SUBSTR2(*string*, *start* [, *count*])	CLOB	Alias for SUBSTRB.	SUBSTR2('abc',3) (assuming a double-byte character set)	c
SUBSTR4(*string*, *start* [, *count*])	CLOB	Alias for SUBSTR.	SUBSTR4('This is a test',6,2)	is
SUBSTR4(*string*, *start* [, *count*])	CLOB	Alias for SUBSTRB.	SUBSTR4('abc',3) (assuming a double-byte character set)	c
SUBSTRC(*string*, *start* [, *count*])	CLOB	Alias for SUBSTR.	SUBSTRC('This is a test',6,2)	is
SUBSTRC(*string*, *start* [, *count*])	CLOB	Alias for SUBSTRB.	SUBSTRC('abc',3) (assuming a double-byte character set)	c
TRIM([LEADING | TRAILING | BOTH] [*characters*] FROM *string*)	CLOB	Remove the longest string containing only the characters (a space by default) from the start/end/both ends of the string.	TRIM(BOTH 'x' FROM 'xTomxx')	Tom
LTRIM(*string* [, *set*])	CLOB	Removes all the characters specified in *set* from the left of a given *string*. If *set* is not specified, a blank space is used as default.	LTRIM('abcdefghi', 'abc')	defghi
RTRIM(*string* [, *set*])	CLOB	Removes all the characters specified in *set* from the right of a given *string*. If *set* is not specified, a blank space is used as default.	RTRIM('abcdefghi', 'ghi')	abcdef
UPPER(*string*)	CLOB	Convert *string* to upper case	UPPER('tom')	TOM

Additional string manipulation functions are available and are listed in the following table. Some of them are used internally to implement the SQL-standard string functions listed in Table 3-3-7.

Table 3-3-8 Other String Functions

Function	Return Type	Description	Example	Result
ASCII(*string*)	INTEGER	ASCII code of the first byte of the argument	ASCII('x')	120
CHR(INTEGER)	CLOB	Character with the given ASCII code	CHR(65)	A
DECODE(*expr*, *expr1a*, *expr1b* [, *expr2a*, *expr2b*]... [, *default*])	Same as argument types of *expr1b*, *expr2b*,..., *default*	Finds first match of *expr* with *expr1a*, *expr2a*, etc. When match found, returns corresponding parameter pair, *expr1b*, *expr2b*, etc. If no match found, returns	DECODE(3, 1,'One', 2,'Two', 3,'Three', 'Not found')	Three

Function	Return Type	Description	Example	Result
		default. If no match found and *default* not specified, returns null.		
INITCAP(*string*)	CLOB	Convert the first letter of each word to uppercase and the rest to lowercase. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.	INITCAP('hi THOMAS')	Hi Thomas
LENGTH	INTEGER	Returns the number of characters in a string value.	LENGTH('Côte d''Azur')	11
LENGTHC	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTHC('Côte d''Azur')	11
LENGTH2	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTH2('Côte d''Azur')	11
LENGTH4	INTEGER	This function is identical in functionality to LENGTH; the function name is supported for compatibility.	LENGTH4('Côte d''Azur')	11
LENGTHB	INTEGER	Returns the number of bytes required to hold the given value.	LENGTHB('Côte d''Azur')	12
LPAD(*string*, *length* INTEGER [, *fill*])	CLOB	Fill up *string* to size, *length* by prepending the characters, *fill* (a space by default). If *string* is already longer than *length* then it is truncated (on the right).	LPAD('hi', 5, 'xy')	xyxhi
REPLACE(*string*, *search_string* [, *replace_string*]	CLOB	Replaces one value in a string with another. If you do not specify a value for *replace_string*, the *search_string* value when found, is removed.	REPLACE('GEORGE', 'GE', 'EG')	EGOREG
RPAD(*string*, *length* INTEGER [, *fill*])	CLOB	Fill up *string* to size, *length* by appending the characters, *fill* (a space by default). If *string* is already longer than *length* then it is truncated.	RPAD('hi', 5, 'xy')	hixyx
TRANSLATE(*string*, *from*, *to*)	CLOB	Any character in *string* that matches a character in the *from* set is replaced by the corresponding character in the *to* set.	TRANSLATE('12345', '14', 'ax')	a23x5

3.3.5 Pattern Matching String Functions

Advanced Server offers support for the `REGEXP_COUNT`, `REGEXP_INSTR` and `REGEXP_SUBSTR` functions. These functions search a string for a pattern specified by a regular expression, and return information about occurrences of the pattern within the string. The pattern should be a POSIX-style regular expression; for more information about forming a POSIX-style regular expression, please refer to the core documentation at:

http://www.enterprisedb.com/docs/en/9.4/pg/functions-matching.html

3.3.5.1 REGEXP_COUNT

`REGEXP_COUNT` searches a string for a regular expression, and returns a count of the times that the regular expression occurs. The signature is:

```
INTEGER REGEXP_COUNT
(
  srcstr    TEXT,
  pattern   TEXT,
  position  DEFAULT 1
  modifier  DEFAULT NULL
)
```

Parameters

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which `REGEXP_COUNT` will search.

position

position is an integer value that indicates the position in the source string at which `REGEXP_COUNT` will begin searching. The default value is `1`.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/functions-matching.html

Example

In the following simple example, REGEXP_COUNT returns a count of the number of times the letter i is used in the character string 'reinitializing':

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 1) FROM DUAL;
 regexp_count
--------------
            5
(1 row)
```

In the first example, the command instructs REGEXP_COUNT begins counting in the first position; if we modify the command to start the count on the 6th position:

```
edb=# SELECT REGEXP_COUNT('reinitializing', 'i', 6) FROM DUAL;
 regexp_count
--------------
            3
(1 row)
```

REGEXP_COUNT returns 3; the count now excludes any occurrences of the letter i that occur before the 6th position.

3.3.5.2 REGEXP_INSTR

REGEXP_INSTR searches a string for a POSIX-style regular expression. This function returns the position within the string where the match was located. The signature is:

```
INTEGER REGEXP_INSTR
(
  srcstr        TEXT,
  pattern       TEXT,
  position      INT  DEFAULT 1,
  occurrence    INT  DEFAULT 1,
  returnparam   INT  DEFAULT 0,
  modifier      TEXT DEFAULT NULL,
  subexpression INT  DEFAULT 0,
)
```

Parameters

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which `REGEXP_INSTR` will search.

position

position specifies an integer value that indicates the start position in a source string. The default value is `1`.

occurrence

occurrence specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is `1`.

returnparam

returnparam is an integer value that specifies the location within the string that `REGEXP_INSTR` should return. The default value is `0`. Specify:

`0` to return the location within the string of the first character that matches *pattern*.

A value greater than `0` to return the position of the first character following the end of the *pattern*.

modifier

modifier specifies values that control the pattern matching behavior. The default value is `NULL`. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/functions-matching.html

subexpression

subexpression is an integer value that identifies the portion of the *pattern* that will be returned by `REGEXP_INSTR`. The default value of *subexpression* is `0`.

If you specify a value for *subexpression*, you must include one (or more) set of parentheses in the *pattern* that isolate a portion of the value being searched for. The value specified by *subexpression* indicates which set of parentheses should be returned; for example, if *subexpression* is 2, REGEXP_INSTR will return the position of the second set of parentheses.

Example

In the following simple example, REGEXP_INSTR searches a string that contains the a phone number for the first occurrence of a pattern that contains three consecutive digits:

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM DUAL;
 regexp_instr
--------------
            1
(1 row)
```

The command instructs REGEXP_INSTR to return the position of the first occurrence. If we modify the command to return the start of the second occurrence of three consecutive digits:

```
edb=# SELECT REGEXP_INSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM DUAL;
 regexp_instr
--------------
            5
(1 row)
```

REGEXP_INSTR returns 5; the second occurrence of three consecutive digits begins in the fifth position.

3.3.5.3 REGEXP_SUBSTR

The REGEXP_SUBSTR function searches a string for a pattern specified by a POSIX compliant regular expression. REGEXP_SUBSTR returns the string that matches the pattern specified in the call to the function. The signature of the function is:

```
TEXT REGEXP_SUBSTR
(
  srcstr        TEXT,
  pattern       TEXT,
  position      INT  DEFAULT 1,
  occurrence    INT  DEFAULT 1,
  modifier      TEXT DEFAULT NULL,
  subexpression INT  DEFAULT 0
)
```

Parameters

srcstr

srcstr specifies the string to search.

pattern

pattern specifies the regular expression for which REGEXP_SUBSTR will search.

position

position specifies an integer value that indicates the start position in a source string. The default value is 1.

occurrence

occurrence specifies which match is returned if more than one occurrence of the pattern occurs in the string that is searched. The default value is 1.

modifier

modifier specifies values that control the pattern matching behavior. The default value is NULL. For a complete list of the modifiers supported by Advanced Server, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/functions-matching.html

subexpression

subexpression is an integer value that identifies the portion of the *pattern* that will be returned by REGEXP_SUBSTR. The default value of *subexpression* is 0.

If you specify a value for *subexpression*, you must include one (or more) set of parentheses in the *pattern* that isolate a portion of the value being searched for. The value specified by *subexpression* indicates which set of parentheses should be returned; for example, if *subexpression* is 2, REGEXP_SUBSTR will return the value contained within the second set of parentheses.

Example

In the following simple example, REGEXP_SUBSTR searches a string that contains a phone number for the first set of three consecutive digits:

```
edb=# SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 1) FROM
DUAL;
 regexp_substr
---------------
 800
(1 row)
```


It locates the first occurrence of three digits and returns the string (800); if we modify the command to check for the second occurrence of three consecutive digits:

```
edb=# SELECT REGEXP_SUBSTR('800-555-1212', '[0-9][0-9][0-9]', 1, 2) FROM
DUAL;
 regexp_substr
---------------
 555
(1 row)
```

REGEXP_SUBSTR returns 555, the contents of the second substring.

3.3.6 Pattern Matching Using the LIKE Operator

Postgres Plus Advanced Server provides pattern matching using the traditional SQL LIKE operator. The syntax for the LIKE operator is as follows.

```
string LIKE pattern [ ESCAPE escape-character ]
string NOT LIKE pattern [ ESCAPE escape-character ]
```

Every *pattern* defines a set of strings. The LIKE expression returns TRUE if *string* is contained in the set of strings represented by *pattern*. As expected, the NOT LIKE expression returns FALSE if LIKE returns TRUE, and vice versa. An equivalent expression is NOT (*string* LIKE *pattern*).

If *pattern* does not contain percent signs or underscore, then the pattern only represents the string itself; in that case LIKE acts like the equals operator. An underscore (_) in *pattern* stands for (matches) any single character; a percent sign (%) matches any string of zero or more characters.

Some examples:

```
'abc' LIKE 'abc'     true
'abc' LIKE 'a%'      true
'abc' LIKE '_b_'     true
'abc' LIKE 'c'       false
```

LIKE pattern matches always cover the entire string. To match a pattern anywhere within a string, the pattern must therefore start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in *pattern* must be preceded by the escape character. The default escape character is the backslash but a different one may be selected by using the ESCAPE clause. To match the escape character itself, write two escape characters.

Note that the backslash already has a special meaning in string literals, so to write a pattern constant that contains a backslash you must write two backslashes in an SQL statement. Thus, writing a pattern that actually matches a literal backslash means writing four backslashes in the statement. You can avoid this by selecting a different escape character with ESCAPE; then a backslash is not special to LIKE anymore. (But it is still special to the string literal parser, so you still need two of them.)

It's also possible to select no escape character by writing ESCAPE ''. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

3.3.7 Data Type Formatting Functions

The Postgres Plus Advanced Server formatting functions (described in Table 3-3-9) provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a string template that defines the output or input format.

Table 3-3-9 Formatting Functions

Function	Return Type	Description	Example	Result
`TO_CHAR(DATE [, *format* ])`	`VARCHAR2`	Convert a date/time to a string with output, *format*. If omitted default format is DD-MON-YY.	`TO_CHAR(SYSDATE, 'MM/DD/YYYY HH12:MI:SS AM')`	`07/25/2007 09:43:02 AM`
`TO_CHAR(TIMESTAMP [, *format* ])`	`VARCHAR2`	Convert a timestamp to a string with output, *format*. If omitted default format is DD-MON-YY.	`TO_CHAR(CURRENT_TIMESTAMP, 'MM/DD/YYYY HH12:MI:SS AM')`	`08/13/2014 08:55:22 PM`
`TO_CHAR(INTEGER [, *format* ])`	`VARCHAR2`	Convert an integer to a string with output, *format*	`TO_CHAR(2412, '999,999S')`	`2,412+`
`TO_CHAR(NUMBER [, *format* ])`	`VARCHAR2`	Convert a decimal number to a string with output, *format*	`TO_CHAR(10125.35, '999,999.99')`	`10,125.35`
`TO_CHAR(DOUBLE PRECISION, *format*)`	`VARCHAR2`	Convert a floating-point number to a string with output, *format*	`TO_CHAR(CAST(123.5282 AS REAL), '999.99')`	`123.53`
`TO_DATE(*string* [, *format* ])`	`DATE`	Convert a date formatted string to a `DATE` data type	`TO_DATE('2007-07-04 13:39:10', 'YYYY-MM-DD HH24:MI:SS')`	`04-JUL-07 13:39:10`
`TO_NUMBER(*string* [, *format* ])`	`NUMBER`	Convert a number formatted string to a `NUMBER` data type	`TO_NUMBER('2,412-', '999,999S')`	`-2412`
`TO_TIMESTAMP(*string*, *format*)`	`TIMESTAMP`	Convert a timestamp formatted string to a `TIMESTAMP` data type	`TO_TIMESTAMP('05 Dec 2000 08:30:25 pm', 'DD Mon YYYY hh12:mi:ss pm')`	`05-DEC-00 20:30:25`

In an output template string (for `TO_CHAR`), there are certain patterns that are recognized and replaced with appropriately-formatted data from the value to be formatted. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for anything but TO_CHAR), template patterns identify the parts of the input data string to be looked at and the values to be found there.

The following table shows the template patterns available for formatting date values using the TO_CHAR and TO_DATE functions.

Table 3-3-10 Template Date/Time Format Patterns

Pattern	Description
HH	Hour of day (01-12)
HH12	Hour of day (01-12)
HH24	Hour of day (00-23)
MI	Minute (00-59)
SS	Second (00-59)
SSSSS	Seconds past midnight (0-86399)
FF*n*	Fractional seconds where *n* is an optional integer from 1 to 9 for the number of digits to return. If omitted, the default is 6.
AM or A.M. or PM or P.M.	Meridian indicator (uppercase)
am or a.m. or pm or p.m.	Meridian indicator (lowercase)
Y,YYY	Year (4 and more digits) with comma
YEAR	Year (spelled out)
SYEAR	Year (spelled out) (BC dates prefixed by a minus sign)
YYYY	Year (4 and more digits)
SYYYY	Year (4 and more digits) (BC dates prefixed by a minus sign)
YYY	Last 3 digits of year
YY	Last 2 digits of year
Y	Last digit of year
IYYY	ISO year (4 and more digits)
IYY	Last 3 digits of ISO year
IY	Last 2 digits of ISO year
I	Last 1 digit of ISO year
BC or B.C. or AD or A.D.	Era indicator (uppercase)
bc or b.c. or ad or a.d.	Era indicator (lowercase)
MONTH	Full uppercase month name
Month	Full mixed-case month name
month	Full lowercase month name
MON	Abbreviated uppercase month name (3 chars in English, localized lengths vary)
Mon	Abbreviated mixed-case month name (3 chars in English, localized lengths vary)
mon	Abbreviated lowercase month name (3 chars in English, localized lengths vary)
MM	Month number (01-12)
DAY	Full uppercase day name
Day	Full mixed-case day name
day	Full lowercase day name
DY	Abbreviated uppercase day name (3 chars in English, localized lengths vary)
Dy	Abbreviated mixed-case day name (3 chars in English, localized lengths vary)
dy	Abbreviated lowercase day name (3 chars in English, localized lengths vary)

Pattern	Description
`DDD`	Day of year (001-366)
`DD`	Day of month (01-31)
`D`	Day of week (1-7; Sunday is 1)
`W`	Week of month (1-5) (The first week starts on the first day of the month)
`WW`	Week number of year (1-53) (The first week starts on the first day of the year)
`IW`	ISO week number of year; the first Thursday of the new year is in week 1
`CC`	Century (2 digits); the 21st century starts on 2001-01-01
`SCC`	Same as CC except BC dates are prefixed by a minus sign
`J`	Julian Day (days since January 1, 4712 BC)
`Q`	Quarter
`RM`	Month in Roman numerals (I-XII; I=January) (uppercase)
`rm`	Month in Roman numerals (i-xii; i=January) (lowercase)
`RR`	First 2 digits of the year when given only the last 2 digits of the year. Result is based upon an algorithm using the current year and the given 2-digit year. The first 2 digits of the given 2-digit year will be the same as the first 2 digits of the current year with the following exceptions: If the given 2-digit year is < 50 and the last 2 digits of the current year is >= 50, then the first 2 digits for the given year is 1 greater than the first 2 digits of the current year. If the given 2-digit year is >= 50 and the last 2 digits of the current year is < 50, then the first 2 digits for the given year is 1 less than the first 2 digits of the current year.
`RRRR`	Only affects TO_DATE function. Allows specification of 2-digit or 4-digit year. If 2-digit year given, then returns first 2 digits of year like RR format. If 4-digit year given, returns the given 4-digit year.

Certain modifiers may be applied to any template pattern to alter its behavior. For example, `FMMonth` is the `Month` pattern with the `FM` modifier. The following table shows the modifier patterns for date/time formatting.

Table 3-3-11 Template Pattern Modifiers for Date/Time Formatting

Modifier	Description	Example
`FM` prefix	Fill mode (suppress padding blanks and zeros)	`FMMonth`
`TH` suffix	Uppercase ordinal number suffix	`DDTH`
`th` suffix	Lowercase ordinal number suffix	`DDth`
`FX` prefix	Fixed format global option (see usage notes)	`FX Month DD Day`
`SP` suffix	Spell mode	`DDSP`

Usage notes for date/time formatting:

- `FM` suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern fixed-width.
- `TO_TIMESTAMP` and `TO_DATE` skip multiple blank spaces in the input string if the `FX` option is not used. `FX` must be specified as the first item in the template. For example `TO_TIMESTAMP('2000    JUN', 'YYYY MON')` is correct, but

TO_TIMESTAMP('2000   JUN', 'FXYYYY MON') returns an error, because TO_TIMESTAMP expects one space only.

- Ordinary text is allowed in TO_CHAR templates and will be output literally.
- In conversions from string to timestamp or date, the CC field is ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y then the year is computed as (CC-1)*100+YY.

The following table shows the template patterns available for formatting numeric values.

Table 3-3-12 Template Patterns for Numeric Formatting

Pattern	Description
9	Value with the specified number of digits
0	Value with leading zeroes
. (period)	Decimal point
, (comma)	Group (thousand) separator
$	Dollar sign
PR	Negative value in angle brackets
S	Sign anchored to number (uses locale)
L	Currency symbol (uses locale)
D	Decimal point (uses locale)
G	Group separator (uses locale)
MI	Minus sign specified in right-most position (if number < 0)
RN or rn	Roman numeral (input between 1 and 3999)
V	Shift specified number of digits (see notes)

Usage notes for numeric formatting:

- 9 results in a value with the same number of digits as there are 9s. If a digit is not available it outputs a space.
- TH does not convert values less than zero and does not convert fractional numbers.

V effectively multiplies the input values by 10^n, where n is the number of digits following V. TO_CHAR does not support the use of V combined with a decimal point. (E.g., 99.9V99 is not allowed.)

The following table shows some examples of the use of the TO_CHAR and TO_DATE functions.

Table 3-3-13 TO_CHAR Examples

Expression	Result
TO_CHAR(CURRENT_TIMESTAMP, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
TO_CHAR(CURRENT_TIMESTAMP, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
TO_CHAR(-0.1, '99.99')	' -.10'
TO_CHAR(-0.1, 'FM9.99')	'-.1'

Expression	Result
TO_CHAR(0.1, '0.9')	' 0.1'
TO_CHAR(12, '9990999.9')	' 0012.0'
TO_CHAR(12, 'FM9990999.9')	'0012.'
TO_CHAR(485, '999')	' 485'
TO_CHAR(-485, '999')	'-485'
TO_CHAR(1485, '9,999')	' 1,485'
TO_CHAR(1485, '9G999')	' 1,485'
TO_CHAR(148.5, '999.999')	' 148.500'
TO_CHAR(148.5, 'FM999.999')	'148.5'
TO_CHAR(148.5, 'FM999.990')	'148.500'
TO_CHAR(148.5, '999D999')	' 148.500'
TO_CHAR(3148.5, '9G999D999')	' 3,148.500'
TO_CHAR(-485, '999S')	'485-'
TO_CHAR(-485, '999MI')	'485-'
TO_CHAR(485, '999MI')	'485 '
TO_CHAR(485, 'FM999MI')	'485'
TO_CHAR(-485, '999PR')	'<485>'
TO_CHAR(485, 'L999')	'$ 485'
TO_CHAR(485, 'RN')	' CDLXXXV'
TO_CHAR(485, 'FMRN')	'CDLXXXV'
TO_CHAR(5.2, 'FMRN')	'V'
TO_CHAR(12, '99V999')	' 12000'
TO_CHAR(12.4, '99V999')	' 12400'
TO_CHAR(12.45, '99V9')	' 125'

3.3.7.1 IMMUTABLE TO_CHAR(TIMESTAMP, format) Function

There are certain cases of the TO_CHAR function that can result in usage of an IMMUTABLE form of the function. Basically, a function is IMMUTABLE if the function does not modify the database, and the function returns the same, consistent value dependent upon only its input parameters. That is, the settings of configuration parameters, the locale, the content of the database, etc. do not affect the results returned by the function.

For more information about function volatility categories VOLATILE, STABLE, and IMMUTABLE, please see the PostgreSQL Core documentation, available from EnterpriseDB at:

http://www.enterprisedb.com/docs/en/9.4/pg/xfunc-volatility.html

A particular advantage of an IMMUTABLE function is that it can be used in the CREATE INDEX command to create an index based on that function.

In order for the TO_CHAR function to use the IMMUTABLE form the following conditions must be satisfied:

- The first parameter of the TO_CHAR function must be of data type TIMESTAMP.
- The format specified in the second parameter of the TO_CHAR function must not affect the return value of the function based on factors such as language, locale, etc. For example a format of 'YYYY-MM-DD HH24:MI:SS' can be used for an IMMUTABLE form of the function since, regardless of locale settings, the result of the function is the date and time expressed solely in numeric form. However, a format of 'DD-MON-YYYY' cannot be used for an IMMUTABLE form of the function because the 3-character abbreviation of the month may return different results depending upon the locale setting.

Format patterns that result in a non-immutable function include any variations of spelled out or abbreviated months (MONTH, MON), days (DAY, DY), median indicators (AM, PM), or era indicators (BC, AD).

Note: The condition specified in the second bullet point applies only for an Oracle compatible installation of Advanced Server. For a PostgreSQL compatible installation of Advanced Server, the TO_CHAR(TIMESTAMP, *format*) function is locale independent and thus categorized as IMMUTABLE unless the format forces locale dependence with the TM (translation mode) prefix. For example, 'DD-MON-YYYY' would allow the function to be IMMUTABLE, but 'DD-TMMON-YYYY' would not.

For more information about the PostgreSQL TM date/time formatting prefix, please see the PostgreSQL Core documentation, available from EnterpriseDB at:

http://www.enterprisedb.com/docs/en/9.4/pg/functions-formatting.html

For the following example, a table with a TIMESTAMP column is created.

```
CREATE TABLE ts_tbl (ts_col TIMESTAMP);
```

The following shows the successful creation of an index with the IMMUTABLE form of the TO_CHAR function. This applies to both an Oracle compatible installation as well as a PostgreSQL compatible installation.

```
edb=# CREATE INDEX ts_idx ON ts_tbl (TO_CHAR(ts_col,'YYYY-MM-DD HH24:MI:SS'));
CREATE INDEX
edb=# \dS ts_idx
                                   Index "public.ts_idx"
 Column  |       Type        |                        Definition
---------+-------------------+---------------------------------------------------------
----
 to_char | character varying | to_char(ts_col, 'YYYY-MM-DD HH24:MI:SS'::character
varying)
btree, for table "public.ts_tbl"
```

In an Oracle compatible installation, the following results in an error because the format specified in the TO_CHAR function prevents the use of the IMMUTABLE form since the 3-character month abbreviation, MON, may result in different return values based on the locale setting.

```
edb=# CREATE INDEX ts_idx_2 ON ts_tbl (TO_CHAR(ts_col, 'DD-MON-YYYY'));
ERROR:  functions in index expression must be marked IMMUTABLE
```

However, for a PostgreSQL compatible installation, the `CREATE INDEX` command rejected in the preceding example would be accepted because the function is `IMMUTABLE` since there is no `TM` prefix in the format.

```
postgres=# CREATE INDEX ts_idx_2 ON ts_tbl (TO_CHAR(ts_col, 'DD-MON-YYYY'));
CREATE INDEX
postgres=# \d ts_idx_2
               Index "postgres.ts_idx_2"
 Column  | Type |              Definition
---------+------+--------------------------------------
 to_char | text | to_char(ts_col, 'DD-MON-YYYY'::text)
btree, for table "postgres.ts_tbl"
```

But when the `TM` prefix is included, the function is not `IMMUTABLE`, and thus, the index is rejected.

```
postgres=# CREATE INDEX ts_idx_3 ON ts_tbl (TO_CHAR(ts_col, 'DD-TMMON-
YYYY'));
ERROR:  functions in index expression must be marked IMMUTABLE
```

3.3.8 Date/Time Functions and Operators

Table 3-3-15 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 3-3-14 illustrates the behaviors of the basic arithmetic operators (+, -).

Table 3-3-14 Date/Time Operators

Operator	Example	Result
+	DATE '2001-09-28' + 7	05-OCT-01 00:00:00
+	TIMESTAMP '2001-09-28 13:30:00' + 3	01-OCT-01 13:30:00
-	DATE '2001-10-01' - 7	24-SEP-01 00:00:00
-	TIMESTAMP '2001-09-28 13:30:00' - 3	25-SEP-01 13:30:00
-	TIMESTAMP '2001-09-29 03:00:00' - TIMESTAMP '2001-09-27 12:00:00'	@ 1 day 15 hours

In the date/time functions of Table 3-3-15 the use of the `DATE` and `TIMESTAMP` data types are interchangeable.

Table 3-3-15 Date/Time Functions

Function	Return Type	Description	Example	Result
ADD_MONTHS(DATE, NUMBER)	DATE	Add months to a date; see Section 3.3.8.1.	ADD_MONTHS('28-FEB-97', 3.8)	31-MAY-97 00:00:00
CURRENT_DATE	DATE	Current date; see Section 3.3.8.8.	CURRENT_DATE	04-JUL-07
CURRENT_TIMESTAMP	TIMESTAMP	Returns the current date and time; see Section 3.3.8.8.	CURRENT_TIMESTAMP	04-JUL-07 15:33:23.484
EXTRACT(*field* FROM TIMESTAMP)	DOUBLE PRECISION	Get subfield; see Section 3.3.8.2.	EXTRACT(hour FROM TIMESTAMP '2001-02-16 20:38:40')	20
LAST_DAY(DATE)	DATE	Returns the last day of the month represented by the given date. If the given date contains a time portion, it is carried forward to the result unchanged.	LAST_DAY('14-APR-98')	30-APR-98 00:00:00
LOCALTIMESTAMP [(*precision*)]	TIMESTAMP	Current date and time (start of current transaction); see Section 3.3.8.8.	LOCALTIMESTAMP	04-JUL-07 15:33:23.484
MONTHS_BETWEEN(DATE, DATE)	NUMBER	Number of months between two dates; see Section 3.3.8.3.	MONTHS_BETWEEN('28-FEB-07', '30-NOV-06')	3
NEXT_DAY(DATE, *dayofweek*)	DATE	Date falling on *dayofweek* following specified date; see Section	NEXT_DAY('16-APR-07','FRI')	20-APR-07 00:00:00

Function	Return Type	Description	Example	Result
		3.3.8.4.		
NEW_TIME(DATE, VARCHAR, VARCHAR)	DATE	Converts a date and time to an alternate time zone	NEW_TIME(TO_DATE '2005/05/29 01:45', 'AST', 'PST')	2005/05/29 21:45:00
NUMTODSINTERVAL(NUMBER, INTERVAL)	INTERVAL	Converts a number to a specified day or second interval; see Section 3.3.8.9.	SELECT numtodsinterval(100, 'hour');	4 days 04:00:00
NUMTOYMINTERVAL(NUMBER, INTERVAL)	INTERVAL	Converts a number to a specified year or month interval; see Section 3.3.8.10.	SELECT numtoyminterval(100, 'month');	8 years 4 mons
ROUND(DATE [, *format*])	DATE	Date rounded according to *format*; see Section 3.3.8.6.	ROUND(TO_DATE('29-MAY-05'),'MON')	01-JUN-05 00:00:00
SYS_EXTRACT_UTC(TIMESTAMP WITH TIME ZONE)	TIMESTAMP	Returns Coordinated Universal Time	SYS_EXTRACT_UTC(CAST('24-MAR-11 12:30:00PM -04:00' AS TIMESTAMP WITH TIME ZONE))	24-MAR-11 16:30:00
SYSDATE	DATE	Returns current date and time	SYSDATE	01-AUG-12 11:12:34
SYSTIMESTAMP()	TIMESTAMP	Returns current date and time	SYSTIMESTAMP	01-AUG-12 11:11:23.665229 -07:00
TRUNC(DATE [*format*])	DATE	Truncate according to format; see Section 3.3.8.7.	TRUNC(TO_DATE('29-MAY-05'), 'MON')	01-MAY-05 00:00:00

3.3.8.1 ADD_MONTHS

The ADD_MONTHS functions adds (or subtracts if the second parameter is negative) the specified number of months to the given date. The resulting day of the month is the same as the day of the month of the given date except when the day is the last day of the month in which case the resulting date always falls on the last day of the month.

Any fractional portion of the number of months parameter is truncated before performing the calculation.

If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the ADD_MONTHS function.

```
SELECT ADD_MONTHS('13-JUN-07',4) FROM DUAL;

     add_months
--------------------
 13-OCT-07 00:00:00
(1 row)
```



```
SELECT ADD_MONTHS('31-DEC-06',2) FROM DUAL;

     add_months
--------------------
 28-FEB-07 00:00:00
(1 row)

SELECT ADD_MONTHS('31-MAY-04',-3) FROM DUAL;

     add_months
--------------------
 29-FEB-04 00:00:00
(1 row)
```

3.3.8.2 EXTRACT

The `EXTRACT` function retrieves subfields such as year or hour from date/time values. The `EXTRACT` function returns values of type `DOUBLE PRECISION`. The following are valid field names:

`YEAR`

The year field

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
      2001
(1 row)
```

`MONTH`

The number of the month within the year (1 - 12)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
         2
(1 row)
```

`DAY`

The day (of the month) field (1 - 31)

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
```

```
-----------
        16
(1 row)
```

HOUR

The hour field (0 - 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
        20
(1 row)
```

MINUTE

The minutes field (0 - 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
        38
(1 row)
```

SECOND

The seconds field, including fractional parts (0 - 59)

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40') FROM DUAL;

 date_part
-----------
        40
(1 row)
```

3.3.8.3 MONTHS_BETWEEN

The MONTHS_BETWEEN function returns the number of months between two dates. The result is a numeric value which is positive if the first date is greater than the second date or negative if the first date is less than the second date.

The result is always a whole number of months if the day of the month of both date parameters is the same, or both date parameters fall on the last day of their respective months.

The following are some examples of the MONTHS_BETWEEN function.

```
SELECT MONTHS_BETWEEN('15-DEC-06','15-OCT-06') FROM DUAL;
```

```
 months_between
----------------
              2
(1 row)

SELECT MONTHS_BETWEEN('15-OCT-06','15-DEC-06') FROM DUAL;

 months_between
----------------
             -2
(1 row)

SELECT MONTHS_BETWEEN('31-JUL-00','01-JUL-00') FROM DUAL;

 months_between
----------------
    0.967741935
(1 row)

SELECT MONTHS_BETWEEN('01-JAN-07','01-JAN-06') FROM DUAL;

 months_between
----------------
             12
(1 row)
```

3.3.8.4 NEXT_DAY

The NEXT_DAY function returns the first occurrence of the given weekday strictly greater than the given date. At least the first three letters of the weekday must be specified - e.g., SAT. If the given date contains a time portion, it is carried forward to the result unchanged.

The following are examples of the NEXT_DAY function.

```
SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'SUNDAY') FROM DUAL;

      next_day
--------------------
 19-AUG-07 00:00:00
(1 row)

SELECT NEXT_DAY(TO_DATE('13-AUG-07','DD-MON-YY'),'MON') FROM DUAL;

      next_day
--------------------
 20-AUG-07 00:00:00
(1 row)
```


3.3.8.5 NEW_TIME

The NEW_TIME function converts a date and time from one time zone to another. NEW_TIME returns a value of type DATE. The syntax is:

```
NEW_TIME(DATE, time_zone1, time_zone2)
```

time_zone1 and *time_zone2* must be string values from the Time Zone column of the following table:

Time Zone	Offset from UTC	Description
AST	UTC+4	Atlantic Standard Time
ADT	UTC+3	Atlantic Daylight Time
BST	UTC+11	Bering Standard Time
BDT	UTC+10	Bering Daylight Time
CST	UTC+6	Central Standard Time
CDT	UTC+5	Central Daylight Time
EST	UTC+5	Eastern Standard Time
EDT	UTC+4	Eastern Daylight Time
GMT	UTC	Greenwich Mean Time
HST	UTC+10	Alaska-Hawaii Standard Time
HDT	UTC+9	Alaska-Hawaii Daylight Time
MST	UTC+7	Mountain Standard Time
MDT	UTC+6	Mountain Daylight Time
NST	UTC+3:30	Newfoundland Standard Time
PST	UTC+8	Pacific Standard Time
PDT	UTC+7	Pacific Daylight Time
YST	UTC+9	Yukon Standard Time
YDT	UTC+8	Yukon Daylight Time

Following is an example of the NEW_TIME function.

```
SELECT NEW_TIME(TO_DATE('08-13-07 10:35:15','MM-DD-YY HH24:MI:SS'),'AST',
'PST') "Pacific Standard Time" FROM DUAL;

Pacific Standard Time
---------------------
 13-AUG-07 06:35:15
(1 row)
```

3.3.8.6 ROUND

The ROUND function returns a date rounded according to a specified template pattern. If the template pattern is omitted, the date is rounded to the nearest day. The following table shows the template patterns for the ROUND function.

Table 3-3-16 Template Date Patterns for the ROUND Function

Pattern	Description
`CC, SCC`	Returns January 1, *cc*01 where *cc* is first 2 digits of the given year if last 2 digits <= 50, or 1 greater than the first 2 digits of the given year if last 2 digits > 50; (for AD years)
`SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y`	Returns January 1, *yyyy* where *yyyy* is rounded to the nearest year; rounds down on June 30, rounds up on July 1
`IYYY, IYY, IY, I`	Rounds to the beginning of the ISO year which is determined by rounding down if the month and day is on or before June 30th, or by rounding up if the month and day is July 1st or later
`Q`	Returns the first day of the quarter determined by rounding down if the month and day is on or before the 15th of the second month of the quarter, or by rounding up if the month and day is on the 16th of the second month or later of the quarter
`MONTH, MON, MM, RM`	Returns the first day of the specified month if the day of the month is on or prior to the 15th; returns the first day of the following month if the day of the month is on the 16th or later
`WW`	Round to the nearest date that corresponds to the same day of the week as the first day of the year
`IW`	Round to the nearest date that corresponds to the same day of the week as the first day of the ISO year
`W`	Round to the nearest date that corresponds to the same day of the week as the first day of the month
`DDD, DD, J`	Rounds to the start of the nearest day; 11:59:59 AM or earlier rounds to the start of the same day; 12:00:00 PM or later rounds to the start of the next day
`DAY, DY, D`	Rounds to the nearest Sunday
`HH, HH12, HH24`	Round to the nearest hour
`MI`	Round to the nearest minute

Following are examples of usage of the `ROUND` function.

The following examples round to the nearest hundred years.

```
SELECT TO_CHAR(ROUND(TO_DATE('1950','YYYY'),'CC'),'DD-MON-YYYY') "Century"
FROM DUAL;

   Century
-------------
 01-JAN-1901
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century"
FROM DUAL;

   Century
-------------
 01-JAN-2001
(1 row)
```

The following examples round to the nearest year.

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY')
"Year" FROM DUAL;
```



```
    Year
-------------
 01-JAN-1999
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY')
"Year" FROM DUAL;

    Year
-------------
 01-JAN-2000
(1 row)
```

The following examples round to the nearest ISO year. The first example rounds to 2004 and the ISO year for 2004 begins on December 29th of 2003. The second example rounds to 2005 and the ISO year for 2005 begins on January 3rd of that same year.

(An ISO year begins on the first Monday from which a 7 day span, Monday thru Sunday, contains at least 4 days of the new year. Thus, it is possible for the beginning of an ISO year to start in December of the prior year.)

```
SELECT TO_CHAR(ROUND(TO_DATE('30-JUN-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-
YYYY') "ISO Year" FROM DUAL;

  ISO Year
-------------
 29-DEC-2003
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-
YYYY') "ISO Year" FROM DUAL;

  ISO Year
-------------
 03-JAN-2005
(1 row)
```

The following examples round to the nearest quarter.

```
SELECT ROUND(TO_DATE('15-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-APR-07 00:00:00
(1 row)
```

The following examples round to the nearest month.

```
SELECT ROUND(TO_DATE('15-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;
```

```
       Month
--------------------
 01-DEC-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
 01-JAN-08 00:00:00
(1 row)
```

The following examples round to the nearest week. The first day of 2007 lands on a Monday so in the first example, January 18th is closest to the Monday that lands on January 15th. In the second example, January 19th is closer to the Monday that falls on January 22nd.

```
SELECT ROUND(TO_DATE('18-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 15-JAN-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
 22-JAN-07 00:00:00
(1 row)
```

The following examples round to the nearest ISO week. An ISO week begins on a Monday. In the first example, January 1, 2004 is closest to the Monday that lands on December 29, 2003. In the second example, January 2, 2004 is closer to the Monday that lands on January 5, 2004.

```
SELECT ROUND(TO_DATE('01-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 29-DEC-03 00:00:00
(1 row)

SELECT ROUND(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 05-JAN-04 00:00:00
(1 row)
```

The following examples round to the nearest week where a week is considered to start on the same day as the first day of the month.

```
SELECT ROUND(TO_DATE('05-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
```

```
--------------------
 08-MAR-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
 01-MAR-07 00:00:00
(1 row)
```

The following examples round to the nearest day.

```
SELECT ROUND(TO_DATE('04-AUG-07 11:59:59 AM','DD-MON-YY HH:MI:SS AM'),'J')
"Day" FROM DUAL;

        Day
--------------------
 04-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J')
"Day" FROM DUAL;

        Day
--------------------
 05-AUG-07 00:00:00
(1 row)
```

The following examples round to the start of the nearest day of the week (Sunday).

```
SELECT ROUND(TO_DATE('08-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

    Day of Week
--------------------
 05-AUG-07 00:00:00
(1 row)

SELECT ROUND(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

    Day of Week
--------------------
 12-AUG-07 00:00:00
(1 row)
```

The following examples round to the nearest hour.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:29','DD-MON-YY HH:MI'),'HH'),'DD-
MON-YY HH24:MI:SS') "Hour" FROM DUAL;

        Hour
--------------------
 09-AUG-07 08:00:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-
MON-YY HH24:MI:SS') "Hour" FROM DUAL;

        Hour
```



```
--------------------
 09-AUG-07 09:00:00
(1 row)
```

The following examples round to the nearest minute.

```
SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:29','DD-MON-YY
HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

       Minute
--------------------
 09-AUG-07 08:30:00
(1 row)

SELECT TO_CHAR(ROUND(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY
HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

       Minute
--------------------
 09-AUG-07 08:31:00
(1 row)
```

3.3.8.7 TRUNC

The `TRUNC` function returns a date truncated according to a specified template pattern. If the template pattern is omitted, the date is truncated to the nearest day. The following table shows the template patterns for the `TRUNC` function.

Table 3-3-17 Template Date Patterns for the TRUNC Function

Pattern	Description
`CC, SCC`	Returns January 1, *cc*01 where *cc* is first 2 digits of the given year
`SYYY, YYYY, YEAR, SYEAR, YYY, YY, Y`	Returns January 1, *yyyy* where *yyyy* is the given year
`IYYY, IYY, IY, I`	Returns the start date of the ISO year containing the given date
`Q`	Returns the first day of the quarter containing the given date
`MONTH, MON, MM, RM`	Returns the first day of the specified month
`WW`	Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the year
`IW`	Returns the start of the ISO week containing the given date
`W`	Returns the largest date just prior to, or the same as the given date that corresponds to the same day of the week as the first day of the month
`DDD, DD, J`	Returns the start of the day for the given date
`DAY, DY, D`	Returns the start of the week (Sunday) containing the given date
`HH, HH12, HH24`	Returns the start of the hour
`MI`	Returns the start of the minute

Following are examples of usage of the `TRUNC` function.

The following example truncates down to the hundred years unit.

```
SELECT TO_CHAR(TRUNC(TO_DATE('1951','YYYY'),'CC'),'DD-MON-YYYY') "Century"
FROM DUAL;

   Century
-------------
 01-JAN-1901
(1 row)
```

The following example truncates down to the year.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-1999','DD-MON-YYYY'),'Y'),'DD-MON-YYYY')
"Year" FROM DUAL;

    Year
-------------
 01-JAN-1999
(1 row)
```

The following example truncates down to the beginning of the ISO year.

```
SELECT TO_CHAR(TRUNC(TO_DATE('01-JUL-2004','DD-MON-YYYY'),'IYYY'),'DD-MON-
YYYY') "ISO Year" FROM DUAL;

  ISO Year
-------------
 29-DEC-2003
(1 row)
```

The following example truncates down to the start date of the quarter.

```
SELECT TRUNC(TO_DATE('16-FEB-07','DD-MON-YY'),'Q') "Quarter" FROM DUAL;

      Quarter
--------------------
 01-JAN-07 00:00:00
(1 row)
```

The following example truncates to the start of the month.

```
SELECT TRUNC(TO_DATE('16-DEC-07','DD-MON-YY'),'MONTH') "Month" FROM DUAL;

       Month
--------------------
 01-DEC-07 00:00:00
(1 row)
```

The following example truncates down to the start of the week determined by the first day of the year. The first day of 2007 lands on a Monday so the Monday just prior to January 19th is January 15th.

```
SELECT TRUNC(TO_DATE('19-JAN-07','DD-MON-YY'),'WW') "Week" FROM DUAL;

        Week
--------------------
```

```
 15-JAN-07 00:00:00
(1 row)
```

The following example truncates to the start of an ISO week. An ISO week begins on a Monday. January 2, 2004 falls in the ISO week that starts on Monday, December 29, 2003.

```
SELECT TRUNC(TO_DATE('02-JAN-04','DD-MON-YY'),'IW') "ISO Week" FROM DUAL;

      ISO Week
--------------------
 29-DEC-03 00:00:00
(1 row)
```

The following example truncates to the start of the week where a week is considered to start on the same day as the first day of the month.

```
SELECT TRUNC(TO_DATE('21-MAR-07','DD-MON-YY'),'W') "Week" FROM DUAL;

        Week
--------------------
 15-MAR-07 00:00:00
(1 row)
```

The following example truncates to the start of the day.

```
SELECT TRUNC(TO_DATE('04-AUG-07 12:00:00 PM','DD-MON-YY HH:MI:SS AM'),'J')
"Day" FROM DUAL;

        Day
--------------------
 04-AUG-07 00:00:00
(1 row)
```

The following example truncates to the start of the week (Sunday).

```
SELECT TRUNC(TO_DATE('09-AUG-07','DD-MON-YY'),'DAY') "Day of Week" FROM DUAL;

    Day of Week
--------------------
 05-AUG-07 00:00:00
(1 row)
```

The following example truncates to the start of the hour.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30','DD-MON-YY HH:MI'),'HH'),'DD-
MON-YY HH24:MI:SS') "Hour" FROM DUAL;

        Hour
--------------------
 09-AUG-07 08:00:00
(1 row)
```

The following example truncates to the minute.

```
SELECT TO_CHAR(TRUNC(TO_DATE('09-AUG-07 08:30:30','DD-MON-YY
HH:MI:SS'),'MI'),'DD-MON-YY HH24:MI:SS') "Minute" FROM DUAL;

       Minute
--------------------
 09-AUG-07 08:30:00
(1 row)
```

3.3.8.8 CURRENT DATE/TIME

Postgres Plus Advanced Server provides a number of functions that return values related to the current date and time. These functions all return values based on the start time of the current transaction.

- `CURRENT_DATE`
- `CURRENT_TIMESTAMP`
- `LOCALTIMESTAMP`
- `LOCALTIMESTAMP(precision)`

`CURRENT_DATE` returns the current date and time based on the start time of the current transaction. The value of `CURRENT_DATE` will not change if called multiple times within a transaction.

```
SELECT CURRENT_DATE FROM DUAL;

   date
-----------
 06-AUG-07
```

`CURRENT_TIMESTAMP` returns the current date and time. When called from a single SQL statement, it will return the same value for each occurrence within the statement. If called from multiple statements within a transaction, may return different values for each occurrence. If called from a function, may return a different value than the value returned by `CURRENT_TIMESTAMP` in the caller.

```
SELECT CURRENT_TIMESTAMP, CURRENT_TIMESTAMP FROM DUAL;

                current_timestamp | current_timestamp
----------------------------------+----------------------------------
 02-SEP-13 17:52:28.361473 +05:00 | 02-SEP-13 17:52:28.361474 +05:00
```

`LOCALTIMESTAMP` can optionally be given a precision parameter which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

```
SELECT LOCALTIMESTAMP FROM DUAL;

       timestamp
```

```
------------------------
 06-AUG-07 16:11:35.973
(1 row)

SELECT LOCALTIMESTAMP(2) FROM DUAL;

       timestamp
-----------------------
 06-AUG-07 16:11:44.58
(1 row)
```

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp. Other database systems may advance these values more frequently.

3.3.8.9 NUMTODSINTERVAL

The `NUMTODSINTERVAL` function converts a numeric value to a time interval that includes day through second interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are `DAY`, `HOUR`, `MINUTE`, and `SECOND`.

The following example converts a numeric value to a time interval that includes days and hours:

```
SELECT numtodsinterval(100, 'hour');
numtodsinterval
---------------
4 days 04:00:00
(1 row)
```

The following example converts a numeric value to a time interval that includes minutes and seconds:

```
SELECT numtodsinterval(100, 'second');
numtodsinterval
---------------
1 min 40 secs
(1 row)
```

3.3.8.10 NUMTOYMINTERVAL

The `NUMTOYMINTERVAL` function converts a numeric value to a time interval that includes year through month interval units. When calling the function, specify the smallest fractional interval type to be included in the result set. The valid interval types are YEAR and MONTH.

The following example converts a numeric value to a time interval that includes years and months:

```
SELECT numtoyminterval(100, 'month');
numtoyminterval
---------------
8 years 4 mons
(1 row)
```

The following example converts a numeric value to a time interval that includes years only:

```
SELECT numtoyminterval(100, 'year');
numtoyminterval
---------------
100 years
(1 row)
```


3.3.9 Sequence Manipulation Functions

This section describes Postgres Plus Advanced Server's functions for operating on sequence objects. Sequence objects (also called sequence generators or just sequences) are special single-row tables created with the `CREATE SEQUENCE` command. A sequence object is usually used to generate unique identifiers for rows of a table. The sequence functions, listed below, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

```
sequence.NEXTVAL
sequence.CURRVAL
```

sequence is the identifier assigned to the sequence in the `CREATE SEQUENCE` command. The following describes the usage of these functions.

`NEXTVAL`

> Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute `NEXTVAL` concurrently, each will safely receive a distinct sequence value.

`CURRVAL`

> Return the value most recently obtained by `NEXTVAL` for this sequence in the current session. (An error is reported if `NEXTVAL` has never been called for this sequence in this session.) Notice that because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed `NEXTVAL` since the current session did.

If a sequence object has been created with default parameters, `NEXTVAL` calls on it will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the `CREATE SEQUENCE` command.

Important: To avoid blocking of concurrent transactions that obtain numbers from the same sequence, a `NEXTVAL` operation is never rolled back; that is, once a value has been fetched it is considered used, even if the transaction that did the `NEXTVAL` later aborts. This means that aborted transactions may leave unused "holes" in the sequence of assigned values.

3.3.10 Conditional Expressions

The following section describes the SQL-compliant conditional expressions available in Postgres Plus Advanced Server.

3.3.10.1 CASE

The SQL `CASE` expression is a generic conditional expression, similar to if/else statements in other languages:

```
CASE WHEN condition THEN result
    [ WHEN ... ]
    [ ELSE result ]
END
```

`CASE` clauses can be used wherever an expression is valid. *`condition`* is an expression that returns a `BOOLEAN` result. If the result is `TRUE` then the value of the `CASE` expression is the *`result`* that follows the condition. If the result is `FALSE` any subsequent `WHEN` clauses are searched in the same manner. If no `WHEN` *`condition`* is `TRUE` then the value of the `CASE` expression is the *`result`* in the `ELSE` clause. If the `ELSE` clause is omitted and no condition matches, the result is `NULL`.

An example:

```
SELECT * FROM test;

 a
---
 1
 2
 3
(3 rows)

SELECT a,
    CASE WHEN a=1 THEN 'one'
         WHEN a=2 THEN 'two'
         ELSE 'other'
    END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

The data types of all the *`result`* expressions must be convertible to a single output type.

The following “simple” `CASE` expression is a specialized variant of the general form above:

```
CASE expression
    WHEN value THEN result
  [ WHEN ... ]
  [ ELSE result ]
END
```

The *expression* is computed and compared to all the *value* specifications in the `WHEN` clauses until one is found that is equal. If no match is found, the *result* in the `ELSE` clause (or a null value) is returned.

The example above can be written using the simple `CASE` syntax:

```
SELECT a,
    CASE a WHEN 1 THEN 'one'
           WHEN 2 THEN 'two'
           ELSE 'other'
    END
FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
(3 rows)
```

A `CASE` expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

3.3.10.2 COALESCE

The `COALESCE` function returns the first of its arguments that is not null. Null is returned only if all arguments are null.

```
COALESCE(value [, value2 ] ... )
```

It is often used to substitute a default value for null values when data is retrieved for display or further computation. For example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Like a `CASE` expression, `COALESCE` will not evaluate arguments that are not needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

3.3.10.3 NULLIF

The NULLIF function returns a null value if *value1* and *value2* are equal; otherwise it returns *value1*.

```
NULLIF(value1, value2)
```

This can be used to perform the inverse operation of the COALESCE example given above:

```
SELECT NULLIF(value1, '(none)') ...
```

If *value1* is (none), return a null, otherwise return *value1*.

3.3.10.4 NVL

The NVL function returns the first of its arguments that is not null. NVL evaluates the first expression; if that expression evaluates to NULL, NVL returns the second expression.

```
NVL(expr1, expr2)
```

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type). NVL returns NULL if all arguments are NULL.

The following example computes a bonus for non-commissioned employees, If an employee is a commissioned employee, this expression returns the employees commission; if the employee is not a commissioned employee (that is, his commission is NULL), this expression returns a bonus that is 10% of his salary.

```
bonus = NVL(emp.commission, emp.salary * .10)
```

3.3.10.5 NVL2

NVL2 evaluates an expression, and returns either the second or third expression, depending on the value of the first expression. If the first expression is not NULL, NVL2 returns the value in *expr2*; if the first expression is NULL, NVL2 returns the value in *expr3*.

```
NVL2(expr1, expr2, expr3)
```

The return type is the same as the argument types; all arguments must have the same data type (or be coercible to a common type).

The following example computes a bonus for commissioned employees - if a given employee is a commissioned employee, this expression returns an amount equal to 110% of his commission; if the employee is not a commissioned employee (that is, his commission is `NULL`), this expression returns `0`.

```
bonus = NVL2(emp.commission, emp.commission * 1.1, 0)
```

3.3.10.6 GREATEST and LEAST

The `GREATEST` and `LEAST` functions select the largest or smallest value from a list of any number of expressions.

```
GREATEST(value [, value2 ] ... )
LEAST(value [, value2 ] ... )
```

The expressions must all be convertible to a common data type, which will be the type of the result. Null values in the list are ignored. The result will be null only if all the expressions evaluate to null.

Note that `GREATEST` and `LEAST` are not in the SQL standard, but are a common extension.

3.3.11 Aggregate Functions

Aggregate functions compute a single result value from a set of input values. The built-in aggregate functions are listed in the following tables.

Table 3-3-18 General-Purpose Aggregate Functions

Function	Argument Type	Return Type	Description
AVG(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	NUMBER for any integer type, DOUBLE PRECISION for a floating-point argument, otherwise the same as the argument data type	The average (arithmetic mean) of all input values
COUNT(*)		BIGINT	Number of input rows
COUNT(*expression*)	Any	BIGINT	Number of input rows for which the value of expression is not null
MAX(*expression*)	Any numeric, string, date/time, or bytea type	Same as argument type	Maximum value of expression across all input values
MIN(*expression*)	Any numeric, string, date/time, or bytea type	Same as argument type	Minimum value of expression across all input values
SUM(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	BIGINT for SMALLINT or INTEGER arguments, NUMBER for BIGINT arguments, DOUBLE PRECISION for floating-point arguments, otherwise the same as the argument data type	Sum of expression across all input values

It should be noted that except for COUNT, these functions return a null value when no rows are selected. In particular, SUM of no rows returns null, not zero as one might expect. The COALESCE function may be used to substitute zero for null when necessary.

The following table shows the aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Where the description mentions *N*, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when *N* is zero.

Table 3-3-19 Aggregate Functions for Statistics

Function	Argument Type	Return Type	Description
CORR(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Correlation coefficient
COVAR_POP(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Population covariance
COVAR_SAMP(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sample covariance
REGR_AVGX(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Average of the independent variable (sum(*X*) / *N*)

Function	Argument Type	Return Type	Description
REGR_AVGY(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Average of the dependent variable (sum(*Y*) / *N*)
REGR_COUNT(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Number of input rows in which both expressions are nonnull
REGR_INTERCEPT(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	y-intercept of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
REGR_R2(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Square of the correlation coefficient
REGR_SLOPE(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Slope of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
REGR_SXX(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (X^2) – sum (X)^2 / N ("sum of squares" of the independent variable)
REGR_SXY(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (X*Y) – sum (X) * sum (Y) / N ("sum of products" of independent times dependent variable)
REGR_SYY(*Y*, *X*)	DOUBLE PRECISION	DOUBLE PRECISION	Sum (Y^2) – sum (Y)^2 / N ("sum of squares" of the dependent variable)
STDDEV(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Historic alias for STDDEV_SAMP
STDDEV_POP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Population standard deviation of the input values
STDDEV_SAMP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Sample standard deviation of the input values
VARIANCE(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Historical alias for VAR_SAMP
VAR_POP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Population variance of the input values (square of the population standard deviation)
VAR_SAMP(*expression*)	INTEGER, REAL, DOUBLE PRECISION, NUMBER	DOUBLE PRECISION for floating-point arguments, otherwise NUMBER	Sample variance of the input values (square of the sample standard deviation)

3.3.12 Subquery Expressions

This section describes the SQL-compliant subquery expressions available in Postgres Plus Advanced Server. All of the expression forms documented in this section return Boolean (`TRUE`/`FALSE`) results.

3.3.12.1 EXISTS

The argument of `EXISTS` is an arbitrary `SELECT` statement, or subquery. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of `EXISTS` is `TRUE`; if the subquery returns no rows, the result of `EXISTS` is `FALSE`.

```
EXISTS(subquery)
```

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed far enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has any side effects (such as calling sequence functions); whether the side effects occur or not may be difficult to predict.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally uninteresting. A common coding convention is to write all `EXISTS` tests in the form `EXISTS(SELECT 1 WHERE ...)`. There are exceptions to this rule however, such as subqueries that use `INTERSECT`.

This simple example is like an inner join on `deptno`, but it produces at most one output row for each `dept` row, even though there are multiple matching `emp` rows:

```
SELECT dname FROM dept WHERE EXISTS (SELECT 1 FROM emp WHERE emp.deptno =
dept.deptno);

   dname
------------
 ACCOUNTING
 RESEARCH
 SALES
(3 rows)
```

3.3.12.2 IN

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of `IN` is `TRUE` if any equal subquery row is found. The result is `FALSE` if no equal row is found (including the special case where the subquery returns no rows).

```
expression IN (subquery)
```

Note that if the left-hand expression yields `NULL`, or if there are no equal right-hand values and at least one right-hand row yields `NULL`, the result of the `IN` construct will be `NULL`, not `FALSE`. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, it's unwise to assume that the subquery will be evaluated completely.

3.3.12.3 NOT IN

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of `NOT IN` is `TRUE` if only unequal subquery rows are found (including the special case where the subquery returns no rows). The result is `FALSE` if any equal row is found.

```
expression NOT IN (subquery)
```

Note that if the left-hand expression yields `NULL`, or if there are no equal right-hand values and at least one right-hand row yields `NULL`, the result of the `NOT IN` construct will be `NULL`, not `TRUE`. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, it's unwise to assume that the subquery will be evaluated completely.

3.3.12.4 ANY/SOME

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of `ANY` is `TRUE` if any true result is obtained. The result is `FALSE` if no true result is found (including the special case where the subquery returns no rows).

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

`SOME` is a synonym for `ANY`. `IN` is equivalent to `= ANY`.

Note that if there are no successes and at least one right-hand row yields `NULL` for the operator's result, the result of the `ANY` construct will be `NULL`, not `FALSE`. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with `EXISTS`, it's unwise to assume that the subquery will be evaluated completely.

3.3.12.5 ALL

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of `ALL` is `TRUE` if all rows yield true (including the special case where the subquery returns no rows). The result is `FALSE` if any false result is found. The result is `NULL` if the comparison does not return `FALSE` for any row, and it returns `NULL` for at least one row.

```
expression operator ALL (subquery)
```

`NOT IN` is equivalent to `<> ALL`. As with `EXISTS`, it's unwise to assume that the subquery will be evaluated completely.

3.3.13 Uniform Resource Locator Functions

This section describes functions that perform operations based on the Uniform Resource Locator. The *Uniform Resource Locator* (URL) is a character string that provides the address of a network resource. Typical usage of URLs is for web pages, which are accessed using the *Hypertext Transfer Protocol* (HTTP) protocol.

3.3.13.1 EDB_GET_URL_AS_BYTEA

The `EDB_GET_URL_AS_BYTEA` function returns content from the user-specified URL in one continuous `BYTEA` string. The signature is:

```
BYTEA EDB_GET_URL_AS_BYTEA
(
  url     TEXT
)
```

Parameters

url

url is the Uniform Resource Locator from which the function will return content.

Example

The following function retrieves the content of the specified URL and displays the first few lines in chunks of 40 bytes (80 hexadecimal characters) per line. The function also returns the total content length in number of bytes.

```
CREATE OR REPLACE FUNCTION get_url_bytea(
    p_url           TEXT
) RETURNS INTEGER
AS $$
DECLARE
    v_data          BYTEA;
    v_line          TEXT;
    v_start         INTEGER;
    v_line_count    INTEGER := 0;
    v_line_length   INTEGER := 40;
BEGIN
    v_data := EDB_GET_URL_AS_BYTEA(p_url);
    FOR i IN 1 .. 10 LOOP
        v_start := (v_line_count * v_line_length) + 1;
        v_line := SUBSTR(v_data, v_start, v_line_length);
        RAISE INFO '%', v_line;
        v_line_count := v_line_count + 1;
    END LOOP;
    RETURN OCTET_LENGTH(v_data);
END;
$$ LANGUAGE 'plpgsql';
```


The following is the output from the example.

```
edb=# SELECT get_url_bytea('http://www.enterprisedb.com');
INFO:  \x3c21444f43545950452068746d6c205055424c494320222d2f2f5733432f2f445444205848544d4c
INFO:  \x20312e30205374726963742f2f454e220d0a202022687474703a2f2f7777772e77332e6f72672f54
INFO:  \x522f7868746d6c312f4454442f7868746d6c312d7374726963742e647464223e0d0a3c68746d6c20
INFO:  \x786d6c6e733d22687474703a2f2f7777772e77332e6f72672f313939392f7868746d6c2220786d6c
INFO:  \x3a6c616e673d22656e22206c616e673d22656e22206469723d226c7472223e0d0a0d0a20203c212d
INFO:  \x2d205f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f2048454144205f5f5f5f5f
INFO:  \x5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f5f202d2d3e0d0a0d0a20203c686561643e0a3c
INFO:  \x6d65746120687474702d65717569763d22436f6e74656e742d547970652220636f6e74656e743d22
INFO:  \x746578742f68746d6c3b20636861727365743d7574662d3822202f3e0d0a0d0a0d0a202020203c74
INFO:  \x69746c653e456e74657270726973654442207c2054686520506f73746772657320446174616173
 get_url_bytea
---------------
         84216
(1 row)
```

3.3.13.2 EDB_GET_URL_AS_TEXT

The `EDB_GET_URL_AS_TEXT` function returns content from the user-specified URL in one continuous `TEXT` string. The signature is:

```
TEXT EDB_GET_URL_AS_TEXT
(
  url     TEXT
)
```

Parameters

url

> *url* is the Uniform Resource Locator from which the function will return content.

Example

The following function retrieves the content of the specified URL and displays the first few lines in chunks of 80 characters per line. The function also returns the total content length in number of bytes.

```
CREATE OR REPLACE FUNCTION get_url_text(
    p_url           TEXT
) RETURNS INTEGER
AS $$
DECLARE
    v_data          TEXT;
    v_line          TEXT;
    v_start         INTEGER;
    v_line_count    INTEGER := 0;
    v_line_length   INTEGER := 80;
BEGIN
    v_data := EDB_GET_URL_AS_TEXT(p_url);
    FOR i IN 1 .. 5 LOOP
        v_start := (v_line_count * v_line_length) + 1;
```



```
        v_line := SUBSTR(v_data, v_start, v_line_length);
        RAISE INFO '%', v_line;
        v_line_count := v_line_count + 1;
    END LOOP;
    RETURN OCTET_LENGTH(v_data);
END;
$$ LANGUAGE 'plpgsql';
```

The following is the output from the example.

```
edb=# SELECT get_url_text('http://www.enterprisedb.com');
INFO:  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/T
INFO:  R/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml
INFO:  :lang="en" lang="en" dir="ltr">

  <!-- ___________________________ HEAD _____
INFO:  ______________________ -->

  <head>
<meta http-equiv="Content-Type" content="
INFO:  text/html; charset=utf-8" />


    <title>EnterpriseDB | The Postgres Databas
 get_url_text
--------------
        83892
(1 row)
```

3.4 Table Partitioning

In a partitioned table, one logically large table is broken into smaller physical pieces. Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning allows you to omit the partition column from the front of an index, reducing index size and making it more likely that the heavily used parts of the index fits in memory.

- When a query or update accesses a large percentage of a single partition, performance may improve because the server will perform a sequential scan of the partition instead of using an index and random access reads scattered across the whole table.

- A bulk load (or unload) can be implemented by adding or removing partitions, if you plan that requirement into the partitioning design. `ALTER TABLE` is far faster than a bulk operation. It also entirely avoids the `VACUUM` overhead caused by a bulk `DELETE`.

- Seldom-used data can be migrated to less-expensifve (or slower) storage media.

Table partitioning is worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application; a good rule of thumb is that the size of the table should exceed the physical memory of the database server.

This document discusses the Oracle-compatible aspects of table partitioning that are supported by Advanced Server.

The PostgreSQL 9.5 `INSERT… ON CONFLICT DO NOTHING/UPDATE` clause (commonly known as UPSERT) is not supported on Oracle-styled partitioned tables. If you include the `ON CONFLICT DO NOTHING/UPDATE` clause when invoking the `INSERT` command to add data to a partitioned table, the server will return an error.

3.4.1 Selecting a Partition Type

When you create a partitioned table, you specify LIST, RANGE, or HASH partitioning rules. The partitioning rules provide a set of constraints that define the data that is stored in each partition. As new rows are added to the partitioned table, the server uses the partitioning rules to decide which partition should contain each row.
Advanced Server can also use the partitioning rules to enforce partition pruning, improving performance when responding to user queries. When selecting a partitioning type and partitioning keys for a table, you should take into consideration how the data that is stored within a table will be queried, and include often-queried columns in the partitioning rules.

List Partitioning

When you create a list-partitioned table, you specify a single partitioning key column. When adding a row to the table, the server compares the key values specified in the partitioning rule to the corresponding column within the row. If the column value matches a value in the partitioning rule, the row is stored in the partition named in the rule.

Range Partitioning

When you create a range-partitioned table, you specify one or more partitioning key columns. When you add a new row to the table, the server compares the value of the partitioning key (or keys) to the corresponding column (or columns) in a table entry. If the column values satisfy the conditions specified in the partitioning rule, the row is stored in the partition named in the rule.

Hash Partitioning

When you create a hash-partitioned table, you specify one or more partitioning key columns. Data is divided into (approx.) equal-sized partitions amongst the specified partitions. When you add a row to a hash-partitioned table, the server computes a hash value for the data in the specified column (or columns), and stores the row in a partition according to the hash value.

Subpartitioning

Subpartitioning breaks a partitioned table into smaller subsets that may or may not be stored on the same server. A table is typically subpartitioned by a different set of columns, and may be a different subpartitioning type than the parent partition. If one partition is subpartitioned, then each partition will have at least one subpartition.

If a table is subpartitioned, no data will be stored in any of the partition tables; the data will be stored instead in the corresponding subpartitions.

3.4.2 Using Partition Pruning

Advanced Server's query planner uses *partition pruning* to compute an efficient plan to locate a row (or rows) that matches the conditions specified in the `WHERE` clause of a `SELECT` statement. To successfully prune partitions from an execution plan, the `WHERE` clause must constrain the information that is compared to the partitioning key column specified when creating the partitioned table. When querying a:

- list-partitioned table, partition pruning is effective when the `WHERE` clause compares a literal value to the partitioning key using operators like equal (=) or `AND`.

- range-partitioned table, partition pruning is effective when the `WHERE` clause compares a literal value to a partitioning key using operators such as equal (=), less than (<), or greater than (>).

- hash-partitioned table, partition pruning is effective when the `WHERE` clause compares a literal value to the partitioning key using an operator such as equal (=).

The partition pruning mechanism uses two optimization techniques:

- Fast Pruning

- Constraint exclusion

Partition pruning techniques limit the search for data to only those partitions in which the values for which you are searching might reside. Both pruning techniques remove partitions from a query's execution plan, increasing performance.

The difference between the fast pruning and constraint exclusion is that fast pruning understands the relationship between the partitions in an Oracle-partitioned table, while constraint exclusion does not. For example, when a query searches for a specific value within a list-partitioned table, fast pruning can reason that only a specific partition may hold that value, while constraint exclusion must examine the constraints defined for each partition. Fast pruning occurs early in the planning process to reduce the number of partitions that the planner must consider, while constraint exclusion occurs late in the planning process.

Using Constraint Exclusion

The `constraint_exclusion` parameter controls constraint exclusion. The `constraint_exclusion` parameter may have a value of `on`, `off`, or `partition`. To enable constraint exclusion, the parameter must be set to *either* `partition` or `on`. By default, the parameter is set to `partition`.

For more information about constraint exclusion, see:

http://www.enterprisedb.com/docs/en/9.4/pg/ddl-partitioning.html

When constraint exclusion is enabled, the server examines the constraints defined for each partition to determine if that partition can satisfy a query.

When you execute a `SELECT` statement that *does not* contain a `WHERE` clause, the query planner must recommend an execution plan that searches the entire table. When you execute a `SELECT` statement that *does* contain a `WHERE` clause, the query planner determines in which partition that row would be stored, and sends query fragments to that partition, pruning the partitions that could not contain that row from the execution plan. If you are are not using partitioned tables, disabling constraint exclusion may improve performance.

Fast Pruning

Like constraint exclusion, fast pruning can only optimize queries that include a `WHERE` (or join) clause, and only when the qualifiers in the `WHERE` clause match a certain form. In both cases, the query planner will avoid searching for data within partitions that cannot possibly hold the data required by the query.

Fast pruning is controlled by a boolean configuration parameter named `edb_enable_pruning`. If `edb_enable_pruning` is `ON`, Advanced Server will fast prune certain queries. If `edb_enable_pruning` is `OFF`, the server will disable fast pruning.

Please note: Fast pruning cannot optimize queries against subpartitioned tables or optimize queries against range-partitioned tables that are partitioned on more than one column.

For LIST-partitioned tables, Advanced Server can fast prune queries that contain a `WHERE` clause that constrains a partitioning column to a literal value. For example, given a LIST-partitioned table such as:

```
CREATE TABLE sales_hist(..., country text, ...)
PARTITION BY LIST(country)
(
    PARTITION americas VALUES('US', 'CA', 'MX'),
    PARTITION europe VALUES('BE', 'NL', 'FR'),
    PARTITION asia VALUES('JP', 'PK', 'CN'),
    PARTITION others VALUES(DEFAULT)
)
```


Fast pruning can reason about `WHERE` clauses such as:

```
WHERE country = 'US'
WHERE country IS NULL;
```

Given the first `WHERE` clause, fast pruning would eliminate partitions `europe`, `asia`, and `others` because those partitions cannot hold rows that satisfy the qualifier: `WHERE country = 'US'`.

Given the second `WHERE` clause, fast pruning would eliminate partitions `americas`, `europe`, and `asia` because because those partitions cannot hold rows where `country IS NULL`.

The operator specified in the `WHERE` clause must be an equal sign (=) or the equality operator appropriate for the data type of the partitioning column.

For range-partitioned tables, Advanced Server can fast prune queries that contain a `WHERE` clause that constrains a partitioning column to a literal value, but the operator may be any of the following:

```
>
>=
=
<=
<
```

Fast pruning will also reason about more complex expressions involving `AND` and `BETWEEN` operators, such as:

```
WHERE size > 100 AND size <= 200
WHERE size BETWEEN 100 AND 200
```

But cannot prune based on expressions involving `OR` or `IN`.

For example, when querying a RANGE-partitioned table, such as:

```
CREATE TABLE boxes(id int, size int, color text)
  PARTITION BY RANGE(size)
(
    PARTITION small VALUES LESS THAN(100),
    PARTITION medium VALUES LESS THAN(200),
    PARTITION large VALUES LESS THAN(300)
)
```

Fast pruning can reason about `WHERE` clauses such as:

```
WHERE size > 100     -- scan partitions 'medium' and 'large'
```

```
WHERE size >= 100    -- scan partitions 'medium' and 'large'
WHERE size = 100     -- scan partition 'medium'
WHERE size <= 100    -- scan partitions 'small' and 'medium'
WHERE size < 100     -- scan partition 'small'
WHERE size > 100 AND size < 199     -- scan partition 'medium'
WHERE size BETWEEN 100 AND 199      -- scan partition 'medium'
WHERE color = 'red' AND size = 100  -- scan 'medium'
WHERE color = 'red' AND (size > 100 AND size < 199) -- scan
'medium'
```

In each case, fast pruning requires that the qualifier must refer to a partitioning column and literal value (or `IS NULL`/`IS NOT NULL`).

Note that fast pruning can also optimize `DELETE` and `UPDATE` statements containing `WHERE` clauses of the forms described above.

3.4.3 Example - Partition Pruning

The `EXPLAIN` statement displays the execution plan of a statement. You can use the `EXPLAIN` statement to confirm that Advanced Server is pruning partitions from the execution plan of a query.

To demonstrate the efficiency of partition pruning, first create a simple table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Then, perform a constrained query that includes the `EXPLAIN` statement:

```
EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'INDIA';
```

The resulting query plan shows that the server will scan only the `sales_asia` table - the table in which a row with a `country` value of `INDIA` would be stored:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'INDIA';
                    QUERY PLAN
---------------------------------------------------
 Append
   ->  Seq Scan on sales
         Filter: ((country)::text = 'INDIA'::text)
   ->  Seq Scan on sales_asia
         Filter: ((country)::text = 'INDIA'::text)
(5 rows)
```

If you perform a query that searches for a row that matches a value not included in the partitioning key:

```
EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE dept_no = '30';
```

The resulting query plan shows that the server must look in all of the partitions to locate the rows that satisfy the query:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE dept_no = '30';
               QUERY PLAN
-----------------------------------------
 Append
   ->  Seq Scan on sales
         Filter: (dept_no = 30::numeric)
   ->  Seq Scan on sales_europe
         Filter: (dept_no = 30::numeric)
   ->  Seq Scan on sales_asia
         Filter: (dept_no = 30::numeric)
   ->  Seq Scan on sales_americas
         Filter: (dept_no = 30::numeric)
(9 rows)
```

Constraint exclusion also applies when querying subpartitioned tables:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date) SUBPARTITION BY LIST(country)
(
  PARTITION "2011" VALUES LESS THAN('01-JAN-2012')
  (
    SUBPARTITION europe_2011 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2011 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2011 VALUES ('US', 'CANADA')
  ),
  PARTITION "2012" VALUES LESS THAN('01-JAN-2013')
  (
    SUBPARTITION europe_2012 VALUES ('ITALY', 'FRANCE'),
```



```
    SUBPARTITION asia_2012 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2012 VALUES ('US', 'CANADA')
  ),
  PARTITION "2013" VALUES LESS THAN('01-JAN-2014')
  (
    SUBPARTITION europe_2013 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2013 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2013 VALUES ('US', 'CANADA')
  )
);
```

When you query the table, the query planner prunes any partitions or subpartitions from the search path that cannot possibly contain the desired result set:

```
edb=# EXPLAIN (COSTS OFF) SELECT * FROM sales WHERE country = 'US' AND date =
'Dec 12, 2012';
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Append
   ->  Seq Scan on sales
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12
00:00:00'::timestamp without time zone))
   ->  Seq Scan on sales_2012
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12
00:00:00'::timestamp without time zone))
   ->  Seq Scan on sales_americas_2012
         Filter: (((country)::text = 'US'::text) AND (date = '12-DEC-12
00:00:00'::timestamp without time zone))
(7 rows)
```


3.5 Partitioning Command Syntax

The following sections provide information about using the table partitioning syntax supported by Advanced Server.

3.5.1 CREATE TABLE…PARTITION BY

Use the `PARTITION BY` clause of the `CREATE TABLE` command to create a partitioned table with data distributed amongst one or more partitions (and subpartitions). The command syntax comes in the following forms:

List Partitioning Syntax

Use the first form to create a list-partitioned table:

```
CREATE TABLE [ schema. ]table_name
   table_definition
   PARTITION BY LIST(column)
   [SUBPARTITION BY {RANGE|LIST} (column[, column ]...)]
   (list_partition_definition[, list_partition_definition]...);
```

Where *list_partition_definition* is:

```
      PARTITION [partition_name]
        VALUES (value[, value]...)
        [TABLESPACE tablespace_name]
        [(subpartition, ...)]
```

Range Partitioning Syntax

Use the second form to create a range-partitioned table:

```
CREATE TABLE [ schema. ]table_name
   table_definition
   PARTITION BY RANGE(column[, column ]...)
   [SUBPARTITION BY {RANGE|LIST} (column[, column ]...)]
   (range_partition_definition[, range_partition_definition]...);
```

Where *range_partition_definition* is:

```
      PARTITION [partition_name]
        VALUES LESS THAN (value[, value]...)
        [TABLESPACE tablespace_name]
        [(subpartition, ...)]
```

Hash Partitioning Syntax

Use the third form to create a hash-partitioned table:

```
CREATE TABLE [ schema. ]table_name
   table_definition
   PARTITION BY HASH(column[, column ]...)
   [SUBPARTITION BY {RANGE|LIST|HASH} (column[, column ]...)]
   (hash_partition_definition[, hash_partition_definition]...);
```

Where `hash_partition_definition` is:

```
      [PARTITION partition_name]
        [TABLESPACE tablespace_name]
        [(subpartition, ...)]
```

Subpartitioning Syntax

`subpartition` may be one of the following:

```
{list_subpartition | range_subpartition | hash_subpartition}
```

where `list_subpartition` is:

```
      SUBPARTITION [subpartition_name]
        VALUES (value[, value]...)
        [TABLESPACE tablespace_name]
```

where `range_subpartition` is:

```
      SUBPARTITION [subpartition_name]
        VALUES LESS THAN (value[, value]...)
        [TABLESPACE tablespace_name]
```

where `hash_subpartition` is:

```
      [SUBPARTITION subpartition_name]
        [TABLESPACE tablespace_name]
```

Description

The `CREATE TABLE… PARTITION BY` command creates a table with one or more partitions; each partition may have one or more subpartitions. There is no upper limit to the number of defined partitions, but if you include the `PARTITION BY` clause, you must specify at least one partitioning rule. The resulting table will be owned by the user that creates it.

Use the `PARTITION BY LIST` clause to divide a table into partitions based on the values entered in a specified column. Each partitioning rule must specify at least one literal value, but there is no upper limit placed on the number of values you may specify. Include a rule that specifies a matching value of `DEFAULT` to direct any un-qualified rows to the given partition; for more information about using the `DEFAULT` keyword, see Section 3.6.

Use the `PARTITION BY RANGE` clause to specify boundary rules by which to create partitions. Each partitioning rule must contain at least one column of a data type that has two operators (i.e., a greater-than or equal to operator, and a less-than operator). Range boundaries are evaluated against a `LESS THAN` clause and are non-inclusive; a date boundary of January 1, 2013 will include only those date values that fall on or before December 31, 2012.

Range partition rules must be specified in ascending order. `INSERT` commands that store rows with values that exceed the top boundary of a range-partitioned table will fail unless the partitioning rules include a boundary rule that specifies a value of `MAXVALUE`. If you do not include a `MAXVALUE` partitioning rule, any row that exceeds the maximum limit specified by the boundary rules will result in an error.

For more information about using the `MAXVALUE` keyword, see Section 3.6.

Use the `TABLESPACE` keyword to specify the name of a tablespace on which a partition or subpartition will reside; if you do not specify a tablespace, the partition or subpartition will reside in the default tablespace.

If a table definition includes the `SUBPARTITION BY` clause, each partition within that table will have at least one subpartition. Each subpartition may be explicitly defined or system-defined.

If the subpartition is system-defined, the server-generated subpartition will reside in the default tablespace, and the name of the subpartition will be assigned by the server. The server will create:

- A `DEFAULT` subpartition if the `SUBPARTITION BY` clause specifies `LIST`.

- A `MAXVALUE` subpartition if the `SUBPARTITION BY` clause specifies `RANGE`.

The server will generate a subpartition name that is a combination of the partition table name and a unique identifier. You can query the `ALL_TAB_SUBPARTITIONS` table to review a complete list of subpartition names.

Parameters

`table_name`

The name (optionally schema-qualified) of the table to be created.

`table_definition`

The column names, data types, and constraint information as described in the PostgreSQL core documentation for the `CREATE TABLE` statement, available at the EnterpriseDB website at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-createtable.html

`partition_name`

The name of the partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

subpartition_name

The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

column

The name of a column on which the partitioning rules are based. Each row will be stored in a partition that corresponds to the *value* of the specified column(s).

(*value*[, *value*]...)

Use `value` to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. `value` may be `NULL`, `DEFAULT` (if specifying a `LIST` partition), or `MAXVALUE` (if specifying a `RANGE` partition).

When specifying rules for a list-partitioned table, include the `DEFAULT` keyword in the last partition rule to direct any un-matched rows to the given partition. If you do not include a rule that includes a value of `DEFAULT`, any `INSERT` statement that attempts to add a row that does not match the specified rules of at least one partition will fail, and return an error.
When specifying rules for a list-partitioned table, include the `MAXVALUE` keyword in the last partition rule to direct any un-categorized rows to the given partition. If you do not include a `MAXVALUE` partition, any `INSERT` statement that attempts to add a row where the partitioning key is greater than the highest value specified will fail, and return an error.
tablespace_name

The name of the tablespace in which the partition or subpartition resides.

3.5.1.1 Example - PARTITION BY LIST

The following example creates a partitioned table (`sales`) using the `PARTITION BY LIST` clause. The `sales` table stores information in three partitions (`europe`, `asia`, and `americas`):

```
CREATE TABLE sales
(
  dept_no     number,
```



```
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The resulting table is partitioned by the value specified in the `country` column:

```
acctg=# SELECT partition_name, high_value from ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 americas       | 'US', 'CANADA'
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(3 rows)
```

- Rows with a value of `US` or `CANADA` in the `country` column are stored in the `americas` partition.
- Rows with a value of `INDIA` or `PAKISTAN` in the `country` column are stored in the `asia` partition.
- Rows with a value of `FRANCE` or `ITALY` in the `country` column are stored in the `europe` partition.

The server would evaluate the following statement against the partitioning rules, and store the row in the `europe` partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012',
'650000');
```

3.5.1.2 Example - PARTITION BY RANGE

The following example creates a partitioned table (`sales`) using the `PARTITION BY RANGE` clause. The `sales` table stores information in four partitions (`q1_2012`, `q2_2012`, `q3_2012` and `q4_2012`) :

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
```

```
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The resulting table is partitioned by the value specified in the `date` column:

```
acctg=# SELECT partition_name, high_value from ALL_TAB_PARTITIONS;
 partition_name |  high_value
----------------+---------------
 q4_2012        | '2013-Jan-01'
 q3_2012        | '2012-Oct-01'
 q2_2012        | '2012-Jul-01'
 q1_2012        | '2012-Apr-01'
(4 rows)
```

- Any row with a value in the `date` column before April 1, 2012 is stored in a partition named `q1_2012`.
- Any row with a value in the `date` column before July 1, 2012 is stored in a partition named `q2_2012`.
- Any row with a value in the `date` column before October 1, 2012 is stored in a partition named `q3_2012`.
- Any row with a value in the `date` column before January 1, 2013 is stored in a partition named `q4_2012`.

The server would evaluate the following statement against the partitioning rules and store the row in the `q3_2012` partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012',
'650000');
```

3.5.1.3 Example - PARTITION BY HASH

The following example creates a partitioned table (`sales`) using the `PARTITION BY HASH` clause. The `sales` table stores information in three partitions (`p1`, `p2`, and `p3`:


```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY HASH (part_no)
(
  PARTITION p1,
  PARTITION p2,
  PARTITION p3
);
```

The table is partitioned by the hash value of the value specified in the `part_no` column:

```
acctg=# SELECT partition_name, partition_position from ALL_TAB_PARTITIONS;
 partition_name | partition_position
----------------+--------------------
 p3             |                  3
 p2             |                  2
 p1             |                  1
(3 rows)
```

The server will evaluate the hash value of the `part_no` column, and distribute the rows into approximately equal partitions.

3.5.1.4 Example - PARTITION BY RANGE, SUBPARTITION BY LIST

The following example creates a partitioned table (`sales`) that is first partitioned by the transaction date; the range partitions (`q1_2012`, `q2_2012`, `q3_2012` and `q4_2012`) are then list-subpartitioned using the value of the `country` column.

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST(country)
  (
    PARTITION q1_2012
      VALUES LESS THAN('2012-Apr-01')
      (
        SUBPARTITION q1_europe VALUES ('FRANCE', 'ITALY'),
```

```
        SUBPARTITION q1_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q1_americas VALUES ('US', 'CANADA')
       ),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01')
      (
        SUBPARTITION q2_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q2_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q2_americas VALUES ('US', 'CANADA')
       ),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01')
      (
        SUBPARTITION q3_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q3_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q3_americas VALUES ('US', 'CANADA')
       ),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
      (
        SUBPARTITION q4_europe VALUES ('FRANCE', 'ITALY'),
        SUBPARTITION q4_asia VALUES ('INDIA', 'PAKISTAN'),
        SUBPARTITION q4_americas VALUES ('US', 'CANADA')
       )
);
```

This statement creates a table with four partitions; each partition has three subpartitions:

```
acctg=# SELECT subpartition_name, high_value, partition_name FROM
ALL_TAB_SUBPARTITIONS;
 subpartition_name |     high_value      | partition_name
-------------------+---------------------+----------------
 q4_asia           | 'INDIA', 'PAKISTAN' | q4_2012
 q4_europe         | 'FRANCE', 'ITALY'   | q4_2012
 q4_americas       | 'US', 'CANADA'      | q4_2012
 q3_americas       | 'US', 'CANADA'      | q3_2012
 q3_asia           | 'INDIA', 'PAKISTAN' | q3_2012
 q3_europe         | 'FRANCE', 'ITALY'   | q3_2012
 q2_americas       | 'US', 'CANADA'      | q2_2012
 q2_asia           | 'INDIA', 'PAKISTAN' | q2_2012
 q2_europe         | 'FRANCE', 'ITALY'   | q2_2012
 q1_americas       | 'US', 'CANADA'      | q1_2012
 q1_asia           | 'INDIA', 'PAKISTAN' | q1_2012
 q1_europe         | 'FRANCE', 'ITALY'   | q1_2012
(12 rows)
```

When a row is added to this table, the value in the `date` column is compared to the values specified in the range partitioning rules, and the server selects the partition in which the row should reside. The value in the `country` column is then compared to the values specified in the list subpartitioning rules; when the server locates a match for the value, the row is stored in the corresponding subpartition.

Any row added to the table will be stored in a subpartition, so the partitions will contain no data.

The server would evaluate the following statement against the partitioning and subpartitioning rulesand store the row in the `q3_europe` partition:

```
INSERT INTO sales VALUES (10, '9519a', 'FRANCE', '18-Aug-2012',
'650000');
```

3.5.2 ALTER TABLE...ADD PARTITION

Use the `ALTER TABLE… ADD PARTITION` command to add a partition to an existing partitioned table. The syntax is:

```
ALTER TABLE table_name ADD PARTITION partition_definition;
```

Where *partition_definition* is:

```
{list_partition | range_partition }
```

and *list_partition* is:

```
PARTITION [partition_name]
  VALUES (value[, value]...)
  [TABLESPACE tablespace_name]
  [(subpartition, ...)]
```

and *range_partition* is:

```
PARTITION [partition_name]
  VALUES LESS THAN (value[, value]...)
  [TABLESPACE tablespace_name]
  [(subpartition, ...)]
```

Where *subpartition* is:

```
{list_subpartition | range_subpartition | hash_subpartition}
```

and *list_subpartition* is:

```
SUBPARTITION [subpartition_name]
  VALUES (value[, value]...)
  [TABLESPACE tablespace_name]
```

and *range_subpartition* is:

```
SUBPARTITION [subpartition_name]
  VALUES LESS THAN (value[, value]...)
  [TABLESPACE tablespace_name]
```

Description

The `ALTER TABLE… ADD PARTITION` command adds a partition to an existing partitioned table. There is no upper limit to the number of defined partitions in a partitioned table.

New partitions must be of the same type (`LIST`, `RANGE` or `HASH`) as existing partitions. The new partition rules must reference the same column specified in the partitioning rules that define the existing partition(s).

You cannot use the `ALTER TABLE… ADD PARTITION` statement to add a partition to a table with a `MAXVALUE` or `DEFAULT` rule. Note that you can alternatively use the `ALTER TABLE… SPLIT PARTITION` statement to split an existing partition, effectively increasing the number of partitions in a table.

`RANGE` partitions must be specified in ascending order. You cannot add a new partition that precedes existing partitions in a `RANGE` partitioned table.

Include the `TABLESPACE` clause to specify the tablespace in which the new partition will reside. If you do not specify a tablespace, the partition will reside in the default tablespace.

If the table is indexed, the index will be created on the new partition.

To use the `ALTER TABLE... ADD PARTITION` command you must be the table owner, or have superuser (or administrative) privileges.

Parameters

`table_name`

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

`partition_name`

> The name of the partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`subpartition_name`

> The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`(value[, value]...)`

Use *value* to specify a quoted literal value (or comma-delimited list of literal values) by which rows will be distributed into partitions. Each partitioning rule must specify at least one *value*, but there is no limit placed on the number of values specified within a rule. *value* may also be NULL, DEFAULT (if specifying a LIST partition), or MAXVALUE (if specifying a RANGE partition).

For information about creating a DEFAULT or MAXVALUE partition, see Section 3.6.

tablespace_name

The name of the tablespace in which a partition or subpartition resides.

3.5.2.1 Example - Adding a Partition to a LIST Partitioned Table

The example that follows adds a partition to the list-partitioned sales table. The table was created using the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The table contains three partitions (americas, asia, and europe) :

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 americas       | 'US', 'CANADA'
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(3 rows)
```

The following command adds a partition named east_asia to the sales table:

```
ALTER TABLE sales ADD PARTITION east_asia
  VALUES ('CHINA', 'KOREA');
```

After invoking the command, the table includes the `east_asia` partition:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 east_asia      | 'CHINA', 'KOREA'
 americas       | 'US', 'CANADA'
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(4 rows)
```

3.5.2.2 Example - Adding a Partition to a RANGE Partitioned Table

The example that follows adds a partition to a range-partitioned table named `sales`:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The table contains four partitions (`q1_2012`, `q2_2012, q3_2012`, and `q4_2012`):

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |  high_value
----------------+---------------
 q4_2012        | '2013-Jan-01'
 q3_2012        | '2012-Oct-01'
 q2_2012        | '2012-Jul-01'
 q1_2012        | '2012-Apr-01'
(4 rows)
```

The following command adds a partition named `q1_2013` to the `sales` table:

```
ALTER TABLE sales ADD PARTITION q1_2013
  VALUES LESS THAN('01-APR-2013');
```

After invoking the command, the table includes the `q1_2013` partition:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |  high_value
----------------+---------------
 q1_2012        | '2012-Apr-01'
 q2_2012        | '2012-Jul-01'
 q3_2012        | '2012-Oct-01'
 q4_2012        | '2013-Jan-01'
 q1_2013        | '01-APR-2013'
(5 rows)
```

3.5.3 ALTER TABLE… ADD SUBPARTITION

The `ALTER TABLE… ADD SUBPARTITION` command adds a subpartition to an existing subpartitioned partition. The syntax is:

```
ALTER TABLE table_name MODIFY PARTITION partition_name
      ADD SUBPARTITION subpartition_definition;
```

Where *subpartition_definition* is:

```
{list_subpartition | range_subpartition}
```

and *list_subpartition* is:

```
SUBPARTITION [subpartition_name]
  VALUES (value[, value]...)
  [TABLESPACE tablespace_name]
```

and *range_subpartition* is:

```
SUBPARTITION [subpartition_name]
  VALUES LESS THAN (value[, value]...)
  [TABLESPACE tablespace_name]
```

Description

The `ALTER TABLE… ADD SUBPARTITION` command adds a subpartition to an existing partition; the partition must already be subpartitioned. There is no upper limit to the number of defined subpartitions.

New subpartitions must be of the same type (`LIST`, `RANGE` or `HASH`) as existing subpartitions. The new subpartition rules must reference the same column specified in the subpartitioning rules that define the existing subpartition(s).

You cannot use the `ALTER TABLE… ADD SUBPARTITION` statement to add a subpartition to a table with a `MAXVALUE` or `DEFAULT` rule , but you can split an existing subpartition with the `ALTER TABLE… SPLIT SUBPARTITION` statement, effectively adding a subpartition to a table.

You cannot add a new subpartition that precedes existing subpartitions in a range subpartitioned table; range subpartitions must be specified in ascending order.

Include the `TABLESPACE` clause to specify the tablespace in which the subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be created on the new subpartition.

To use the `ALTER TABLE... ADD SUBPARTITION` command you must be the table owner, or have superuser (or administrative) privileges.

Parameters

`table_name`

The name (optionally schema-qualified) of the partitioned table in which the subpartition will reside.

`partition_name`

The name of the partition in which the new subpartition will reside.

`subpartition_name`

The name of the subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`(`*`value`*`[, `*`value`*`]...)`

Use `value` to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. `value` may also be `NULL`, `DEFAULT` (if specifying a `LIST` partition), or `MAXVALUE` (if specifying a `RANGE` partition).

For information about creating a `DEFAULT` or `MAXVALUE` partition, see Section 3.6.

`tablespace_name`

The name of the tablespace in which the subpartition resides.

3.5.3.1 Example - Adding a Subpartition to a LIST-RANGE Partitioned Table

The following example adds a `RANGE` subpartition to the list-partitioned `sales` table. The `sales` table was created with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
  SUBPARTITION BY RANGE(date)
(
  PARTITION europe VALUES('FRANCE', 'ITALY')
    (
      SUBPARTITION europe_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION europe_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION asia VALUES('INDIA', 'PAKISTAN')
    (
      SUBPARTITION asia_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION asia_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION americas VALUES('US', 'CANADA')
    (
      SUBPARTITION americas_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION americas_2012
        VALUES LESS THAN('2013-Jan-01')
    )
);
```

The `sales` table has three partitions, named `europe`, `asia`, and `americas`. Each partition has two range-defined subpartitions:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2011       | '2012-Jan-01'
 europe         | europe_2012       | '2013-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
 asia           | asia_2012         | '2013-Jan-01'
```

```
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_2012     | '2013-Jan-01'
(6 rows)
```

The following command adds a subpartition named `europe_2013`:

```
ALTER TABLE sales MODIFY PARTITION europe
  ADD SUBPARTITION europe_2013
  VALUES LESS THAN('2014-Jan-01');
```

After invoking the command, the table includes a subpartition named `europe_2013`:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2011       | '2012-Jan-01'
 europe         | europe_2012       | '2013-Jan-01'
 europe         | europe_2013       | '2014-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
 asia           | asia_2012         | '2013-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_2012     | '2013-Jan-01'
(7 rows)
```

Note that when adding a new range subpartition, the subpartitioning rules must specify a range that falls *after* any existing subpartitions.

3.5.3.2 Example - Adding a Subpartition to a RANGE-LIST Partitioned Table

The following example adds a `LIST` subpartition to the `RANGE` partitioned `sales` table. The `sales` table was created with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST (country)
  (
    PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012')
    (
      SUBPARTITION europe VALUES ('ITALY', 'FRANCE'),
      SUBPARTITION americas VALUES ('US', 'CANADA')
```

```
    ),
    PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
    (
      SUBPARTITION asia VALUES ('INDIA', 'PAKISTAN')
    )
  );
```

After executing the above command, the sales table will have two partitions, named first_half_2012 and second_half_2012. The first_half_2012 partition has two subpartitions, named europe and americas, and the second_half_2012 partition has one partition, named asia:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |     high_value
------------------+-------------------+---------------------
 first_half_2012  | europe            | 'ITALY', 'FRANCE'
 first_half_2012  | americas          | 'US', 'CANADA'
 second_half_2012 | asia              | 'INDIA', 'PAKISTAN'
(3 rows)
```

The following command adds a subpartition to the second_half_2012 partition, named east_asia:

```
ALTER TABLE sales MODIFY PARTITION second_half_2012
  ADD SUBPARTITION east_asia VALUES ('CHINA');
```

After invoking the command, the table includes a subpartition named east_asia:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |     high_value
------------------+-------------------+---------------------
 first_half_2012  | europe            | 'ITALY', 'FRANCE'
 first_half_2012  | americas          | 'US', 'CANADA'
 second_half_2012 | asia              | 'INDIA', 'PAKISTAN'
 second_half_2012 | east_asia         | 'CHINA'
(4 rows)
```

3.5.4 ALTER TABLE...SPLIT PARTITION

Use the `ALTER TABLE… SPLIT PARTITION` command to divide a single partition into two partitions, and redistribute the partition's contents between the new partitions. The command syntax comes in two forms.

The first form splits a `RANGE` partition into two partitions:

```
ALTER TABLE table_name SPLIT PARTITION partition_name
  AT (range_part_value)
  INTO
  (
    PARTITION new_part1
      [TABLESPACE tablespace_name],
    PARTITION new_part2
      [TABLESPACE tablespace_name]
  );
```

The second form splits a `LIST` partition into two partitions:

```
ALTER TABLE table_name SPLIT PARTITION partition_name
  VALUES (value[, value]...)
  INTO
  (
    PARTITION new_part1
      [TABLESPACE tablespace_name],
    PARTITION new_part2
      [TABLESPACE tablespace_name]
  );
```

Description

The `ALTER TABLE...SPLIT PARTITION` command adds a partition to an existing `LIST` or `RANGE` partitioned table. Please note that the `ALTER TABLE… SPLIT PARTITION` command cannot add a partition to a `HASH` partitioned table. There is no upper limit to the number of partitions that a table may have.

When you execute an `ALTER TABLE...SPLIT PARTITION` command, Advanced Server creates two new partitions, and redistributes the content of the old partition between them (as constrained by the partitioning rules).

Include the `TABLESPACE` clause to specify the tablespace in which a partition will reside. If you do not specify a tablespace, the partition will reside in the default tablespace.

If the table is indexed, the index will be created on the new partition.

To use the `ALTER TABLE... SPLIT PARTITION` command you must be the table owner, or have superuser (or administrative) privileges.

Parameters

`table_name`

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

`partition_name`

The name of the partition that is being split.

`new_part1`

The name of the first new partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_part1` will receive the rows that meet the subpartitioning constraints specified in the `ALTER TABLE… SPLIT SUBPARTITION` command.

`new_part2`

The name of the second new partition to be created. Partition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_part2` will receive the rows are not directed to *`new_part1`* by the partitioning constraints specified in the `ALTER TABLE… SPLIT PARTITION` command.

`range_part_value`

Use *`range_part_value`* to specify the boundary rules by which to create the new partition. The partitioning rule must contain at least one column of a data type that has two operators (i.e., a greater-than-or-equal to operator, and a less-than operator). Range boundaries are evaluated against a `LESS THAN` clause and are non-inclusive; a date boundary of January 1, 2010 will include only those date values that fall on or before December 31, 2009.

`(value[, value]...)`

Use *`value`* to specify a quoted literal value (or comma-delimited list of literal values) by which rows will be distributed into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule.

For information about creating a `DEFAULT` or `MAXVALUE` partition, see Section 3.6.

`tablespace_name`

The name of the tablespace in which the partition or subpartition resides.

3.5.4.1 Example - Splitting a LIST Partition

Our example will divide one of the partitions in the list-partitioned `sales` table into two new partitions, and redistribute the contents of the partition between them. The `sales` table is created with the statement:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

The table definition creates three partitions (`europe`, `asia`, and `americas`). The following command adds rows to each partition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
```



```
(40, '4788a', 'US', '23-Sept-2012', '4950'),
(40, '4788b', 'US', '09-Oct-2012', '15000'),
(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
(20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

The rows are distributed amongst the partitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date        | amount
----------------+---------+---------+----------+--------------------+-------
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
(17 rows)
```

The following command splits the `americas` partition into two partitions named `us` and `canada`:

```
ALTER TABLE sales SPLIT PARTITION americas
  VALUES ('US')
  INTO (PARTITION us, PARTITION canada);
```

A `SELECT` statement confirms that the rows have been redistributed across the correct partitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid   | dept_no | part_no | country  |        date        | amount
--------------+---------+---------+----------+--------------------+--------
 sales_europe |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia   |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia   |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia   |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia   |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia   |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_us     |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_us     |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_us     |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_us     |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_us     |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
 sales_canada |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
```



```
 sales_canada |       30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_canada |       30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
(17 rows)
```

3.5.4.2 Example - Splitting a RANGE Partition

This example divides the `q4_2012` partition (of the range-partitioned `sales` table) into two partitions, and redistribute the partition's contents. Use the following command to create the `sales` table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012
    VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012
    VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012
    VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012
    VALUES LESS THAN('2013-Jan-01')
);
```

The table definition creates four partitions (`q1_2012`, `q2_2012`, `q3_2012`, and `q4_2012`). The following command adds rows to each partition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
```

```
(40, '4788b', 'US', '09-Oct-2012', '15000'),
(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
(20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

A `SELECT` statement confirms that the rows are distributed amongst the partitions as expected:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date         | amount
---------------+---------+---------+----------+---------------------+--------
 sales_q1_2012 |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_q1_2012 |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_q1_2012 |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
 sales_q2_2012 |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_q2_2012 |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_q2_2012 |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_q2_2012 |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_q3_2012 |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_q3_2012 |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_q3_2012 |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_q3_2012 |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_q4_2012 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_q4_2012 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_q4_2012 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |   15000
 sales_q4_2012 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_q4_2012 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(17 rows)
```

The following command splits the `q4_2012` partition into two partitions named `q4_2012_p1` and `q4_2012_p2`:

```
ALTER TABLE sales SPLIT PARTITION q4_2012
  AT ('15-Nov-2012')
  INTO
  (
    PARTITION q4_2012_p1,
    PARTITION q4_2012_p2
  );
```

A `SELECT` statement confirms that the rows have been redistributed across the new partitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid      | dept_no | part_no | country  |        date         |amount
-------------------+---------+---------+----------+---------------------+-------
 sales_q1_2012     |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 | 45000
 sales_q1_2012     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 | 75000
 sales_q1_2012     |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 | 75000
 sales_q2_2012     |      40 | 9519b   | US       | 12-APR-12 00:00:00 |145000
 sales_q2_2012     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 | 37500
 sales_q2_2012     |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |120000
 sales_q2_2012     |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |  4950
 sales_q3_2012     |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 | 15000
 sales_q3_2012     |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |650000
 sales_q3_2012     |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |650000
```

```
 sales_q3_2012    |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |  5090
 sales_q3_2012    |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |  4950
 sales_q4_2012_p1 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 | 25000
 sales_q4_2012_p1 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 | 15000
 sales_q4_2012_p1 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |650000
 sales_q4_2012_p2 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 | 50000
 sales_q4_2012_p2 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |  5090
(17 rows)
```

3.5.5 ALTER TABLE...SPLIT SUBPARTITION

Use the `ALTER TABLE… SPLIT SUBPARTITION` command to divide a single subpartition into two subpartitions, and redistribute the subpartition's contents. The command comes in two variations.

The first variation splits a range subpartition into two subpartitions:

```
ALTER TABLE table_name SPLIT SUBPARTITION subpartition_name
  AT (range_part_value)
  INTO
  (
    SUBPARTITION new_subpart1
      [TABLESPACE tablespace_name],
    SUBPARTITION new_subpart2
      [TABLESPACE tablespace_name]
  );
```

The second variation splits a list subpartition into two subpartitions:

```
ALTER TABLE table_name SPLIT SUBPARTITION subpartition_name
  VALUES (value[, value]...)
  INTO
  (
    SUBPARTITION new_subpart1
      [TABLESPACE tablespace_name],
    SUBPARTITION new_subpart2
      [TABLESPACE tablespace_name]
  );
```

Description

The `ALTER TABLE...SPLIT SUBPARTITION` command adds a subpartition to an existing subpartitioned table. There is no upper limit to the number of defined subpartitions. When you execute an `ALTER TABLE...SPLIT SUBPARTITION` command, Advanced Server creates two new subpartitions, moving any rows that contain values that are constrained by the specified subpartition rules into *`new_subpart1`*, and any remaining rows into *`new_subpart2`*.

The new subpartition rules must reference the column specified in the rules that define the existing subpartition(s).

Include the `TABLESPACE` clause to specify a tablespace in which a new subpartition will reside. If you do not specify a tablespace, the subpartition will be created in the default tablespace.

If the table is indexed, the index will be created on the new subpartition.

To use the `ALTER TABLE... SPLIT SUBPARTITION` command you must be the table owner, or have superuser (or administrative) privileges.

Parameters

`table_name`

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

`subpartition_name`

The name of the subpartition that is being split.

`new_subpart1`

The name of the first new subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_subpart1` will receive the rows that meet the subpartitioning constraints specified in the `ALTER TABLE… SPLIT SUBPARTITION` command.

`new_subpart2`

The name of the second new subpartition to be created. Subpartition names must be unique amongst all partitions and subpartitions, and must follow the naming conventions for object identifiers.

`new_subpart2` will receive the rows are not directed to *`new_subpart1`* by the subpartitioning constraints specified in the `ALTER TABLE… SPLIT SUBPARTITION` command.

`(`*`value`*`[, `*`value`*`]...)`

Use *`value`* to specify a quoted literal value (or comma-delimited list of literal values) by which table entries will be grouped into partitions. Each partitioning rule must specify at least one value, but there is no limit placed on the number of values specified within a rule. *`value`* may also be `NULL`, `DEFAULT` (if specifying a `LIST` subpartition), or `MAXVALUE` (if specifying a `RANGE` subpartition).

For information about creating a DEFAULT or MAXVALUE partition, see Section 3.6.

tablespace_name

The name of the tablespace in which the partition or subpartition resides.

3.5.5.1 Example - Splitting a LIST Subpartition

The following example splits a list subpartition, redistributing the subpartition's contents between two new subpartitions. The sample table (sales) was created with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST (country)
  (
    PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012')
    (
      SUBPARTITION p1_europe VALUES ('ITALY', 'FRANCE'),
      SUBPARTITION p1_americas VALUES ('US', 'CANADA')
    ),
    PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
    (
      SUBPARTITION p2_europe VALUES ('ITALY', 'FRANCE'),
      SUBPARTITION p2_americas VALUES ('US', 'CANADA')
    )
  );
```

The sales table has two partitions, named first_half_2012, and second_half_2012. Each partition has two range-defined subpartitions that distribute the partition's contents into subpartitions based on the value of the country column:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |    high_value
------------------+-------------------+-------------------
 second_half_2012 | p2_europe         | 'ITALY', 'FRANCE'
 first_half_2012  | p1_europe         | 'ITALY', 'FRANCE'
 second_half_2012 | p2_americas       | 'US', 'CANADA'
```

```
 first_half_2012  | p1_americas       | 'US', 'CANADA'
(4 rows)
```

The following command adds rows to each subpartition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000');
```

A `SELECT` statement confirms that the rows are correctly distributed amongst the subpartitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid      | dept_no | part_no | country|        date        |amount
-------------------+---------+---------+--------+--------------------+------
 sales_p1_europe   |      10 | 4519b   | FRANCE | 17-JAN-12 00:00:00 |  45000
 sales_p1_europe   |      10 | 4519b   | FRANCE | 17-JAN-12 00:00:00 |  45000
 sales_p1_americas |      40 | 9519b   | US     | 12-APR-12 00:00:00 | 145000
 sales_p1_americas |      30 | 9519b   | CANADA | 01-FEB-12 00:00:00 |  75000
 sales_p1_americas |      30 | 4519b   | CANADA | 08-APR-12 00:00:00 | 120000
 sales_p1_americas |      40 | 3788a   | US     | 12-MAY-12 00:00:00 |   4950
 sales_p2_europe   |      10 | 9519b   | ITALY  | 07-JUL-12 00:00:00 |  15000
 sales_p2_europe   |      10 | 9519a   | FRANCE | 18-AUG-12 00:00:00 | 650000
 sales_p2_europe   |      10 | 9519b   | FRANCE | 18-AUG-12 00:00:00 | 650000
 sales_p2_americas |      40 | 4577b   | US     | 11-NOV-12 00:00:00 |  25000
 sales_p2_americas |      30 | 7588b   | CANADA | 14-DEC-12 00:00:00 |  50000
 sales_p2_americas |      40 | 4788a   | US     | 23-SEP-12 00:00:00 |   4950
 sales_p2_americas |      40 | 4788b   | US     | 09-OCT-12 00:00:00 |  15000
(13 rows)
```

The following command splits the `p2_americas` subpartition into two new subpartitions, and redistributes the contents:

```
ALTER TABLE sales SPLIT SUBPARTITION p2_americas
  VALUES ('US')
  INTO
  (
    SUBPARTITION p2_us,
    SUBPARTITION p2_canada
  );
```

After invoking the command, the `p2_americas` subpartition has been deleted; in it's place, the server has created two new subpartitions (`p2_us` and `p2_canada`):

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
  partition_name  | subpartition_name |    high_value
------------------+-------------------+-------------------
 first_half_2012  | p1_europe         | 'ITALY', 'FRANCE'
 first_half_2012  | p1_americas       | 'US', 'CANADA'
 second_half_2012 | p2_europe         | 'ITALY', 'FRANCE'
 second_half_2012 | p2_canada         | 'CANADA'
 second_half_2012 | p2_us             | 'US'
(5 rows)
```

Querying the `sales` table demonstrates that the content of the `p2_americas` subpartition has been redistributed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
     tableoid      | dept_no | part_no | country |        date        |amount
-------------------+---------+---------+---------+--------------------+------
 sales_p1_europe   |      10 | 4519b   | FRANCE  | 17-JAN-12 00:00:00 | 45000
 sales_p1_europe   |      10 | 4519b   | FRANCE  | 17-JAN-12 00:00:00 | 45000
 sales_p1_americas |      40 | 9519b   | US      | 12-APR-12 00:00:00 |145000
 sales_p1_americas |      30 | 9519b   | CANADA  | 01-FEB-12 00:00:00 | 75000
 sales_p1_americas |      30 | 4519b   | CANADA  | 08-APR-12 00:00:00 |120000
 sales_p1_americas |      40 | 3788a   | US      | 12-MAY-12 00:00:00 |  4950
 sales_p2_europe   |      10 | 9519b   | ITALY   | 07-JUL-12 00:00:00 | 15000
 sales_p2_europe   |      10 | 9519a   | FRANCE  | 18-AUG-12 00:00:00 |650000
 sales_p2_europe   |      10 | 9519b   | FRANCE  | 18-AUG-12 00:00:00 |650000
 sales_p2_us       |      40 | 4577b   | US      | 11-NOV-12 00:00:00 | 25000
 sales_p2_us       |      40 | 4788a   | US      | 23-SEP-12 00:00:00 |  4950
 sales_p2_us       |      40 | 4788b   | US      | 09-OCT-12 00:00:00 | 15000
 sales_p2_canada   |      30 | 7588b   | CANADA  | 14-DEC-12 00:00:00 | 50000
(13 rows)
```

3.5.5.2 Example - Splitting a RANGE Subpartition

The following example splits a range subpartition, redistributing the subpartition's contents between two new subpartitions. The sample table (`sales`) was created with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
  SUBPARTITION BY RANGE(date)
(
  PARTITION europe VALUES('FRANCE', 'ITALY')
    (
      SUBPARTITION europe_2011
        VALUES LESS THAN('2012-Jan-01'),
```

```
      SUBPARTITION europe_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION asia VALUES('INDIA', 'PAKISTAN')
    (
      SUBPARTITION asia_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION asia_2012
        VALUES LESS THAN('2013-Jan-01')
    ),
  PARTITION americas VALUES('US', 'CANADA')
    (
      SUBPARTITION americas_2011
        VALUES LESS THAN('2012-Jan-01'),
      SUBPARTITION americas_2012
        VALUES LESS THAN('2013-Jan-01')
    )
);
```

The `sales` table has three partitions (`europe`, `asia`, and `americas`). Each partition has two range-defined subpartitions that sort the partitions contents into subpartitions by the value of the `date` column:

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2011       | '2012-Jan-01'
 europe         | europe_2012       | '2013-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
 asia           | asia_2012         | '2013-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_2012     | '2013-Jan-01'
(6 rows)
```

The following command adds rows to each subpartition:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
```

```
(40, '4788b', 'US', '09-Oct-2012', '15000'),
(20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
(20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

A `SELECT` statement confirms that the rows are distributed amongst the subpartions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
      tableoid       | dept_no|part_no| country |        date         |amount
---------------------+--------+-------+---------+---------------------+---
 sales_europe_2012   |      10| 4519b | FRANCE  | 17-JAN-12 00:00:00 | 45000
 sales_europe_2012   |      10| 9519b | ITALY   | 07-JUL-12 00:00:00 | 15000
 sales_europe_2012   |      10| 9519a | FRANCE  | 18-AUG-12 00:00:00 | 650000
 sales_europe_2012   |      10| 9519b | FRANCE  | 18-AUG-12 00:00:00 | 650000
 sales_asia_2012     |      20| 3788a | INDIA   | 01-MAR-12 00:00:00 | 75000
 sales_asia_2012     |      20| 3788a | PAKISTAN| 04-JUN-12 00:00:00 | 37500
 sales_asia_2012     |      20| 3788b | INDIA   | 21-SEP-12 00:00:00 | 5090
 sales_asia_2012     |      20| 4519a | INDIA   | 18-OCT-12 00:00:00 | 650000
 sales_asia_2012     |      20| 4519b | INDIA   | 02-DEC-12 00:00:00 | 5090
 sales_americas_2012 |      40| 9519b | US      | 12-APR-12 00:00:00 | 145000
 sales_americas_2012 |      40| 4577b | US      | 11-NOV-12 00:00:00 | 25000
 sales_americas_2012 |      30| 7588b | CANADA  | 14-DEC-12 00:00:00 | 50000
 sales_americas_2012 |      30| 9519b | CANADA  | 01-FEB-12 00:00:00 | 75000
 sales_americas_2012 |      30| 4519b | CANADA  | 08-APR-12 00:00:00 | 120000
 sales_americas_2012 |      40| 3788a | US      | 12-MAY-12 00:00:00 | 4950
 sales_americas_2012 |      40| 4788a | US      | 23-SEP-12 00:00:00 | 4950
 sales_americas_2012 |      40| 4788b | US      | 09-OCT-12 00:00:00 | 15000
(17 rows)
```

The following command splits the `americas_2012` subpartition into two new subpartitions, and redistributes the contents:

```
ALTER TABLE sales
  SPLIT SUBPARTITION americas_2012
  AT('2012-Jun-01')
  INTO
   (
     SUBPARTITION americas_p1_2012,
     SUBPARTITION americas_p2_2012
   );
```

After invoking the command, the `americas_2012` subpartition has been deleted; in it's place, the server has created two new subpartitions (`americas_p1_2012` and `americas_p2_2012`):

```
acctg=# SELECT partition_name, subpartition_name, high_value FROM
ALL_TAB_SUBPARTITIONS;
 partition_name | subpartition_name |  high_value
----------------+-------------------+---------------
 europe         | europe_2012       | '2013-Jan-01'
 europe         | europe_2011       | '2012-Jan-01'
 americas       | americas_2011     | '2012-Jan-01'
 americas       | americas_p2_2012  | '2013-Jan-01'
 americas       | americas_p1_2012  | '2012-Jun-01'
 asia           | asia_2012         | '2013-Jan-01'
 asia           | asia_2011         | '2012-Jan-01'
```



```
(7 rows)
```

Querying the `sales` table demonstrates that the subpartition's contents are redistributed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
      tableoid         | dept_no|part_no|country |      date         |amount
-----------------------+--------+-------+--------+-------------------+-------
sales_europe_2012      |      10| 4519b |FRANCE  | 17-JAN-12 00:00:00|  45000
 sales_europe_2012     |      10| 9519b |ITALY   | 07-JUL-12 00:00:00|  15000
 sales_europe_2012     |      10| 9519a |FRANCE  | 18-AUG-12 00:00:00| 650000
 sales_europe_2012     |      10| 9519b |FRANCE  | 18-AUG-12 00:00:00| 650000
 sales_asia_2012       |      20| 3788a |INDIA   | 01-MAR-12 00:00:00|  75000
 sales_asia_2012       |      20| 3788a |PAKISTAN| 04-JUN-12 00:00:00|  37500
 sales_asia_2012       |      20| 3788b |INDIA   | 21-SEP-12 00:00:00|   5090
 sales_asia_2012       |      20| 4519a |INDIA   | 18-OCT-12 00:00:00| 650000
 sales_asia_2012       |      20| 4519b |INDIA   | 02-DEC-12 00:00:00|   5090
 sales_americas_p1_2012|      40| 9519b |US      | 12-APR-12 00:00:00| 145000
 sales_americas_p1_2012|      30| 9519b |CANADA  | 01-FEB-12 00:00:00|  75000
 sales_americas_p1_2012|      30| 4519b |CANADA  | 08-APR-12 00:00:00| 120000
 sales_americas_p1_2012|      40| 3788a |US      | 12-MAY-12 00:00:00|   4950
 sales_americas_p2_2012|      40| 4577b |US      | 11-NOV-12 00:00:00|  25000
 sales_americas_p2_2012|      30| 7588b |CANADA  | 14-DEC-12 00:00:00|  50000
 sales_americas_p2_2012|      40| 4788a |US      | 23-SEP-12 00:00:00|   4950
 sales_americas_p2_2012|      40| 4788b |US      | 09-OCT-12 00:00:00|  15000
(17 rows)
```


3.5.6 ALTER TABLE… EXCHANGE PARTITION

The `ALTER TABLE…EXCHANGE PARTITION` command swaps an existing table with a partition or subpartition. If you plan to add a large quantity of data to a partitioned table, you can use the `ALTER TABLE… EXCHANGE PARTITION` command to implement a bulk load. You can also use the `ALTER TABLE… EXCHANGE PARTITION` command to remove old or unneeded data for storage.

The command syntax is available in two forms. The first form swaps a table for a partition:

```
ALTER TABLE target_table
  EXCHANGE PARTITION target_partition
  WITH TABLE source_table
  [(WITH | WITHOUT) VALIDATION];
```

The second form swaps a table for a subpartition:

```
ALTER TABLE target_table
  EXCHANGE SUBPARTITION target_subpartition
  WITH TABLE source_table
  [(WITH | WITHOUT) VALIDATION];
```

This command makes no distinction between a partition and a subpartition:

- You can exchange a partition with the `EXCHANGE PARTITION` or `EXCHANGE SUBPARTITION` clause.

- You can exchange a subpartition with `EXCHANGE PARTITION` or `EXCHANGE SUBPARTITION` clause.

Description

When the `ALTER TABLE… EXCHANGE PARTITION` command completes, the data originally located in the *`target_partition`* will be located in the *`source_table`*, and the data originally located in the *`source_table`* will be located in the *`target_partition`*.

The `ALTER TABLE… EXCHANGE PARTITION` command can exchange partitions in a `LIST`, `RANGE` or `HASH` partitioned table. The structure of the *`source_table`* must match the structure of the *`target_table`* (both tables must have matching columns and data types), and the data contained within the table must adhere to the partitioning constraints.

Advanced Server accepts the `WITHOUT VALIDATION` clause, but ignores it; the new table is always validated.

You must own a table to invoke `ALTER TABLE… EXCHANGE PARTITION` or `ALTER TABLE… EXCHANGE SUBPARTITION` against that table.

Parameters

target_table

The name (optionally schema-qualified) of the table in which the partition resides.

target_partition

The name of the partition or subpartition to be replaced.

source_table

The name of the table that will replace the *target_partition*.

3.5.6.1 Example - Exchanging a Table for a Partition

The example that follows demonstrates swapping a table for a partition (`americas`) of the `sales` table. You can create the `sales` table with the following command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Use the following command to add sample data to the `sales` table:

```
INSERT INTO sales VALUES
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
```

```
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that only one row resides in the `americas` partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no| part_no | country |      date         | amount
---------------+--------+---------+---------+-------------------+-----------
 sales_europe  |      10| 4519b   | FRANCE  | 17-JAN-12 00:00:00|      45000
 sales_europe  |      10| 9519b   | ITALY   | 07-JUL-12 00:00:00|      15000
 sales_europe  |      10| 9519a   | FRANCE  | 18-AUG-12 00:00:00|     650000
 sales_europe  |      10| 9519b   | FRANCE  | 18-AUG-12 00:00:00|     650000
 sales_asia    |      20| 3788a   | INDIA   | 01-MAR-12 00:00:00|      75000
 sales_asia    |      20| 3788a   | PAKISTAN| 04-JUN-12 00:00:00|      37500
 sales_asia    |      20| 3788b   | INDIA   | 21-SEP-12 00:00:00|       5090
 sales_asia    |      20| 4519a   | INDIA   | 18-OCT-12 00:00:00|     650000
 sales_asia    |      20| 4519b   | INDIA   | 02-DEC-12 00:00:00|       5090
 sales_americas|      40| 9519b   | US      | 12-APR-12 00:00:00|     145000
(10 rows)
```

The following command creates a table (`n_america`) that matches the definition of the `sales` table:

```
CREATE TABLE n_america
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
);
```

The following command adds data to the `n_america` table. The data conforms to the partitioning rules of the `americas` partition:

```
INSERT INTO n_america VALUES
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000');
```

The following command swaps the table into the partitioned table:

```
ALTER TABLE sales
  EXCHANGE PARTITION americas
  WITH TABLE n_america;
```

Querying the `sales` table shows that the contents of the `n_america` table has been exchanged for the content of the `americas` partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no| part_no | country |        date         | amount
----------------+--------+---------+---------+---------------------+-----------
 sales_europe   |      10| 4519b   | FRANCE  | 17-JAN-12 00:00:00 |     45000
 sales_europe   |      10| 9519b   | ITALY   | 07-JUL-12 00:00:00 |     15000
 sales_europe   |      10| 9519a   | FRANCE  | 18-AUG-12 00:00:00 |    650000
 sales_europe   |      10| 9519b   | FRANCE  | 18-AUG-12 00:00:00 |    650000
 sales_asia     |      20| 3788a   | INDIA   | 01-MAR-12 00:00:00 |     75000
 sales_asia     |      20| 3788a   | PAKISTAN| 04-JUN-12 00:00:00 |     37500
 sales_asia     |      20| 3788b   | INDIA   | 21-SEP-12 00:00:00 |      5090
 sales_asia     |      20| 4519a   | INDIA   | 18-OCT-12 00:00:00 |    650000
 sales_asia     |      20| 4519b   | INDIA   | 02-DEC-12 00:00:00 |      5090
 sales_americas|      40| 9519b   | US      | 12-APR-12 00:00:00 |    145000
 sales_americas|      40| 4577b   | US      | 11-NOV-12 00:00:00 |     25000
 sales_americas|      30| 7588b   | CANADA  | 14-DEC-12 00:00:00 |     50000
 sales_americas|      30| 9519b   | CANADA  | 01-FEB-12 00:00:00 |     75000
 sales_americas|      30| 4519b   | CANADA  | 08-APR-12 00:00:00 |    120000
 sales_americas|      40| 3788a   | US      | 12-MAY-12 00:00:00 |      4950
 sales_americas|      40| 4788a   | US      | 23-SEP-12 00:00:00 |      4950
 sales_americas|      40| 4788b   | US      | 09-OCT-12 00:00:00 |     15000
(17 rows)
```

Querying the `n_america` table shows that the row that was previously stored in the `americas` partition has been moved to the `n_america` table:

```
acctg=# SELECT tableoid::regclass, * FROM n_america;
 tableoid  | dept_no | part_no | country |        date        | amount
-----------+---------+---------+---------+--------------------+------------
 n_america |      40 | 9519b   | US      | 12-APR-12 00:00:00 |     145000
(1 row)
```


3.5.7 ALTER TABLE… MOVE PARTITION

Use the `ALTER TABLE… MOVE PARTITION` command to move a partition or subpartition to a different tablespace. The command takes two forms.

The first form moves a partition to a new tablespace:

```
ALTER TABLE table_name
  MOVE PARTITION partition_name
   TABLESPACE tablespace_name;
```

The second form moves a subpartition to a new tablespace:

```
ALTER TABLE table_name
  MOVE SUBPARTITION subpartition_name
   TABLESPACE tablespace_name;
```

The command syntax makes no distinctions between a partition and a subpartition:

- You can move a partition with the `MOVE PARTITION` or `MOVE SUBPARTITION` clause.

- You can move a subpartition with `MOVE PARTITION` or `MOVE SUBPARTITION` clause.

Description

The `ALTER TABLE…MOVE PARTITION` command moves a partition or subpartition from it's current tablespace to a different tablespace. The `ALTER TABLE… MOVE PARTITION` command can move partitions (or subpartitions) of a `LIST`, `RANGE` or `HASH` partitioned (or subpartitioned) table. You must own a table to invoke `ALTER TABLE… MOVE PARTITION` or `ALTER TABLE… MOVE SUBPARTITION`.

Parameters

`table_name`

The name (optionally schema-qualified) of the table in which the partition resides.

`partition_name`

The name of the partition or subpartition to be moved.

`tablespace_name`

The name of the tablespace to which the partition or subpartition will be moved.

3.5.7.1 Example - Moving a Partition to a Different Tablespace

The following example moves a partition of the `sales` table from one tablespace to another. First, create the `sales` table with the command:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012 VALUES LESS THAN ('2012-Apr-01'),
  PARTITION q2_2012 VALUES LESS THAN ('2012-Jul-01'),
  PARTITION q3_2012 VALUES LESS THAN ('2012-Oct-01'),
  PARTITION q4_2012 VALUES LESS THAN ('2013-Jan-01') TABLESPACE ts_1,
  PARTITION q1_2013 VALUES LESS THAN ('2013-Mar-01') TABLESPACE ts_2
);
```

Querying the `ALL_TAB_PARTITIONS` view confirms that the partitions reside on the expected servers and tablespaces:

```
acctg=# SELECT partition_name, tablespace_name FROM ALL_TAB_PARTITIONS;
 partition_name | tablespace_name
----------------+-------------+-----------------
 q1_2013        | ts_2
 q4_2012        | ts_1
 q3_2012        |
 q2_2012        |
 q1_2012        |
(5 rows)
```

After preparing the target tablespace, invoke the `ALTER TABLE… MOVE PARTITION` command to move the `q1_2013` partition from a tablespace named `ts_2` to a tablespace named `ts_3`:

```
ALTER TABLE sales MOVE PARTITION q1_2013 TABLESPACE ts_3;
```

Querying the `ALL_TAB_PARTITIONS` view shows that the move was successful:

```
acctg=# SELECT partition_name, tablespace_name FROM ALL_TAB_PARTITIONS;
 partition_name | tablespace_name
----------------+-----------------
 q1_2013        | ts_3
 q4_2012        | ts_1
 q3_2012        |
```



```
 q2_2012         |
 q1_2012         |
(5 rows)
```

3.5.8 ALTER TABLE… RENAME PARTITION

Use the `ALTER TABLE… RENAME PARTITION` command to rename a table partition. The syntax takes two forms:

```
ALTER TABLE table_name
 RENAME PARTITION partition_name
 TO new_name;
```

and

```
ALTER TABLE table_name
 RENAME SUBPARTITION subpartition_name
 TO new_name;
```

This command makes no distinctions between a partition and a subpartition:

- You can rename a partition with the `RENAME PARTITION` or `RENAME SUBPARTITION` clause.
- You can rename a subpartition with `RENAME PARTITION` or `RENAME SUBPARTITION` clause.

Description

The `ALTER TABLE… RENAME PARTITION` and `ALTER TABLE… RENAME SUBPARTITION` commands rename a partition or subpartition. You must own the specified table to invoke `ALTER TABLE… RENAME PARTITION` or `ALTER TABLE… RENAME SUBPARTITION`.

Parameters

table_name

The name (optionally schema-qualified) of the table in which the partition resides.

partition_name

The name of the partition or subpartition to be renamed.

new_name

The new name of the partition or subpartition.

3.5.8.1 Example - Renaming a Partition

The following command creates a list-partitioned table named `sales`:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Query the `ALL_TAB_PARTITIONS` view to display the partition names:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 americas       | 'US', 'CANADA'
(3 rows)
```

The following command renames the `americas` partition to `n_america`:

```
ALTER TABLE sales
 RENAME PARTITION americas TO n_america;
```

Querying the `ALL_TAB_PARTITIONS` view demonstrates that the partition has been successfully renamed:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 n_america      | 'US', 'CANADA'
(3 rows)
```

3.5.9 DROP TABLE

Use the PostgreSQL `DROP TABLE` command to remove a partitioned table definition, it's partitions and subpartitions, and delete the table contents. The syntax is:

```
DROP TABLE table_name
```

Description

The `DROP TABLE` command removes an entire table, and the data that resides in that table. When you delete a table, any partitions or subpartitions (of that table) are deleted as well.

To use the `DROP TABLE` command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

Parameters

table_name

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

Example

To delete a table, connect to the controller node (the host of the partitioning root), and invoke the `DROP TABLE` command. For example, to delete the `sales` table, invoke the following command:

```
DROP TABLE sales;
```

The server will confirm that the table has been dropped:

```
acctg=# drop table sales;
DROP TABLE
acctg=#
```

For more information about the `DROP TABLE` command, please see the PostgreSQL core documentation at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-droptable.html

3.5.10 ALTER TABLE… DROP PARTITION

Use the `ALTER TABLE… DROP PARTITION` command to delete a partition definition, and the data stored in that partition. The syntax is:

```
ALTER TABLE table_name DROP PARTITION partition_name;
```

Parameters

`table_name`

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

`partition_name`

The name of the partition to be deleted.

Description

The `ALTER TABLE… DROP PARTITION` command deletes a partition and any data stored on that partition. The `ALTER TABLE… DROP PARTITION` command can drop partitions of a `LIST` or `RANGE` partitioned table; please note that this command does not work on a `HASH` partitioned table. When you delete a partition, any subpartitions (of that partition) are deleted as well.

To use the `DROP PARTITION` clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have database superuser or administrative privileges.

3.5.10.1 Example - Deleting a Partition

The example that follows deletes a partition of the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
```

```
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Querying the `ALL_TAB_PARTITIONS` view displays the partition names:

```
acctg=# SELECT partition_name, server_name, high_value FROM
ALL_TAB_PARTITIONS;
 partition_name | server_name |     high_value
----------------+-------------+---------------------
 europe         | seattle     | 'FRANCE', 'ITALY'
 asia           | chicago     | 'INDIA', 'PAKISTAN'
 americas       | boston      | 'US', 'CANADA'
(3 rows)
```

To delete the `americas` partition from the `sales` table, invoke the following command:

```
ALTER TABLE sales DROP PARTITION americas;
```

Querying the `ALL_TAB_PARTITIONS` view demonstrates that the partition has been successfully deleted:

```
acctg=# SELECT partition_name, server_name, high_value FROM
ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 asia           | 'INDIA', 'PAKISTAN'
 europe         | 'FRANCE', 'ITALY'
(2 rows)
```


3.5.11 ALTER TABLE… DROP SUBPARTITION

Use the `ALTER TABLE… DROP SUBPARTITION` command to drop a subpartition definition, and the data stored in that subpartition. The syntax is:

```
ALTER TABLE table_name DROP SUBPARTITION subpartition_name;
```

Parameters

table_name

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

subpartition_name

The name of the subpartition to be deleted.

Description

The `ALTER TABLE… DROP SUBPARTITION` command deletes a subpartition, and the data stored in that subpartition. To use the `DROP SUBPARTITION` clause, you must be the owner of the partitioning root, a member of a group that owns the table, or have superuser or administrative privileges.

3.5.11.1 Example - Deleting a Subpartition

The example that follows deletes a subpartition of the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY RANGE(date)
  SUBPARTITION BY LIST (country)
  (
    PARTITION first_half_2012 VALUES LESS THAN('01-JUL-2012')
    (
      SUBPARTITION europe VALUES ('ITALY', 'FRANCE'),
```

```
    SUBPARTITION americas VALUES ('CANADA', 'US'),
    SUBPARTITION asia VALUES ('PAKISTAN', 'INDIA')
  ),
  PARTITION second_half_2012 VALUES LESS THAN('01-JAN-2013')
);
```

Querying the `ALL_TAB_SUBPARTITIONS` view displays the subpartition names:

```
acctg=# SELECT subpartition_name, high_value, server_name FROM
ALL_TAB_SUBPARTITIONS; subpartition_name |     high_value      | server_name
-------------------+---------------------+-------------
 europe            | 'ITALY', 'FRANCE'   | chicago
 americas          | 'CANADA', 'US'      | seattle
 asia              | 'PAKISTAN', 'INDIA' | boston
(3 rows)
```

To delete the `americas` subpartition from the `sales` table, invoke the following command:

```
ALTER TABLE sales DROP SUBPARTITION americas;
```

Querying the `ALL_TAB_SUBPARTITIONS` view demonstrates that the subpartition has been successfully deleted:

```
acctg=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |     high_value
-------------------+---------------------
 europe            | 'ITALY', 'FRANCE'
 asia              | 'PAKISTAN', 'INDIA'
(2 rows)
```

3.5.12 TRUNCATE TABLE

Use the `TRUNCATE TABLE` command to remove the contents of a table, while preserving the table definition. When you truncate a table, any partitions or subpartitions of that table are also truncated. The syntax is:

```
TRUNCATE TABLE table_name
```

Description

The `TRUNCATE TABLE` command removes an entire table, and the data that resides in that table. When you delete a table, any partitions or subpartitions (of that table) are deleted as well.

To use the `TRUNCATE TABLE` command, you must be the owner of the partitioning root, a member of a group that owns the table, the schema owner, or a database superuser.

Parameters

table_name

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

3.5.12.1 Example - Emptying a Table

The example that follows removes the data from the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date    date,
  amount  number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Populate the `sales` table with the command:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that the partitions are populated with data:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid    |dept_no | part_no | country  |        date         | amount
---------------+--------+---------+----------+---------------------+-----------
sales_europe  |     10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |     45000
sales_europe  |     10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |     15000
sales_europe  |     10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |    650000
 sales_europe |     10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |    650000
sales_asia    |     20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |     75000
sales_asia    |     20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |     37500
sales_asia    |     20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |      5090
sales_asia    |     20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |    650000
sales_asia    |     20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |      5090
sales_americas|     40 | 9519b   | US       | 12-APR-12 00:00:00 |    145000
sales_americas|     40 | 4577b   | US       | 11-NOV-12 00:00:00 |     25000
sales_americas|     30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |     50000
sales_americas|     30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |     75000
sales_americas|     30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |    120000
sales_americas|     40 | 3788a   | US       | 12-MAY-12 00:00:00 |      4950
sales_americas|     40 | 4788a   | US       | 23-SEP-12 00:00:00 |      4950
sales_americas|     40 | 4788b   | US       | 09-OCT-12 00:00:00 |     15000
(17 rows)
```

To delete the contents of the `sales` table, invoke the following command:

```
TRUNCATE TABLE sales;
```

Now, querying the `sales` table shows that the data has been removed but the structure is intact:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
 tableoid | dept_no | part_no | country |   date   | amount
```

```
----------+---------+---------+---------+----------+------------
(0 rows)
```

For more information about the `TRUNCATE TABLE` command, please see the PostgreSQL documentation at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-truncate.html

3.5.13 ALTER TABLE… TRUNCATE PARTITION

Use the `ALTER TABLE… TRUNCATE PARTITION` command to remove the data from the specified partition, leaving the partition structure intact. The syntax is:

```
ALTER TABLE table_name TRUNCATE PARTITION partition_name
  [{DROP|REUSE} STORAGE]
```

Description

Use the `ALTER TABLE… TRUNCATE PARTITION` command to remove the data from the specified partition, leaving the partition structure intact. When you truncate a partition, any subpartitions of that partition are also truncated.

`ALTER TABLE… TRUNCATE PARTITION` will not cause `ON DELETE` triggers that might exist for the table to fire, but it will fire `ON TRUNCATE` triggers. If an `ON TRUNCATE` trigger is defined for the partition, all `BEFORE TRUNCATE` triggers are fired before any truncation happens, and all `AFTER TRUNCATE` triggers are fired after the last truncation occurs.

You must have the `TRUNCATE` privilege on a table to invoke `ALTER TABLE… TRUNCATE PARTITION`.

Parameters

`table_name`

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

`partition_name`

The name of the partition to be deleted.

`DROP STORAGE` and `REUSE STORAGE` are included for compatibility only; the clauses are parsed and ignored.

3.5.13.1 Example - Emptying a Partition

The example that follows removes the data from a partition of the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
```

```
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA')
);
```

Populate the `sales` table with the command:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2012', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2012', '50000'),
  (30, '9519b', 'CANADA', '01-Feb-2012', '75000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2012', '4950'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (10, '9519a', 'FRANCE', '18-Aug-2012', '650000'),
  (10, '9519b', 'FRANCE', '18-Aug-2012', '650000'),
  (20, '3788b', 'INDIA', '21-Sept-2012', '5090'),
  (40, '4788a', 'US', '23-Sept-2012', '4950'),
  (40, '4788b', 'US', '09-Oct-2012', '15000'),
  (20, '4519a', 'INDIA', '18-Oct-2012', '650000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that the partitions are populated with data:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date        | amount
----------------+---------+---------+----------+--------------------+--------
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 | 145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |  25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |  50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |  75000
```

```
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 | 120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |   4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |   4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |  15000
(17 rows)
```

To delete the contents of the `americas` partition, invoke the following command:

```
ALTER TABLE sales TRUNCATE PARTITION americas;
```

Now, querying the `sales` table shows that the content of the `americas` partition has been removed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid   | dept_no | part_no | country  |        date        | amount
--------------+---------+---------+----------+--------------------+--------
 sales_europe |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |  45000
 sales_europe |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_europe |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_europe |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 | 650000
 sales_asia   |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_asia   |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_asia   |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |   5090
 sales_asia   |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 | 650000
 sales_asia   |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |   5090
(9 rows)
```

While the rows have been removed, the structure of the `americas` partition is still intact:

```
acctg=# SELECT partition_name, high_value FROM ALL_TAB_PARTITIONS;
 partition_name |     high_value
----------------+---------------------
 europe         | 'FRANCE', 'ITALY'
 asia           | 'INDIA', 'PAKISTAN'
 americas       | 'US', 'CANADA'
(3 rows)
```


3.5.14 ALTER TABLE… TRUNCATE SUBPARTITION

Use the `ALTER TABLE… TRUNCATE SUBPARTITION` command to remove all of the data from the specified subpartition, leaving the subpartition structure intact. The syntax is:

```
ALTER TABLE table_name
  TRUNCATE SUBPARTITION subpartition_name
  [{DROP|REUSE} STORAGE]
```

Description

The `ALTER TABLE… TRUNCATE SUBPARTITION` command removes all data from a specified subpartition, leaving the subpartition structure intact.

`ALTER TABLE… TRUNCATE SUBPARTITION` will not cause `ON DELETE` triggers that might exist for the table to fire, but it will fire `ON TRUNCATE` triggers. If an `ON TRUNCATE` trigger is defined for the subpartition, all `BEFORE TRUNCATE` triggers are fired before any truncation happens, and all `AFTER TRUNCATE` triggers are fired after the last truncation occurs.

You must have the `TRUNCATE` privilege on a table to invoke `ALTER TABLE… TRUNCATE SUBPARTITION`.

Parameters

table_name

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

subpartition_name

The name of the subpartition to be truncated.

The `DROP STORAGE` and `REUSE STORAGE` clauses are included for compatibility only; the clauses are parsed and ignored.

3.5.14.1 Example - Emptying a Subpartition

The example that follows removes the data from a subpartition of the `sales` table. Use the following command to create the `sales` table:

```
CREATE TABLE sales
```

```
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date) SUBPARTITION BY LIST (country)
(
  PARTITION "2011" VALUES LESS THAN('01-JAN-2012')
  (
    SUBPARTITION europe_2011 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2011 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2011 VALUES ('US', 'CANADA')
  ),
  PARTITION "2012" VALUES LESS THAN('01-JAN-2013')
  (
    SUBPARTITION europe_2012 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2012 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2012 VALUES ('US', 'CANADA')
  ),
  PARTITION "2013" VALUES LESS THAN('01-JAN-2014')
  (
    SUBPARTITION europe_2013 VALUES ('ITALY', 'FRANCE'),
    SUBPARTITION asia_2013 VALUES ('PAKISTAN', 'INDIA'),
    SUBPARTITION americas_2013 VALUES ('US', 'CANADA')
  )
);
```

Populate the `sales` table with the command:

```
INSERT INTO sales VALUES
  (10, '4519b', 'FRANCE', '17-Jan-2011', '45000'),
  (20, '3788a', 'INDIA', '01-Mar-2012', '75000'),
  (40, '9519b', 'US', '12-Apr-2012', '145000'),
  (20, '3788a', 'PAKISTAN', '04-Jun-2012', '37500'),
  (40, '4577b', 'US', '11-Nov-2012', '25000'),
  (30, '7588b', 'CANADA', '14-Dec-2011', '50000'),
  (30, '4519b', 'CANADA', '08-Apr-2012', '120000'),
  (40, '3788a', 'US', '12-May-2011', '4950'),
  (20, '3788a', 'US', '04-Apr-2012', '37500'),
  (40, '4577b', 'INDIA', '11-Jun-2011', '25000'),
  (10, '9519b', 'ITALY', '07-Jul-2012', '15000'),
  (20, '4519b', 'INDIA', '2-Dec-2012', '5090');
```

Querying the `sales` table shows that the rows have been distributed amongst the subpartitions:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
```



```
      tableoid      | dept_no| part_no| country  |        date       |amount
--------------------+--------+--------+----------+-------------------+-------
 sales_2011_europe  |      10| 4519b  | FRANCE   | 17-JAN-11 00:00:00|  45000
 sales_2011_asia    |      40| 4577b  | INDIA    | 11-JUN-11 00:00:00|  25000
 sales_2011_americas|      30| 7588b  | CANADA   | 14-DEC-11 00:00:00|  50000
 sales_2011_americas|      40| 3788a  | US       | 12-MAY-11 00:00:00|   4950
 sales_2012_europe  |      10| 9519b  | ITALY    | 07-JUL-12 00:00:00|  15000
 sales_2012_asia    |      20| 3788a  | INDIA    | 01-MAR-12 00:00:00|  75000
 sales_2012_asia    |      20| 3788a  | PAKISTAN | 04-JUN-12 00:00:00|  37500
 sales_2012_asia    |      20| 4519b  | INDIA    | 02-DEC-12 00:00:00|   5090
 sales_2012_americas|      40| 9519b  | US       | 12-APR-12 00:00:00| 145000
 sales_2012_americas|      40| 4577b  | US       | 11-NOV-12 00:00:00| 25000
 sales_2012_americas|      30| 4519b  | CANADA   | 08-APR-12 00:00:00| 120000
 sales_2012_americas|      20| 3788a  | US       | 04-APR-12 00:00:00|  37500
(12 rows)
```

To delete the contents of the `2012_americas` partition, invoke the following command:

```
ALTER TABLE sales TRUNCATE SUBPARTITION "americas_2012";
```

Now, querying the `sales` table shows that the content of the `americas_2012` partition has been removed:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
      tableoid      | dept_no|part_no| country  |        date        | amount
--------------------+--------+-------+----------+--------------------+--------
 sales_2011_europe  |      10| 4519b | FRANCE   | 17-JAN-11 00:00:00 |  45000
 sales_2011_asia    |      40| 4577b | INDIA    | 11-JUN-11 00:00:00 |  25000
 sales_2011_americas|      30| 7588b | CANADA   | 14-DEC-11 00:00:00 |  50000
 sales_2011_americas|      40| 3788a | US       | 12-MAY-11 00:00:00 |   4950
 sales_2012_europe  |      10| 9519b | ITALY    | 07-JUL-12 00:00:00 |  15000
 sales_2012_asia    |      20| 3788a | INDIA    | 01-MAR-12 00:00:00 |  75000
 sales_2012_asia    |      20| 3788a | PAKISTAN | 04-JUN-12 00:00:00 |  37500
 sales_2012_asia    |      20| 4519b | INDIA    | 02-DEC-12 00:00:00 |   5090
(8 rows)
```

While the rows have been removed, the structure of the `2012_americas` partition is still intact:

```
acctg=# SELECT subpartition_name, high_value FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name |     high_value
-------------------+---------------------
 2013_europe       | 'ITALY', 'FRANCE'
 2012_europe       | 'ITALY', 'FRANCE'
 2011_europe       | 'ITALY', 'FRANCE'
 2013_asia         | 'PAKISTAN', 'INDIA'
 2012_asia         | 'PAKISTAN', 'INDIA'
 2011_asia         | 'PAKISTAN', 'INDIA'
 2013_americas     | 'US', 'CANADA'
 2012_americas     | 'US', 'CANADA'
 2011_americas     | 'US', 'CANADA'
(9
rows)
```

3.6 Handling Stray Values in a LIST or RANGE Partitioned Table

A DEFAULT or MAXVALUE partition or subpartition will capture any rows that do not meet the other partitioning rules defined for a table.

Defining a DEFAULT Partition

A DEFAULT partition will capture any rows that do not fit into any other partition in a LIST partitioned (or subpartitioned) table. If you do not include a DEFAULT rule, any row that does not match one of the values in the partitioning constraints will result in an error. Each LIST partition or subpartition may have its own DEFAULT rule.

The syntax of a DEFAULT rule is:

```
PARTITION [partition_name] VALUES (DEFAULT)
```

Where *partition_name* specifies the name of the partition or subpartition that will store any rows that do not match the rules specified for other partitions.

The last example created a list partitioned table in which the server decided which partition to store the data based upon the value of the country column. If you attempt to add a row in which the value of the country column contains a value not listed in the rules, Advanced Server reports an error:

```
acctg=# INSERT INTO sales VALUES
acctg-#   (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');
ERROR:  inserted partition key does not map to any partition
```

The following example creates the same table, but adds a DEFAULT partition. The server will store any rows that do not match a value specified in the partitioning rules for europe, asia, or americas partitions in the others partition:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY LIST(country)
(
  PARTITION europe VALUES('FRANCE', 'ITALY'),
  PARTITION asia VALUES('INDIA', 'PAKISTAN'),
  PARTITION americas VALUES('US', 'CANADA'),
  PARTITION others VALUES (DEFAULT)
);
```


To test the `DEFAULT` partition, add row with a value in the `country` column that does not match one of the countries specified in the partitioning constraints:

```
INSERT INTO sales VALUES
  (40, '3000x', 'IRELAND', '01-Mar-2012', '45000');
```

Querying the contents of the `sales` table confirms that the previously rejected row is now stored in the `sales_others` partition:

```
acctg=# SELECT tableoid::regclass, * FROM sales;
    tableoid    | dept_no | part_no | country  |        date         |  amount
----------------+---------+---------+----------+---------------------+--------
 sales_europe   |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |   45000
 sales_europe   |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |   15000
 sales_europe   |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |  650000
 sales_europe   |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |  650000
 sales_asia     |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |   75000
 sales_asia     |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |   37500
 sales_asia     |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |    5090
 sales_asia     |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |  650000
 sales_asia     |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |    5090
 sales_americas |      40 | 9519b   | US       | 12-APR-12 00:00:00 |  145000
 sales_americas |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |   25000
 sales_americas |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |   50000
 sales_americas |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |   75000
 sales_americas |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |  120000
 sales_americas |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |    4950
 sales_americas |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |    4950
 sales_americas |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |   15000
 sales_others   |      40 | 3000x   | IRELAND  | 01-MAR-12 00:00:00 |   45000
(18 rows)
```

Please note that Advanced Server does not have a way to re-assign the contents of a `DEFAULT` partition or subpartition:

- You cannot use the `ALTER TABLE… ADD PARTITION` command to add a partition to a table with a `DEFAULT` rule, but you can use the `ALTER TABLE… SPLIT PARTITION` command to split an existing partition.

- You cannot use the `ALTER TABLE… ADD SUBPARTITION` command to add a subpartition to a table with a `DEFAULT` rule, but you can use the `ALTER TABLE… SPLIT SUBPARTITION` command to split an existing subpartition.

Defining a MAXVALUE Partition

A `MAXVALUE` partition (or subpartition) will capture any rows that do not fit into any other partition in a range-partitioned (or subpartitioned) table. If you do not include a `MAXVALUE` rule, any row that exceeds the maximum limit specified by the partitioning rules will result in an error. Each partition or subpartition may have its own `MAXVALUE` partition.

The syntax of a MAXVALUE rule is:

```
PARTITION [partition_name] VALUES LESS THAN (MAXVALUE)
```

Where *partition_name* specifies the name of the partition that will store any rows that do not match the rules specified for other partititions.

The last example created a range-partitioned table in which the data was partitioned based upon the value of the date column. If you attempt to add a row with a date that exceeds a date listed in the partitioning constraints, Advanced Server reports an error:

```
acctg=# INSERT INTO sales VALUES
acctg-#   (40, '3000x', 'IRELAND', '01-Mar-2013', '45000');
ERROR:  inserted partition key does not map to any partition
```

The following CREATE TABLE command creates the same table, but with a MAXVALUE partition. Instead of throwing an error, the server will store any rows that do not match the previous partitioning constraints in the others partition:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  date        date,
  amount      number
)
PARTITION BY RANGE(date)
(
  PARTITION q1_2012 VALUES LESS THAN('2012-Apr-01'),
  PARTITION q2_2012 VALUES LESS THAN('2012-Jul-01'),
  PARTITION q3_2012 VALUES LESS THAN('2012-Oct-01'),
  PARTITION q4_2012 VALUES LESS THAN('2013-Jan-01')
  PARTITION others VALUES LESS THAN (MAXVALUE)
);
```

To test the MAXVALUE partition, add a row with a value in the date column that exceeds the last date value listed in a partitioning rule. The server will store the row in the others partition:

```
INSERT INTO sales VALUES
  (40, '3000x', 'IRELAND', '01-Mar-2013', '45000');
```

Querying the contents of the sales table confirms that the previously rejected row is now stored in the sales_others partition :

```
acctg=# SELECT tableoid::regclass, * FROM sales;
   tableoid    | dept_no | part_no | country  |        date        | amount
```



```
---------------+---------+---------+----------+--------------------+---------
 sales_q1_2012 |      10 | 4519b   | FRANCE   | 17-JAN-12 00:00:00 |    45000
 sales_q1_2012 |      20 | 3788a   | INDIA    | 01-MAR-12 00:00:00 |    75000
 sales_q1_2012 |      30 | 9519b   | CANADA   | 01-FEB-12 00:00:00 |    75000
 sales_q2_2012 |      40 | 9519b   | US       | 12-APR-12 00:00:00 |   145000
 sales_q2_2012 |      20 | 3788a   | PAKISTAN | 04-JUN-12 00:00:00 |    37500
 sales_q2_2012 |      30 | 4519b   | CANADA   | 08-APR-12 00:00:00 |   120000
 sales_q2_2012 |      40 | 3788a   | US       | 12-MAY-12 00:00:00 |     4950
 sales_q3_2012 |      10 | 9519b   | ITALY    | 07-JUL-12 00:00:00 |    15000
 sales_q3_2012 |      10 | 9519a   | FRANCE   | 18-AUG-12 00:00:00 |   650000
 sales_q3_2012 |      10 | 9519b   | FRANCE   | 18-AUG-12 00:00:00 |   650000
 sales_q3_2012 |      20 | 3788b   | INDIA    | 21-SEP-12 00:00:00 |     5090
 sales_q3_2012 |      40 | 4788a   | US       | 23-SEP-12 00:00:00 |     4950
 sales_q4_2012 |      40 | 4577b   | US       | 11-NOV-12 00:00:00 |    25000
 sales_q4_2012 |      30 | 7588b   | CANADA   | 14-DEC-12 00:00:00 |    50000
 sales_q4_2012 |      40 | 4788b   | US       | 09-OCT-12 00:00:00 |    15000
 sales_q4_2012 |      20 | 4519a   | INDIA    | 18-OCT-12 00:00:00 |   650000
 sales_q4_2012 |      20 | 4519b   | INDIA    | 02-DEC-12 00:00:00 |     5090
 sales_others  |      40 | 3000x   | IRELAND  | 01-MAR-13 00:00:00 |    45000
(18 rows)
```

Please note that Advanced Server does not have a way to re-assign the contents of a `MAXVALUE` partition or subpartition:

- You cannot use the `ALTER TABLE… ADD PARTITION` statement to add a partition to a table with a `MAXVALUE` rule, but you can use the `ALTER TABLE… SPLIT PARTITION` statement to split an existing partition.

- You cannot use the `ALTER TABLE… ADD SUBPARTITION` statement to add a subpartition to a table with a `MAXVALUE` rule , but you can split an existing subpartition with the `ALTER TABLE… SPLIT SUBPARTITION` statement.

3.7 Specifying Multiple Partitioning Keys in a RANGE Partitioned Table

You can often improve performance by specifying multiple key columns for a RANGE partitioned table. If you often select rows using comparison operators (based on a greater-than or less-than value) on a small set of columns, consider using those columns in RANGE partitioning rules.

Specifying Multiple Keys in a Range-Partitioned Table

Range-partitioned table definitions may include multiple columns in the partitioning key. To specify multiple partitioning keys for a range-partitioned table, include the column names in a comma-separated list after the PARTITION BY RANGE clause:

```
CREATE TABLE sales
(
  dept_no     number,
  part_no     varchar2,
  country     varchar2(20),
  sale_year    number,
  sale_month   number,
  sale_day     number,
  amount      number
)
PARTITION BY RANGE(sale_year, sale_month)
(
  PARTITION q1_2012
    VALUES LESS THAN(2012, 4),
  PARTITION q2_2012
    VALUES LESS THAN(2012, 7),
  PARTITION q3_2012
    VALUES LESS THAN(2012, 10),
  PARTITION q4_2012
    VALUES LESS THAN(2013, 1)
);
```

If a table is created with multiple partitioning keys, you must specify multiple key values when querying the table to take full advantage of partition pruning:

```
acctg=# EXPLAIN SELECT * FROM sales WHERE sale_year = 2012 AND sale_month =
8;
                                   QUERY PLAN
-----------------------------------------------------------------------------
Result  (cost=0.00..14.35 rows=2 width=250)
   ->  Append  (cost=0.00..14.35 rows=2 width=250)
         ->  Seq Scan on sales  (cost=0.00..0.00 rows=1 width=250)
               Filter: ((sale_year = 2012::numeric) AND (sale_month =
8::numeric))
         ->  Seq Scan on sales_q3_2012 sales  (cost=0.00..14.35 rows=1
width=250)
```

```
               Filter: ((sale_year = 2012::numeric) AND (sale_month = 
8::numeric))
(6 rows)
```

Since all rows with a value of 8 in the sale_month column and a value of 2012 in the sale_year column will be stored in the q3_2012 partition, Advanced Server searches only that partition.

3.8 Retrieving Information about a Partitioned Table

Advanced Server provides five system catalog views that you can use to view information about the structure of partitioned tables.

Querying the Partitioning Views

You can query the following views to retrieve information about partitioned and subpartitioned tables:

- `ALL_PART_TABLES`
- `ALL_TAB_PARTITIONS`
- `ALL_TAB_SUBPARTITIONS`
- `ALL_PART_KEY_COLUMNS`
- `ALL_SUBPART_KEY_COLUMNS`

The structure of each view is explained in Section 13.5.1, *Table Partitioning Views*. If you are using the EDB-PSQL client, you can also discover the structure of a view by entering:

```
\d view_name
```

Where `view_name` specifies the name of the table partitioning view.

Querying a view can provide information about the structure of a partitioned or subpartitioned table. For example, the following code snippit displays the system-assigned names of a subpartitioned table:

```
acctg=# SELECT subpartition_name, partition_name FROM ALL_TAB_SUBPARTITIONS;
 subpartition_name | partition_name
-------------------+----------------
 SYS_SUBP107       | americas
 SYS_SUBP104       | asia
 SYS_SUBP101       | europe
 SYS_SUBP108       | americas
 SYS_SUBP105       | asia
 SYS_SUBP102       | europe
 SYS_SUBP109       | americas
 SYS_SUBP106       | asia
 SYS_SUBP103       | europe
(9 rows)
```

3.8.1 Table Partitioning Views - Reference

Query the following catalog views to review detailed information about your partitioned tables.

3.8.1.1 ALL_PART_TABLES

The following table lists the information available in the ALL_PART_TABLES view:

Column	Type	Description
owner	name	The owner of the table.
table_name	name	The name of the table.
schema_name	name	The schema in which the table resides.
partitioning_type	text	RANGE, LIST or HASH
subpartitioning_type	text	RANGE, LIST, HASH, or NONE
partition_count	bigint	The number of partitions.
def_subpartition_count	integer	The default subpartition count - this will always be 0.
partitioning_key_count	integer	The number of columns listed in the partition by clause.
subpartitioning_key_count	integer	The number of columns in the subpartition by clause.
status	character varying(8)	This column will always be VALID.
def_tablespace_name	character varying(30)	This column will always be NULL.
def_pct_free	numeric	This column will always be NULL.
def_pct_used	numeric	This column will always be NULL.
def_ini_trans	numeric	This column will always be NULL.
def_max_trans	numeric	This column will always be NULL.
def_initial_extent	character varying(40)	This column will always be NULL.
def_next_extent	character varying(40)	This column will always be NULL.
def_min_extents	character varying(40)	This column will always be NULL.
def_max_extents	character varying(40)	This column will always be NULL.
def_pct_increase	character varying(40)	This column will always be NULL.
def_freelists	numeric	This column will always be NULL.
def_freelist_groups	numeric	This column will always be NULL.
def_logging	character varying(7)	This column will always be YES
def_compression	character varying(8)	This column will always be NONE
def_buffer_pool	character varying(7)	This column will always be DEFAULT
ref_ptn_constraint_name	character varying(30)	This column will always be NULL.
interval	character varying(1000)	This column will always be NULL.

3.8.1.2 ALL_TAB_PARTITIONS

The following table lists the information available in the ALL_TAB_PARTITIONS view:

Column	Type	Description
table_owner	name	The owner of the table.
table_name	name	The name of the table.
schema_name	name	The schema in which the table resides.
composite	text	YES if the table is subpartioned; NO if it is not subpartitioned.
partition_name	name	The name of the partition.
subpartition_count	bigint	The number of subpartitions for this partition.
high_value	text	The partition limit for RANGE partitions, or the partition value for LIST partitions.
high_value_length	integer	The length of high_value.
partition_position	integer	The ordinal position of this partition.
tablespace_name	name	The tablespace in which this partition resides.
pct_free	numeric	This column will always be 0.
pct_used	numeric	This column will always be 0.
ini_trans	numeric	This column will always be 0.
max_trans	numeric	This column will always be 0.
initial_extent	numeric	This column will always be NULL.
next_extent	numeric	This column will always be NULL.
min_extent	numeric	This column will always be 0.
max_extent	numeric	This column will always be 0.
pct_increase	numeric	This column will always be 0.
freelists	numeric	This column will always be NULL.
freelist_groups	numeric	This column will always be NULL.
logging	character varying(7)	This column will always be YES.
compression	character varying(8)	This column will always be NONE.
num_rows	numeric	The approx. number of rows in this partition.
blocks	integer	The approx. number of blocks in this partition.
empty_blocks	numeric	This column will always be NULL.
avg_space	numeric	This column will always be NULL.
chain_cnt	numeric	This column will always be NULL.
avg_row_len	numeric	This column will always be NULL.
sample_size	numeric	This column will always be NULL.
last_analyzed	timestamp without time zone	This column will always be NULL.
buffer_pool	character varying(7)	This column will always be NULL.
global_stats	character varying(3)	This column will always be YES.
user_stats	character varying(3)	This column will always be NO.
backing_table	regclass	OID of the backing table for this partition.
server_name	name	The name of the server on which the partition resides.

3.8.1.3 ALL_TAB_SUBPARTITIONS

The following table lists the information available in the `ALL_TAB_SUBPARTITIONS` view:

Column	Type	Description
table_owner	name	The name of the owner of the table.
table_name	name	The name of the table.
schema_name	name	The name of the schema in which the table resides.
partition_name	name	The name of the partition.
high_value	text	The subpartition limit for RANGE, or the subpartition value for LIST subpartitions.
high_value_length	integer	The length of high_value.
subpartition_name	name	The name of the subpartition.
subpartition_position	integer	The ordinal position of this subpartition.
tablespace_name	name	The tablespace in which this subpartition resides.
pct_free	numeric	This column will always be 0.
pct_used	numeric	This column will always be 0.
ini_trans	numeric	This column will always be 0.
max_trans	numeric	This column will always be 0.
initial_extent	numeric	This column will always be NULL.
next_extent	numeric	This column will always be NULL.
min_extent	numeric	This column will always be 0.
max_extent	numeric	This column will always be 0.
pct_increase	numeric	This column will always be 0.
freelists	numeric	This column will always be NULL.
freelist_groups	numeric	This column will always be NULL.
logging	character varying(7)	This column will always be YES.
compression	character varying(8)	This column will always be NONE.
num_rows	numeric	The approx. number of rows in this subpartition.
blocks	integer	The approx. number of blocks in this subpartition.
empty_blocks	numeric	This column will always be NULL.
avg_space	numeric	This column will always be NULL.
chain_cnt	numeric	This column will always be NULL.
avg_row_len	numeric	This column will always be NULL.
sample_size	numeric	This column will always be NULL.
last_analyzed	timestamp without time zone	This column will always be NULL.
buffer_pool	character varying(7)	This column will always be NULL.
global_stats	character varying(3)	This column will always be YES.
user_stats	character varying(3)	Thiscolumn will always be NO.
backing_table	regclass	OID of the backing table for this subpartition.
server_name	name	The name of the server on which the subpartition resides.

3.8.1.4 ALL_PART_KEY_COLUMNS

The following table lists the information available in the `ALL_PART_KEY_COLUMNS` view:

Column	Type	Description
`owner`	`name`	The name of the table owner.
`name`	`name`	The name of the table.
`schema`	`name`	The name of the schema on which the table resides.
`object_type`	`character(5)`	This column will always be TABLE.
`column_name`	`name`	The name of the partitioning key column.
`column_position`	`integer`	The position of this column within the partitioning key (the first column has a column position of 1, the second column has a column position of 2...)

3.8.1.5 ALL_SUBPART_KEY_COLUMNS

The following table lists the information available in the `ALL_SUBPART_KEY_COLUMNS` view:

Column	Type	Description
`owner`	`name`	The name of the table owner.
`name`	`name`	The name of the table.
`schema`	`name`	The name of the schema on which the table resides.
`object_type`	`character(5)`	This column will always be TABLE.
`column_name`	`name`	The name of the partitioning key column.
`column_position`	`integer`	The position of this column within the subpartitioning key (the first column has a column position of 1, the second column has a column position of 2...)

4 Security

The chapter describes various features providing for added security.

4.1 Protecting Against SQL Injection Attacks

Postgres Plus Advanced Server provides protection against SQL injection attacks. A *SQL injection attack* is an attempt to compromise a database by running SQL statements whose results provide clues to the attacker as to the content, structure, or security of that database.

Preventing a SQL injection attack is normally the responsibility of the application developer. The database administrator typically has little or no control over the potential threat. The difficulty for database administrators is that the application must have access to the data to function properly.

SQL/Protect is a module that allows a database administrator to protect a database from SQL injection attacks. SQL/Protect provides a layer of security in addition to the normal database security policies by examining incoming queries for common SQL injection profiles.

SQL/Protect gives the control back to the database administrator by alerting the administrator to potentially dangerous queries and by blocking these queries.

4.1.1 SQL/Protect Overview

This section contains an introduction to the different types of SQL injection attacks and describes how SQL/Protect guards against them.

4.1.1.1 Types of SQL Injection Attacks

There are a number of different techniques used to perpetrate SQL injection attacks. Each technique is characterized by a certain *signature*. SQL/Protect examines queries for the following signatures:

Unauthorized Relations

While Postgres Plus Advanced Server allows administrators to restrict access to relations (tables, views, etc.), many administrators do not perform this tedious task. SQL/Protect provides a *learn* mode that tracks the relations a user accesses.

This allows administrators to examine the workload of an application, and for SQL/Protect to learn which relations an application should be allowed to access for a given user or group of users in a role.

When SQL/Protect is switched to either *passive* or *active* mode, the incoming queries are checked against the list of learned relations.

Utility Commands

A common technique used in SQL injection attacks is to run utility commands, which are typically SQL Data Definition Language (DDL) statements. An example is creating a user-defined function that has the ability to access other system resources.

SQL/Protect can block the running of all utility commands, which are not normally needed during standard application processing.

SQL Tautology

The most frequent technique used in SQL injection attacks is issuing a tautological `WHERE` clause condition (that is, using a condition that is always true).

The following is an example:

```
WHERE password = 'x' OR 'x'='x'
```

Attackers will usually start identifying security weaknesses using this technique. SQL/Protect can block queries that use a tautological conditional clause.

Unbounded DML Statements

A dangerous action taken during SQL injection attacks is the running of unbounded DML statements. These are `UPDATE` and `DELETE` statements with no `WHERE` clause. For example, an attacker may update all users' passwords to a known value or initiate a denial of service attack by deleting all of the data in a key table.

4.1.1.2 Monitoring SQL Injection Attacks

This section describes how SQL/Protect monitors and reports on SQL injection attacks.

4.1.1.2.1 Protected Roles

Monitoring for SQL injection attacks involves analyzing SQL statements originating in database sessions where the current user of the session is a protected role. A *protected role* is a Postgres Plus Advanced Server user or group that the database administrator has chosen to monitor using SQL/Protect. (In Postgres Plus Advanced Server, users and groups are collectively referred to as *roles*.)

Each protected role can be customized for the types of SQL injection attacks (discussed in Section 4.1.1.1) for which it is to be monitored, thus providing different levels of protection by role and significantly reducing the user maintenance load for DBAs.

Note: A role with the superuser privilege cannot be made a protected role. If a protected non-superuser role is subsequently altered to become a superuser, certain behaviors are exhibited whenever an attempt is made by that superuser to issue any command:

- A warning message is issued by SQL/Protect on every command issued by the protected superuser.
- The statistic in column `superusers` of `edb_sql_protect_stats` is incremented with every command issued by the protected superuser. See Section 4.1.1.2.2 for information on the `edb_sql_protect_stats` view.
- When SQL/Protect is in active mode, all commands issued by the protected superuser are prevented from running.

A protected role that has the superuser privilege should either be altered so that it is no longer a superuser, or it should be reverted back to an unprotected role.

4.1.1.2.2 Attack Attempt Statistics

Each usage of a command by a protected role that is considered an attack by SQL/Protect is recorded. Statistics are collected by type of SQL injection attack as discussed in Section 4.1.1.1.

These statistics are accessible from view `edb_sql_protect_stats` that can be easily monitored to identify the start of a potential attack.

The columns in `edb_sql_protect_stats` monitor the following:

- **username.** Name of the protected role.
- **superusers.** Number of SQL statements issued when the protected role is a superuser. In effect, any SQL statement issued by a protected superuser increases this statistic. See Section 4.1.1.2.1 for information on protected superusers.
- **relations.** Number of SQL statements issued referencing relations that were not learned by a protected role. (That is, relations that are not in a role's protected relations list.)
- **commands.** Number of DDL statements issued by a protected role.
- **tautology.** Number of SQL statements issued by a protected role that contained a tautological condition.
- **dml.** Number of `UPDATE` and `DELETE` statements issued by a protected role that did not contain a `WHERE` clause.

This gives database administrators the opportunity to react proactively in preventing theft of valuable data or other malicious actions.

If a role is protected in more than one database, the role's statistics for attacks in each database are maintained separately and are viewable only when connected to the respective database.

Note: SQL/Protect statistics are maintained in memory while the database server is running. When the database server is shut down, the statistics are saved to a binary file named `edb_sqlprotect.stat` in the `data/global` subdirectory of the Postgres Plus Advanced Server home directory.

4.1.1.2.3 Attack Attempt Queries

Each usage of a command by a protected role that is considered an attack by SQL/Protect is recorded in view `edb_sql_protect_queries`.

View `edb_sql_protect_queries` contains the following columns:

- **username.** Database user name of the attacker used to log into the database server.
- **ip_address.** IP address of the machine from which the attack was initiated.
- **port.** Port number from which the attack originated.
- **machine_name.** Name of the machine, if known, from which the attack originated.
- **date_time.** Date and time at which the query was received by the database server. The time is stored to the precision of a minute.

- **query.** The query string sent by the attacker.

The maximum number of offending queries that are saved in `edb_sql_protect_queries` is controlled by configuration parameter `edb_sql_protect.max_queries_to_save`.

If a role is protected in more than one database, the role's queries for attacks in each database are maintained separately and are viewable only when connected to the respective database.

4.1.2 Configuring SQL/Protect

The library file (`sqlprotect.so` on Linux, `sqlprotect.dll` on Windows) necessary to run SQL/Protect should already be installed in the `lib` subdirectory of your Postgres Plus Advanced Server home directory.

You will also need the SQL script file `sqlprotect.sql` located in the `share/contrib` subdirectory of your Postgres Plus Advanced Server home directory.

You must configure the database server to use SQL/Protect, and you must configure each database that you want SQL/Protect to monitor:

- The database server configuration file, `postgresql.conf`, must be modified by adding and enabling configuration parameters used by SQL/Protect.
- Database objects used by SQL/Protect must be installed in each database that you want SQL/Protect to monitor.

Step 1: Edit the following configuration parameters in the `postgresql.conf` file located in the `data` subdirectory of your Postgres Plus Advanced Server home directory.

- **shared_preload_libraries.** Add `$libdir/sqlprotect` to the list of libraries.
- **edb_sql_protect.enabled.** Controls whether or not SQL/Protect is actively monitoring protected roles by analyzing SQL statements issued by those roles and reacting according to the setting of `edb_sql_protect.level`. When you are ready to begin monitoring with SQL/Protect set this parameter to `on`. If this parameter is omitted, the default is `off`.
- **edb_sql_protect.level.** Sets the action taken by SQL/Protect when a SQL statement is issued by a protected role. If this parameter is omitted, the default behavior is `passive`. Initially, set this parameter to `learn`. See Section 4.1.2.1.2 for further explanation of this parameter.
- **edb_sql_protect.max_protected_roles.** Sets the maximum number of roles that can be protected. If this parameter is omitted, the default setting is `64`. See Section 2.1.3.12.8 for information on the maximum range of this parameter.
- **edb_sql_protect.max_protected_relations.** Sets the maximum number of relations that can be protected per role. If this parameter is omitted, the default setting is `1024`.
  Please note the total number of protected relations for the server will be the number of protected relations times the number of protected roles. Every protected relation consumes space in shared memory. The space for the maximum possible protected relations is reserved during database server startup.
  See Section 2.1.3.12.7 for information about the maximum range of this parameter.

- **edb_sql_protect.max_queries_to_save.** Sets the maximum number of offending queries to save in the edb_sql_protect_queries view. If this parameter is omitted, the default setting is 5000. If the number of offending queries reaches the limit, additional queries are not saved in the view, but are accessible in the database server log file. **Note:** The minimum valid value for this parameter is 100. If a value less than 100 is specified, the database server starts using the default setting of 5000. A warning message is recorded in the database server log file. See Section 2.1.3.12.9 for information on the maximum range of this parameter.

The following example shows the settings of these parameters in the postgresql.conf file:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/sqlprotect'
                                        # (change requires restart)
                    .
                    .
                    .
edb_sql_protect.enabled = off
edb_sql_protect.level = learn
edb_sql_protect.max_protected_roles = 64
edb_sql_protect.max_protected_relations = 1024
edb_sql_protect.max_queries_to_save = 5000
```

Step 2: Restart the database server after you have modified the postgresql.conf file.

For Linux only: Run the /etc/init.d/ppas-9.4 script with the restart option as shown by the following:

```
$ su root
Password:
$ /etc/init.d/ppas-9.4 restart
Restarting Postgres Plus Advanced Server 9.4:
waiting for server to shut down.... done
server stopped
waiting for server to start.... done
server started
Postgres Plus Advanced Server 9.4 restarted successfully
```

For Windows only: Open Control Panel, Administrative Tools, and then Services. Restart the service named ppas-9.4.

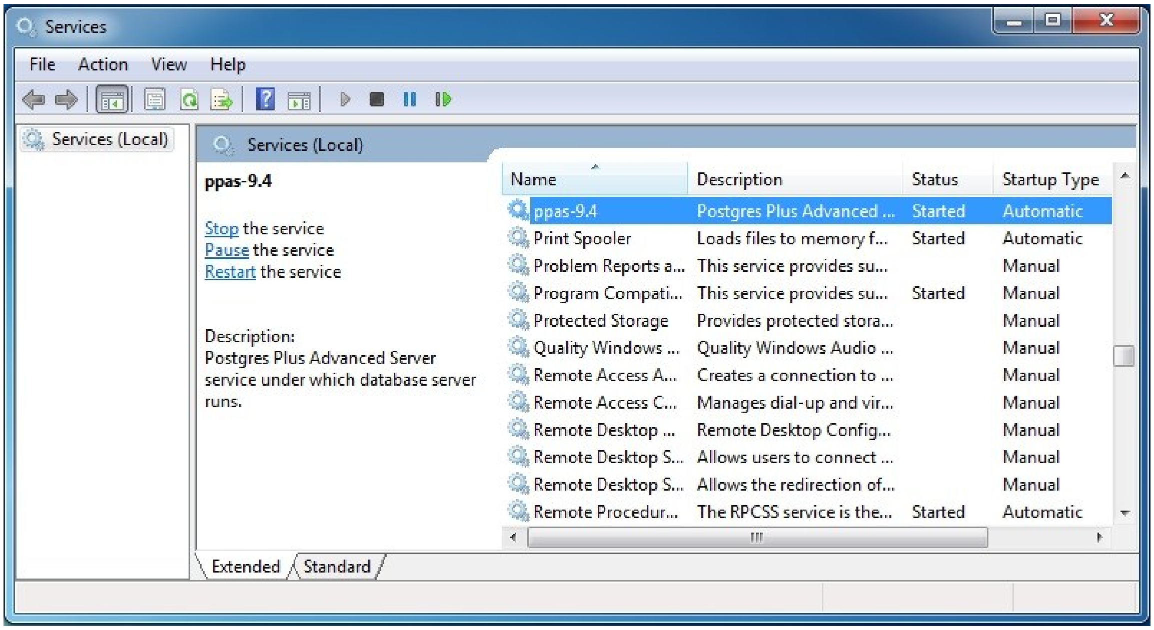


Figure 4.2 - Starting the ppas-9.4 service.

Step 3: For each database that you want to protect from SQL injection attacks, connect to the database as a superuser (either `enterprisedb` or `postgres`, depending upon your installation options) and run the script `sqlprotect.sql` located in the `share/contrib` subdirectory of your Postgres Plus Advanced Server home directory.

The script creates the SQL/Protect database objects in a schema named `sqlprotect`.

The following example shows this process to set up protection for a database named `edb`:

```
$ /opt/PostgresPlus/9.4AS/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (9.4.0.0)
Type "help" for help.

edb=# \i /opt/PostgresPlus/9.4AS/share/contrib/sqlprotect.sql
CREATE SCHEMA
GRANT
SET
CREATE TABLE
GRANT
CREATE TABLE
GRANT
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
DO
CREATE FUNCTION
CREATE FUNCTION
```



```
DO
CREATE VIEW
GRANT
DO
CREATE VIEW
GRANT
CREATE VIEW
GRANT
CREATE FUNCTION
CREATE FUNCTION
SET
```

4.1.2.1 Selecting Roles to Protect

After the SQL/Protect database objects have been created in a database, you select the roles for which SQL queries are to be monitored for protection, and the level of protection.

4.1.2.1.1 Setting the Protected Roles List

For each database that you want to protect, you must determine the roles you want to monitor and then add those roles to the *protected roles list* of that database.

Step 1: Connect as a superuser to a database that you wish to protect using either `psql` or Postgres Enterprise Manager Client.

```
$ /opt/PostgresPlus/9.4AS/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (9.4.0.0)
Type "help" for help.

edb=#
```

Step 2: Since the SQL/Protect tables, functions, and views are built under the `sqlprotect` schema, use the `SET search_path` command to include the `sqlprotect` schema in your search path. This eliminates the need to schema-qualify any operation or query involving SQL/Protect database objects.

```
edb=# SET search_path TO sqlprotect;
SET
```

Step 3: Each role that you wish to protect must be added to the protected roles list. This list is maintained in the table `edb_sql_protect`.

To add a role, use the function `protect_role('`*`rolename`*`')`.

The following example protects a role named `appuser`.

```
edb=# SELECT protect_role('appuser');
 protect_role
--------------
```

```
(1 row)
```

You can list the roles that have been added to the protected roles list by issuing the following query:

```
edb=# SELECT * FROM edb_sql_protect;
 dbid  | roleid | protect_relations | allow_utility_cmds | allow_tautology | allow_empty_dml
-------+--------+-------------------+--------------------+-----------------+-----------------
 13917 |  16671 | t                 | f                  | f               | f
(1 row)
```

A view is also provided that gives the same information using the object names instead of the Object Identification numbers (OIDs).

```
edb=# \x
Expanded display is on.
edb=# SELECT * FROM list_protected_users;
-[ RECORD 1 ]------+--------
dbname             | edb
username           | appuser
protect_relations  | t
allow_utility_cmds | f
allow_tautology    | f
allow_empty_dml    | f
```

4.1.2.1.2 Setting the Protection Level

Configuration parameter `edb_sql_protect.level` sets the protection level, which defines the behavior of SQL/Protect when a protected role issues a SQL statement. **The defined behavior applies to all roles in the protected roles lists of all databases configured with SQL/Protect in the database server.**

In the `postgresql.conf` file the `edb_sql_protect.level` configuration parameter can be set to one of the following values to use either learn mode, passive mode, or active mode:

- **learn.** Tracks the activities of protected roles and records the relations used by the roles. This is used when initially configuring SQL/Protect so the expected behaviors of the protected applications are learned.
- **passive.** Issues warnings if protected roles are breaking the defined rules, but does not stop any SQL statements from executing. This is the next step after SQL/Protect has learned the expected behavior of the protected roles. This essentially behaves in intrusion detection mode and can be run in production when properly monitored.
- **active.** Stops all invalid statements for a protected role. This behaves as a SQL firewall preventing dangerous queries from running. This is particularly effective against early penetration testing when the attacker is trying to determine the vulnerability point and the type of database behind the application. Not only does SQL/Protect close those vulnerability points, but it tracks the blocked queries

allowing administrators to be alerted before the attacker finds an alternate method of penetrating the system.

If the `edb_sql_protect.level` parameter is not set or is omitted from the configuration file, the default behavior of SQL/Protect is passive.

If you are using SQL/Protect for the first time, set `edb_sql_protect.level` to `learn`.

4.1.2.2 Monitoring Protected Roles

Once you have configured SQL/Protect in a database, added roles to the protected roles list, and set the desired protection level, you can then activate SQL/Protect in one of learn mode, passive mode, or active mode. You can then start running your applications.

With a new SQL/Protect installation, the first step is to determine the relations that protected roles should be permitted to access during normal operation. Learn mode allows a role to run applications during which time SQL/Protect is recording the relations that are accessed. These are added to the role's *protected relations list* stored in table `edb_sql_protect_rel`.

Monitoring for protection against attack begins when SQL/Protect is run in passive or active mode. In passive and active modes, the role is permitted to access the relations in its protected relations list as these were determined to be the relations the role should be able to access during typical usage.

However, if a role attempts to access a relation that is not in its protected relations list, a `WARNING` or `ERROR` severity level message is returned by SQL/Protect. The role's attempted action on the relation may or may not be carried out depending upon whether the mode is passive or active.

4.1.2.2.1 Learn Mode

Step 1: To activate SQL/Protect in learn mode, set the following parameters in the `postgresql.conf` file as shown below:

```
edb_sql_protect.enabled = on
edb_sql_protect.level = learn
```

Step 2: Reload the `postgresql.conf` file.

Choose Expert Configuration, then Reload Configuration from the Postgres Plus Advanced Server application menu.

Note: For an alternative method of reloading the configuration file, use the pg_reload_conf function. Be sure you are connected to a database as a superuser and execute function pg_reload_conf as shown by the following example:

```
edb=# SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

Step 3: Allow the protected roles to run their applications.

As an example the following queries are issued in the psql application by protected role appuser:

```
edb=> SELECT * FROM dept;
NOTICE:  SQLPROTECT: Learned relation: 16384
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)

edb=> SELECT empno, ename, job FROM emp WHERE deptno = 10;
NOTICE:  SQLPROTECT: Learned relation: 16391
 empno | ename  |    job
-------+--------+-----------
  7782 | CLARK  | MANAGER
  7839 | KING   | PRESIDENT
  7934 | MILLER | CLERK
(3 rows)
```

SQL/Protect generates a NOTICE severity level message indicating the relation has been added to the role's protected relations list.

In SQL/Protect learn mode, SQL statements that are cause for suspicion are not prevented from executing, but a message is issued to alert the user to potentially dangerous statements as shown by the following example:

```
edb=> CREATE TABLE appuser_tab (f1 INTEGER);
NOTICE:  SQLPROTECT: This command type is illegal for this user
CREATE TABLE
edb=> DELETE FROM appuser_tab;
NOTICE:  SQLPROTECT: Learned relation: 16672
NOTICE:  SQLPROTECT: Illegal Query: empty DML
DELETE 0
```

Step 4: As a protected role runs applications, the SQL/Protect tables can be queried to observe the addition of relations to the role's protected relations list.

Connect as a superuser to the database you are monitoring and set the search path to include the sqlprotect schema.

```
edb=# SET search_path TO sqlprotect;
SET
```

Query the edb_sql_protect_rel table to see the relations added to the protected relations list:

```
edb=# SELECT * FROM edb_sql_protect_rel;
 dbid  | roleid | relid
-------+--------+-------
 13917 |  16671 | 16384
 13917 |  16671 | 16391
 13917 |  16671 | 16672
(3 rows)
```

The view list_protected_rels is provided that gives more comprehensive information along with the object names instead of the OIDs.

```
edb=# SELECT * FROM list_protected_rels;
 Database | Protected User | Schema |    Name     | Type  |    Owner
----------+----------------+--------+-------------+-------+--------------
 edb      | appuser        | public | dept        | Table | enterprisedb
 edb      | appuser        | public | emp         | Table | enterprisedb
 edb      | appuser        | public | appuser_tab | Table | appuser
(3 rows)
```

4.1.2.2.2 Passive Mode

Once you have determined that a role's applications have accessed all relations they will need, you can now change the protection level so that SQL/Protect can actively monitor the incoming SQL queries and protect against SQL injection attacks.

Passive mode is the less restrictive of the two protection modes, passive and active.

Step 1: To activate SQL/Protect in passive mode, set the following parameters in the postgresql.conf file as shown below:

```
edb_sql_protect.enabled = on
edb_sql_protect.level = passive
```

Step 2: Reload the configuration file as shown in Step 2 of Section 4.1.2.2.1.

Now SQL/Protect is in passive mode. For relations that have been learned such as the dept and emp tables of the prior examples, SQL statements are permitted with no special notification to the client by SQL/Protect as shown by the following queries run by user appuser:

```
edb=> SELECT * FROM dept;
 deptno |   dname    |   loc
```



```
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BOSTON
(4 rows)

edb=> SELECT empno, ename, job FROM emp WHERE deptno = 10;
 empno | ename  |    job
-------+--------+-----------
  7782 | CLARK  | MANAGER
  7839 | KING   | PRESIDENT
  7934 | MILLER | CLERK
(3 rows)
```

SQL/Protect does not prevent any SQL statement from executing, but issues a message of `WARNING` severity level for SQL statements executed against relations that were not learned, or for SQL statements that contain a prohibited signature as shown in the following example:

```
edb=> CREATE TABLE appuser_tab_2 (f1 INTEGER);
WARNING:  SQLPROTECT: This command type is illegal for this user
CREATE TABLE
edb=> INSERT INTO appuser_tab_2 VALUES (1);
WARNING:  SQLPROTECT: Illegal Query: relations
INSERT 0 1
edb=> INSERT INTO appuser_tab_2 VALUES (2);
WARNING:  SQLPROTECT: Illegal Query: relations
INSERT 0 1
edb=> SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
WARNING:  SQLPROTECT: Illegal Query: relations
WARNING:  SQLPROTECT: Illegal Query: tautology
 f1
----
  1
  2
(2 rows)
```

Step 3: Monitor the statistics for suspicious activity.

By querying the view `edb_sql_protect_stats`, you can see the number of times SQL statements were executed that referenced relations that were not in a role's protected relations list, or contained SQL injection attack signatures. See Section 4.1.1.2.2 for more information on view `edb_sql_protect_stats`.

The following is a query on `edb_sql_protect_stats`:

```
edb=# SET search_path TO sqlprotect;
SET
edb=# SELECT * FROM edb_sql_protect_stats;
 username | superusers | relations | commands | tautology | dml
----------+------------+-----------+----------+-----------+-----
 appuser  |          0 |         3 |        1 |         1 |   0
(1 row)
```

Step 4: View information on specific attacks.

By querying the view `edb_sql_protect_queries`, you can see the SQL statements that were executed that referenced relations that were not in a role's protected relations list, or contained SQL injection attack signatures. See Section 4.1.1.2.3 for more information on view `edb_sql_protect_queries`.

The following is a query on `edb_sql_protect_queries`:

```
edb=# SELECT * FROM edb_sql_protect_queries;
-[ RECORD 1 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (1);
-[ RECORD 2 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | CREATE TABLE appuser_tab_2 (f1 INTEGER);
-[ RECORD 3 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (2);
-[ RECORD 4 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

Note: The `ip_address` and `port` columns do not return any information if the attack originated on the same host as the database server using the Unix-domain socket (that is, `pg_hba.conf` connection type `local`).

4.1.2.2.3 Active Mode

In active mode, disallowed SQL statements are prevented from executing. Also, the message issued by SQL/Protect has a higher severity level of `ERROR` instead of `WARNING`.

Step 1: To activate SQL/Protect in active mode, set the following parameters in the `postgresql.conf` file as shown below:

```
edb_sql_protect.enabled = on
edb_sql_protect.level = active
```

Step 2: Reload the configuration file as shown in Step 2 of Section 4.1.2.2.1.

The following example illustrates SQL statements similar to those given in the examples of Step 2 in Section 4.1.2.2.2, but executed by user `appuser` when `edb_sql_protect.level` is set to `active`:

```
edb=> CREATE TABLE appuser_tab_3 (f1 INTEGER);
ERROR:  SQLPROTECT: This command type is illegal for this user
edb=> INSERT INTO appuser_tab_2 VALUES (1);
ERROR:  SQLPROTECT: Illegal Query: relations
edb=> SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
ERROR:  SQLPROTECT: Illegal Query: relations
```

The following shows the resulting statistics:

```
edb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
 username | superusers | relations | commands | tautology | dml
----------+------------+-----------+----------+-----------+-----
 appuser  |          0 |         5 |        2 |         1 |   0
(1 row)
```

The following is a query on `edb_sql_protect_queries`:

```
edb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
-[ RECORD 1 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | CREATE TABLE appuser_tab_2 (f1 INTEGER);
-[ RECORD 2 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (2);
-[ RECORD 3 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | CREATE TABLE appuser_tab_3 (f1 INTEGER);
-[ RECORD 4 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (1);
-[ RECORD 5 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```


4.1.3 Common Maintenance Operations

The following describes how to perform other common operations.

You must be connected as a superuser to perform these operations and have included schema `sqlprotect` in your search path.

4.1.3.1 Adding a Role to the Protected Roles List

To add a role to the protected roles list run `protect_role('rolename')`.

```
protect_role('rolename')
```

This is shown by the following example:

```
edb=# SELECT protect_role('newuser');
 protect_role
--------------

(1 row)
```

4.1.3.2 Removing a Role From the Protected Roles List

To remove a role from the protected roles list use either of the following functions:

```
unprotect_role('rolename')
unprotect_role(roleoid)
```

Note: The variation of the function using the OID is useful if you remove the role using the `DROP ROLE` or `DROP USER` SQL statement before removing the role from the protected roles list. If a query on a SQL/Protect relation returns a value such as `unknown (OID=16458)` for the user name, use the `unprotect_role(roleoid)` form of the function to remove the entry for the deleted role from the protected roles list.

Removing a role using these functions also removes the role's protected relations list.

The statistics for a role that has been removed are not deleted until you use the `drop_stats` function as described in Section 4.1.3.5.

The offending queries for a role that has been removed are not deleted until you use the `drop_queries` function as described in Section 4.1.3.6.

The following is an example of the `unprotect_role` function:

```
edb=# SELECT unprotect_role('newuser');
 unprotect_role
----------------

(1 row)
```

Alternatively, the role could be removed by giving its OID of `16693`:

```
edb=# SELECT unprotect_role(16693);
 unprotect_role
----------------

(1 row)
```

4.1.3.3 Setting the Types of Protection for a Role

You can change whether or not a role is protected from a certain type of SQL injection attack.

Change the Boolean value for the column in `edb_sql_protect` corresponding to the type of SQL injection attack for which protection of a role is to be disabled or enabled.

Be sure to qualify the following columns in your `WHERE` clause of the statement that updates `edb_sql_protect`:

- **dbid.** OID of the database for which you are making the change
- **roleid.** OID of the role for which you are changing the Boolean settings

For example, to allow a given role to issue utility commands, update the `allow_utility_cmds` column as follows:

```
UPDATE edb_sql_protect SET allow_utility_cmds = TRUE WHERE dbid = 13917 AND
roleid = 16671;
```

You can verify the change was made by querying `edb_sql_protect` or `list_protected_users`. In the following query note that column `allow_utility_cmds` now contains `t`.

```
edb=# SELECT dbid, roleid, allow_utility_cmds FROM edb_sql_protect;
 dbid  | roleid | allow_utility_cmds
-------+--------+--------------------
 13917 |  16671 | t
(1 row)
```

The updated rules take effect on new sessions started by the role since the change was made.

4.1.3.4 Removing a Relation From the Protected Relations List

If SQL/Protect has learned that a given relation is accessible for a given role, you can subsequently remove that relation from the role's protected relations list.

Delete its entry from the `edb_sql_protect_rel` table using any of the following functions:

```
unprotect_rel('rolename', 'relname')
unprotect_rel('rolename', 'schema', 'relname')
unprotect_rel(roleoid, reloid)
```

If the relation given by *`relname`* is not in your current search path, specify the relation's schema using the second function format.

The third function format allows you to specify the OIDs of the role and relation, respectively, instead of their text names.

The following example illustrates the removal of the `public.emp` relation from the protected relations list of the role `appuser`.

```
edb=# SELECT unprotect_rel('appuser', 'public', 'emp');
 unprotect_rel
---------------

(1 row)
```

The following query shows there is no longer an entry for the `emp` relation.

```
edb=# SELECT * FROM list_protected_rels;
 Database | Protected User | Schema |     Name      | Type  |     Owner
----------+----------------+--------+---------------+-------+--------------
 edb      | appuser        | public | dept          | Table | enterprisedb
 edb      | appuser        | public | appuser_tab   | Table | appuser
(2 rows)
```

SQL/Protect will now issue a warning or completely block access (depending upon the setting of `edb_sql_protect.level`) whenever the role attempts to utilize that relation.

4.1.3.5 Deleting Statistics

You can delete statistics from view `edb_sql_protect_stats` using either of the two following functions:

```
drop_stats('rolename')
drop_stats(roleoid)
```

Note: The variation of the function using the OID is useful if you remove the role using the DROP ROLE or DROP USER SQL statement before deleting the role's statistics using drop_stats('*rolename*'). If a query on edb_sql_protect_stats returns a value such as unknown (OID=16458) for the user name, use the drop_stats(*roleoid*) form of the function to remove the deleted role's statistics from edb_sql_protect_stats.

The following is an example of the drop_stats function:

```
edb=# SELECT drop_stats('appuser');
 drop_stats
------------

(1 row)

edb=# SELECT * FROM edb_sql_protect_stats;
 username | superusers | relations | commands | tautology | dml
----------+------------+-----------+----------+-----------+-----
(0 rows)
```

The following is an example of using the drop_stats(*roleoid*) form of the function when a role is dropped before deleting its statistics:

```
edb=# SELECT * FROM edb_sql_protect_stats;
      username       | superusers | relations | commands | tautology | dml
---------------------+------------+-----------+----------+-----------+-----
 unknown (OID=16693) |          0 |         5 |        3 |         1 |   0
 appuser             |          0 |         5 |        2 |         1 |   0
(2 rows)

edb=# SELECT drop_stats(16693);
 drop_stats
------------

(1 row)

edb=# SELECT * FROM edb_sql_protect_stats;
 username | superusers | relations | commands | tautology | dml
----------+------------+-----------+----------+-----------+-----
 appuser  |          0 |         5 |        2 |         1 |   0
(1 row)
```

4.1.3.6 Deleting Offending Queries

You can delete offending queries from view edb_sql_protect_queries using either of the two following functions:

```
drop_queries('rolename')
drop_queries(roleoid)
```

Note: The variation of the function using the OID is useful if you remove the role using the DROP ROLE or DROP USER SQL statement before deleting the role's offending queries using drop_queries('*rolename*'). If a query on edb_sql_protect_queries returns a value such as unknown (OID=16454) for the user name, use the drop_queries(*roleoid*) form of the function to remove the deleted role's offending queries from edb_sql_protect_queries.

The following is an example of the drop_queries function:

```
edb=# SELECT drop_queries('appuser');
 drop_queries
--------------
            5
(1 row)

edb=# SELECT * FROM edb_sql_protect_queries;
 username | ip_address | port | machine_name | date_time | query
----------+------------+------+--------------+-----------+-------
(0 rows)
```

The following is an example of using the drop_queries(*roleoid*) form of the function when a role is dropped before deleting its queries:

```
edb=# SELECT username, query FROM edb_sql_protect_queries;
      username       |                    query
---------------------+----------------------------------------------
 unknown (OID=16454) | CREATE TABLE appuser_tab_2 (f1 INTEGER);
 unknown (OID=16454) | INSERT INTO appuser_tab_2 VALUES (2);
 unknown (OID=16454) | CREATE TABLE appuser_tab_3 (f1 INTEGER);
 unknown (OID=16454) | INSERT INTO appuser_tab_2 VALUES (1);
 unknown (OID=16454) | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
(5 rows)

edb=# SELECT drop_queries(16454);
 drop_queries
--------------
            5
(1 row)

edb=# SELECT * FROM edb_sql_protect_queries;
 username | ip_address | port | machine_name | date_time | query
----------+------------+------+--------------+-----------+-------
(0 rows)
```

4.1.3.7 Disabling and Enabling Monitoring

If you wish to turn off SQL/Protect monitoring once you have enabled it, perform the following steps:

Step 1: Set the configuration parameter edb_sql_protect.enabled to off in the postgresql.conf file.

The entry for edb_sql_protect.enabled should look like the following:

```
edb_sql_protect.enabled = off
```

Step 2: Reload the configuration file as shown in Step 2 of Section 4.1.2.2.1.

To re-enable SQL/Protect monitoring perform the following steps:

Step 1: Set the configuration parameter `edb_sql_protect.enabled` to `on` in the `postgresql.conf` file.

The entry for `edb_sql_protect.enabled` should look like the following:

```
edb_sql_protect.enabled = on
```

Step 2: Reload the configuration file as shown in Step 2 of Section 4.1.2.2.1.

4.1.4 Backing Up and Restoring a SQL/Protect Database

Backing up a database that is configured with SQL/Protect, and then restoring the backup file to a new database require additional considerations to what is normally associated with backup and restore procedures. This is primarily due to the use of Object Identification numbers (OIDs) in the SQL/Protect tables as explained in this section.

Note: This section is applicable if your backup and restore procedures result in the re-creation of database objects in the new database with new OIDs such as is the case when using the `pg_dump` backup program.

If you are backing up your Postgres Plus Advanced Server database server by simply using the operating system's copy utility to create a binary image of the Postgres Plus Advanced Server data files (file system backup method), then this section does not apply.

4.1.4.1 Object Identification Numbers in SQL/Protect Tables

SQL/Protect uses two tables, `edb_sql_protect` and `edb_sql_protect_rel`, to store information on database objects such as databases, roles, and relations. References to these database objects in these tables are done using the objects' OIDs, and not the objects' text names. The OID is a numeric data type used by Postgres Plus Advanced Server to uniquely identify each database object.

When a database object is created, Postgres Plus Advanced Server assigns an OID to the object, which is then used whenever a reference is needed to the object in the database catalogs. If you create the same database object in two databases, such as a table with the same `CREATE TABLE` statement, each table is assigned a different OID in each database.

In a backup and restore operation that results in the re-creation of the backed up database objects, the restored objects end up with different OIDs in the new database than what they were assigned in the original database. As a result, the OIDs referencing databases, roles, and relations stored in the `edb_sql_protect` and `edb_sql_protect_rel` tables are no longer valid when these tables are simply dumped to a backup file and then restored to a new database.

The following sections describe two functions, `export_sqlprotect` and `import_sqlprotect`, that are used specifically for backing up and restoring SQL/Protect tables in order to ensure the OIDs in the SQL/Protect tables reference the correct database objects after the SQL/Protect tables are restored.

4.1.4.2 Backing Up the Database

The following are the steps to back up a database that has been configured with SQL/Protect.

Step 1: Create a backup file using `pg_dump`.

The following example shows a plain-text backup file named `/tmp/edb.dmp` created from database `edb` using the `pg_dump` utility program:

```
$ cd /opt/PostgresPlus/9.4AS/bin
$ ./pg_dump -U enterprisedb -Fp -f /tmp/edb.dmp edb
Password:
$
```

Step 2: Connect to the database as a superuser and export the SQL/Protect data using the `export_sqlprotect('`*`sqlprotect_file`*`')` function where *`sqlprotect_file`* is the fully qualified path to a file where the SQL/Protect data is to be saved.

The `enterprisedb` operating system account (`postgres` if you installed Postgres Plus Advanced Server in PostgreSQL compatibility mode) must have read and write access to the directory specified in *`sqlprotect_file`*.

```
edb=# SELECT sqlprotect.export_sqlprotect('/tmp/sqlprotect.dmp');
 export_sqlprotect
-------------------

(1 row)
```

The files `/tmp/edb.dmp` and `/tmp/sqlprotect.dmp` comprise your total database backup.

4.1.4.3 Restoring From the Backup Files

Step 1: Restore the backup file to the new database.

The following example uses the `psql` utility program to restore the plain-text backup file `/tmp/edb.dmp` to a newly created database named `newdb`:

```
$ /opt/PostgresPlus/9.4AS/bin/psql -d newdb -U enterprisedb -f /tmp/edb.dmp
Password for user enterprisedb:
SET
SET
SET
SET
SET
COMMENT
CREATE SCHEMA
    .
    .
    .
```

Step 2: Connect to the new database as a superuser and delete all rows from the `edb_sql_protect_rel` table.

This step removes any existing rows in the `edb_sql_protect_rel` table that were backed up from the original database. These rows do not contain the correct OIDs relative to the database where the backup file has been restored.

```
$ /opt/PostgresPlus/9.4AS/bin/psql -d newdb -U enterprisedb
Password for user enterprisedb:
psql.bin (9.4.0.0)
Type "help" for help.

newdb=# DELETE FROM sqlprotect.edb_sql_protect_rel;
DELETE 2
```

Step 3: Delete all rows from the `edb_sql_protect` table.

This step removes any existing rows in the `edb_sql_protect` table that were backed up from the original database. These rows do not contain the correct OIDs relative to the database where the backup file has been restored.

```
newdb=# DELETE FROM sqlprotect.edb_sql_protect;
DELETE 1
```

Step 4: Delete any statistics that may exist for the database.

This step removes any existing statistics that may exist for the database to which you are restoring the backup. The following query displays any existing statistics:

```
newdb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
 username | superusers | relations | commands | tautology | dml
----------+------------+-----------+----------+-----------+-----
(0 rows)
```

For each row that appears in the preceding query, use the `drop_stats` function specifying the role name of the entry.

For example, if a row appeared with `appuser` in the `username` column, issue the following command to remove it:

```
newdb=# SELECT sqlprotect.drop_stats('appuser');
 drop_stats
------------

(1 row)
```

Step 5: Delete any offending queries that may exist for the database.

This step removes any existing queries that may exist for the database to which you are restoring the backup. The following query displays any existing queries:


```
edb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
 username | ip_address | port | machine_name | date_time | query
----------+------------+------+--------------+-----------+-------
(0 rows)
```

For each row that appears in the preceding query, use the `drop_queries` function specifying the role name of the entry.

For example, if a row appeared with `appuser` in the `username` column, issue the following command to remove it:

```
edb=# SELECT sqlprotect.drop_queries('appuser');
 drop_queries
--------------

(1 row)
```

Step 6: Make sure the role names that were protected by SQL/Protect in the original database exist in the database server where the new database resides.

If the original and new databases reside in the same database server, then nothing needs to be done assuming you have not deleted any of these roles from the database server.

Step 7: Run the function `import_sqlprotect('`*`sqlprotect_file`*`')` where *`sqlprotect_file`* is the fully qualified path to the file you created in Step 2 of Section 4.1.4.2.

```
newdb=# SELECT sqlprotect.import_sqlprotect('/tmp/sqlprotect.dmp');
 import_sqlprotect
-------------------

(1 row)
```

Tables `edb_sql_protect` and `edb_sql_protect_rel` are now populated with entries containing the OIDs of the database objects as assigned in the new database. The statistics view `edb_sql_protect_stats` also now displays the statistics imported from the original database.

The SQL/Protect tables and statistics are now properly restored for this database. This is verified by the following queries on the Postgres Plus Advanced Server system catalogs:

```
newdb=# SELECT datname, oid FROM pg_database;
  datname  |  oid
-----------+-------
 template1 |     1
 template0 | 13909
 edb       | 13917
 newdb     | 16679
(4 rows)

newdb=# SELECT rolname, oid FROM pg_roles;
   rolname    |  oid
--------------+-------
 enterprisedb |    10
```



```
 appuser      | 16671
 newuser      | 16678
(3 rows)

newdb=# SELECT relname, oid FROM pg_class WHERE relname IN ('dept','emp','appuser_tab');
   relname   |  oid
-------------+-------
 appuser_tab | 16803
 dept        | 16809
 emp         | 16812
(3 rows)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect;
 dbid  | roleid | protect_relations | allow_utility_cmds | allow_tautology | allow_empty_dml
-------+--------+-------------------+--------------------+-----------------+-----------------
 16679 |  16671 | t                 | t                  | f               | f
(1 row)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect_rel;
 dbid  | roleid | relid
-------+--------+-------
 16679 |  16671 | 16809
 16679 |  16671 | 16803
(2 rows)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
 username | superusers | relations | commands | tautology | dml
----------+------------+-----------+----------+-----------+-----
 appuser  |          0 |         5 |        2 |         1 |   0
(1 row)

newedb=# \x
Expanded display is on.
nwedb=# SELECT * FROM sqlprotect.edb_sql_protect_queries;
-[ RECORD 1 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:21:00 -04:00
 query        | CREATE TABLE appuser_tab_2 (f1 INTEGER);
-[ RECORD 2 ]+---------------------------------------------
 username     | appuser
 ip_address   |
 port         |
 machine_name |
 date_time    | 20-JUN-14 13:22:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (2);
-[ RECORD 3 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | CREATE TABLE appuser_tab_3 (f1 INTEGER);
-[ RECORD 4 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | INSERT INTO appuser_tab_2 VALUES (1);
-[ RECORD 5 ]+---------------------------------------------
 username     | appuser
 ip_address   | 192.168.2.6
 port         | 50098
 machine_name |
 date_time    | 20-JUN-14 13:39:00 -04:00
 query        | SELECT * FROM appuser_tab_2 WHERE 'x' = 'x';
```

Note the following about the columns in tables `edb_sql_protect` and `edb_sql_protect_rel`:

- **dbid.** Matches the value in the `oid` column from `pg_database` for `newdb`
- **roleid.** Matches the value in the `oid` column from `pg_roles` for `appuser`

Also note that in table `edb_sql_protect_rel`, the values in the `relid` column match the values in the `oid` column of `pg_class` for relations `dept` and `appuser_tab`.

Step 8: Verify that the SQL/Protect configuration parameters are set as desired in the `postgresql.conf` file for the database server running the new database as described throughout sections 0, 4.1.2.1, and 4.1.2.2. Restart the database server or reload the configuration file as appropriate.

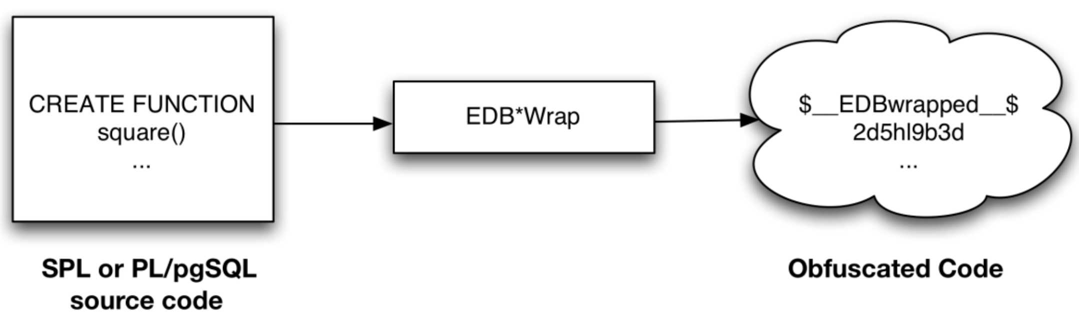
You can now monitor the database using SQL/Protect.

4.2 EDB*Wrap

The EDB*Wrap utility protects proprietary source code and programs (functions, stored procedures, triggers, and packages) from unauthorized scrutiny. The EDB*Wrap program translates a file that contains SPL or PL/pgSQL source code (the plaintext) into a file that contains the same code in a form that is nearly impossible to read. Once you have the obfuscated form of the code, you can send that code to Advanced Server and it will store those programs in obfuscated form. While EDB*Wrap does obscure code, table definitions are still exposed.

Everything you wrap is stored in obfuscated form. If you wrap an entire package, the package body source, as well as the prototypes contained in the package header and the functions and procedures contained in the package body are stored in obfuscated form.

If you wrap a `CREATE PACKAGE` statement, you hide the package API from other developers. You may want to wrap the package body, but not the package header so users can see the package prototypes and other public variables that are defined in the package body. To allow users to see what prototypes the package contains, use EDB*Wrap to obfuscate only the `'CREATE PACKAGE BODY'` statement in the edbwrap input file, omitting the `'CREATE PACKAGE'` statement. The package header source will be stored plaintext, while the package body source and package functions and procedures will be stored obfuscated.



Once wrapped, source code and programs cannot be unwrapped or debugged. Reverse engineering is possible, but would be very difficult.

The entire source file is wrapped into one unit. Any `psql` meta-commands included in the wrapped file will not be recognized when the file is executed; executing an obfuscated file that contains a psql meta-command will cause a syntax error. `edbwrap` does not validate SQL source code - if the plaintext form contains a syntax error, `edbwrap` will not complain. Instead, the server will report an error and abort the entire file when you try to execute the obfuscated form.

4.2.1 Using EDB*Wrap to Obfuscate Source Code

EDB*Wrap is a command line utility; it accepts a single input source file, obfuscates the contents and returns a single output file. When you invoke the `edbwrap` utility, you must provide the name of the file that contains the source code to obfuscate. You may also specify the name of the file where `edbwrap` will write the obfuscated form of the code. `edbwrap` offers three different command-line styles. The first style is shown by the following:

```
edbwrap iname=input_file [oname=output_file]
```

The `iname=input_file` argument specifies the name of the input file; if `input_file` does not contain an extension, `edbwrap` will search for a file named *`input_file`*`.sql`

The `oname=`*`output_file`* argument (which is optional) specifies the name of the output file; if *`output_file`* does not contain an extension, `edbwrap` will append `.plb` to the name.

If you do not specify an output file name, `edbwrap` writes to a file whose name is derived from the input file name: `edbwrap` strips the suffix (typically `.sql`) from the input file name and adds `.plb`.

`edbwrap` offers two other command-line styles that may feel more familiar:

```
edbwrap --iname input_file [--oname output_file]
edbwrap -i input_file [-o output_file]
```

You may mix command-line styles; the rules for deriving input and output file names are identical regardless of which style you use.

Once `edbwrap` has produced a file that contains obfuscated code, you typically feed that file into Advanced Server using a client application such as `edb-psql`. The server executes the obfuscated code line by line and stores the source code for SPL and PL/pgSQL programs in wrapped form.

In summary, to obfuscate code with EDB*Wrap, you:

- Create the source code file.
- Invoke EDB*Wrap to obfuscate the code.
- Import the file as if it were in plaintext form.

The following sequence demonstrates `edbwrap` functionality.

First, create the source code for the `list_emp` function (in plaintext form):

```
$ cat listemp.sql
```

```
CREATE OR REPLACE FUNCTION list_emp() RETURNS VOID
AS $$
DECLARE
    v_empno         NUMERIC(4);
    v_ename         VARCHAR(10);
    emp_cur CURSOR FOR
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    RAISE INFO 'EMPNO    ENAME';
    RAISE INFO '-----    -------';
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN NOT FOUND;
        RAISE INFO '%     %', v_empno, v_ename;
    END LOOP;
    CLOSE emp_cur;
    RETURN;
END;
$$ LANGUAGE 'plpgsql';
```

You can import the `list_emp` function with a client application such as `psql`:

```
$ psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (9.4.0.0)
Type "help" for help.

edb=# \i listemp.sql
CREATE FUNCTION
```

You can view the plaintext source code (stored in the server) by examining the `pg_function` system table:

```
edb=# SELECT funsrc FROM pg_function WHERE funname = 'list_emp';
                        funsrc
------------------------------------------------------
                                                     +
 DECLARE                                             +
     v_empno         NUMERIC(4);                     +
     v_ename         VARCHAR(10);                    +
     emp_cur CURSOR FOR                              +
         SELECT empno, ename FROM emp ORDER BY empno;+
 BEGIN                                               +
     OPEN emp_cur;                                   +
     RAISE INFO 'EMPNO    ENAME';                    +
     RAISE INFO '-----    -------';                  +
     LOOP                                            +
         FETCH emp_cur INTO v_empno, v_ename;        +
         EXIT WHEN NOT FOUND;                        +
         RAISE INFO '%     %', v_empno, v_ename;     +
     END LOOP;                                       +
     CLOSE emp_cur;                                  +
     RETURN;                                         +
 END;                                                +

(1 row)
```

Next, obfuscate the plaintext file with EDB*Wrap:

```
$ edbwrap -i listemp.sql

EDB*Wrap Utility: Release 9.4.0.0

Copyright (c) 2004-2016, EnterpriseDB Corporation.  All Rights Reserved.

Using encoding UTF8 for input
Processing listemp.sql to listemp.plb
$
$ cat listemp.plb
$__EDBwrapped__$
UTF8
d6UEwTa69kFnCNAFVOgJqNyQMH7HwCn8dPPFJlkMFSb6YB4meTCGpIIoBnhYpcnxtAU+ZJMAu0Xe
WOTKG5iU9jpjqlwuioYVNa4EHFrf5JtNRTSL8tWhbi78li8ET5SWdU9eSGZiOfSVGi43b21ZWuGc
F8a342iMTy0bozbdl0r1dYku/f2kHnMYoBCi6EukmHik3j/iO1mJp06GHH71FG7BCOgCSW6L4B4x
BDje0MMVbBJveYyHWxBH12Bi8p4KDGy1HDLC8MK9S9EbfKPJbwKPZK37J8Ci9fhWBorfrTtz1k2f
vO1UKaTZGkYH0MIFvcZw6BG24dFL1kH5E2Rk5x4RzRsV2Hm+2LwTuDexs8hgleA3sPB/oZF9umb2
hZYkT5v1Ja7cKBnowdJrJNj/DOFoJcI1pFgG3DgJ
$__EDBwrapped__$
```

You may notice that the second line of the wrapped file contains an encoding name (in this case, the encoding is UTF8). When you obfuscate a file, edbwrap infers the encoding of the input file by examining the locale. For example, if you are running edbwrap while your locale is set to en_US.utf8, edbwrap assumes that the input file is encoded in UTF8. Be sure to examine the output file after running edbwrap; if the locale contained in the wrapped file does not match the encoding of the input file, you should change your locale and rewrap the input file.

You can import the obfuscated code into Advanced Server using the same tools that work with plaintext code:

```
$ psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (9.4.0.0)
Type "help" for help.

edb=# \i listemp.plb
CREATE FUNCTION
edb=# SELECT funsrc FROM pg_function WHERE funname = 'list_emp';
                                   funsrc
------------------------------------------------------------------------------
                                                                             +
 $__EDBwrapped__$                                                            +
 UTF8                                                                        +
 d6UNH3OTrROsTCLF6NKWq5gWsZxi5giSpg6SmNgWDqHutT8OqqpJZnL5wNtaBxs4B6+inA6qeWCA+
 QsTKvmcDNHk3yFneWI33Jeo/DsdVqkIEMrlUsu2ogymEJedHcM1YQFARyx+l0mWBI+yqixE4BNZw+
 jSeqiVKAhAckek8JzL9pf0QLFT8TTzzTG61KN7iFQQii0B6C4/GpDlZCmC5oDXt94PR15YcZ5fJq+
 p+UThN/uahwIaDu+FQ2AhSxNCxJH1aqjJEnwE9S7jsRvQXQ/yRt4zc7WbfeQMhhLA0E9w+hOy3aS+
 CKb6bHF3pVVQLiG6tWpjdWwgTZ7neG+T1EounZC8bKwn                                +
 $__EDBwrapped__$
(1 row)
```

Invoke the obfuscated code in the same way that you would invoke the plaintext form:

```
edb=# 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)
```

When you use `pg_dump` to back up a database, wrapped programs remain obfuscated in the archive file.

Be aware that audit logs produced by Advanced Server will show wrapped programs in plaintext form. Source code is also displayed in plaintext in SQL error messages generated during the execution of a program.

Note: At this time, the bodies of the objects created by the following statements will not be stored in obfuscated form:

```
CREATE [OR REPLACE] TYPE type_name AS OBJECT
CREATE [OR REPLACE] TYPE type_name UNDER type_name
CREATE [OR REPLACE] TYPE BODY type_name
```

4.3 Virtual Private Database

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 and PL/pgSQL in addition to SPL.

Note: The database objects currently supported by Advanced Server Virtual Private Database are tables. Policies cannot be applied to views or synonyms.

The advantages of using Virtual Private Database are the following:

- Provides a fine-grained level of security. Database object level privileges given by the `GRANT` command determine access privileges to the entire instance of a database object, while Virtual Private Database provides access control for the individual rows of a database object instance.
- A different security policy can be applied depending upon the type of SQL command (`INSERT`, `UPDATE`, `DELETE`, or `SELECT`).
- The security policy can vary dynamically for each applicable SQL command affecting the database object depending upon factors such as the session user of the application accessing the database object.
- Invocation of the security policy is transparent to all applications that access the database object and thus, individual applications do not have to be modified to apply the security policy.
- Once a security policy is enabled, it is not possible for any application (including new applications) to circumvent the security policy except by the system privilege noted by the following.
- Even superusers cannot circumvent the security policy except by the system privilege noted by the following.

Note: The only way security policies can be circumvented is if the `EXEMPT ACCESS POLICY` system privilege has been granted to a user. The `EXEMPT ACCESS POLICY` privilege should be granted with extreme care as a user with this privilege is exempted from all policies in the database.

The `DBMS_RLS` package provides procedures to create policies, remove policies, enable policies, and disable policies. See Section 9.11 for details on using the `DBMS_RLS` package.

5 EDB Resource Manager

EDB Resource Manager is an Advanced Server feature that provides the capability to control the usage of operating system resources used by Advanced Server processes.

This capability allows you to protect the system from processes that may uncontrollably overuse and monopolize certain system resources.

The following are some key points about using EDB Resource Manager.

- The basic component of EDB Resource Manager is a resource group. A *resource group* is a named, global group, available to all databases in an Advanced Server instance, on which various resource usage limits can be defined. Advanced Server processes that are assigned as members of a given resource group are then controlled by EDB Resource Manager so that the aggregate resource usage of all processes in the group is kept near the limits defined on the group.
- Data definition language commands are used to create, alter, and drop resource groups. These commands can only be used by a database user with superuser privileges.
- The desired, aggregate consumption level of all processes belonging to a resource group is defined by *resource type parameters*. There are different resource type parameters for the different types of system resources currently supported by EDB Resource Manager.
- Multiple resource groups can be created, each with different settings for its resource type parameters, thus defining different consumption levels for each resource group.
- EDB Resource Manager throttles processes in a resource group to keep resource consumption near the limits defined by the resource type parameters. If there are multiple resource type parameters with defined settings in a resource group, the actual resource consumption may be significantly lower for certain resource types than their defined resource type parameter settings. This is because EDB Resource Manager throttles processes attempting to keep *all resources with defined resource type settings within their defined limits*.
- The definition of available resource groups and their resource type settings are stored in a shared global system catalog. Thus, resource groups can be utilized by all databases in a given Advanced Server instance.
- The `edb_max_resource_groups` configuration parameter sets the maximum number of resource groups that can be active simultaneously with running processes. The default setting is 16 resource groups. Changes to this parameter take effect on database server restart.
- Use the `SET edb_resource_group TO` *`group_name`* command to assign the current process to a specified resource group. Use the `RESET edb_resource_group` command or `SET edb_resource_group TO DEFAULT` to remove the current process from a resource group.

- A default resource group can be assigned to a role using the `ALTER ROLE ... SET` command, or to a database by the `ALTER DATABASE ... SET` command. The entire database server instance can be assigned a default resource group by setting the parameter in the `postgresql.conf` file.
- In order to include resource groups in a backup file of the database server instance, use the `pg_dumpall` backup utility with default settings (That is, do not specify any of the `--globals-only`, `--roles-only`, or `--tablespaces-only` options.)

5.1 Creating and Managing Resource Groups

The data definition language commands described in this section provide for the creation and management of resource groups.

5.1.1 CREATE RESOURCE GROUP

Use the `CREATE RESOURCE GROUP` command to create a new resource group.

```
CREATE RESOURCE GROUP group_name;
```

Description

The `CREATE RESOURCE GROUP` command creates a resource group with the specified name. Resource limits can then be defined on the group with the `ALTER RESOURCE GROUP` command. The resource group is accessible from all databases in the Advanced Server instance.

To use the `CREATE RESOURCE GROUP` command you must have superuser privileges.

Parameters

group_name

The name of the resource group.

Example

The following example results in the creation of three resource groups named `resgrp_a`, `resgrp_b`, and `resgrp_c`.

```
edb=# CREATE RESOURCE GROUP resgrp_a;
CREATE RESOURCE GROUP
edb=# CREATE RESOURCE GROUP resgrp_b;
CREATE RESOURCE GROUP
```

```
edb=# CREATE RESOURCE GROUP resgrp_c;
CREATE RESOURCE GROUP
```

The following query shows the entries for the resource groups in the `edb_resource_group` catalog.

```
edb=# SELECT * FROM edb_resource_group;
 rgrpname | rgrpcpuratelimit | rgrpdirtyratelimit
----------+------------------+--------------------
 resgrp_a |                0 |                  0
 resgrp_b |                0 |                  0
 resgrp_c |                0 |                  0
(3 rows)
```

5.1.2 ALTER RESOURCE GROUP

Use the `ALTER RESOURCE GROUP` command to change the attributes of an existing resource group. The command syntax comes in three forms.

The first form renames the resource group:

```
ALTER RESOURCE GROUP group_name RENAME TO new_name;
```

The second form assigns a resource type to the resource group:

```
ALTER RESOURCE GROUP group_name SET
  resource_type { TO | = } { value | DEFAULT };
```

The third form resets the assignment of a resource type to its default within the group:

```
ALTER RESOURCE GROUP group_name RESET resource_type;
```

Description

The `ALTER RESOURCE GROUP` command changes certain attributes of an existing resource group.

The first form with the `RENAME TO` clause assigns a new name to an existing resource group.

The second form with the `SET resource_type TO` clause either assigns the specified literal value to a resource type, or resets the resource type when `DEFAULT` is specified. Resetting or setting a resource type to `DEFAULT` means that the resource group has no defined limit on that resource type.

The third form with the `RESET resource_type` clause resets the resource type for the group as described previously.

To use the `ALTER RESOURCE GROUP` command you must have superuser privileges.

Parameters

group_name

The name of the resource group to be altered.

new_name

The new name to be assigned to the resource group.

resource_type

The resource type parameter specifying the type of resource to which a usage value is to be set.

value | DEFAULT

When *value* is specified, the literal value to be assigned to *resource_type*. When `DEFAULT` is specified, the assignment of *resource_type* is reset for the resource group.

Example

The following are examples of the `ALTER RESOURCE GROUP` command.

```
edb=# ALTER RESOURCE GROUP resgrp_a RENAME TO newgrp;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_b SET cpu_rate_limit = .5;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_b SET dirty_rate_limit = 6144;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_c RESET cpu_rate_limit;
ALTER RESOURCE GROUP
```

The following query shows the results of the `ALTER RESOURCE GROUP` commands to the entries in the `edb_resource_group` catalog.

```
edb=# SELECT * FROM edb_resource_group;
 rgrpname | rgrpcpuratelimit | rgrpdirtyratelimit
----------+------------------+--------------------
 newgrp   |                0 |                  0
 resgrp_b |              0.5 |               6144
 resgrp_c |                0 |                  0
(3 rows)
```


5.1.3 DROP RESOURCE GROUP

Use the `DROP RESOURCE GROUP` command to remove a resource group.

```
DROP RESOURCE GROUP [ IF EXISTS ] group_name;
```

Description

The `DROP RESOURCE GROUP` command removes a resource group with the specified name.

To use the `DROP RESOURCE GROUP` command you must have superuser privileges.

Parameters

`group_name`

The name of the resource group to be removed.

`IF EXISTS`

Do not throw an error if the resource group does not exist. A notice is issued in this case.

Example

The following example removes resource group `newgrp`.

```
edb=# DROP RESOURCE GROUP newgrp;
DROP RESOURCE GROUP
```

5.1.4 Assigning a Process to a Resource Group

Use the `SET edb_resource_group TO` *`group_name`* command to assign the current process to a specified resource group as shown by the following.

```
edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_b
(1 row)
```

The resource type settings of the group immediately take effect on the current process. If the command is used to change the resource group assigned to the current process, the resource type settings of the newly assigned group immediately take effect.

Processes can be included by default in a resource group by assigning a default resource group to roles, databases, or an entire database server instance.

A default resource group can be assigned to a role using the `ALTER ROLE ... SET` command. For more information about the `ALTER ROLE` command, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-alterrole.html

A default resource group can be assigned to a database by the `ALTER DATABASE ... SET` command. For more information about the `ALTER DATABASE` command, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-alterdatabase.html

The entire database server instance can be assigned a default resource group by setting the `edb_resource_group` configuration parameter in the `postgresql.conf` file as shown by the following.

```
# - EDB Resource Manager -
#edb_max_resource_groups = 16           # 0-65536 (change requires restart)
edb_resource_group = 'resgrp_b'
```

A change to `edb_resource_group` in the `postgresql.conf` file requires a configuration file reload before it takes effect on the database server instance.

5.1.5 Removing a Process from a Resource Group

Set `edb_resource_group` to `DEFAULT` or use `RESET edb_resource_group` to remove the current process from a resource group as shown by the following.

```
edb=# SET edb_resource_group TO DEFAULT;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------

(1 row)
```

For removing a default resource group from a role, use the `ALTER ROLE ... RESET` form of the `ALTER ROLE` command.

For removing a default resource group from a database, use the ALTER DATABASE ... RESET form of the ALTER DATABASE command.

For removing a default resource group from the database server instance, set the edb_resource_group configuration parameter to an empty string in the postgresql.conf file and reload the configuration file.

5.1.6 Monitoring Processes in Resource Groups

After resource groups have been created, the number of processes actively using these resource groups can be obtained from the view edb_all_resource_groups.

The columns in edb_all_resource_groups are the following:

- **group_name.** Name of the resource group.
- **active_processes.** Number of active processes in the resource group.
- **cpu_rate_limit.** The value of the CPU rate limit resource type assigned to the resource group.
- **per_process_cpu_rate_limit.** The CPU rate limit applicable to an individual, active process in the resource group.
- **dirty_rate_limit.** The value of the dirty rate limit resource type assigned to the resource group.
- **per_process_dirty_rate_limit.** The dirty rate limit applicable to an individual, active process in the resource group.

Note: Columns per_process_cpu_rate_limit and per_process_dirty_rate_limit do not show the *actual* resource consumption used by the processes, but indicate how EDB Resource Manager sets the resource limit for an individual process based upon the number of active processes in the resource group.

The following shows edb_all_resource_groups when resource group resgrp_a contains no active processes, resource group resgrp_b contains two active processes, and resource group resgrp_c contains one active process.

```
edb=# SELECT * FROM edb_all_resource_groups ORDER BY group_name;
-[ RECORD 1 ]----------------+------------------
 group_name                   | resgrp_a
 active_processes             | 0
 cpu_rate_limit               | 0.5
 per_process_cpu_rate_limit   |
 dirty_rate_limit             | 12288
 per_process_dirty_rate_limit |
-[ RECORD 2 ]----------------+------------------
 group_name                   | resgrp_b
 active_processes             | 2
 cpu_rate_limit               | 0.4
```

```
 per_process_cpu_rate_limit   | 0.195694289022895
 dirty_rate_limit             | 6144
 per_process_dirty_rate_limit | 3785.92924684337
-[ RECORD 3 ]----------------+------------------
 group_name                   | resgrp_c
 active_processes             | 1
 cpu_rate_limit               | 0.3
 per_process_cpu_rate_limit   | 0.292342129631091
 dirty_rate_limit             | 3072
 per_process_dirty_rate_limit | 3072
```

The CPU rate limit and dirty rate limit settings that are assigned to these resource groups are as follows.

```
edb=# SELECT * FROM edb_resource_group;
 rgrpname | rgrpcpuratelimit | rgrpdirtyratelimit
----------+------------------+--------------------
 resgrp_a |              0.5 |              12288
 resgrp_b |              0.4 |               6144
 resgrp_c |              0.3 |               3072
(3 rows)
```

In the `edb_all_resource_groups` view, note that the `per_process_cpu_rate_limit` and `per_process_dirty_rate_limit` values are roughly the corresponding CPU rate limit and dirty rate limit divided by the number of active processes.

5.2 CPU Usage Throttling

CPU usage of a resource group is controlled by setting the `cpu_rate_limit` resource type parameter.

Set the `cpu_rate_limit` parameter to the fraction of CPU time over wall-clock time to which the combined, simultaneous CPU usage of all processes in the group should not exceed. Thus, the value assigned to `cpu_rate_limit` should typically be less than or equal to 1.

The valid range of the `cpu_rate_limit` parameter is 0 to 1.67772e+07. A setting of 0 means no CPU rate limit has been set for the resource group.

When multiplied by 100, the `cpu_rate_limit` can also be interpreted as the CPU usage percentage for a resource group.

EDB Resource Manager utilizes *CPU throttling* to keep the aggregate CPU usage of all processes in the group within the limit specified by the `cpu_rate_limit` parameter. A process in the group may be interrupted and put into sleep mode for a short interval of time to maintain the defined limit. When and how such interruptions occur is defined by a proprietary algorithm used by EDB Resource Manager.

5.2.1 Setting the CPU Rate Limit for a Resource Group

The `ALTER RESOURCE GROUP` command with the `SET cpu_rate_limit` clause is used to set the CPU rate limit for a resource group.

In the following example the CPU usage limit is set to 50% for `resgrp_a`, 40% for `resgrp_b` and 30% for `resgrp_c`. This means that the combined CPU usage of all processes assigned to `resgrp_a` is maintained at approximately 50%. Similarly, for all processes in `resgrp_b`, the combined CPU usage is kept to approximately 40%, etc.

```
edb=# ALTER RESOURCE GROUP resgrp_a SET cpu_rate_limit TO .5;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_b SET cpu_rate_limit TO .4;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_c SET cpu_rate_limit TO .3;
ALTER RESOURCE GROUP
```

The following query shows the settings of `cpu_rate_limit` in the catalog.

```
edb=# SELECT rgrpname, rgrpcpuratelimit FROM edb_resource_group;
 rgrpname | rgrpcpuratelimit
----------+------------------
 resgrp_a |              0.5
 resgrp_b |              0.4
```

```
 resgrp_c |                 0.3
(3 rows)
```

Changing the `cpu_rate_limit` of a resource group not only affects new processes that are assigned to the group, but any currently running processes that are members of the group are immediately affected by the change. That is, if the `cpu_rate_limit` is changed from .5 to .3, currently running processes in the group would be throttled downward so that the aggregate group CPU usage would be near 30% instead of 50%.

To illustrate the effect of setting the CPU rate limit for resource groups, the following examples use a CPU-intensive calculation of 20000 factorial (multiplication of 20000 * 19999 * 19998, etc.) performed by the query `SELECT 20000!;` run in the `psql` command line utility.

The resource groups with the CPU rate limit settings shown in the previous query are used in these examples.

5.2.2 Example – Single Process in a Single Group

The following shows that the current process is set to use resource group `resgrp_b`. The factorial calculation is then started.

```
edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_b
(1 row)
edb=# SELECT 20000!;
```

In a second session, the Linux `top` command is used to display the CPU usage as shown under the `%CPU` column. The following is a snapshot at an arbitrary point in time as the `top` command output periodically changes.

```
$ top
top - 16:37:03 up  4:15,  7 users,  load average: 0.49, 0.20, 0.38
Tasks: 202 total,   1 running, 201 sleeping,   0 stopped,   0 zombie
Cpu(s): 42.7%us,  2.3%sy,  0.0%ni, 55.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0
Mem:   1025624k total,   791160k used,   234464k free,    23400k buffers
Swap:   103420k total,    13404k used,    90016k free,   373504k cached

  PID USER       PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
28915 enterpri   20   0  195m 5900 4212 S 39.9  0.6   3:36.98 edb-postgres
 1033 root       20   0  171m  77m 2960 S  1.0  7.8   3:43.96 Xorg
 3040 user       20   0  278m  22m  14m S  1.0  2.2   3:41.72 knotify4
    .
    .
    .
```

The `psql` session performing the factorial calculation is shown by the row where `edb-postgres` appears under the `COMMAND` column. The CPU usage of the session shown under the %CPU column shows 39.9, which is close to the 40% CPU limit set for resource group resgrp_b.

By contrast, if the psql session is removed from the resource group and the factorial calculation is performed again, the CPU usage is much higher.

```
edb=# SET edb_resource_group TO DEFAULT;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------

(1 row)

edb=# SELECT 20000!;
```

Under the %CPU column for edb-postgres, the CPU usage is now 93.6, which is significantly higher than the 39.9 when the process was part of the resource group.

```
$ top
top - 16:43:03 up  4:21,  7 users,  load average: 0.66, 0.33, 0.37
Tasks: 202 total,   5 running, 197 sleeping,   0 stopped,   0 zombie
Cpu(s): 96.7%us,  3.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0
Mem:   1025624k total,   791228k used,   234396k free,    23560k buffers
Swap:   103420k total,    13404k used,    90016k free,   373508k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
28915 enterpri  20   0  195m 5900 4212 R 93.6  0.6   5:01.56 edb-postgres
 1033 root      20   0  171m  77m 2960 S  1.0  7.8   3:48.15 Xorg
 2907 user      20   0 98.7m  11m 9100 S  0.3  1.2   0:46.51 vmware-user-lo
    .
    .
    .
```

5.2.3 Example – Multiple Processes in a Single Group

As stated previously, the CPU rate limit applies to the aggregate of all processes in the resource group. This concept is illustrated in the following example.

The factorial calculation is performed simultaneously in two separate psql sessions, each of which has been added to resource group resgrp_b that has cpu_rate_limit set to .4 (CPU usage of 40%).

Session 1:

```
edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_b
(1 row)

edb=# SELECT 20000!;
```

Session 2:

```
edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_b
(1 row)

edb=# SELECT 20000!;
```

A third session monitors the CPU usage.

```
$ top
top - 16:53:03 up  4:31,  7 users,  load average: 0.31, 0.19, 0.27
Tasks: 202 total,   1 running, 201 sleeping,   0 stopped,   0 zombie
Cpu(s): 41.2%us,  3.0%sy,  0.0%ni, 55.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0
Mem:   1025624k total,   792020k used,   233604k free,    23844k buffers
Swap:   103420k total,    13404k used,    90016k free,   373508k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
29857 enterpri  20   0  195m 4708 3312 S 19.9  0.5   0:57.35 edb-postgres
28915 enterpri  20   0  195m 5900 4212 S 19.6  0.6   5:35.49 edb-postgres
 3040 user      20   0  278m  22m  14m S  1.0  2.2   3:54.99 knotify4
 1033 root      20   0  171m  78m 2960 S  0.3  7.8   3:55.71 Xorg
    .
    .
    .
```

There are now two processes named `edb-postgres` with `%CPU` values of 19.9 and 19.6, whose sum is close to the 40% CPU usage set for resource group `resgrp_b`.

The following command sequence displays the sum of all `edb-postgres` processes sampled over half second time intervals. This shows how the total CPU usage of the processes in the resource group changes over time as EDB Resource Manager throttles the processes to keep the total resource group CPU usage near 40%.

```
$ while [[ 1 -eq 1 ]]; do  top -d0.5 -b -n2 | grep edb-postgres | awk '{ SUM
+= $9} END { print SUM / 2 }'; done
37.2
39.1
38.9
38.3
44.7
39.2
42.5
39.1
39.2
39.2
41
42.85
46.1
    .
    .
    .
```

5.2.4 Example – Multiple Processes in Multiple Groups

In this example, two additional `psql` sessions are used along with the previous two sessions. The third and fourth sessions perform the same factorial calculation within resource group `resgrp_c` with a `cpu_rate_limit` of `.3` (30% CPU usage).

Session 3:

```
edb=# SET edb_resource_group TO resgrp_c;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_c
(1 row)

edb=# SELECT 20000!;
```

Session 4:

```
edb=# SET edb_resource_group TO resgrp_c;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_c
(1 row)

edb=# SELECT 20000!;
```

The `top` command displays the following output.

```
$ top
top - 17:45:09 up  5:23,  8 users,  load average: 0.47, 0.17, 0.26
Tasks: 203 total,   4 running, 199 sleeping,   0 stopped,   0 zombie
Cpu(s): 70.2%us,  0.0%sy,  0.0%ni, 29.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0
Mem:   1025624k total,   806140k used,   219484k free,    25296k buffers
Swap:   103420k total,    13404k used,    90016k free,   374092k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
29857 enterpri  20   0  195m 4820 3324 S 19.9  0.5   4:25.02 edb-postgres
28915 enterpri  20   0  195m 5900 4212 R 19.6  0.6   9:07.50 edb-postgres
29023 enterpri  20   0  195m 4744 3248 R 16.3  0.5   4:01.73 edb-postgres
11019 enterpri  20   0  195m 4120 2764 R 15.3  0.4   0:04.92 edb-postgres
 2907 user      20   0 98.7m  12m 9112 S  1.3  1.2   0:56.54 vmware-user-lo
 3040 user      20   0  278m  22m  14m S  1.3  2.2   4:38.73 knotify4
```

The two resource groups in use have CPU usage limits of 40% and 30%. The sum of the `%CPU` column for the first two `edb-postgres` processes is 39.5 (approximately 40%, which is the limit for `resgrp_b`) and the sum of the `%CPU` column for the third and fourth `edb-postgres` processes is 31.6 (approximately 30%, which is the limit for `resgrp_c`).

The sum of the CPU usage limits of the two resource groups to which these processes belong is 70%. The following output shows that the sum of the four processes borders around 70%.

```
$ while [[ 1 -eq 1 ]]; do  top -d0.5 -b -n2 | grep edb-postgres | awk '{ SUM
+= $9} END { print SUM / 2 }'; done
61.8
76.4
72.6
69.55
64.55
79.95
68.55
71.25
74.85
62
74.85
76.9
72.4
65.9
74.9
68.25
```

By contrast, if three sessions are processing where two sessions remain in `resgrp_b`, but the third session does not belong to any resource group, the `top` command shows the following output.

```
$ top
top - 17:24:55 up  5:03,  7 users,  load average: 1.00, 0.41, 0.38
Tasks: 199 total,   3 running, 196 sleeping,   0 stopped,   0 zombie
Cpu(s): 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0
Mem:   1025624k total,   797692k used,   227932k free,    24724k buffers
Swap:   103420k total,    13404k used,    90016k free,   374068k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
29023 enterpri  20   0  195m 4744 3248 R 58.6  0.5   2:53.75 edb-postgres
28915 enterpri  20   0  195m 5900 4212 S 18.9  0.6   7:58.45 edb-postgres
29857 enterpri  20   0  195m 4820 3324 S 18.9  0.5   3:14.85 edb-postgres
 1033 root      20   0  174m  81m 2960 S  1.7  8.2   4:26.50 Xorg
 3040 user      20   0  278m  22m  14m S  1.0  2.2   4:21.20 knotify4
```

The second and third `edb-postgres` processes belonging to the resource group where the CPU usage is limited to 40%, have a total CPU usage of 37.8. However, the first `edb-postgres` process has a 58.6% CPU usage as it is not within a resource group, and basically utilizes the remaining, available CPU resources on the system.

Likewise, the following output shows the sum of all three sessions is around 95% since one of the sessions has no set limit on its CPU usage.

```
$ while [[ 1 -eq 1 ]]; do  top -d0.5 -b -n2 | grep edb-postgres | awk '{ SUM
+= $9} END { print SUM / 2 }'; done
96
90.35
92.55
96.4
94.1
```

```
90.7
95.7
95.45
93.65
87.95
96.75
94.25
95.45
97.35
92.9
96.05
96.25
94.95
    .
    .
    .
```

5.3 Dirty Buffer Throttling

Writing to shared buffers is controlled by setting the `dirty_rate_limit` resource type parameter.

Set the `dirty_rate_limit` parameter to the number of kilobytes per second for the combined rate at which all the processes in the group should write to or "dirty" the shared buffers. An example setting would be 3072 kilobytes per seconds.

The valid range of the `dirty_rate_limit` parameter is 0 to 1.67772e+07. A setting of 0 means no dirty rate limit has been set for the resource group.

EDB Resource Manager utilizes *dirty buffer throttling* to keep the aggregate, shared buffer writing rate of all processes in the group near the limit specified by the `dirty_rate_limit` parameter. A process in the group may be interrupted and put into sleep mode for a short interval of time to maintain the defined limit. When and how such interruptions occur is defined by a proprietary algorithm used by EDB Resource Manager.

5.3.1 Setting the Dirty Rate Limit for a Resource Group

The `ALTER RESOURCE GROUP` command with the `SET dirty_rate_limit` clause is used to set the dirty rate limit for a resource group.

In the following example the dirty rate limit is set to 12288 kilobytes per second for `resgrp_a`, 6144 kilobytes per second for `resgrp_b` and 3072 kilobytes per second for `resgrp_c`. This means that the combined writing rate to the shared buffer of all processes assigned to `resgrp_a` is maintained at approximately 12288 kilobytes per second. Similarly, for all processes in `resgrp_b`, the combined writing rate to the shared buffer is kept to approximately 6144 kilobytes per second, etc.

```
edb=# ALTER RESOURCE GROUP resgrp_a SET dirty_rate_limit TO 12288;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_b SET dirty_rate_limit TO 6144;
ALTER RESOURCE GROUP
edb=# ALTER RESOURCE GROUP resgrp_c SET dirty_rate_limit TO 3072;
ALTER RESOURCE GROUP
```

The following query shows the settings of `dirty_rate_limit` in the catalog.

```
edb=# SELECT rgrpname, rgrpdirtyratelimit FROM edb_resource_group;
 rgrpname | rgrpdirtyratelimit
----------+--------------------
 resgrp_a |              12288
 resgrp_b |               6144
 resgrp_c |               3072
(3 rows)
```

Changing the dirty_rate_limit of a resource group not only affects new processes that are assigned to the group, but any currently running processes that are members of the group are immediately affected by the change. That is, if the dirty_rate_limit is changed from 12288 to 3072, currently running processes in the group would be throttled downward so that the aggregate group dirty rate would be near 3072 kilobytes per second instead of 12288 kilobytes per second.

To illustrate the effect of setting the dirty rate limit for resource groups, the following examples use the following table for intensive I/O operations.

```
CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
```

The FILLFACTOR = 10 clause results in INSERT commands packing rows up to only 10% per page. This results in a larger sampling of dirty shared blocks for the purpose of these examples.

The pg_stat_statements module is used to display the number of shared buffer blocks that are dirtied by a SQL command and the amount of time the command took to execute. This provides the information to calculate the actual kilobytes per second writing rate for the SQL command, and thus compare it to the dirty rate limit set for a resource group.

In order to use the pg_stat_statements module, perform the following steps.

Step 1: In the postgresql.conf file, add $libdir/pg_stat_statements to the shared_preload_libraries configuration parameter as shown by the following.

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/pg_stat_statements'
```

Step 2: Restart the database server.

Step 3: Use the CREATE EXTENSION command to complete the creation of the pg_stat_statements module.

```
edb=# CREATE EXTENSION pg_stat_statements SCHEMA public;
CREATE EXTENSION
```

The pg_stat_statements_reset() function is used to clear out the pg_stat_statements view for clarity of each example.

The resource groups with the dirty rate limit settings shown in the previous query are used in these examples.

5.3.2 Example – Single Process in a Single Group

The following sequence of commands shows the creation of table `t1`. The current process is set to use resource group `resgrp_b`. The `pg_stat_statements` view is cleared out by running the `pg_stat_statements_reset()` function.

Finally, the `INSERT` command generates a series of integers from 1 to 10,000 to populate the table, and dirty approximately 10,000 blocks.

```
edb=# CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_b
(1 row)

edb=# SELECT pg_stat_statements_reset();
 pg_stat_statements_reset
--------------------------

(1 row)

edb=# INSERT INTO t1 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

The following shows the results from the `INSERT` command.

```
edb=# SELECT query, rows, total_time, shared_blks_dirtied FROM
pg_stat_statements;
-[ RECORD 1 ]-------+--------------------------------------------------
 query               | INSERT INTO t1 VALUES (generate_series (?,?), ?);
 rows                | 10000
 total_time          | 13496.184
 shared_blks_dirtied | 10003
```

The actual dirty rate is calculated as follows.

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 13496.184 ms, which yields *0.74117247 blocks per millisecond*.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields *741.17247 blocks per second*.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately *6072 kilobytes per second*.

Note that the actual dirty rate of 6072 kilobytes per second is close to the dirty rate limit for the resource group, which is 6144 kilobytes per second.

By contrast, if the steps are repeated again without the process belonging to any resource group, the dirty buffer rate is much higher.

```
edb=# CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------

(1 row)

edb=# SELECT pg_stat_statements_reset();
 pg_stat_statements_reset
--------------------------

(1 row)

edb=# INSERT INTO t1 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

The following shows the results from the `INSERT` command without the usage of a resource group.

```
edb=# SELECT query, rows, total_time, shared_blks_dirtied FROM
pg_stat_statements;
-[ RECORD 1 ]-------+--------------------------------------------------
 query               | INSERT INTO t1 VALUES (generate_series (?,?), ?);
 rows                | 10000
 total_time          | 2432.165
 shared_blks_dirtied | 10003
```

First, note the total time was only 2432.165 milliseconds as compared to 13496.184 milliseconds when a resource group with a dirty rate limit set to 6144 kilobytes per second was used.

The actual dirty rate without the use of a resource group is calculated as follows.

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 2432.165 ms, which yields *4.112797 blocks per millisecond*.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields *4112.797 blocks per second*.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately *33692 kilobytes per second*.

Note that the actual dirty rate of 33692 kilobytes per second is significantly higher than when the resource group with a dirty rate limit of 6144 kilobytes per second was used.

5.3.3 Example – Multiple Processes in a Single Group

As stated previously, the dirty rate limit applies to the aggregate of all processes in the resource group. This concept is illustrated in the following example.

For this example the inserts are performed simultaneously on two different tables in two separate `psql` sessions, each of which has been added to resource group `resgrp_b` that has a `dirty_rate_limit` set to 6144 kilobytes per second.

Session 1:

```
edb=# CREATE TABLE t1 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_b
(1 row)

edb=# INSERT INTO t1 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

Session 2:

```
edb=# CREATE TABLE t2 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SET edb_resource_group TO resgrp_b;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_b
(1 row)

edb=# SELECT pg_stat_statements_reset();
 pg_stat_statements_reset
--------------------------

(1 row)

edb=# INSERT INTO t2 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

Note: The `INSERT` commands in session 1 and session 2 were started after the `SELECT pg_stat_statements_reset()` command in session 2 was run.

The following shows the results from the `INSERT` commands in the two sessions. `RECORD 3` shows the results from session 1. `RECORD 2` shows the results from session 2.

```
edb=# SELECT query, rows, total_time, shared_blks_dirtied FROM
pg_stat_statements;
-[ RECORD 1 ]-------+--------------------------------------------------
 query              | SELECT pg_stat_statements_reset();
 rows               | 1
 total_time         | 0.43
 shared_blks_dirtied | 0
-[ RECORD 2 ]-------+--------------------------------------------------
 query              | INSERT INTO t2 VALUES (generate_series (?,?), ?);
 rows               | 10000
 total_time         | 30591.551
 shared_blks_dirtied | 10003
-[ RECORD 3 ]-------+--------------------------------------------------
 query              | INSERT INTO t1 VALUES (generate_series (?,?), ?);
 rows               | 10000
 total_time         | 33215.334
```



```
 shared_blks_dirtied | 10003
```

First, note the total time was 33215.334 milliseconds for session 1 and 30591.551 milliseconds for session 2. When only one session was active in the same resource group as shown in the first example, the time was 13496.184 milliseconds. Thus more active processes in the resource group result in a slower dirty rate for each active process in the group. This is shown in the following calculations.

The actual dirty rate for session 1 is calculated as follows.

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 33215.334 ms, which yields *0.30115609 blocks per millisecond*.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields *301.15609 blocks per second*.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately *2467 kilobytes per second*.

The actual dirty rate for session 2 is calculated as follows.

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 30591.551 ms, which yields *0.32698571 blocks per millisecond*.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields *326.98571 blocks per second*.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately *2679 kilobytes per second*.

The combined dirty rate from session 1 (2467 kilobytes per second) and from session 2 (2679 kilobytes per second) yields 5146 kilobytes per second, which is below the set dirty rate limit of the resource group (6144 kilobytes per seconds).

5.3.4 Example – Multiple Processes in Multiple Groups

In this example, two additional `psql` sessions are used along with the previous two sessions. The third and fourth sessions perform the same `INSERT` command in resource group `resgrp_c` with a `dirty_rate_limit` of 3072 kilobytes per second.

Sessions 1 and 2 are repeated as illustrated in the prior example using resource group `resgrp_b`. with a `dirty_rate_limit` of 6144 kilobytes per second.

Session 3:

```
edb=# CREATE TABLE t3 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SET edb_resource_group TO resgrp_c;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
```

```
--------------------
 resgrp_c
(1 row)

edb=# INSERT INTO t3 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

Session 4:

```
edb=# CREATE TABLE t4 (c1 INTEGER, c2 CHARACTER(500)) WITH (FILLFACTOR = 10);
CREATE TABLE
edb=# SET edb_resource_group TO resgrp_c;
SET
edb=# SHOW edb_resource_group;
 edb_resource_group
--------------------
 resgrp_c
(1 row)

edb=# SELECT pg_stat_statements_reset();
 pg_stat_statements_reset
--------------------------

(1 row)

edb=# INSERT INTO t4 VALUES (generate_series (1,10000), 'aaa');
INSERT 0 10000
```

Note: The `INSERT` commands in all four sessions were started after the `SELECT pg_stat_statements_reset()` command in session 4 was run.

The following shows the results from the `INSERT` commands in the four sessions. `RECORD 3` shows the results from session 1. `RECORD 2` shows the results from session 2. `RECORD 4` shows the results from session 3. `RECORD 5` shows the results from session 4.

```
edb=# SELECT query, rows, total_time, shared_blks_dirtied FROM
pg_stat_statements;
-[ RECORD 1 ]-------+--------------------------------------------------
 query              | SELECT pg_stat_statements_reset();
 rows               | 1
 total_time         | 0.467
 shared_blks_dirtied | 0
-[ RECORD 2 ]-------+--------------------------------------------------
 query              | INSERT INTO t2 VALUES (generate_series (?,?), ?);
 rows               | 10000
 total_time         | 31343.458
 shared_blks_dirtied | 10003
-[ RECORD 3 ]-------+--------------------------------------------------
 query              | INSERT INTO t1 VALUES (generate_series (?,?), ?);
 rows               | 10000
 total_time         | 28407.435
 shared_blks_dirtied | 10003
-[ RECORD 4 ]-------+--------------------------------------------------
 query              | INSERT INTO t3 VALUES (generate_series (?,?), ?);
 rows               | 10000
 total_time         | 52727.846
 shared_blks_dirtied | 10003
-[ RECORD 5 ]-------+--------------------------------------------------
```



```
query              | INSERT INTO t4 VALUES (generate_series (?,?), ?);
rows               | 10000
total_time         | 56063.697
shared_blks_dirtied | 10003
```

First note that the times of session 1 (28407.435) and session 2 (31343.458) are close to each other as they are both in the same resource group with `dirty_rate_limit` set to 6144, as compared to the times of session 3 (52727.846) and session 4 (56063.697), which are in the resource group with `dirty_rate_limit` set to 3072. The latter group has a slower dirty rate limit so the expected processing time is longer as is the case for sessions 3 and 4.

The actual dirty rate for session 1 is calculated as follows.

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 28407.435 ms, which yields *0.35212612 blocks per millisecond*.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields *352.12612 blocks per second*.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately *2885 kilobytes per second*.

The actual dirty rate for session 2 is calculated as follows.

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 31343.458 ms, which yields *0.31914156 blocks per millisecond*.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields *319.14156 blocks per second*.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately *2614 kilobytes per second*.

The combined dirty rate from session 1 (2885 kilobytes per second) and from session 2 (2614 kilobytes per second) yields 5499 kilobytes per second, which is near the set dirty rate limit of the resource group (6144 kilobytes per seconds).

The actual dirty rate for session 3 is calculated as follows.

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 52727.846 ms, which yields *0.18971001 blocks per millisecond*.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields *189.71001 blocks per second*.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately *1554 kilobytes per second*.

The actual dirty rate for session 4 is calculated as follows.

- The number of blocks dirtied per millisecond (ms) is 10003 blocks / 56063.697 ms, which yields *0.17842205 blocks per millisecond*.
- Multiply the result by 1000 to give the number of shared blocks dirtied per second (1 second = 1000 ms), which yields *178.42205 blocks per second*.
- Multiply the result by 8.192 to give the number of kilobytes dirtied per second (1 block = 8.192 kilobytes), which yields approximately *1462 kilobytes per second*.

The combined dirty rate from session 3 (1554 kilobytes per second) and from session 4 (1462 kilobytes per second) yields 3016 kilobytes per second, which is near the set dirty rate limit of the resource group (3072 kilobytes per seconds).

Thus, this demonstrates how EDB Resource Manager keeps the aggregate dirty rate of the active processes in its groups close to the dirty rate limit set for each group.

5.4 System Catalogs

This section describes the system catalogs that store the resource group information used by EDB Resource Manager.

5.4.1 edb_all_resource_groups

The following table lists the information available in the `edb_all_resource_groups` catalog:

Column	Type	Description
`group_name`	`name`	The name of the resource group.
`active_processes`	`integer`	Number of currently active processes in the resource group.
`cpu_rate_limit`	`float8`	Maximum CPU rate limit for the resource group. `0` means no limit.
`per_process_cpu_rate_limit`	`float8`	Maximum CPU rate limit per currently active process in the resource group.
`dirty_rate_limit`	`float8`	Maximum dirty rate limit for a resource group. `0` means no limit.
`per_process_dirty_rate_limit`	`float8`	Maximum dirty rate limit per currently active process in the resource group.

5.4.2 edb_resource_group

The following table lists the information available in the `edb_resource_group` catalog:

Column	Type	Description
`rgrpname`	`name`	The name of the resource group.
`rgrpcpuratelimit`	`float8`	Maximum CPU rate limit for a resource group. `0` means no limit.
`rgrpdirtyratelimit`	`float8`	Maximum dirty rate limit for a resource group. `0` means no limit.

6 Database Utilities

This chapter describes various database utilities that provide many usage benefits with Postgres Plus Advanced Server.

6.1 EDB*Loader

EDB*Loader is a high-performance bulk data loader that provides an Oracle compatible interface for Postgres Plus Advanced Server. The EDB*Loader command line utility loads data from an input source, typically a file, into one or more tables using a subset of the parameters offered by Oracle SQL*Loader.

EDB*Loader features include:

- Support for the Oracle SQL*Loader data loading methods - conventional path load, direct path load, and parallel direct path load
- Oracle SQL*Loader compatible syntax for control file directives
- Input data with delimiter-separated or fixed-width fields
- Bad file for collecting rejected records
- Loading of multiple target tables
- Discard file for collecting records that do not meet the selection criteria of any target table
- Log file for recording the EDB*Loader session and any error messages
- Data loading from standard input and remote loading, particularly useful for large data sources on remote hosts

These features are explained in detail in the following sections.

Note: The following are important version compatibility restrictions between the EDB*Loader client and the database server.

- Invoking EDB*Loader is done using a client program called `edbldr`, which is used to pass parameters and directive information to the database server. **It is strongly recommended that the 9.5 EDB*Loader client (that is, the edbldr program supplied with Postgres Plus Advanced Server 9.5) be used to load data only into version 9.5 of the database server. In general, the EDB*Loader client and database server should be the same version.**
- It is possible to use a 9.5 EDB*Loader client to load data into a 9.5 database server, but the new 9.5 EDB*Loader features may not be available under those circumstances.
- Use of a 9.5, 9.4 or 9.3 EDB*Loader client is not supported for database servers version 9.2 or earlier.

6.1.1 Data Loading Methods

As with Oracle SQL*Loader, EDB*Loader supports three data loading methods:

- Conventional path load
- Direct path load
- Parallel direct path load

Conventional path load is the default method used by EDB*Loader. Basic insert processing is used to add rows to the table.

The advantage of a conventional path load over the other methods is that table constraints and database objects defined on the table such as primary keys, not null constraints, check constraints, unique indexes, foreign key constraints, and triggers are enforced during a conventional path load.

One exception is that Postgres Plus Advanced Server *rules* defined on the table are not enforced. EDB*Loader can load tables on which rules are defined, but the rules are not executed. As a consequence, partitioned tables implemented using rules cannot be loaded using EDB*Loader.

Note: Postgres Plus Advanced Server rules are created by the `CREATE RULE` command. Postgres Plus Advanced Server rules are not the same database objects as rules and rule sets used in Oracle.

EDB*Loader also supports direct path loads. A direct path load is faster than a conventional path load, but requires the removal of most types of constraints and triggers from the table. See Section 6.1.5 for information on direct path loads.

Finally, EDB*Loader supports parallel direct path loads. A parallel direct path load provides even greater performance improvement by permitting multiple EDB*Loader sessions to run simultaneously to load a single table. See Section 6.1.6 for information on parallel direct path loads.

6.1.2 General Usage

EDB*Loader can load data files with either delimiter-separated or fixed-width fields, in single-byte or multi-byte character sets. The delimiter can be a string consisting of one or more single-byte or multi-byte characters. Data file encoding and the database encoding may be different. Character set conversion of the data file to the database encoding is supported.

Each EDB*Loader session runs as a single, independent transaction. If an error should occur during the EDB*Loader session that aborts the transaction, all changes made during the session are rolled back.

Generally, formatting errors in the data file do not result in an aborted transaction. Instead, the badly formatted records are written to a text file called the *bad file*. The reason for the error is recorded in the *log file*.

Records causing database integrity errors do result in an aborted transaction and rollback. As with formatting errors, the record causing the error is written to the bad file and the reason is recorded in the log file.

Note: EDB*Loader differs from Oracle SQL*Loader in that a database integrity error results in a rollback in EDB*Loader. In Oracle SQL*Loader, only the record causing the error is rejected. Records that were previously inserted into the table are retained and loading continues after the rejected record.

The following are examples of types of formatting errors that do not abort the transaction:

- Attempt to load non-numeric value into a numeric column
- Numeric value is too large for a numeric column
- Character value is too long for the maximum length of a character column
- Attempt to load improperly formatted date value into a date column

The following are examples of types of database errors that abort the transaction and result in the rollback of all changes made in the EDB*Loader session:

- Violation of a unique constraint such as a primary key or unique index
- Violation of a referential integrity constraint
- Violation of a check constraint
- Error thrown by a trigger fired as a result of inserting rows

6.1.3 Building the EDB*Loader Control File

When you invoke EDB*Loader, the list of arguments provided must include the name of a control file. The control file includes the instructions that EDB*Loader uses to load the table (or tables) from the input data file. The control file includes information such as:

- The name of the input data file containing the data to be loaded.
- The name of the table or tables to be loaded from the data file.
- Names of the columns within the table or tables and their corresponding field placement in the data file.
- Specification of whether the data file uses a delimiter string to separate the fields, or if the fields occupy fixed column positions.
- Optional selection criteria to choose which records from the data file to load into a given table.
- The name of the file that will collect illegally formatted records.
- The name of the discard file that will collect records that do not meet the selection criteria of any table.

The syntax for the EDB*Loader control file is as follows:

```
[ OPTIONS (param=value [, param=value ] ...) ]
LOAD DATA
  [ CHARACTERSET charset ]
  [ INFILE '{ data_file | stdin }' ]
  [ BADFILE 'bad_file' ]
  [ DISCARDFILE 'discard_file' ]
  [ { DISCARDMAX | DISCARDS } max_discard_recs ]
[ INSERT | APPEND | REPLACE | TRUNCATE ]
[ PRESERVE BLANKS ]
{ INTO TABLE target_table
  [ WHEN field_condition [ AND field_condition ] ...]
  [ FIELDS TERMINATED BY 'termstring'
    [ OPTIONALLY ENCLOSED BY 'enclstring' ] ]
  [ TRAILING NULLCOLS ]
   (field_def [, field_def ] ...)
} ...
```

where `field_def` defines a *field* in the specified `data_file` that describes the location, data format, or value of the data to be inserted into `column_name` of the `target_table`. The syntax of `field_def` is the following:

```
column_name {
  CONSTANT val |
  FILLER [ POSITION (start:end) ] [ fieldtype ] |
  [ POSITION (start:end) ] [ fieldtype ]
```

```
    [ PRESERVE BLANKS ] [ "expr" ]
  }
```

where *fieldtype* is one of:

```
  CHAR | DATE [ "datemask" ] | INTEGER EXTERNAL |
  FLOAT EXTERNAL | DECIMAL EXTERNAL | ZONED EXTERNAL |
  ZONED [(precision[,scale])]
```

Description

The specification of *data_file*, *bad_file*, and *discard_file* may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is then assumed to exist (in the case of *data_file*), or is created (in the case of *bad_file* or *discard_file*), relative to the current working directory from which `edbldr` is invoked.

You can include references to environment variables within the EDB*Loader control file when referring to a directory path and/or file name. Environment variable references are formatted differently on Windows systems than on Linux systems:

- On Linux, the format is `$`*ENV_VARIABLE* or `${`*ENV_VARIABLE*`}`
- On Windows, the format is `%`*ENV_VARIABLE*`%`

Where *ENV_VARIABLE* is the environment variable that is set to the directory path and/or file name.

The `EDBLDR_ENV_STYLE` environment variable instructs Advanced Server to interpret environment variable references as Windows-styled references or Linux-styled references irregardless of the operating system on which EDB*Loader resides. You can use this environment variable to create portable control files for EDB*Loader.

- On a Windows system, set `EDBLDR_ENV_STYLE` to `linux` or `unix` to instruct Advanced Server to recognize Linux-style references within the control file.
- On a Linux system, set `EDBLDR_ENV_STYLE` to `windows` to instruct Advanced Server to recognize Windows-style references within the control file.

The operating system account `enterprisedb` must have read permission on the directory and file specified by *data_file*.

The operating system account `enterprisedb` must have write permission on the directories where *bad_file* and *discard_file* are to be written.

Note: It is suggested that the file names for *data_file*, *bad_file*, and *discard_file* include extensions of .dat, .bad, and .dsc, respectively. If the provided file name does not contain an extension, EDB*Loader assumes the actual file name includes the appropriate aforementioned extension.

If an EDB*Loader session results in data format errors and the BADFILE clause is not specified, nor is the BAD parameter given on the command line when edbldr is invoked, a bad file is created with the name *control_file_base*.bad in the current working directory from which edbldr is invoked. *control_file_base* is the base name of the control file (that is, the file name without any extension) used in the edbldr session.

If all of the following conditions are true, the discard file is not created even if the EDB*Loader session results in discarded records:

- The DISCARDFILE clause for specifying the discard file is not included in the control file.
- The DISCARD parameter for specifying the discard file is not included on the command line.
- The DISCARDMAX clause for specifying the maximum number of discarded records is not included in the control file.
- The DISCARDS clause for specifying the maximum number of discarded records is not included in the control file.
- The DISCARDMAX parameter for specifying the maximum number of discarded records is not included on the command line.

If neither the DISCARDFILE clause nor the DISCARD parameter for explicitly specifying the discard file name are specified, but DISCARDMAX or DISCARDS is specified, then the EDB*Loader session creates a discard file using the data file name with an extension of .dsc.

Note: There is a distinction between keywords DISCARD and DISCARDS. DISCARD is an EDB*Loader command line parameter used to specify the discard file name (see Section 6.1.4). DISCARDS is a clause of the LOAD DATA directive that may only appear in the control file. Keywords DISCARDS and DISCARDMAX provide the same functionality of specifying the maximum number of discarded records allowed before terminating the EDB*Loader session. Records loaded into the database before termination of the EDB*Loader session due to exceeding the DISCARDS or DISCARDMAX settings are kept in the database and are not rolled back.

If one of INSERT, APPEND, REPLACE, or TRUNCATE is specified, it establishes the default action of how rows are to be added to target tables. If omitted, the default action is as if INSERT had been specified.

If the `FIELDS TERMINATED BY` clause is specified, then the `POSITION (`*`start`*`:`*`end`*`)` clause may not be specified for any *`field_def`*. Alternatively if the `FIELDS TERMINATED BY` clause is not specified, then every *`field_def`* must contain the `POSITION (`*`start`*`:`*`end`*`)` clause, excluding those with the `CONSTANT` clause.

Parameters

`OPTIONS param=value`

Use the *`OPTIONS`* clause to specify *`param=value`* pairs that represent an EDB*Loader directive. If a parameter is specified in both the `OPTIONS` clause and on the command line when `edbldr` is invoked, the command line setting is used.

Specify one or more of the following parameter/value pairs:

`DIRECT= { FALSE | TRUE }`

If `DIRECT` is set to `TRUE` EDB*Loader performs a direct path load instead of a conventional path load. The default value of `DIRECT` is `FALSE`.

See Section 6.1.5 for information on direct path loads.

`ERRORS=`*`error_count`*

`error_count` specifies the number of errors permitted before aborting the EDB*Loader session. The default is `50`.

`FREEZE= { FALSE | TRUE }`

Set `FREEZE` to `TRUE` to indicate that the data should be copied with the rows *frozen*. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, see the PostgreSQL core documentation at:

http://www.enterprisedb.com/docs/en/9.4/pg/routine-vacuuming.html

You must specify a data-loading type of `TRUNCATE` in the control file when using the `FREEZE` option. `FREEZE` is not supported for direct loading.

By default, `FREEZE` is `FALSE`.

`PARALLEL= { FALSE | TRUE }`

Set `PARALLEL` to `TRUE` to indicate that this EDB*Loader session is one of a number of concurrent EDB*Loader sessions participating in a parallel direct path load. The default value of `PARALLEL` is `FALSE`.

When `PARALLEL` is `TRUE`, the `DIRECT` parameter must also be set to `TRUE` . See Section 6.1.6 for more information about parallel direct path loads.

`ROWS=`*`n`*

`n` specifies the number of rows that EDB*Loader will commit before loading the next set of *`n`* rows.

If EDB*Loader encounters an invalid row during a load (in which the `ROWS` parameter is specified), those rows committed prior to encountering the error will remain in the destination table.

`SKIP=`*`skip_count`*

`skip_count` specifies the number of records at the beginning of the input data file that should be skipped before loading begins. The default is `0`.

`SKIP_INDEX_MAINTENANCE={ FALSE | TRUE }`

If `SKIP_INDEX_MAINTENANCE` is `TRUE`, index maintenance is not performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of `SKIP_INDEX_MAINTENANCE` is `FALSE`.

Please note: During a parallel direct path load, target table indexes are not updated, and are marked as invalid after the load is complete.

You can use the `REINDEX` command to rebuild an index. For more information about the `REINDEX` command, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-reindex.html

`charset`

Use the `CHARACTERSET` clause to identify the character set encoding of *`data_file`* where *`charset`* is the character set name. This clause is required if the data file encoding differs from the control file encoding. (The control file encoding must always be in the encoding of the client where `edbldr` is invoked.)

Examples of *`charset`* settings are `UTF8`, `SQL_ASCII`, and `SJIS`.

For more information about client to database character set conversion, see the PostgreSQL core documentation available at:

http://www.enterprisedb.com/docs/en/9.4/pg/multibyte.html

`data_file`

File containing the data to be loaded into *`target_table`*. Each record in the data file corresponds to a row to be inserted into *`target_table`*.

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of `.dat`, for example, `mydatafile.dat`.

Note: If the `DATA` parameter is specified on the command line when `edbldr` is invoked, the file given by the command line `DATA` parameter is used instead.

If the `INFILE` clause is omitted as well as the command line `DATA` parameter, then the data file name is assumed to be identical to the control file name, but with an extension of `.dat`.

`stdin`

Specify `stdin` (all lowercase letters) if you want to use standard input to pipe the data to be loaded directly to EDB*Loader. This is useful for data sources generating a large number of records to be loaded.

`bad_file`

File that receives *`data_file`* records that cannot be loaded due to errors.

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of `.bad`, for example, `mybadfile.bad`.

Note: If the `BAD` parameter is specified on the command line when `edbldr` is invoked, the file given by the command line `BAD` parameter is used instead.

`discard_file`

File that receives input data records that are not loaded into any table because none of the selection criteria are met for tables with the `WHEN` clause, and there are no tables without a `WHEN` clause. (All records meet the selection criteria of a table without a `WHEN` clause.)

If an extension is not provided in the file name, EDB*Loader assumes the file has an extension of `.dsc`, for example, `mydiscardfile.dsc`.

Note: If the `DISCARD` parameter is specified on the command line when `edbldr` is invoked, the file given by the command line `DISCARD` parameter is used instead.

`{ DISCARDMAX | DISCARDS }` *`max_discard_recs`*

Maximum number of discarded records that may be encountered from the input data records before terminating the EDB*Loader session. (A discarded record is described in the preceding description of the *`discard_file`* parameter.) Either keyword `DISCARDMAX` or `DISCARDS` may be used preceding the integer value specified by *`max_discard_recs`*.

For example, if *`max_discard_recs`* is `0`, then the EDB*Loader session is terminated if and when a first discarded record is encountered. If *`max_discard_recs`* is `1`, then the EDB*Loader session is terminated if and when a second discarded record is encountered.

When the EDB*Loader session is terminated due to exceeding *`max_discard_recs`*, prior input data records that have been loaded into the database are retained. They are not rolled back.

`INSERT | APPEND | REPLACE | TRUNCATE`

Specifies how data is to be loaded into the target tables. If one of `INSERT`, `APPEND`, `REPLACE`, or `TRUNCATE` is specified, it establishes the default action for all tables, overriding the default of `INSERT`.

`INSERT`

Data is to be loaded into an empty table. EDB*Loader throws an exception and does not load any data if the table is not initially empty.

Note: If the table contains rows, the `TRUNCATE` command must be used to empty the table prior to invoking EDB*Loader. EDB*Loader throws an exception if the `DELETE` command is used to empty the table instead of the `TRUNCATE` command. Oracle SQL*Loader allows the table to be emptied by using either the `DELETE` or `TRUNCATE` command.

`APPEND`

Data is to be added to any existing rows in the table. The table may be initially empty as well.

`REPLACE`

The `REPLACE` keyword and `TRUNCATE` keywords are functionally identical. The table is truncated by EDB*Loader prior to loading the new data.

Note: Delete triggers on the table are not fired as a result of the `REPLACE` operation.

`TRUNCATE`

The table is truncated by EDB*Loader prior to loading the new data. Delete triggers on the table are not fired as a result of the `TRUNCATE` operation.

`PRESERVE BLANKS`

For all target tables, retains leading white space when the optional enclosure delimiters are not present and leaves trailing white space intact when fields are specified with a predetermined size. When omitted, the default behavior is to trim leading and trailing white space.

`target_table`

Name of the table into which data is to be loaded. The table name may be schema-qualified (for example, `enterprisedb.emp`). The specified target must not be a view.

`field_condition`

Conditional clause taking the following form:

```
[ ( ] (start:end) { = | != | <> } 'val' [ ) ]
```

`start` and *`end`* are positive integers specifying the column positions in *`data_file`* that mark the beginning and end of a field that is to be compared with the constant *`val`*. The first character in each record begins with a *`start`* value of `1`.

In the `WHEN` *`field_condition`* `[ AND` *`field_condition`* `]` clause, if all such conditions evaluate to true for a given record, then EDB*Loader attempts to insert that record into *target_table*. If the insert operation fails, the record is written to *bad_file*.

All characters used in the *field_condition* text (particularly in the *val* string) must be valid in the database encoding. (For performing data conversion, EDB*Loader first converts the characters in *val* string to the database encoding and then to the data file encoding.)

If for a given record, none of the `WHEN` clauses evaluate to true for all `INTO TABLE` clauses, the record is written to *discard_file*, if a discard file was specified for the EDB*Loader session.

termstring

String of one or more characters that separates each field in *data_file*. The characters may be single-byte or multi-byte as long as they are valid in the database encoding. Two consecutive appearances of *termstring* with no intervening character results in the corresponding column set to null.

enclstring

String of one or more characters used to enclose a field value in *data_file*. The characters may be single-byte or multi-byte as long as they are valid in the database encoding. Use *enclstring* on fields where *termstring* appears as part of the data.

`TRAILING NULLCOLS`

If `TRAILING NULLCOLS` is specified, then the columns in the column list for which there is no data in *data_file* for a given record, are set to null when the row is inserted. This applies only to one or more consecutive columns at the end of the column list.

If fields are omitted at the end of a record and `TRAILING NULLCOLS` is not specified, EDB*Loader assumes the record contains formatting errors and writes it to the bad file.

column_name

Name of a column in *target_table* into which a field value defined by *field_def* is to be inserted.

`CONSTANT` *`val`*

Specifies a constant that is type-compatible with the column data type to which it is assigned in a field definition. Single or double quotes may enclose *`val`*. If *`val`* contains white space, then enclosing quotation marks must be used.

The use of the `CONSTANT` clause completely determines the value to be assigned to a column in each inserted row. No other clause may appear in the same field definition.

If the `TERMINATED BY` clause is used to delimit the fields in *`data_file`*, there must be no delimited field in *`data_file`* corresponding to any field definition with a `CONSTANT` clause. In other words, EDB*Loader assumes there is no field in *`data_file`* for any field definition with a `CONSTANT` clause.

`FILLER`

Specifies that the data in the field defined by the field definition is not to be loaded into the associated column. The column is set to null.

A column name defined with the `FILLER` clause must not be referenced in a SQL expression. See the discussion of the *`expr`* parameter.

`POSITION (`*`start`*`:`*`end`*`)`

Defines the location of the field in a record in a fixed-width field data file. *`start`* and *`end`* are positive integers. The first character in the record has a start value of `1`.

```
CHAR | DATE [ "datemask" ] | INTEGER EXTERNAL |
FLOAT EXTERNAL | DECIMAL EXTERNAL | ZONED EXTERNAL |
ZONED [(precision[,scale])]
```

Field type that describes the format of the data field in *`data_file`*.

Note: Specification of a field type is optional (for descriptive purposes only) and has no effect on whether or not EDB*Loader successfully inserts the data in the field into the table column. Successful loading depends upon the compatibility of the column data type and the field value. For example, a column with data type `NUMBER(7,2)` successfully accepts a field containing `2600`, but if the field contains a value such as `26XX`, the insertion fails and the record is written to *`bad_file`*.

Please note that `ZONED` data is not human-readable; `ZONED` data is stored in an internal format where each digit is encoded in a separate nibble/nybble/4-bit field.

In each `ZONED` value, the last byte contains a single digit (in the high-order 4 bits) and the sign (in the low-order 4 bits).

precision

Use precision to specify the length of the `ZONED` value.

If the precision value specified for `ZONED` conflicts with the length calculated by the server based on information provided with the `POSITION` clause, EDB*Loader will use the value specified for precision.

scale

scale specifies the number of digits to the right of the decimal point in a `ZONED` value.

datemask

Specifies the ordering and abbreviation of the day, month, and year components of a date field.

Note: If the `DATE` field type is specified along with a SQL expression for the column, then *datemask* must be specified after `DATE` and before the SQL expression. See the following discussion of the *expr* parameter.

`PRESERVE BLANKS`

For the column on which this option appears, retains leading white space when the optional enclosure delimiters are not present and leaves trailing white space intact when fields are specified with a predetermined size. When omitted, the default behavior is to trim leading and trailing white space.

expr

A SQL expression returning a scalar value that is type-compatible with the column data type to which it is assigned in a field definition. Double quotes must enclose *expr*. *expr* may contain a reference to any column in the field list (except for fields with the `FILLER` clause) by prefixing the column name by a colon character (`:`).

Examples

The following are some examples of control files and their corresponding data files.

The following control file uses a delimiter-separated data file that appends rows to the `emp` table:

```
LOAD DATA
  INFILE    'emp.dat'
    BADFILE 'emp.bad'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

In the preceding control file, the `APPEND` clause is used to allow the insertion of additional rows into the `emp` table.

The following is the corresponding delimiter-separated data file:

```
9101,ROGERS,CLERK,7902,17-DEC-10,1980.00,20
9102,PETERSON,SALESMAN,7698,20-DEC-10,2600.00,30,2300.00
9103,WARREN,SALESMAN,7698,22-DEC-10,5250.00,30,2500.00
9104,"JONES, JR.",MANAGER,7839,02-APR-09,7975.00,20
```

The use of the `TRAILING NULLCOLS` clause allows the last field supplying the `comm` column to be omitted from the first and last records. The `comm` column is set to null for the rows inserted from these records.

The double quotation mark enclosure character surrounds the value `JONES, JR.` in the last record since the comma delimiter character is part of the field value.

The following query displays the rows added to the table after the EDB*Loader session:

```
SELECT * FROM emp WHERE empno > 9100;

empno |   ename    |   job    | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+------------+----------+------+--------------------+---------+---------+-------
-
  9101 | ROGERS     | CLERK    | 7902 | 17-DEC-10 00:00:00 | 1980.00 |         |     20
  9102 | PETERSON   | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2300.00 |     30
  9103 | WARREN     | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 2500.00 |     30
  9104 | JONES, JR. | MANAGER  | 7839 | 02-APR-09 00:00:00 | 7975.00 |         |     20
(4 rows)
```

The following example is a control file that loads the same rows into the `emp` table, but uses a data file containing fixed-width fields:

```
LOAD DATA
  INFILE        'emp_fixed.dat'
    BADFILE     'emp_fixed.bad'
  APPEND
  INTO TABLE emp
```

```
    TRAILING NULLCOLS
  (
    empno        POSITION (1:4),
    ename        POSITION (5:14),
    job          POSITION (15:23),
    mgr          POSITION (24:27),
    hiredate     POSITION (28:38),
    sal          POSITION (39:46),
    deptno       POSITION (47:48),
    comm         POSITION (49:56)
  )
```

In the preceding control file, the `FIELDS TERMINATED BY` and `OPTIONALLY ENCLOSED BY` clauses are absent. Instead, each field now includes the `POSITION` clause.

The following is the corresponding data file containing fixed-width fields:

```
9101ROGERS    CLERK    790217-DEC-10   1980.0020
9102PETERSON  SALESMAN 769820-DEC-10   2600.0030 2300.00
9103WARREN    SALESMAN 769822-DEC-10   5250.0030 2500.00
9104JONES, JR.MANAGER  783902-APR-09   7975.0020
```

The following control file illustrates the use of the `FILLER` clause in the data fields for the `sal` and `comm` columns. EDB*Loader ignores the values in these fields and sets the corresponding columns to null.

```
LOAD DATA
  INFILE         'emp_fixed.dat'
    BADFILE      'emp_fixed.bad'
  APPEND
  INTO TABLE emp
    TRAILING NULLCOLS
  (
    empno        POSITION (1:4),
    ename        POSITION (5:14),
    job          POSITION (15:23),
    mgr          POSITION (24:27),
    hiredate     POSITION (28:38),
    sal          FILLER POSITION (39:46),
    deptno       POSITION (47:48),
    comm         FILLER POSITION (49:56)
  )
```

Using the same fixed-width data file as in the prior example, the resulting rows in the table appear as follows:

```
SELECT * FROM emp WHERE empno > 9100;

empno |      ename       |   job    | mgr  |      hiredate      | sal | comm | deptno
-------+------------------+----------+------+--------------------+-----+------+--------
  9101 | ROGERS           | CLERK    | 7902 | 17-DEC-10 00:00:00 |     |      |     20
  9102 | PETERSON         | SALESMAN | 7698 | 20-DEC-10 00:00:00 |     |      |     30
  9103 | WARREN           | SALESMAN | 7698 | 22-DEC-10 00:00:00 |     |      |     30
  9104 | JONES, JR.       | MANAGER  | 7839 | 02-APR-09 00:00:00 |     |      |     20
(4 rows)
```

The following example illustrates the use of multiple `INTO TABLE` clauses. For this example, two empty tables are created with the same data definition as the `emp` table. The following `CREATE TABLE` commands create these two empty tables, while inserting no rows from the original `emp` table:

```
CREATE TABLE emp_research AS SELECT * FROM emp WHERE deptno = 99;
CREATE TABLE emp_sales AS SELECT * FROM emp WHERE deptno = 99;
```

The following control file contains two `INTO TABLE` clauses. Also note that there is no `APPEND` clause so the default operation of `INSERT` is used, which requires that tables `emp_research` and `emp_sales` be empty.

```
LOAD DATA
  INFILE        'emp_multitbl.dat'
    BADFILE     'emp_multitbl.bad'
    DISCARDFILE 'emp_multitbl.dsc'
  INTO TABLE emp_research
    WHEN (47:48) = '20'
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      CONSTANT '20',
    comm        POSITION (49:56)
  )
  INTO TABLE emp_sales
    WHEN (47:48) = '30'
    TRAILING NULLCOLS
  (
    empno       POSITION (1:4),
    ename       POSITION (5:14),
    job         POSITION (15:23),
    mgr         POSITION (24:27),
    hiredate    POSITION (28:38),
    sal         POSITION (39:46),
    deptno      CONSTANT '30',
    comm        POSITION (49:56) "ROUND(:comm + (:sal * .25), 0)"
  )
```

The `WHEN` clauses specify that when the field designated by columns 47 thru 48 contains `20`, the record is inserted into the `emp_research` table and when that same field contains `30`, the record is inserted into the `emp_sales` table. If neither condition is true, the record is written to the discard file named `emp_multitbl.dsc`.

The `CONSTANT` clause is given for column `deptno` so the specified constant value is inserted into `deptno` for each record. When the `CONSTANT` clause is used, it must be the only clause in the field definition other than the column name to which the constant value is assigned.

Finally, column `comm` of the `emp_sales` table is assigned a SQL expression. Column names may be referenced in the expression by prefixing the column name with a colon character (:).

The following is the corresponding data file:

```
9101ROGERS     CLERK    790217-DEC-10   1980.0020
9102PETERSON   SALESMAN 769820-DEC-10   2600.0030 2300.00
9103WARREN     SALESMAN 769822-DEC-10   5250.0030 2500.00
9104JONES, JR.MANAGER  783902-APR-09   7975.0020
9105ARNOLDS    CLERK    778213-SEP-10   3750.0010
9106JACKSON    ANALYST  756603-JAN-11   4500.0040
```

Since the records for employees `ARNOLDS` and `JACKSON` contain `10` and `40` in columns 47 thru 48, which do not satisfy any of the `WHEN` clauses, EDB*Loader writes these two records to the discard file, `emp_multitbl.dsc`, whose content is shown by the following:

```
9105ARNOLDS    CLERK    778213-SEP-10   3750.0010
9106JACKSON    ANALYST  756603-JAN-11   4500.0040
```

The following are the rows loaded into the `emp_research` and `emp_sales` tables:

```
SELECT * FROM emp_research;

empno |   ename    |   job   | mgr  |      hiredate      |   sal   | comm | deptno
-------+------------+---------+------+--------------------+---------+------+--------
  9101 | ROGERS     | CLERK   | 7902 | 17-DEC-10 00:00:00 | 1980.00 |      |  20.00
  9104 | JONES, JR. | MANAGER | 7839 | 02-APR-09 00:00:00 | 7975.00 |      |  20.00
(2 rows)

SELECT * FROM emp_sales;

empno |  ename   |   job    | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+----------+----------+------+--------------------+---------+---------+--------
  9102 | PETERSON | SALESMAN | 7698 | 20-DEC-10 00:00:00 | 2600.00 | 2950.00 |  30.00
  9103 | WARREN   | SALESMAN | 7698 | 22-DEC-10 00:00:00 | 5250.00 | 3813.00 |  30.00
(2 rows)
```

6.1.4 Invoking EDB*Loader

You must have superuser privileges to run EDB*Loader. Use the following command to invoke EDB*Loader from the command line:

```
edbldr [ -d dbname ] [ -p port ] [ -h host ]
[ USERID={ username/password | username/ | username | / } ]
  CONTROL=control_file
[ DATA=data_file ]
[ BAD=bad_file ]
[ DISCARD=discard_file ]
[ DISCARDMAX=max_discard_recs ]
[ LOG=log_file ]
[ PARFILE=param_file ]
[ DIRECT={ FALSE | TRUE } ]
[ FREEZE={ FALSE | TRUE } ]
[ ERRORS=error_count ]
[ PARALLEL={ FALSE | TRUE } ]
[ ROWS=n ]
[ SKIP=skip_count ]
[ SKIP_INDEX_MAINTENANCE={ FALSE | TRUE } ]
[ edb_resource_group=group_name ]
```

Description

If the `-d` option, the `-p` option, or the `-h` option are omitted, the defaults for the database, port, and host are determined according to the same rules as other Postgres Plus Advanced Server utility programs such as `edb-psql`, for example.

Any parameter listed in the preceding syntax diagram except for the `-d` option, `-p` option, `-h` option, and the `PARFILE` parameter may be specified in a *parameter file*. The parameter file is specified on the command line when `edbldr` is invoked using `PARFILE=param_file`. Some parameters may be specified in the `OPTIONS` clause in the control file. See the description of the control file in Section 6.1.3.

The specification of *control_file*, *data_file*, *bad_file*, *discard_file*, *log_file*, and *param_file* may include the full directory path or a relative directory path to the file name. If the file name is specified alone or with a relative directory path, the file is assumed to exist (in the case of *control_file*, *data_file*, or *param_file*), or to be created (in the case of *bad_file*, *discard_file*, or *log_file*) relative to the current working directory from which `edbldr` is invoked.

Note: The control file must exist in the character set encoding of the client where `edbldr` is invoked. If the client is in a different encoding than the database encoding, then the `PGCLIENTENCODING` environment variable must be set on the client to the client's encoding prior to invoking `edbldr`. This must be done to ensure character set conversion is properly done between the client and the database server.

The operating system account used to invoke `edbldr` must have read permission on the directories and files specified by *`control_file`*, *`data_file`*, and *`param_file`*.

The operating system account `enterprisedb` must have write permission on the directories where *`bad_file`*, *`discard_file`*, and *`log_file`* are to be written.

Note: It is suggested that the file names for *`control_file`*, *`data_file`*, *`bad_file`*, *`discard_file`*, and *`log_file`* include extensions of `.ctl`, `.dat`, `.bad`, `.dsc`, and `.log`, respectively. If the provided file name does not contain an extension, EDB*Loader assumes the actual file name includes the appropriate aforementioned extension.

Parameters

`dbname`

Name of the database containing the tables to be loaded.

`port`

Port number on which the database server is accepting connections.

`host`

IP address of the host on which the database server is running.

`USERID={ `*`username/password`*` | `*`username/`*` | `*`username`*` | / }`

EDB*Loader connects to the database with *`username`*. *`username`* must be a superuser. *`password`* is the password for *`username`*.

If the `USERID` parameter is omitted, EDB*Loader prompts for *`username`* and *`password`*. If `USERID=`*`username/`* is specified, then EDB*Loader 1) uses the password file specified by environment variable `PGPASSFILE` if `PGPASSFILE` is set, or 2) uses the `.pgpass` password file (`pgpass.conf` on Windows systems) if `PGPASSFILE` is not set. If `USERID=`*`username`* is specified, then EDB*Loader prompts for *`password`*. If `USERID=/` is specified, the connection is attempted using the operating system account as the user name.

Note: The Postgres Plus Advanced Server connection environment variables `PGUSER` and `PGPASSWORD` are ignored by EDB*Loader. See the PostgreSQL core documentation for information on the `PGPASSFILE` environment variable and the password file.

CONTROL=control_file

> *control_file* specifies the name of the control file containing EDB*Loader directives. If a file extension is not specified, an extension of `.ctl` is assumed. See Section 6.1.3 for a description of the control file.

DATA=data_file

> *data_file* specifies the name of the file containing the data to be loaded into the target table. If a file extension is not specified, an extension of `.dat` is assumed. See Section 6.1.3 for a description of the *data_file*.
>
> **Note:** Specifying a *data_file* on the command line overrides the `INFILE` clause specified in the control file.

BAD=bad_file

> *bad_file* specifies the name of a file that receives input data records that cannot be loaded due to errors. See Section 6.1.3 for a description of the *bad_file*.
>
> **Note:** Specifying a *bad_file* on the command line overrides any `BADFILE` clause specified in the control file.

DISCARD=discard_file

> *discard_file* is the name of the file that receives input data records that do not meet any table's selection criteria. See the description of *discard_file* in Section 6.1.3.
>
> **Note:** Specifying a *discard_file* using the command line `DISCARD` parameter overrides the `DISCARDFILE` clause in the control file.

DISCARDMAX=max_discard_recs

> *max_discard_recs* is the maximum number of discarded records that may be encountered from the input data records before terminating the EDB*Loader session. See the description of *max_discard_recs* in Section 6.1.3.
>
> **Note:** Specifying *max_discard_recs* using the command line `DISCARDMAX` parameter overrides the `DISCARDMAX` or `DISCARDS` clause in the control file.

LOG=log_file

> *log_file* specifies the name of the file in which EDB*Loader records the results of the EDB*Loader session.

If the LOG parameter is omitted, EDB*Loader creates a log file with the name *control_file_base*.log in the directory from which edbldr is invoked. *control_file_base* is the base name of the control file used in the EDB*Loader session. The operating system account enterprisedb must have write permission on the directory where the log file is to be written.

PARFILE=param_file

param_file specifies the name of the file that contains command line parameters for the EDB*Loader session. Any command line parameter listed in this section except for the -d, -p, and -h options, and the PARFILE parameter itself, can be specified in *param_file* instead of on the command line.

Any parameter given in *param_file* overrides the same parameter supplied on the command line before the PARFILE option. Any parameter given on the command line that appears after the PARFILE option overrides the same parameter given in *param_file*.

Note: Unlike other EDB*Loader files, there is no default file name or extension assumed for *param_file*, though by Oracle SQL*Loader convention, .par is typically used, but not required, as an extension.

DIRECT= { FALSE | TRUE }

If DIRECT is set to TRUE EDB*Loader performs a direct path load instead of a conventional path load. The default value of DIRECT is FALSE.

See Section 6.1.5 for information on direct path loads.

FREEZE= { FALSE | TRUE }

Set FREEZE to TRUE to indicate that the data should be copied with the rows *frozen*. A tuple guaranteed to be visible to all current and future transactions is marked as frozen to prevent transaction ID wrap-around. For more information about frozen tuples, see the PostgreSQL core documentation at:

http://www.enterprisedb.com/docs/en/9.4/pg/routine-vacuuming.html

You must specify a data-loading type of TRUNCATE in the control file when using the FREEZE option. FREEZE is not supported for direct loading.

By default, FREEZE is FALSE.

ERRORS=*error_count*

error_count specifies the number of errors permitted before aborting the EDB*Loader session. The default is 50.

PARALLEL= { FALSE | TRUE }

Set PARALLEL to TRUE to indicate that this EDB*Loader session is one of a number of concurrent EDB*Loader sessions participating in a parallel direct path load. The default value of PARALLEL is FALSE.

When PARALLEL is TRUE, the DIRECT parameter must also be set to TRUE . See Section 6.1.6 for more information about parallel direct path loads.

ROWS=*n*

n specifies the number of rows that EDB*Loader will commit before loading the next set of *n* rows.

SKIP=skip_count

Number of records at the beginning of the input data file that should be skipped before loading begins. The default is 0.

SKIP_INDEX_MAINTENANCE= { FALSE | TRUE }

If set to TRUE, index maintenance is not performed as part of a direct path load, and indexes on the loaded table are marked as invalid. The default value of SKIP_INDEX_MAINTENANCE is FALSE.

Please note: During a parallel direct path load, target table indexes are not updated, and are marked as invalid after the load is complete.

You can use the REINDEX command to rebuild an index. For more information about the REINDEX command, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-reindex.html

edb_resource_group=*group_name*

group_name specifies the name of an EDB Resource Manager resource group to which the EDB*Loader session is to be assigned.

Any default resource group that may have been assigned to the session (for example, a database user running the EDB*Loader session who had been assigned a default resource group with the ALTER ROLE ... SET edb_resource_group command) is overridden by the resource group given by the edb_resource_group parameter specified on the edbldr command line.

For information about the EDB Resource Manager, see Chapter 5, "EDB Resource Manager" in the Postgres Plus Enterprise Edition Guide, available at:

http://www.enterprisedb.com/docs/en/9.4/eeguide/Table%2520of%2520Contents.htm

Examples

In the following example EDB*Loader is invoked using a control file named emp.ctl located in the current working directory to load a table in database edb:

```
$ /opt/PostgresPlus/9.4AS/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl
EDB*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

Successfully loaded (4) records
```

In the following example, EDB*Loader prompts for the user name and password since they are omitted from the command line. In addition, the files for the bad file and log file are specified with the BAD and LOG command line parameters.

```
$ /opt/PostgresPlus/9.4AS/bin/edbldr -d edb CONTROL=emp.ctl BAD=/tmp/emp.bad
LOG=/tmp/emp.log
Enter the user name : enterprisedb
Enter the password :
EDB*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

Successfully loaded (4) records
```

The following example runs EDB*Loader with the same parameters as shown in the preceding example, but using a parameter file located in the current working directory. The SKIP and ERRORS parameters are altered from their defaults in the parameter file as well.

The parameter file, emp.par, contains the following:

```
CONTROL=emp.ctl
BAD=/tmp/emp.bad
LOG=/tmp/emp.log
SKIP=1
ERRORS=10
```

EDB*Loader is invoked with the parameter file as shown by the following:

```
$ /opt/PostgresPlus/9.4AS/bin/edbldr -d edb PARFILE=emp.par
Enter the user name : enterprisedb
Enter the password :
```

```
EDB*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

Successfully loaded (3) records
```

6.1.4.1 Exit Codes

When EDB*Loader exits, it will return one of the following codes:

Exit Code	Description
`0`	Indicates that all rows loaded successfully.
`1`	Indicates that EDB*Loader encountered command line or syntax errors, or aborted the load operation due to an unrecoverable error.
`2`	Indicates that the load completed, but some (or all) rows were rejected or discarded.
`3`	Indicates that EDB*Loader encountered fatal errors (such as OS errors). This class of errors is equivalent to the `FATAL` or `PANIC` severity levels of PostgreSQL errors.

6.1.5 Direct Path Load

During a direct path load, EDB*Loader writes the data directly to the database pages, which is then synchronized to disk. The insert processing associated with a conventional path load is bypassed, thereby resulting in a performance improvement.

Bypassing insert processing reduces the types of constraints that may exist on the target table. The following types of constraints are permitted on the target table of a direct path load:

- Primary key
- Not null constraints
- Indexes (unique or non-unique)

The restrictions on the target table of a direct path load are the following:

- Triggers are not permitted
- Check constraints are not permitted
- Foreign key constraints on the target table referencing another table are not permitted
- Foreign key constraints on other tables referencing the target table are not permitted
- The table must not be partitioned
- Rules may exist on the target table, but they are not executed

Note: Currently, a direct path load in EDB*Loader is more restrictive than in Oracle SQL*Loader. The preceding restrictions do not apply to Oracle SQL*Loader in most cases. The following restrictions apply to a control file used in a direct path load:

- Multiple table loads are not supported. That is, only one `INTO TABLE` clause may be specified in the control file.
- SQL expressions may not be used in the data field definitions of the `INTO TABLE` clause.
- The `FREEZE` option is not supported for direct path loading.

To run a direct path load, add the `DIRECT=TRUE` option as shown by the following example:

```
$ /opt/PostgresPlus/9.4AS/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp.ctl DIRECT=TRUE
EDB*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.

Successfully loaded (4) records
```


6.1.6 Parallel Direct Path Load

The performance of a direct path load can be further improved by distributing the loading process over two or more sessions running concurrently. Each session runs a direct path load into the same table.

Since the same table is loaded from multiple sessions, the input records to be loaded into the table must be divided amongst several data files so that each EDB*Loader session uses its own data file and the same record is not loaded more than once into the table.

The target table of a parallel direct path load is under the same restrictions as a direct path load run in a single session.

The restrictions on the target table of a direct path load are the following:

- Triggers are not permitted
- Check constraints are not permitted
- Foreign key constraints on the target table referencing another table are not permitted
- Foreign key constraints on other tables referencing the target table are not permitted
- The table must not be partitioned
- Rules may exist on the target table, but they are not executed

In addition, the `APPEND` clause must be specified in the control file used by each EDB*Loader session.

To run a parallel direct path load, run EDB*Loader in a separate session for each participant of the parallel direct path load. Invocation of each such EDB*Loader session must include the `DIRECT=TRUE` and `PARALLEL=TRUE` parameters.

Each EDB*Loader session runs as an independent transaction so if one of the parallel sessions aborts and rolls back its changes, the loading done by the other parallel sessions are not affected.

Note: In a parallel direct path load, each EDB*Loader session reserves a fixed number of blocks in the target table in a round-robin fashion. Some of the blocks in the last allocated chunk may not be used, and those blocks remain uninitialized. A subsequent use of the `VACUUM` command on the target table may show warnings regarding these uninitialized blocks such as the following:

```
WARNING:  relation "emp" page 98264 is uninitialized --- fixing

WARNING:  relation "emp" page 98265 is uninitialized --- fixing
```



```
WARNING:  relation "emp" page 98266 is uninitialized --- fixing
```

This is an expected behavior and does not indicate data corruption.

Indexes on the target table are not updated during a parallel direct path load and are therefore marked as invalid after the load is complete. You must use the `REINDEX` command to rebuild the indexes.

The following example shows the use of a parallel direct path load on the `emp` table.

Note: If you attempt a parallel direct path load on the sample `emp` table provided with Postgres Plus Advanced Server, you must first remove the triggers and constraints referencing the `emp` table. In addition the primary key column, `empno`, was expanded from `NUMBER(4)` to `NUMBER` in this example to allow for the insertion of a larger number of rows.

The following is the control file used in the first session:

```
LOAD DATA
  INFILE    '/home/user/loader/emp_parallel_1.dat'
  APPEND
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

The `APPEND` clause must be specified in the control file for a parallel direct path load.

The following shows the invocation of EDB*Loader in the first session. The `DIRECT=TRUE` and `PARALLEL=TRUE` parameters must be specified.

```
$ /opt/PostgresPlus/9.4AS/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp_parallel_1.ctl DIRECT=TRUE PARALLEL=TRUE
WARNING:  index maintenance will be skipped with PARALLEL load
EDB*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.
```

The control file used for the second session appears as follows. Note that it is the same as the one used in the first session, but uses a different data file.

```
LOAD DATA
  INFILE    '/home/user/loader/emp_parallel_2.dat'
  APPEND
```

```
  INTO TABLE emp
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    TRAILING NULLCOLS
  (
    empno,
    ename,
    job,
    mgr,
    hiredate,
    sal,
    deptno,
    comm
  )
```

The preceding control file is used in a second session as shown by the following:

```
$ /opt/PostgresPlus/9.4AS/bin/edbldr -d edb USERID=enterprisedb/password
CONTROL=emp_parallel_2.ctl DIRECT=TRUE PARALLEL=TRUE
WARNING:  index maintenance will be skipped with PARALLEL load
EDB*Loader: Copyright (c) 2007-2014, EnterpriseDB Corporation.
```

EDB*Loader displays the following message in each session when its respective load operation completes:

```
Successfully loaded (10000) records
```

The following query shows that the index on the `emp` table has been marked as `INVALID`:

```
SELECT index_name, status FROM user_indexes WHERE table_name = 'EMP';

 index_name | status
------------+---------
 EMP_PK     | INVALID
(1 row)
```

Note: `user_indexes` is the Oracle compatible view of indexes owned by the current user.

Queries on the `emp` table will not utilize the index unless it is rebuilt using the `REINDEX` command as shown by the following:

```
REINDEX INDEX emp_pk;
```

A subsequent query on `user_indexes` shows that the index is now marked as `VALID`:

```
SELECT index_name, status FROM user_indexes WHERE table_name = 'EMP';

 index_name | status
------------+--------
 EMP_PK     | VALID
(1 row)
```

6.1.7 Remote Loading

EDB*Loader supports a feature called *remote loading*. In remote loading, the database containing the table to be loaded is running on a database server on a different host than from where EDB*Loader is invoked with the input data source.

This feature is useful if you have a large amount of data to be loaded, and you do not want to create a large data file on the host running the database server.

In addition, you can use the standard input feature to pipe the data from the data source such as another program or script, directly to EDB*Loader, which then loads the table in the remote database. This bypasses the process of having to create a data file on disk for EDB*Loader.

Performing remote loading along with using standard input requires the following:

- The `edbldr` program must be installed on the client host on which it is to be invoked with the data source for the EDB*Loader session.
- The control file must contain the clause `INFILE 'stdin'` so you can pipe the data directly into EDB*Loader's standard input. See Section 6.1.3 for information on the `INFILE` clause and the EDB*Loader control file.
- All files used by EDB*Loader such as the control file, bad file, discard file, and log file must reside on, or are created on, the client host on which `edbldr` is invoked.
- When invoking EDB*Loader, use the `-h` option to specify the IP address of the remote database server. See Section 6.1.4 for information on invoking EDB*Loader.
- Use the operating system pipe operator (`|`) or input redirection operator (<) to supply the input data to EDB*Loader.

The following example loads a database running on a database server at `192.168.1.14` using data piped from a source named `datasource`.

```
datasource | ./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password
CONTROL=remote.ctl
```

The following is another example of how standard input can be used:

```
./edbldr -d edb -h 192.168.1.14 USERID=enterprisedb/password
CONTROL=remote.ctl < datasource
```

6.1.8 Updating a Table with a Conventional Path Load

You can use EDB*Loader with a conventional path load to update the rows within a table, merging new data with the existing data. When you invoke EDB*Loader to perform an update, the server searches the table for an existing row with a matching primary key:

- If the server locates a row with a matching key, it replaces the existing row with the new row.

- If the server does not locate a row with a matching key, it adds the new row to the table.

To use EDB*Loader to update a table, the table must have a primary key. Please note that you cannot use EDB*Loader to `UPDATE` a partitioned table.

To perform an `UPDATE`, use the same steps as when performing a conventional path load:

1. Create a data file that contains the rows you wish to `UPDATE` or `INSERT`.

2. Define a control file that uses the `INFILE` keyword to specify the name of the data file. For information about building the EDB*Loader control file, see Section 6.1.3.

3. Invoke EDB*Loader, specifying the database name, connection information, and the name of the control file. For information about invoking EDB*Loader, see Section 6.1.4.

The following example uses the `emp` table that is distributed with the Advanced Server sample data. By default, the table contains:

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

```
(14 rows)
```

The following control file (`emp_update.ctl`) specifies the fields in the table in a comma-delimited list. The control file performs an `UPDATE` on the `emp` table:

```
LOAD DATA
  INFILE 'emp_update.dat'
  BADFILE 'emp_update.bad'
  DISCARDFILE 'emp_update.dsc'
UPDATE INTO TABLE emp
FIELDS TERMINATED BY ","
(empno, ename, job, mgr, hiredate, sal, comm, deptno)
```

The data that is being updated or inserted is saved in the `emp_update.dat` file. `emp_update.dat` contains:

```
7521,WARD,MANAGER,7839,22-FEB-81 00:00:00,3000.00,0.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,3500.00,0.00,20
7903,BAKER,SALESMAN,7521,10-JUN-13 00:00:00,1800.00,500.00,20
7904,MILLS,SALESMAN,7839,13-JUN-13 00:00:00,1800.00,500.00,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1500.00,400.00,30
```

Invoke EDB*Loader, specifying the name of the database (`edb`), the name of a database superuser (and their associated password) and the name of the control file (`emp_update.ctl`):

```
edbldr -d edb userid=user_name/password control=emp_update.ctl
```

After performing the update, the `emp` table contains:

```
edb=# select * from emp;
empno|ename |   job    | mgr  |      hiredate       |   sal    | comm   | deptno
-----+------+----------+------+--------------------+---------+-------+--------
7369 |SMITH |CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |        |      20
7499 |ALLEN |SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 |300.00  |      30
7521 |WARD  |MANAGER   | 7839 | 22-FEB-81 00:00:00 | 3000.00 |0.00    |      30
7566 |JONES |MANAGER   | 7839 | 02-APR-81 00:00:00 | 3500.00 |0.00    |      20
7654 |MARTIN|SALESMAN  | 7698 | 28-SEP-81 00:00:00 | 1500.00 |400.00  |      30
7698 |BLAKE |MANAGER   | 7839 | 01-MAY-81 00:00:00 | 2850.00 |        |      30
7782 |CLARK |MANAGER   | 7839 | 09-JUN-81 00:00:00 | 2450.00 |        |      10
7788 |SCOTT |ANALYST   | 7566 | 19-APR-87 00:00:00 | 3000.00 |        |      20
7839 |KING  |PRESIDENT|       | 17-NOV-81 00:00:00 | 5000.00 |        |      10
7844 |TURNER|SALESMAN  | 7698 | 08-SEP-81 00:00:00 | 1500.00 |   0.00 |      30
7876 |ADAMS |CLERK     | 7788 | 23-MAY-87 00:00:00 | 1100.00 |        |      20
7900 |JAMES |CLERK     | 7698 | 03-DEC-81 00:00:00 |  950.00 |        |      30
7902 |FORD  |ANALYST   | 7566 | 03-DEC-81 00:00:00 | 3000.00 |        |      20
7903 |BAKER |SALESMAN |7521   | 10-JUN-13 00:00:00 | 1800.00 |500.00  |      20
7904 |MILLS |SALESMAN |7839   |13-JUN-13 00:00:00  |1800.00  |500.00  |      20
7934 |MILLER|CLERK     | 7782 | 23-JAN-82 00:00:00 | 1300.00 |        |      10
(16 rows)
```

The rows containing information for the three employees that are currently in the `emp` table are updated, while rows are added for the new employees (`BAKER` and `MILLS`).

6.1.9 Loading Empty Strings with EDB*Loader

Advanced Server includes a configuration parameter that controls how EDB*Loader handles a CSV (comma-separated value) file containing empty strings. An empty string within a CSV file may take the form of:

- an unquoted empty string. For example:

```
9001,  , 40
```

- a single-quoted, comma-delimited value. For example:

```
9001, '', 40
```

- a double-quoted, comma-delimited value. For example:

```
9001, "", 40
```

You can use the `edbldr.empty_csv_field` parameter to specify how EDB*Loader will treat an empty string. The valid values for the `edbldr.empty_csv_field` parameter are:

Parameter Setting	EDB*Loader Behavior
`NULL`	An empty field is treated as `NULL`.
`empty_string`	An empty field is treated as a string of length zero.
`pgsql`	An empty field is treated as a `NULL` if it does not contain quotes and as an empty string if it contains quotes.

You can set the `edbldr.empty_csv_field` parameter in any context (i.e. with a `SET` statement, or in the `postgresql.conf` file). You can also use the `PGOPTIONS` environment variable to set the value of `edbldr.empty_csv_field` for an EDB*Loader session. For example, before invoking EDB*Loader, enter the command:

```
$ export PGOPTIONS="-c edbldr.empty_csv_field=empty_string"
```

Then, invoke EDB*Loader, specifying command line options as required.

For more information about setting parameter values, see the PostgreSQL core documentation, available at:

http://www.enterprisedb.com/documentation/english

*6.2 EDB*Plus*

EDB*Plus is a utility program that provides a command line user interface to the Postgres Plus Advanced Server. EDB*Plus accepts SQL commands, SPL anonymous blocks, and EDB*Plus commands. EDB*Plus provides various capabilities including:

- Querying certain database objects
- Executing stored procedures
- Formatting output from SQL commands
- Executing batch scripts
- Executing OS commands
- Recording output

The following section describes how to connect to a Postgres Plus Advanced Server database using EDB*Plus. The final section provides a summary of the EDB*Plus commands.

6.2.1 Starting EDB*Plus

To open an EDB*Plus command line, navigate through the `Applications` (or `Start`) menu to the `Postgres Plus Advanced Server` menu, to the `Run SQL Command Line` menu, and select the `EDB*Plus` option. You can also invoke EDB*Plus from the operating system command line with the following command:

```
edbplus [ -S[ILENT ] ] [ login | /NOLOG ] [ @scriptfile[.ext ] ]
```

`-SILENT`

If specified, the EDB*Plus sign-on banner is suppressed along with all prompts.

`login`

Login information for connecting to the database server and database. *`login`* takes the following format. (There must be no white space within the login information.)

```
username[/password][@{connectstring | variable } ]
```

Where:

`username` is a database username with which to connect to the database.

`password` is the password associated with the specified *`username`*. If a *`password`* is not provided, but a password is required for authentication, EDB*Plus will prompt for the password.

connectstring is the database connection string.

variable is a variable defined in the `login.sql` file that contains a database connection string. The `login.sql` file can be found in the `edbplus` subdirectory of the Postgres Plus Advanced Server home directory.

host[:*port*][/*dbname*]]

host is the hostname on which the database server resides. If neither @*connectstring* nor @*variable* nor `/NOLOG` is specified, the default host is assumed to be the localhost. *port* is the port number receiving connections on the database server. If not specified, the default is 5444. *dbname* is the name of the database to connect to. If not specified the default is `edb`.

/NOLOG

Specify `/NOLOG` to start EDB*Plus without establishing a database connection. SQL commands and EDB*Plus commands that require a database connection cannot be used in this mode. The `CONNECT` command can be subsequently given to connect to a database after starting EDB*Plus with the `/NOLOG` option.

scriptfile[.*ext*]

scriptfile is the name of a file residing in the current working directory, containing SQL and/or EDB*Plus commands that will be automatically executed after startup of EDB*Plus. *ext* is the filename extension. If the filename extension is `sql`, then the `.sql` extension may be omitted when specifying *scriptfile*. When creating a script file, always name the file with an extension, otherwise it will not be accessible by EDB*Plus. (EDB*Plus will always assume a `.sql` extension on filenames that are specified with no extension.)

The following example shows user `enterprisedb` with password, `password`, connecting to database `edb` running on a database server on the localhost at port 5444.

```
C:\Program Files (x86)\PostgresPlus\9.4AS\edbplus>edbplus
enterprisedb/password
Connected to EnterpriseDB 9.4.0.0 (localhost:5444/edb) AS enterprisedb

EDB*Plus: Release 9.4
Copyright (c) 2008-2016, EnterpriseDB Corporation.  All rights reserved.

SQL>
```

The following example shows user `enterprisedb` with password, `password`, connecting to database `edb` running on a database server on the localhost at port 5445.

```
C:\Program Files (x86)\PostgresPlus\9.4AS\edbplus>edbplus
enterprisedb/password@localhost:5445/edb
```



```
Connected to EnterpriseDB 9.4.0.0 (localhost:5445/edb) AS enterprisedb

EDB*Plus: Release 9.4
Copyright (c) 2008-2016, EnterpriseDB Corporation.  All rights reserved.

SQL>
```

Using variable `hr_5445` in the `login.sql` file, the following illustrates how it is used to connect to database `hr` on localhost at port 5445.

```
C:\Program Files (x86)\PostgresPlus\9.4AS\edbplus>edbplus
enterprisedb/password@hr_5445
Connected to EnterpriseDB 9.4.0.0 (localhost:5445/hr) AS enterprisedb

EDB*Plus: Release 9.4 (Build 28)
Copyright (c) 2008-2016, EnterpriseDB Corporation.  All rights reserved.

SQL>
```

The following is the content of the `login.sql` file used in the previous example.

```
define edb="localhost:5445/edb"
define hr_5445="localhost:5445/hr"
```

The following example executes a script file, `dept_query.sql` after connecting to database `edb` on server localhost at port 5444.

```
C:\Program Files (x86)\PostgresPlus\9.4AS\edbplus>edbplus
enterprisedb/password @dept_query
Connected to EnterpriseDB 9.4.0.0 (localhost:5444/edb) AS enterprisedb

SQL> SELECT * FROM dept;

DEPTNO DNAME          LOC
------ -------------- -------------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON

SQL> EXIT
Disconnected from EnterpriseDB Database.
```

The following is the content of file `dept_query.sql` used in the previous example.

```
SET PAGESIZE 9999
SET ECHO ON
SELECT * FROM dept;
EXIT
```

6.2.2 Command Summary

This section contains a summary of EDB*Plus commands.

6.2.2.1 ACCEPT

The `ACCEPT` command displays a prompt and waits for the user's keyboard input. The value input by the user is placed in the specified variable.

ACC[EPT] *variable*

The following example creates a new variable named `my_name`, accepts a value of John Smith, then displays the value using the `DEFINE` command.

```
SQL> ACCEPT my_name
Enter value for my_name: John Smith
SQL> DEFINE my_name
DEFINE MY_NAME = "John Smith"
```

6.2.2.2 APPEND

`APPEND` is a line editor command that appends the given text to the end of the current line in the SQL buffer.

A[PPEND] *text*

In the following example, a `SELECT` command is built-in the SQL buffer using the `APPEND` command. Note that two spaces are placed between the `APPEND` command and the `WHERE` clause in order to separate `dept` and `WHERE` by one space in the SQL buffer.

```
SQL> APPEND SELECT * FROM dept
SQL> LIST
  1* SELECT * FROM dept
SQL> APPEND  WHERE deptno = 10
SQL> LIST
  1* SELECT * FROM dept WHERE deptno = 10
```

6.2.2.3 CHANGE

`CHANGE` is a line editor command performs a search-and-replace on the current line in the SQL buffer.

C[HANGE] /*from*/[*to*/]

If *to/* is specified, the first occurrence of text *from* in the current line is changed to text *to*. If *to/* is omitted, the first occurrence of text *from* in the current line is deleted.

The following sequence of commands makes line 3 the current line, then changes the department number in the WHERE clause from 20 to 30.

```
SQL> LIST
  1   SELECT empno, ename, job, sal, comm
  2   FROM emp
  3   WHERE deptno = 20
  4*  ORDER BY empno
SQL> 3
  3*  WHERE deptno = 20
SQL> CHANGE /20/30/
  3*  WHERE deptno = 30
SQL> LIST
  1   SELECT empno, ename, job, sal, comm
  2   FROM emp
  3   WHERE deptno = 30
  4*  ORDER BY empno
```

6.2.2.4 CLEAR

The CLEAR command removes the contents of the SQL buffer, deletes all column definitions set with the COLUMN command, or clears the screen.

```
CL[EAR ] [ BUFF[ER ] | SQL | COL[UMNS ] | SCR[EEN ] ]
```

BUFFER | SQL

Clears the SQL buffer.

COLUMNS

Removes column definitions.

SCREEN

Clears the screen. This is the default if no options are specified.

6.2.2.5 COLUMN

The COLUMN command controls output formatting. The formatting attributes set by using the COLUMN command remain in effect only for the duration of the current session.

```
COL[UMN ]
  [ column
    { CLE[AR ] |
      { FOR[MAT ] spec |
        HEA[DING ] text |
        { OFF | ON }
      } [...]
    }
```

```
]
```

If the `COLUMN` command is specified with no subsequent options, formatting options for current columns in effect for the session are displayed.

If the `COLUMN` command is followed by a column name, then the column name may be followed by one of the following:

- No other options
- `CLEAR`
- Any combination of `FORMAT`, `HEADING`, and one of `OFF` or `ON`

`column`

Name of a column in a table to which subsequent column formatting options are to apply. If no other options follow *`column`*, then the current column formatting options if any, of *`column`* are displayed.

`CLEAR`

The `CLEAR` option reverts all formatting options back to their defaults for *`column`*. If the `CLEAR` option is specified, it must be the only option specified.

`spec`

Format specification to be applied to *`column`*. For character columns, *`spec`* takes the following format:

`n`

`n` is a positive integer that specifies the column width in characters within which to display the data. Data in excess of *`n`* will wrap around with the specified column width.

For numeric columns, *`spec`* is comprised of the following elements.

Table 10-6-1 Numeric Column Format Elements

Element	Description
`$`	Display a leading dollar sign.
`,`	Display a comma in the indicated position.
`.`	Marks the location of the decimal point.
`0`	Display leading zeros.
`9`	Number of significant digits to display.

If loss of significant digits occurs due to overflow of the format, then all `#`'s are displayed.

`text`

Text to be used for the column heading of *`column`*.

`OFF | ON`

If `OFF` is specified, formatting options are reverted back to their defaults, but are still available within the session. If `ON` is specified, the formatting options specified by previous `COLUMN` commands for *`column`* within the session are re-activated.

The following example shows the effect of changing the display width of the `job` column.

```
SQL> SET PAGESIZE 9999
SQL> COLUMN job FORMAT A5
SQL> COLUMN job
COLUMN   JOB  ON
FORMAT   A5
wrapped
SQL> SELECT empno, ename, job FROM emp;

EMPNO ENAME      JOB
----- ---------- -----
 7369 SMITH      CLERK
 7499 ALLEN      SALES
                 MAN

 7521 WARD       SALES
                 MAN

 7566 JONES      MANAG
                 ER

 7654 MARTIN     SALES
                 MAN

 7698 BLAKE      MANAG
                 ER

 7782 CLARK      MANAG
                 ER

 7788 SCOTT      ANALY
                 ST

 7839 KING       PRESI
                 DENT

 7844 TURNER     SALES
                 MAN

 7876 ADAMS      CLERK
```

```
 7900 JAMES      CLERK
 7902 FORD       ANALY
                 ST

 7934 MILLER     CLERK

14 rows retrieved.
```

The following example applies a format to the `sal` column.

```
SQL> COLUMN sal FORMAT $99,999.00
SQL> COLUMN
COLUMN   JOB  ON
FORMAT   A5
wrapped

COLUMN   SAL  ON
FORMAT   $99,999.00
wrapped
SQL> SELECT empno, ename, job, sal FROM emp;

EMPNO ENAME      JOB           SAL
----- ---------- ----- -----------
 7369 SMITH      CLERK     $800.00
 7499 ALLEN      SALES   $1,600.00
                 MAN

 7521 WARD       SALES   $1,250.00
                 MAN

 7566 JONES      MANAG   $2,975.00
                 ER

 7654 MARTIN     SALES   $1,250.00
                 MAN

 7698 BLAKE      MANAG   $2,850.00
                 ER

 7782 CLARK      MANAG   $2,450.00
                 ER

 7788 SCOTT      ANALY   $3,000.00
                 ST

 7839 KING       PRESI   $5,000.00
                 DENT

 7844 TURNER     SALES   $1,500.00
                 MAN

 7876 ADAMS      CLERK   $1,100.00
 7900 JAMES      CLERK     $950.00
 7902 FORD       ANALY   $3,000.00
                 ST

 7934 MILLER     CLERK   $1,300.00

14 rows retrieved.
```

6.2.2.6 CONNECT

Change the database connection to a different user and/or connect to a different database. There must be no white space between any of the parameters following the `CONNECT` command.

```
CON[NECT] username[/password][@{connectstring | variable } ]
```

Where:

> *username* is a database username with which to connect to the database.
>
> *password* is the password associated with the specified *username*. If a *password* is not provided, but a password is required for authentication, EDB*Plus will prompt for the password.
>
> *connectstring* is the database connection string.
>
> *variable* is a variable defined in the `login.sql` file that contains a database connection string. The `login.sql` file can be found in the `edbplus` subdirectory of the Postgres Plus Advanced Server home directory.

In the following example, the database connection is changed to database `edb` on the localhost at port `5445` with username, `smith`.

```
SQL> CONNECT smith/mypassword@localhost:5445/edb
Disconnected from EnterpriseDB Database.
Connected to EnterpriseDB 9.4.0.0 (localhost:5445/edb) AS smith
```

From within the session shown above, the connection is changed to username `enterprisedb`. Also note that the host defaults to the localhost, the port defaults to 5444 (which is not the same as the port previously used), and the database defaults to `edb`.

```
SQL> CONNECT enterprisedb/password
Disconnected from EnterpriseDB Database.
Connected to EnterpriseDB 9.4.0.0 (localhost:5444/edb) AS enterprisedb
```

6.2.2.7 DEFINE

The `DEFINE` command creates or replaces the value of a *user variable* (also called a *substitution variable*).

```
DEF[INE ] [ variable [ = text ] ]
```

If the `DEFINE` command is given without any parameters, all current variables and their values are displayed.

If `DEFINE` *`variable`* is given, only *`variable`* is displayed with its value.

`DEFINE` *`variable`* `=` *`text`* assigns *`text`* to *`variable`*. *`text`* may be optionally enclosed within single or double quotation marks. Quotation marks must be used if *`text`* contains space characters.

The following example defines two variables, `dept` and `name`.

```
SQL> DEFINE dept = 20
SQL> DEFINE name = 'John Smith'
SQL> DEFINE
DEFINE EDB = "localhost:5445/edb"
DEFINE DEPT = "20"
DEFINE NAME = "John Smith"
```

Note: The variable `EDB` is read from the `login.sql` file located in the `edbplus` subdirectory of the Postgres Plus Advanced Server home directory.

6.2.2.8 DEL

`DEL` is a line editor command that deletes one or more lines from the SQL buffer.

```
DEL [ n | n m | n * | n L[AST ] | * | * n | * L[AST ] |
  L[AST ] ]
```

The parameters specify which lines are to be deleted from the SQL buffer. Two parameters specify the start and end of a range of lines to be deleted. If the `DEL` command is given with no parameters, the current line is deleted.

`n`

> *`n`* is an integer representing the *`n`*th line

`n m`

> *`n`* and *`m`* are integers where *`m`* is greater than *`n`* representing the *`n`*th through the *`m`*th lines

`*`

> Current line

`LAST`

> Last line

In the following example, the fifth and sixth lines containing columns `sal` and `comm`, respectively, are deleted from the `SELECT` command in the SQL buffer.

```
SQL> LIST
  1  SELECT
  2    empno
  3   ,ename
  4   ,job
  5   ,sal
  6   ,comm
  7   ,deptno
  8* FROM emp
SQL> DEL 5 6
SQL> LIST
  1  SELECT
  2    empno
  3   ,ename
  4   ,job
  5   ,deptno
  6* FROM emp
```

6.2.2.9 DESCRIBE

The `DESCRIBE` command displays:

- A list of columns, column data types, and column lengths for a table or view
- A list of parameters for a procedure or function
- A list of procedures and functions and their respective parameters for a package.

The `DESCRIBE` command will also display the structure of the database object referred to by a synonym. The syntax is:

```
DESC[RIBE] [ schema.]object
```

schema

Name of the schema containing the object to be described.

object

Name of the table, view, procedure, function, or package to be displayed, or the synonym of an object.

6.2.2.10 DISCONNECT

The `DISCONNECT` command closes the current database connection, but does not terminate EDB*Plus.

```
DISC[ONNECT ]
```


6.2.2.11 EDIT

The `EDIT` command invokes an external editor to edit the contents of an operating system file or the SQL buffer.

```
ED[IT ] [ filename[.ext ] ]
```

filename[.*ext*]

> *filename* is the name of the file to open with an external editor. *ext* is the filename extension. If the filename extension is `sql`, then the `.sql` extension may be omitted when specifying *filename*. `EDIT` always assumes a `.sql` extension on filenames that are specified with no extension. If the filename parameter is omitted from the `EDIT` command, the contents of the SQL buffer are brought into the editor.

6.2.2.12 EXECUTE

The `EXECUTE` command executes an SPL procedure from EDB*Plus.

```
EXEC[UTE ] spl_procedure [ ([ parameters ]) ]
```

spl_procedure

> The name of the SPL procedure to be executed.

parameters

> Comma-delimited list of parameters. If there are no parameters, then a pair of empty parentheses may optionally be specified.

6.2.2.13 EXIT

The `EXIT` command terminates the EDB*Plus session and returns control to the operating system. `QUIT` is a synonym for `EXIT`. Specifying no parameters is equivalent to `EXIT SUCCESS COMMIT`.

```
{ EXIT | QUIT }
  [ SUCCESS | FAILURE | WARNING | value |variable ]
  [ COMMIT | ROLLBACK ]SUCCESS | FAILURE |WARNING
```

Returns an operating system dependent return code indicating successful operation, failure, or warning for `SUCCESS`, `FAILURE`, and `WARNING`, respectively. The default is `SUCCESS`.

value

An integer value that is returned as the return code.

variable

A variable created with the `DEFINE` command whose value is returned as the return code.

`COMMIT | ROLLBACK`

If `COMMIT` is specified, uncommitted updates are committed upon exit. If `ROLLBACK` is specified, uncommitted updates are rolled back upon exit. The default is `COMMIT`.

6.2.2.14 GET

The `GET` command loads the contents of the given file to the SQL buffer.

```
GET filename[.ext ] [ LIS[T ] | NOL[IST ] ]
```

filename`[.`*ext* `]`

filename is the name of the file to load into the SQL buffer. *ext* is the filename extension. If the filename extension is `sql`, then the `.sql` extension may be omitted when specifying *filename*. `GET` always assumes a `.sql` extension on filenames that are specified with no extension.

`LIST | NOLIST`

If `LIST` is specified, the content of the SQL buffer is displayed after the file is loaded. If `NOLIST` is specified, no listing is displayed. The default is `LIST`.

6.2.2.15 HELP

The `HELP` command obtains an index of topics or help on a specific topic. The question mark (`?`) is synonymous with specifying `HELP`.

```
{ HELP | ? } { INDEX | topic }
```

`INDEX`

Displays an index of available topics.

topic

The name of a specific topic – e.g., an EDB*Plus command, for which help is desired.

6.2.2.16 HOST

The `HOST` command executes an operating system command from EDB*Plus.

```
HO[ST ] [os_command]
```

os_command

The operating system command to be executed. If you do not provide an operating system command, EDB*Plus pauses execution and opens a new shell prompt. When the shell exits, EDB*Plus resumes execution.

6.2.2.17 INPUT

The `INPUT` line editor command adds a line of text to the SQL buffer after the current line.

```
I[NPUT ] text
```

The following sequence of `INPUT` commands constructs a `SELECT` command.

```
SQL> INPUT SELECT empno, ename, job, sal, comm
SQL> INPUT FROM emp
SQL> INPUT WHERE deptno = 20
SQL> INPUT ORDER BY empno
SQL> LIST
  1  SELECT empno, ename, job, sal, comm
  2  FROM emp
  3  WHERE deptno = 20
  4* ORDER BY empno
```

6.2.2.18 LIST

`LIST` is a line editor command that displays the contents of the SQL buffer.

```
L[IST] [ n | n m | n * | n L[AST] | * | * n | * L[AST] |
  L[AST] ]
```

The buffer does not include a history of the EDB*Plus commands.

n

n represents the buffer line number.

n m

n m displays a list of lines between n and m.

n *

n * displays a list of lines that range between line *n* and the current line.

n L[AST]

n L[AST] displays a list of lines that range from line *n* through the last line in the buffer.

*

* displays the current line.

* *n*

* *n* displays a list of lines that range from the current line through line *n*.

* L[AST]

* L[AST] displays a list of lines that range from the current line through the last line.

L[AST]

L[AST] displays the last line.

6.2.2.19 PASSWORD

Use the PASSWORD command to change your database password.

```
PASSW[ORD] [user_name]
```

You must have sufficient privileges to use the PASSWORD command to change another user's password. The following example demonstrates using the PASSWORD command to change the password for a user named acctg:

```
SQL> PASSWORD acctg
Changing password for acctg
    New password:
```



```
    New password again:
Password successfully changed.
```

6.2.2.20 PAUSE

The `PAUSE` command displays a message, and waits for the user to press `ENTER`.

```
PAU[SE]  [optional_text]
```

`optional_text` specifies the text that will be displayed to the user. If the *`optional_text`* is omitted, Advanced Server will display two blank lines. If you double quote the *`optional_text`* string, the quotes will be included in the output.

6.2.2.21 PRINT

The `PRINT` command displays the value of a bind variable.

```
PRI[NT] [bind_variable_name]
```

`bind_variable_name` specifies the name of a bind variable. Omit *`bind_variable_name`* to generate a list that includes the values of all bind variables.

6.2.2.22 PROMPT

The `PROMPT` command displays a message to the user before continuing.

```
PRO[MPT] [message_text]
```

`message_text` specifies the text displayed to the user. Double quote the string to include quotes in the output.

6.2.2.23 QUIT

The `QUIT` command terminates the session and returns control to the operating system. `QUIT` is a synonym for `EXIT`.

```
QUIT
  [SUCCESS | FAILURE | WARNING | value | sub_variable]
  [COMMIT | ROLLBACK]
```

The default value is `QUIT SUCCESS COMMIT`.

6.2.2.24 REMARK

Use `REMARK` to include comments in a script.

```
REM[ARK] [optional_text]
```

You may also use the following convention to include a comment:

```
/*
 *  This is an example of a three line comment.
 */
```

6.2.2.25 SAVE

Use the `SAVE` command to write the SQL Buffer to an operating system file.

```
SAV[E] file_name
  [CRE[ATE] | REP[LACE] | APP[END]]
```

file_name

> *file_name* specifies the name of the file (including the path) where the buffer contents are written. If you do not provide a file extension, `.sql` is appended to the end of the file name.

CREATE

> Include the `CREATE` keyword to create a new file. A new file is created *only* if a file with the specified name does not already exist. This is the default.

REPLACE

> Include the `REPLACE` keyword to specify that Advanced Server should overwrite an existing file.

APPEND

> Include the `APPEND` keyword to specify that Advanced Server should append the contents of the SQL buffer to the end of the specified file.

The following example saves the contents of the SQL buffer to a file named `example.sql`, located in the `temp` directory:

```
SQL> SAVE C:\example.sql CREATE
File "example.sql" written.
```

6.2.2.26 SET

Use the `SET` command to specify a value for a session level variable that controls EDB*Plus behavior. The following forms of the `SET` command are valid:

SET AUTOCOMMIT

Use the `SET AUTOCOMMIT` command to specify `COMMIT` behavior for Advanced Server transactions.

```
SET AUTO[COMMIT]
  {ON | OFF | IMMEDIATE | statement_count}
```

Please note that EDB*Plus always automatically commits DDL statements.

`ON`

Specify `ON` to turn `AUTOCOMMIT` behavior on.

`OFF`

Specify `OFF` to turn `AUTOCOMMIT` behavior off.

`IMMEDIATE`

`IMMEDIATE` has the same effect as `ON`.

`statement_count`

Include a value for *`statement_count`* to instruct EDB*Plus to issue a commit after the specified count of successful SQL statements.

SET COLUMN SEPARATOR

Use the `SET COLUMN SEPARATOR` command to specify the text that Advanced Server displays between columns.

```
SET COLSEP column_separator
```

The default value of *`column_separator`* is a single space.

SET ECHO

Use the `SET ECHO` command to specify if SQL and EDB*Plus script statements should be displayed onscreen as they are executed.


```
SET ECHO {ON | OFF}
```

The default value is `OFF`.

SET FEEDBACK

The `SET FEEDBACK` command controls the display of interactive information after a SQL statement executes.

```
SET FEED[BACK] {ON | OFF | row_threshold}
```

row_threshold

> Specify an integer value for *row_threshold*. Setting *row_threshold* to `0` is same as setting `FEEDBACK` to `OFF`. Setting *row_threshold* equal `1` effectively sets `FEEDBACK` to `ON`.

SET FLUSH

Use the `SET FLUSH` command to control display buffering.

```
SET FLU[SH] {ON | OFF}
```

Set `FLUSH` to `OFF` to enable display buffering. If you enable buffering, messages bound for the screen may not appear until the script completes. Please note that setting `FLUSH` to `OFF` will offer better performance.

Set `FLUSH` to `ON` to disable display buffering. If you disable buffering, messages bound for the screen appear immediately.

SET HEADING

Use the `SET HEADING` variable to specify if Advanced Server should display column headings for `SELECT` statements.

```
SET HEA[DING] {ON | OFF}
```

SET HEAD SEPARATOR

The `SET HEADSEP` command sets the new heading separator character used by the `COLUMN HEADING` command. The default is '|'.

```
SET HEADS[EP]
```


SET LINESIZE

Use the `SET LINESIZE` command to specify the width of a line in characters.

```
SET LIN[ESIZE] width_of_line
```

width_of_line

The default value of *width_of_line* is `132`.

SET NEWPAGE

Use the `SET NEWPAGE` command to specify how many blank lines are printed after a page break.

```
SET NEWP[AGE] lines_per_page
```

lines_per_page

The default value of *lines_per_page* is `1`.

SET NULL

Use the `SET NULL` command to specify a string that is displayed to the user when a `NULL` column value is displayed in the output buffer.

```
SET NULL null_string
```

SET PAGESIZE

Use the `SET PAGESIZE` command to specify the number of printed lines that fit on a page.

```
SET PAGES[IZE] line_count
```

Use the *line_count* parameter to specify the number of lines per page.

SET SQLCASE

The `SET SQLCASE` command specifies if SQL statements transmitted to the server should be converted to upper or lower case.

```
SET SQLC[ASE] {MIX[ED] | UP[PER] | LO[WER]}
```

`UPPER`

Specify `UPPER` to convert the command text to uppercase.

`LOWER`

Specify `LOWER` to convert the command text to lowercase.

`MIXED`

Specify `MIXED` to leave the case of SQL commands unchanged. The default is `MIXED`.

SET PAUSE

The `SET PAUSE` command is most useful when included in a script; the command displays a prompt and waits for the user to press `Return`.

```
SET PAU[SE] {ON | OFF}
```

If `SET PAUSE` is `ON`, the message `Hit ENTER to continue…` will be displayed before each command is executed.

SET SPACE

Use the `SET SPACE` command to specify the number of spaces to display between columns:

```
SET SPACE number_of_spaces
```

SET SQLPROMPT

Use `SET SQLPROMPT` to set a value for a user-interactive prompt:

```
SET SQLP[ROMPT] "prompt"
```

By default, `SQLPROMPT` is set to `"SQL> "`

SQL TERMOUT

Use the `SQL TERMOUT` command to specify if command output should be displayed onscreen.

```
SET TERM[OUT] {ON | OFF}
```

SQL TIMING

The `SQL TIMING` command specifies if Advanced Server should display the execution time for each SQL statement after it is executed.

```
SET TIMI[NG] {ON | OFF}
```

SET VERIFY

Specifies if both the old and new values of a SQL statement are displayed when a substitution variable is encountered.

```
SET VER[IFY] { ON | OFF }
```

6.2.2.27 SHOW

Use the `SHOW` command to display current parameter values.

```
SHO[W] {ALL | parameter_name}
```

Display the current parameter settings by including the `ALL` keyword:

```
SQL> SHOW ALL
autocommit      OFF
colsep          " "
define          "&"
echo            OFF
FEEDBACK ON for 6 row(s).
flush           ON
heading         ON
headsep         "|"
linesize        78
newpage         1
null            " "
pagesize        14
pause           OFF
serveroutput    OFF
spool           OFF
sqlcase         MIXED
sqlprompt       "SQL> "
sqlterminator   ";"
suffix          ".sql"
termout         ON
timing          OFF
verify          ON
USER is         "enterprisedb"
HOST is         "localhost"
PORT is         "5444"
DATABASE is     "edb"
VERSION is      "9.4.0.0"
```

Or display a specific parameter setting by including the *`parameter_name`* in the `SHOW` command:

```
SQL> SHOW VERSION
VERSION is "9.4.0.0"
```

6.2.2.28 SPOOL

The `SPOOL` command sends output from the display to a file.

```
SP[OOL] output_file | OFF
```

Use the *`output_file`* parameter to specify a path name for the output file.

6.2.2.29 START

Use the `START` command to run an EDB*Plus script file; `START` is an alias for `@` command.

```
STA[RT] script_file
```

Specify the name of a script file in the *`script_file`* parameter.

6.2.2.30 UNDEFINE

The `UNDEFINE` command erases a user variable created by the `DEFINE` command.

```
UNDEF[INE] variable_name [ variable_name...]
```

Use the *`variable_name`* parameter to specify the name of a variable or variables.

6.2.2.31 WHENEVER SQLERROR

The `WHENEVER SQLERROR` command provides error handling for SQL errors or PL/SQL block errors. The syntax is:

```
WHENEVER SQLERROR
  {CONTINUE [COMMIT|ROLLBACK|NONE]
  |EXIT [SUCCESS|FAILURE|WARNING|n|sub_variable]
  [COMMIT|ROLLBACK]}
```

If Advanced Server encounters an error during the execution of a SQL command or PL/SQL block, EDB*Plus performs the action specified in the `WHENEVER SQLERROR` command:

> Include the `CONTINUE` clause to instruct EDB*Plus to perform the specified action before continuing.

Include the `COMMIT` clause to instruct EDB*Plus to `COMMIT` the current transaction before exiting or continuing.

Include the `ROLLBACK` clause to instruct EDB*Plus to `ROLLBACK` the current transaction before exiting or continuing.

Include the `NONE` clause to instruct EDB*Plus to continue without committing or rolling back the transaction.

Include the `EXIT` clause to instruct EDB*Plus to perform the specified action and exit if it encounters an error.

Use the following options to specify a status code that EDB*Plus will return before exiting:

```
[SUCCESS|FAILURE|WARNING|n|sub_variable]
```

Please note that EDB*Plus supports substitution variables, but does not support bind variables.

6.3 libpq C Library

libpq is the C application programmer's interface to Postgres Plus Advanced Server. libpq is a set of library functions that allow client programs to pass queries to the Postgres Plus Advanced Server and to receive the results of these queries.

libpq is also the underlying engine for several other EnterpriseDB application interfaces including those written for C++, Perl, Python, Tcl and ECPG. So some aspects of libpq's behavior will be important to the user if one of those packages is used.

Client programs that use libpq must include the header file `libpq-fe.h` and must link with the `libpq` library.

6.3.1 Using libpq with EnterpriseDB SPL

The EnterpriseDB SPL language can be used with the libpq interface library, providing support for:

- Procedures, functions, packages
- Prepared statements
- `REFCURSOR`s
- Static cursors
- `structs` and `typedefs`
- Arrays
- DML and DDL operations
- `IN/OUT/IN OUT` parameters

6.3.2 REFCURSOR Support

In earlier releases, Advanced Server provided support for REFCURSORs through the following libpq functions; these functions should now be considered deprecated:

- `PQCursorResult()`
- `PQgetCursorResult()`
- `PQnCursor()`

You may now use `PQexec()` and `PQgetvalue()` to retrieve a `REFCURSOR` returned by an SPL (or PL/pgSQL) function. A `REFCURSOR` is returned in the form of a null-terminated string indicating the name of the cursor. Once you have the name of the cursor, you can execute one or more `FETCH` statements to retrieve the values exposed through the cursor.

Please note that the samples that follow do not include error-handling code that would be required in a real-world client application.

Returning a Single REFCURSOR

The following example shows an SPL function that returns a value of type `REFCURSOR`:

```
CREATE OR REPLACE FUNCTION getEmployees(p_deptno NUMERIC)
RETURN REFCURSOR AS
  result REFCURSOR;
BEGIN
  OPEN result FOR SELECT * FROM emp WHERE deptno = p_deptno;

  RETURN result;
END;
```

This function expects a single parameter, `p_deptno`, and returns a `REFCURSOR` that holds the result set for the `SELECT` query shown in the `OPEN` statement. The `OPEN` statement executes the query and stores the result set in a cursor. The server constructs a name for that cursor and stores the name in a variable (named `result`). The function then returns the name of the cursor to the caller.

To call this function from a C client using libpq, you can use `PQexec()` and `PQgetvalue()`:

```
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void fetchAllRows(PGconn *conn,
                         const char *cursorName,
                         const char *description);
static void fail(PGconn *conn, const char *msg);

int
main(int argc, char *argv[])
{
  PGconn    *conn = PQconnectdb(argv[1]);
  PGresult  *result;

  if (PQstatus(conn) != CONNECTION_OK)
    fail(conn, PQerrorMessage(conn));

  result = PQexec(conn, "BEGIN TRANSACTION");

  if (PQresultStatus(result) != PGRES_COMMAND_OK)
    fail(conn, PQerrorMessage(conn));

  PQclear(result);
```

```
  result = PQexec(conn, "SELECT * FROM getEmployees(10)");

  if (PQresultStatus(result) != PGRES_TUPLES_OK)
    fail(conn, PQerrorMessage(conn));

  fetchAllRows(conn, PQgetvalue(result, 0, 0), "employees");

  PQclear(result);

  PQexec(conn, "COMMIT");

  PQfinish(conn);

  exit(0);
}

static void
fetchAllRows(PGconn *conn,
             const char *cursorName,
             const char *description)
{
  size_t commandLength = strlen("FETCH ALL FROM ") +
                         strlen(cursorName) + 3;

  char     *commandText = malloc(commandLength);
  PGresult *result;
  int       row;

  sprintf(commandText, "FETCH ALL FROM \"%s\"", cursorName);

  result = PQexec(conn, commandText);

  if (PQresultStatus(result) != PGRES_TUPLES_OK)
    fail(conn, PQerrorMessage(conn));

  printf("-- %s --\n", description);

  for (row = 0; row < PQntuples(result); row++)
  {
    const char *delimiter = "\t";
    int         col;

    for (col = 0; col < PQnfields(result); col++)
    {
      printf("%s%s", delimiter, PQgetvalue(result, row, col));
      delimiter = ",";
    }

    printf("\n");
  }
```



```
  PQclear(result);
  free(commandText);
}

static void
fail(PGconn *conn, const char *msg)
{
  fprintf(stderr, "%s\n", msg);

  if (conn != NULL)
    PQfinish(conn);

  exit(-1);
}
```

The code sample contains a line of code that calls the `getEmployees()` function, and returns a result set that contains all of the employees in department `10`:

```
result = PQexec(conn, "SELECT * FROM getEmployees(10)");
```

The `PQexec()` function returns a result set handle to the C program. The result set will contain exactly one value; that value is the name of the cursor as returned by `getEmployees()`.

Once you have the name of the cursor, you can use the SQL `FETCH` statement to retrieve the rows in that cursor. The function `fetchAllRows()` builds a `FETCH ALL` statement, executes that statement, and then prints the result set of the `FETCH ALL` statement.

The output of this program is shown below:

```
-- employees --
    7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
    7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
    7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
```

Returning Multiple REFCURSORs

The next example returns two `REFCURSOR`s:

- The first `REFCURSOR` contains the name of a cursor (`employees`) that contains all employees who work in a department within the range specified by the caller.

- The second `REFCURSOR` contains the name of a cursor (`departments`) that contains all of the departments in the range specified by the caller.

In this example, instead of returning a single `REFCURSOR`, the function returns a `SETOF REFCURSOR` (which means `0` or more `REFCURSORS`). One other important difference is that the libpq program should not expect a single `REFCURSOR` in the result set, but should expect two rows, each of which will contain a single value (the first row contains the name of the `employees` cursor, and the second row contains the name of the `departments` cursor).

```
CREATE OR REPLACE FUNCTION getEmpsAndDepts(p_min NUMERIC,
                                           p_max NUMERIC)
RETURN SETOF REFCURSOR AS
  employees   REFCURSOR;
  departments REFCURSOR;
BEGIN
  OPEN employees FOR
    SELECT * FROM emp WHERE deptno BETWEEN p_min AND p_max;
  RETURN NEXT employees;

  OPEN departments FOR
    SELECT * FROM dept WHERE deptno BETWEEN p_min AND p_max;
  RETURN NEXT departments;
END;
```

As in the previous example, you can use `PQexec()` and `PQgetvalue()` to call the SPL function:

```
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void fetchAllRows(PGconn *conn,
                         const char *cursorName,
                         const char *description);
static void fail(PGconn *conn, const char *msg);

int
main(int argc, char *argv[])
{
  PGconn   *conn = PQconnectdb(argv[1]);
  PGresult *result;

  if (PQstatus(conn) != CONNECTION_OK)
    fail(conn, PQerrorMessage(conn));

  result = PQexec(conn, "BEGIN TRANSACTION");
```

```
  if (PQresultStatus(result) != PGRES_COMMAND_OK)
    fail(conn, PQerrorMessage(conn));

  PQclear(result);

  result = PQexec(conn, "SELECT * FROM getEmpsAndDepts(20, 30)");

  if (PQresultStatus(result) != PGRES_TUPLES_OK)
    fail(conn, PQerrorMessage(conn));

  fetchAllRows(conn, PQgetvalue(result, 0, 0), "employees");
  fetchAllRows(conn, PQgetvalue(result, 1, 0), "departments");

  PQclear(result);

  PQexec(conn, "COMMIT");

  PQfinish(conn);

  exit(0);
}

static void
fetchAllRows(PGconn *conn,
             const char *cursorName,
             const char *description)
{
  size_t     commandLength = strlen("FETCH ALL FROM ") +
                             strlen(cursorName) + 3;
  char      *commandText   = malloc(commandLength);
  PGresult  *result;
  int        row;

  sprintf(commandText, "FETCH ALL FROM \"%s\"", cursorName);

  result = PQexec(conn, commandText);

  if (PQresultStatus(result) != PGRES_TUPLES_OK)
  fail(conn, PQerrorMessage(conn));

  printf("-- %s --\n", description);

  for (row = 0; row < PQntuples(result); row++)
  {
    const char *delimiter = "\t";
    int         col;

    for (col = 0; col < PQnfields(result); col++)
    {
      printf("%s%s", delimiter, PQgetvalue(result, row, col));
      delimiter = ",";
```

```
    }

    printf("\n");
  }

  PQclear(result);
  free(commandText);
}

static void
fail(PGconn *conn, const char *msg)
{
  fprintf(stderr, "%s\n", msg);

  if (conn != NULL)
    PQfinish(conn);

  exit(-1);
}
```

If you call `getEmpsAndDepts(20, 30)`, the server will return a cursor that contains all employees who work in department `20` or `30`, and a second cursor containing the description of departments `20` and `30`.

```
-- employees --
  7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
  7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
  7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
  7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
  7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
  7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
  7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
  7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
  7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
  7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
  7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
-- departments --
  20,RESEARCH,DALLAS
  30,SALES,CHICAGO
```

6.3.3 Array Binding

Advanced Server's array binding functionality allows you to send an array of data across the network in a single round-trip. When the back end receives the bulk data, it can use the data to perform insert or update operations.

Perform bulk operations with a prepared statement; use the following function to prepare the statement:

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

Details of `PQprepare()` can be found in the prepared statement section.

The following functions can be used to perform bulk operations:

- PQBulkStart
- PQexecBulk
- PQBulkFinish
- PQexecBulkPrepared

6.3.3.1 PQBulkStart

`PQBulkStart()` initializes bulk operations on the server. You must call this function before sending bulk data to the server. `PQBulkStart()` initializes the prepared statement specified in stmtName to receive data in a format specified by `paramFmts`.

API Definition

```
PGresult * PQBulkStart(PGconn *conn,
                       const char * Stmt_Name,
                       unsigned int nCol,
                       const int *paramFmts);
```

6.3.3.2 PQexecBulk

`PQexecBulk()` is used to supply data (`paramValues`) for a statement that was previously initialized for bulk operation using `PQBulkStart()`.

This function can be used more than once after `PQBulkStart()` to send multiple blocks of data. See the example for more details.

API Definition

```
PGresult *PQexecBulk(PGconn *conn,
                     unsigned int nRows,
                     const char *const * paramValues,
                     const int *paramLengths);
```

6.3.3.3 PQBulkFinish

This function completes the current bulk operation. You can use the prepared statement again without re-preparing it.

API Definition

```
PGresult *PQBulkFinish(PGconn *conn);
```

6.3.3.4 PQexecBulkPrepared

Alternatively, you can use the `PQexecBulkPrepared()` function to perform a bulk operation with a single function call. `PQexecBulkPrepared()` sends a request to execute a prepared statement with the given parameters, and waits for the result. This function combines the functionality of `PQbulkStart()`, `PQexecBulk()`, and `PQBulkFinish()`. When using this function, you are not required to initialize or terminate the bulk operation; this function starts the bulk operation, passes the data to the server, and terminates the bulk operation.

Specify a previously prepared statement in the place of `stmtName`. Commands that will be used repeatedly will be parsed and planned just once, rather than each time they are executed.

API Definition

```
PGresult *PQexecBulkPrepared(PGconn *conn,
                             const char *stmtName,
                             unsigned int nCols,
                             unsigned int nRows,
                             const char *const *paramValues,
                             const int *paramLengths,
                             const int *paramFormats);
```

6.3.3.5 Example Code (Using PQBulkStart, PQexecBulk, PQBulkFinish)

The following example uses `PGBulkStart`, `PQexecBulk`, and `PQBulkFinish`.

```
void InsertDataUsingBulkStyle( PGconn *conn )
{
```

```
    PGresult              *res;
    Oid                   paramTypes[2];
    char                  *paramVals[5][2];
    int                   paramLens[5][2];
    int                   paramFmts[2];
    int                   i;

    int                   a[5] = { 10, 20, 30, 40, 50 };
    char                  b[5][10] = { "Test_1", "Test_2", "Test_3",  "Test_4",
"Test_5" };


    paramTypes[0] = 23;
    paramTypes[1] = 1043;
    res = PQprepare( conn, "stmt_1", "INSERT INTO testtable1 values( $1, $2
)", 2, paramTypes );
    PQclear( res );

    paramFmts[0] = 1;   /* Binary format */
    paramFmts[1] = 0;

    for( i = 0; i < 5; i++ )
    {
        a[i] = htonl( a[i] );
        paramVals[i][0] = &(a[i]);
        paramVals[i][1] = b[i];

        paramLens[i][0] = 4;
        paramLens[i][1] = strlen( b[i] );
    }

     res = PQBulkStart(conn, "stmt_1", 2, paramFmts);
     PQclear( res );
     printf( "< -- PQBulkStart -- >\n" );

     res = PQexecBulk(conn, 5, (const char *const *)paramVals, (const int
*)paramLens);
     PQclear( res );
     printf( "< -- PQexecBulk -- >\n" );

     res = PQexecBulk(conn, 5, (const char *const *)paramVals, (const int
*)paramLens);
     PQclear( res );
     printf( "< -- PQexecBulk -- >\n" );

     res = PQBulkFinish(conn);
     PQclear( res );
     printf( "< -- PQBulkFinish -- >\n" );
}
```

6.3.3.6 Example Code (Using PQexecBulkPrepared)

The following example uses `PQexecBulkPrepared`.

```
void InsertDataUsingBulkStyleCombinedVersion( PGconn *conn )
{
     PGresult              *res;
     Oid                   paramTypes[2];
     char                  *paramVals[5][2];
```

```
    int                 paramLens[5][2];
    int                 paramFmts[2];
    int                 i;

    int                 a[5] = { 10, 20, 30, 40, 50 };
    char                b[5][10] = { "Test_1", "Test_2", "Test_3", "Test_4",
"Test_5" };

    paramTypes[0] = 23;
    paramTypes[1] = 1043;
    res = PQprepare( conn, "stmt_2", "INSERT INTO testtable1 values( $1, $2
)", 2, paramTypes );
    PQclear( res );

    paramFmts[0] = 1;   /* Binary format */
    paramFmts[1] = 0;

    for( i = 0; i < 5; i++ )
    {
          a[i] = htonl( a[i] );
          paramVals[i][0] = &(a[i]);
          paramVals[i][1] = b[i];

          paramLens[i][0] = 4;
          paramLens[i][1] = strlen( b[i] );
    }

res = PQexecBulkPrepared(conn, "stmt_2", 2, 5, (const char *const
*)paramVals,(const int *)paramLens, (const int *)paramFmts);
    PQclear( res );
}
```


6.4 ECPGPlus

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) to create ECPGPlus. ECPGPlus allows you to include embedded SQL commands in C applications; when you use ECPGPlus to compile an application that contains embedded SQL commands, the SQL code is syntax-checked and translated into C.

ECPGPlus supports Pro*C compatible syntax in C programs when connected to an Advanced Server database. ECPGPlus supports:

- Oracle Dynamic SQL – Method 4 (ODS-M4)
- Pro*C compatible anonymous blocks
- An Oracle-compatible `CALL` statement

As part of ECPGPlus's Pro*C compatibility, you do not need to include the `BEGIN DECLARE SECTION` and `END DECLARE SECTION` directives.

For more information about using ECPGPlus, please see the Postgres Plus Advanced Server ECPG Connector Guide, available from the EnterpriseDB website at:

http://www.enterprisedb.com/documentation/english

6.4.1 C-preprocessor Directives

The ECPGPlus C-preprocessor enforces two behaviors that are dependent on the mode in which you invoke ECPGPlus:

- `PROC` mode
- non-`PROC` mode

Compiling in PROC mode

In `PROC` mode, ECPGPlus allows you to:

- Declare host variables outside of an `EXEC SQL BEGIN/END DECLARE SECTION`.
- Use any C variable as a host variable as long as it is of a data type compatible with ECPG.

When you invoke ECPGPlus in `PROC` mode (by including the `-C PROC` keywords), the ECPG compiler honors the following C-preprocessor directives:

```
#include
#if expression
#ifdef symbolName
#ifndef symbolName
#else
#elif expression
#endif
#define symbolName expansion
#define symbolName([macro arguments]) expansion
#undef symbolName
#defined(symbolName)
```

Pre-processor directives are used to effect or direct the code that is received by the compiler. For example, using the following code sample:

```
#if HAVE_LONG_LONG == 1
#define BALANCE_TYPE long long
#else
#define BALANCE_TYPE double
#endif
...
BALANCE_TYPE customerBalance;
```

If you invoke ECPGPlus with the following command-line arguments:

```
ecpg -C PROC -DHAVE_LONG_LONG=1
```

ECPGPlus will copy the entire fragment (without change) to the output file, but will only send the following tokens to the ECPG parser:

```
long long customerBalance;
```

On the other hand, if you invoke ECPGPlus with the following command-line arguments:

```
ecpg -C PROC -DHAVE_LONG_LONG=0
```

The ECPG parser will receive the following tokens:

```
double customerBalance;
```

If your code uses preprocessor directives to filter the code that is sent to the compiler, the complete code is retained in the original code, while the ECPG parser sees only the processed token stream.

Compiling in non-PROC mode

If you do not include the `-C PROC` command-line option:

- C preprocessor directives are copied to the output file without change.
- You must declare the type and name of each C variable that you intend to use as a host variable within an `EXEC SQL BEGIN/END DECLARE` section.

When invoked in non-`PROC` mode, ECPG implements the behavior described in the PostgreSQL Core documentation, available at:

http://www.enterprisedb.com/documentation/english

6.4.2 Supported C Data Types

An ECPGPlus application must deal with two sets of data types: SQL data types (such as `SMALLINT`, `DOUBLE PRECISION` and `CHARACTER VARYING`) and C data types (like `short`, `double` and `varchar[n]`). When an application fetches data from the server, ECPGPlus will map each SQL data type to the type of the C variable into which the data is returned.

In general, ECPGPlus can convert most SQL server types into similar C types, but not all combinations are valid. For example, ECPGPlus will try to convert a SQL character value into a C integer value, but the conversion may fail (at execution time) if the SQL character value contains non-numeric characters.

The reverse is also true; when an application sends a value to the server, ECPGPlus will try to convert the C data type into the required SQL type. Again, the conversion may fail (at execution time) if the C value cannot be converted into the required SQL type.

ECPGPlus can convert any SQL type into C character values (`char[n]` or `varchar[n]`). Although it is safe to convert any SQL type to/from `char[n]` or `varchar[n]`, it is often convenient to use more natural C types such as `int`, `double`, or `float`.

The supported C data types are:

- `short`
- `int`
- `unsigned int`
- `long long int`
- `float`
- `double`
- `char[n+1]`
- `varchar[n+1]`
- `bool`
- and any equivalent created by a `typedef`

In addition to the numeric and character types supported by C, the `pgtypeslib` run-time library offers custom data types (and functions to operate on those types) for dealing with date/time and exact numeric values:

- `timestamp`
- `interval`
- `date`
- `decimal`

- `numeric`

To use a data type supplied by `pgtypeslib`, you must `#include` the proper header file.

6.4.3 Type Codes

The following table contains the type codes for *external* data types. An external data type is used to indicate the type of a C host variable. When an application binds a value to a parameter or binds a buffer to a `SELECT`-list item, the type code in the corresponding SQLDA descriptor (`descriptor->T[column]`) should be set to one of the following values:

Type Code	Host Variable Type (C Data Type)
1, 2, 8, 11, 12, 15, 23, 24, 91, 94, 95, 96, 97	char[]
3	int
4, 7, 21	float
5, 6	null-terminated string (char[length+1])
9	varchar
22	double
68	unsigned int

The following table contains the type codes for *internal* data types. An internal type code is used to indicate the type of a value as it resides in the database. The `DESCRIBE SELECT LIST` statement populates the data type array (`descriptor->T[column]`) using the following values.

Internal Type Code	Server Type
1	VARCHAR2
2	NUMBER
8	LONG
11	ROWID
12	DATE
23	RAW
24	LONG RAW
96	CHAR
100	BINARY FLOAT
101	BINARY DOUBLE
104	UROWID
187	TIMESTAMP
188	TIMESTAMP W/TIMEZONE
189	INTERVAL YEAR TO MONTH
190	INTERVAL DAY TO SECOND
232	TIMESTAMP LOCAL_TZ

6.4.4 The SQLDA Structure

Oracle Dynamic SQL method 4 uses the SQLDA data structure to hold the data and metadata for a dynamic SQL statement. A SQLDA structure can describe a set of input parameters corresponding to the parameter markers found in the text of a dynamic statement or the result set of a dynamic statement. The layout of the SQLDA structure is:

```
struct SQLDA
{
  int     N;     /* Number of entries              */
  char  **V;     /* Variables                      */
  int    *L;     /* Variable lengths               */
  short  *T;     /* Variable types                 */
  short **I;     /* Indicators                     */
  int     F;     /* Count of variables discovered by DESCRIBE */
  char  **S;     /* Variable names                 */
  short  *M;     /* Variable name maximum lengths  */
  short  *C;     /* Variable name actual lengths   */
  char  **X;     /* Indicator names                */
  short  *Y;     /* Indicator name maximum lengths */
  short  *Z;     /* Indicator name actual lengths  */
};
```

Parameters

`N` - *maximum number of entries*

The `N` structure member contains the maximum number of entries that the SQLDA may describe. This member is populated by the `sqlald()` function when you allocate the SQLDA structure. Before using a descriptor in an `OPEN` or `FETCH` statement, you must set `N` to the *actual* number of values described.

`V` - *data values*

The `V` structure member is a pointer to an array of data values.

> For a `SELECT`-list descriptor, `V` points to an array of values returned by a `FETCH` statement (each member in the array corresponds to a column in the result set).
>
> For a bind descriptor, `V` points to an array of parameter values (you must populate the values in this array before opening a cursor that uses the descriptor).

Your application must allocate the space required to hold each value.

`L` – *length of each data value*

The `L` structure member is a pointer to an array of lengths. Each member of this array must indicate the amount of memory available in the corresponding member of the `V` array. For example, if `V[5]` points to a buffer large enough to hold a 20-byte NULL-terminated string, `L[5]` should contain the value 21 (20 bytes for the characters in the string plus 1 byte for the NULL-terminator). Your application must set each member of the `L` array.

`T` – *data types*

The `T` structure member points to an array of data types, one for each column (or parameter) described by the descriptor.

> For a bind descriptor, you must set each member of the `T` array to tell ECPGPlus the data type of each parameter.
>
> For a `SELECT`-list descriptor, the `DESCRIBE SELECT LIST` statement sets each member of the `T` array to reflect the type of data found in the corresponding column.

You may change any member of the `T` array before executing a `FETCH` statement to force ECPGPlus to convert the corresponding value to a specific data type. For example, if the `DESCRIBE SELECT LIST` statement indicates that a given column is of type `DATE`, you may change the corresponding `T` member to request that the next `FETCH` statement return that value in the form of a NULL-terminated string. Each member of the T array is a numeric type code. The type codes returned by a `DESCRIBE SELECT LIST` statement differ from those expected by a `FETCH` statement. After executing a `DESCRIBE SELECT LIST` statement, each member of `T` encodes a data type *and* a flag indicating whether the corresponding column is nullable. You can use the `sqlnul()` function to extract the type code and nullable flag from a member of the T array. The signature of the `sqlnul()` function is as follows:

```
void sqlnul(unsigned short *valType,
            unsigned short *typeCode,
            int            *isNull)
```

For example, to find the type code and nullable flag for the third column of a descriptor named results, you would invoke `sqlnul()` as follows:

```
sqlnul(&results->T[2], &typeCode, &isNull);
```

`I` – *indicator variables*

The `I` structure member points to an array of indicator variables. This array is allocated for you when your application calls the `sqlald()` function to allocate the descriptor.

> For a `SELECT`-list descriptor, each member of the `I` array indicates whether the corresponding column contains a NULL (non-zero) or non-NULL (zero) value.
>
> For a bind parameter, your application must set each member of the `I` array to indicate whether the corresponding parameter value is NULL.

`F` - *number of entries*

The `F` structure member indicates how many values are described by the descriptor (the `N` structure member indicates the *maximum* number of values which may be described by the descriptor; `F` indicates the actual number of values). The value of the `F` member is set by ECPGPlus when you execute a `DESCRIBE` statement. `F` may be positive, negative, or zero.

> For a `SELECT`-list descriptor, `F` will contain a positive value if the number of columns in the result set is equal to or less than the maximum number of values permitted by the descriptor (as determined by the `N` structure member); 0 if the statement is *not* a `SELECT` statement, or a negative value if the query returns more columns than allowed by the `N` structure member.
>
> For a bind descriptor, `F` will contain a positive number if the number of parameters found in the statement is less than or equal to the maximum number of values permitted by the descriptor (as determined by the `N` structure member); 0 if the statement contains no parameters markers, or a negative value if the statement contains more parameter markers than allowed by the `N` structure member.

If `F` contains a positive number (after executing a `DESCRIBE` statement), that number reflects the count of columns in the result set (for a `SELECT`-list descriptor) or the number of parameter markers found in the statement (for a bind descriptor). If `F` contains a negative value, you may compute the absolute value of `F` to discover how many values (or parameter markers) are required. For example, if `F` contains `-24` after describing a `SELECT` list, you know that the query returns 24 columns.

`S` - *column/parameter names*

The `S` structure member points to an array of NULL-terminated strings.

> For a `SELECT`-list descriptor, the `DESCRIBE SELECT LIST` statement sets each member of this array to the name of the corresponding column in the result set.
>
> For a bind descriptor, the `DESCRIBE BIND VARIABLES` statement sets each member of this array to the name of the corresponding bind variable.

In this release, the name of each bind variable is determined by the left-to-right order of the parameter marker within the query - for example, the name of the first parameter is always `?0`, the name of the second parameter is always `?1`, and so on.

`M` – *maximum column/parameter name length*

The `M` structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the `S` array (that is, `M[0]` specifies the maximum length of the column/parameter name found at `S[0]`). This array is populated by the `sqlald()` function.

`C` – *actual column/parameter name length*

The `C` structure member points to an array of lengths. Each member in this array specifies the *actual* length of the corresponding member of the `S` array (that is, `C[0]` specifies the actual length of the column/parameter name found at `S[0]`).

This array is populated by the `DESCRIBE` statement.

`X` – *indicator variable names*

The `X` structure member points to an array of NULL-terminated strings - each string represents the name of a NULL indicator for the corresponding value.

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

`Y` – *maximum indicator name length*

The `Y` structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the `X` array (that is, `Y[0]` specifies the maximum length of the indicator name found at `X[0]`).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

`Z` – *actual indicator name length*

The `Z` structure member points to an array of lengths. Each member in this array specifies the *actual* length of the corresponding member of the `X` array (that is, `Z[0]` specifies the actual length of the indicator name found at `X[0]`).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

6.4.5 ECPGPlus Statements

An embedded SQL statement allows your client application to interact with the server, while an embedded directive is an instruction to the ECPGPlus compiler.

You can embed any Advanced Server SQL statement in a C program. Each statement should begin with the keywords `EXEC SQL`, and must be terminated with a semi-colon (`;`). Within the C program, a SQL statement takes the form:

```
EXEC SQL sql_command_body;
```

Where *sql_command_body* represents a standard SQL statement. You can use a host variable anywhere that the SQL statement expects a value expression.

ECPGPlus extends the PostgreSQL server-side syntax for some statements; for those statements, syntax differences are outlined in the following reference sections. For a complete reference to the supported syntax of other SQL commands, please see the *PostgreSQL Core Documentation*, available from the EnterpriseDB website at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-commands.html

6.4.5.1 ALLOCATE DESCRIPTOR

Use the `ALLOCATE DESCRIPTOR` statement to allocate an SQL descriptor area:

```
EXEC SQL [FOR array_size] ALLOCATE DESCRIPTOR descriptor_name
    [WITH MAX variable_count];
```

Where:

array_size is a variable that specifies the number of array elements to allocate for the descriptor. *array_size* may be an `INTEGER` value or a host variable.

descriptor_name is the host variable that contains the name of the descriptor, or the name of the descriptor. This value may take the form of an identifier, a quoted string literal, or of a host variable.

variable_count specifies the maximum number of host variables in the descriptor. The default value of *variable_count* is `100`.

The following code fragment allocates a descriptor named `emp_query` that may be processed as an array (`emp_array`):

```
EXEC SQL FOR :emp_array ALLOCATE DESCRIPTOR emp_query;
```

6.4.5.2 CALL

Use the CALL statement to invoke a procedure or function on the server. The CALL statement works only on Advanced Server. The CALL statement comes in two forms; the first form is used to call a *function*:

```
EXEC SQL CALL program_name '('[actual_arguments]')'
  INTO [[:ret_variable][:ret_indicator]];
```

The second form is used to call a *procedure*:

```
EXEC SQL CALL program_name '('[actual_arguments]')';
```

Where:

> *program_name* is the name of the stored procedure or function that the CALL statement invokes. The program name may be schema-qualified or package-qualified (or both); if you do not specify the schema or package in which the program resides, ECPGPlus will use the value of search_path to locate the program.
>
> *actual_arguments* specifies a comma-separated list of arguments required by the program. Note that each *actual_argument* corresponds to a formal argument expected by the program. Each formal argument may be an IN parameter, an OUT parameter, or an INOUT parameter.
>
> :*ret_variable* specifies a host variable that will receive the value returned if the program is a function.
>
> :*ret_indicator* specifies a host variable that will receive the indicator value returned, if the program is a function.

For example, the following statement invokes the get_job_desc function with the value contained in the :ename host variable, and captures the value returned by that function in the :job host variable:

```
EXEC SQL CALL get_job_desc(:ename)
  INTO :job;
```

6.4.5.3 CLOSE

Use the CLOSE statement to close a cursor, and free any resources currently in use by the cursor. A client application cannot fetch rows from a closed cursor. The syntax of the CLOSE statement is:


```
EXEC SQL CLOSE [cursor_name];
```

Where:

> `cursor_name` is the name of the cursor closed by the statement. The cursor name may take the form of an identifier or of a host variable.

The `OPEN` statement initializes a cursor. Once initialized, a cursor result set will remain unchanged unless the cursor is re-opened. You do not need to `CLOSE` a cursor before re-opening it.

To manually close a cursor named `emp_cursor`, use the command:

```
EXEC SQL CLOSE emp_cursor;
```

A cursor is automatically closed when an application terminates.

6.4.5.4 COMMIT

Use the `COMMIT` statement to complete the current transaction, making all changes permanent and visible to other users. The syntax is:

```
EXEC SQL [AT database_name] COMMIT [WORK]
         [COMMENT 'text'] [COMMENT 'text' RELEASE];
```

Where:

> `database_name` is the name of the database (or host variable that contains the name of the database) in which the work resides. This value may take the form of an unquoted string literal, or of a host variable.

For compatibility, ECPGPlus accepts the `COMMENT` clause without error but does *not* store any text included with the `COMMENT` clause.

Include the `RELEASE` clause to close the current connection after performing the commit.

For example, the following command commits all work performed on the `dept` database and closes the current connection:

```
EXEC SQL AT dept COMMIT RELEASE;
```

By default, statements are committed only when a client application performs a `COMMIT` statement. Include the `-t` option when invoking ECPGPlus to specify that a client application should invoke `AUTOCOMMIT` functionality. You can also control `AUTOCOMMIT` functionality in a client application with the following statements:

```
EXEC SQL SET AUTOCOMMIT TO ON
```

and

```
EXEC SQL SET AUTOCOMMIT TO OFF
```

6.4.5.5 CONNECT

Use the `CONNECT` statement to establish a connection to a database. The `CONNECT` statement is available in two forms.

The following is the first form:

```
EXEC SQL CONNECT
  {{:user_name IDENTIFIED BY :password} | :connection_id}
  [AT database_name]
  [USING :database_string]
  [ALTER AUTHORIZATION :new_password];
```

Where:

> *user_name* is a host variable that contains the role that the client application will use to connect to the server.
>
> *password* is a host variable that contains the password associated with that role.
>
> *connection_id* is a host variable that contains a slash-delimited user name and password used to connect to the database.

Include the `AT` clause to specify the database to which the connection is established. *database_name* is the name of the database to which the client is connecting; specify the value in the form of a variable, or as a string literal.

Include the `USING` clause to specify a host variable that contains a null-terminated string identifying the database to which the connection will be established.

The `ALTER AUTHORIZATION` clause is supported for syntax compatibility only; ECPGPlus parses the `ALTER AUTHORIZATION` clause, and reports a warning.

Using the first form of the `CONNECT` statement, a client application might establish a connection with a host variable named `user` that contains the identity of the connecting role, and a host variable named `password` that contains the associated password using the following command:

```
EXEC SQL CONNECT :user IDENTIFIED BY :password;
```

A client application could also use the first form of the `CONNECT` statement to establish a connection using a single host variable named `:connection_id`. In the following example, `connection_id` contains the slash-delimited role name and associated password for the user:

```
EXEC SQL CONNECT :connection_id;
```

The syntax of the second form of the `CONNECT` statement is:

```
EXEC SQL CONNECT TO database_name
[AS connection_name] [credentials];
```

Where *credentials* is one of the following:

```
USER user_name password
USER user_name IDENTIFIED BY password
USER user_name USING password
```

In the second form:

database_name is the name or identity of the database to which the client is connecting. Specify *database_name* as a variable, or as a string literal, in one of the following forms:

```
database_name[@hostname][:port]

tcp:postgresql://hostname[:port][/database_name][options]

unix:postgresql://hostname[:port][/database_name][options]
```

Where:

hostname is the name or IP address of the server on which the database resides.

port is the port on which the server listens.

You can also specify a value of `DEFAULT` to establish a connection with the default database, using the default role name. If you specify `DEFAULT` as the target database, do not include a *connection_name* or *credentials*.

connection_name is the name of the connection to the database. *connection_name* should take the form of an identifier (that is, not a string literal or a variable). You can open multiple connections, by providing a unique *connection_name* for each connection.

If you do not specify a name for a connection, `ecpglib` assigns a name of `DEFAULT` to the connection. You can refer to the connection by name (`DEFAULT`) in any `EXEC SQL` statement.

`CURRENT` is the most recently opened or the connection mentioned in the most-recent `SET CONNECTION TO` statement. If you do not refer to a connection by name in an `EXEC SQL` statement, ECPG assumes the name of the connection to be `CURRENT`.

`user_name` is the role used to establish the connection with the Advanced Server database. The privileges of the specified role will be applied to all commands performed through the connection.

`password` is the password associated with the specified *`user_name`*.

The following code fragment uses the second form of the `CONNECT` statement to establish a connection to a database named `edb`, using the role `alice` and the password associated with that role, `1safepwd`:

```
EXEC SQL CONNECT TO edb AS acctg_conn
  USER 'alice' IDENTIFIED BY '1safepwd';
```

The name of the connection is `acctg_conn`; you can use the connection name when changing the connection name using the `SET CONNECTION` statement.

6.4.5.6 DEALLOCATE DESCRIPTOR

Use the `DEALLOCATE DESCRIPTOR` statement to free memory in use by an allocated descriptor. The syntax of the statement is:

```
EXEC SQL DEALLOCATE DESCRIPTOR descriptor_name
```

Where:

`descriptor_name` is the name of the descriptor. This value may take the form of a quoted string literal, or of a host variable.

The following example deallocates a descriptor named `emp_query`:

```
EXEC SQL DEALLOCATE DESCRIPTOR emp_query;
```

6.4.5.7 DECLARE CURSOR

Use the `DECLARE CURSOR` statement to define a cursor. The syntax of the statement is:

```
EXEC SQL [AT database_name] DECLARE cursor_name CURSOR FOR
(select_statement | statement_name);
```

Where:

database_name is the name of the database on which the cursor operates. This value may take the form of an identifier or of a host variable. If you do not specify a database name, the default value of *database_name* is the default database.

cursor_name is the name of the cursor.

select_statement is the text of the `SELECT` statement that defines the cursor result set; the `SELECT` statement cannot contain an `INTO` clause.

statement_name is the name of a SQL statement or block that defines the cursor result set.

The following example declares a cursor named `employees`:

```
EXEC SQL DECLARE employees CURSOR FOR
  SELECT
    empno, ename, sal, comm
  FROM
    emp;
```

The cursor generates a result set that contains the employee number, employee name, salary and commission for each employee record that is stored in the `emp` table.

6.4.5.8 DECLARE DATABASE

Use the `DECLARE DATABASE` statement to declare a database identifier for use in subsequent SQL statements (for example, in a `CONNECT` statement). The syntax is:

```
EXEC SQL DECLARE database_name DATABASE;
```

Where:

database_name specifies the name of the database.

The following example demonstrates declaring an identifier for the `acctg` database:

```
EXEC SQL DECLARE acctg DATABASE;
```

After invoking the command declaring `acctg` as a database identifier, the `acctg` database can be referenced by name when establishing a connection or in `AT` clauses.

This statement has no effect and is provided for Pro*C compatibility only.

6.4.5.9 DECLARE STATEMENT

Use the `DECLARE STATEMENT` directive to declare an identifier for an SQL statement. Advanced Server supports two versions of the `DECLARE STATEMENT` directive:

```
EXEC SQL [database_name] DECLARE statement_name STATEMENT;
```

and

```
EXEC SQL DECLARE STATEMENT statement_name;
```

Where:

> *statement_name* specifies the identifier associated with the statement.
>
> *database_name* specifies the name of the database. This value may take the form of an identifier or of a host variable that contains the identifier.

A typical usage sequence that includes the `DECLARE STATEMENT` directive might be:

```
EXEC SQL DECLARE give_raise STATEMENT;      // give_raise
is now a statement handle (not prepared)
EXEC SQL PREPARE give_raise FROM :stmtText; // give_raise
is now associated with a statement
EXEC SQL EXECUTE give_raise;
```

This statement has no effect and is provided for Pro*C compatibility only.

6.4.5.10 DELETE

Use the `DELETE` statement to delete one or more rows from a table. The syntax for the ECPGPlus `DELETE` statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that an expression is allowed. The syntax is:

```
[FOR exec_count] DELETE FROM [ONLY] table [[AS] alias]
  [USING using_list]
  [WHERE condition | WHERE CURRENT OF cursor_name]
  [{RETURNING|RETURN} * | output_expression [[ AS] output_name]
[, ...] INTO host_variable_list ]
```

Where:

Include the FOR *exec_count* clause to specify the number of times the statement will execute; this clause is valid only if the VALUES clause references an array or a pointer to an array.

table is the name (optionally schema-qualified) of an existing table. Include the ONLY clause to limit processing to the specified table; if you do not include the ONLY clause, any tables inheriting from the named table are also processed.

alias is a substitute name for the target table.

using_list is a list of table expressions, allowing columns from other tables to appear in the WHERE condition.

Include the WHERE clause to specify which rows should be deleted. If you do not include a WHERE clause in the statement, DELETE will delete all rows from the table, leaving the table definition intact.

condition is an expression, host variable or parameter marker that returns a value of type BOOLEAN. Those rows for which *condition* returns true will be deleted.

cursor_name is the name of the cursor to use in the WHERE CURRENT OF clause; the row to be deleted will be the one most recently fetched from this cursor. The cursor must be a non-grouping query on the DELETE statements target table. You cannot specify WHERE CURRENT OF in a DELETE statement that includes a Boolean condition.

The RETURN/RETURNING clause specifies an *output_expression* or *host_variable_list* that is returned by the DELETE command after each row is deleted:

> *output_expression* is an expression to be computed and returned by the DELETE command after each row is deleted. *output_name* is the name of the returned column; include * to return all columns.
>
> *host_variable_list* is a comma-separated list of host variables and optional indicator variables. Each host variable receives a corresponding value from the RETURNING clause.

For example, the following statement deletes all rows from the emp table where the sal column contains a value greater than the value specified in the host variable, :max_sal:

```
DELETE FROM emp WHERE sal > :max_sal;
```


For more information about using the DELETE statement, please see the PostgreSQL Core documentation, available from EnterpriseDB at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-delete.html

6.4.5.11 DESCRIBE

Use the DESCRIBE statement to find the number of input values required by a prepared statement or the number of output values returned by a prepared statement. The DESCRIBE statement is used to analyze a SQL statement whose shape is unknown at the time you write your application.

The DESCRIBE statement populates an SQLDA descriptor; to populate a SQL descriptor, use the ALLOCATE DESCRIPTOR and DESCRIBE…DESCRIPTOR statements.

```
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name INTO
descriptor;
```

or

```
EXEC SQL DESCRIBE SELECT LIST FOR statement_name INTO
descriptor;
```

Where:

> *statement_name* is the identifier associated with a prepared SQL statement or PL/SQL block.
>
> *descriptor* is the name of C variable of type SQLDA*. You must allocate the space for the descriptor by calling sqlald() (and initialize the descriptor) before executing the DESCRIBE statement.

When you execute the first form of the DESCRIBE statement, ECPG populates the given descriptor with a description of each input variable *required* by the statement. For example, given two descriptors:

```
SQLDA *query_values_in;
SQLDA *query_values_out;
```

You might prepare a query that returns information from the emp table:

```
EXEC SQL PREPARE get_emp FROM
  "SELECT ename, empno, sal FROM emp WHERE empno = ?";
```

The command requires one input variable (for the parameter marker (?)).

```
EXEC SQL DESCRIBE BIND VARIABLES
  FOR get_emp INTO query_values_in;
```

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

When you execute the second form, ECPG populates the given descriptor with a description of each value *returned* by the statement. For example, the following statement returns three values:

```
EXEC SQL DESCRIBE SELECT LIST
  FOR get_emp  INTO query_values_out;
```

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

Before *executing* the statement, you must bind a variable for each input value and a variable for each output value. The variables that you bind for the input values specify the actual values used by the statement. The variables that you bind for the output values tell ECPGPlus where to put the values when you execute the statement.

This is alternate Pro*C compatible syntax for the `DESCRIBE DESCRIPTOR` statement.

6.4.5.12 DESCRIBE DESCRIPTOR

Use the `DESCRIBE DESCRIPTOR` statement to retrieve information about a SQL statement, and store that information in a SQL descriptor. Before using `DESCRIBE DESCRIPTOR`, you must allocate the descriptor with the `ALLOCATE DESCRIPTOR` statement. The syntax is:

```
EXEC SQL DESCRIBE [INPUT | OUTPUT] statement_identifier
  USING [SQL] DESCRIPTOR descriptor_name;
```

Where:

> *`statement_name`* is the name of a prepared SQL statement.
>
> *`descriptor_name`* is the name of the descriptor. *`descriptor_name`* can be a quoted string value or a host variable that contains the name of the descriptor.

If you include the `INPUT` clause, ECPGPlus populates the given descriptor with a description of each input variable *required* by the statement.

> For example, given two descriptors:


```
EXEC SQL ALLOCATE DESCRIPTOR query_values_in;
EXEC SQL ALLOCATE DESCRIPTOR query_values_out;
```

You might prepare a query that returns information from the `emp` table:

```
EXEC SQL PREPARE get_emp FROM
  "SELECT ename, empno, sal FROM emp WHERE empno = ?";
```

The command requires one input variable (for the parameter marker (`?`)).

```
EXEC SQL DESCRIBE INPUT get_emp USING
'query_values_in';
```

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

If you do not specify the `INPUT` clause, `DESCRIBE DESCRIPTOR` populates the specified descriptor with the values returned by the statement.

If you include the `OUTPUT` clause, ECPGPlus populates the given descriptor with a description of each value *returned* by the statement.

For example, the following statement returns three values:

```
EXEC SQL DESCRIBE OUTPUT FOR get_emp USING
'query_values_out';
```

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

6.4.5.13 DISCONNECT

Use the `DISCONNECT` statement to close the connection to the server. The syntax is:

```
EXEC SQL DISCONNECT [connection_name][CURRENT][DEFAULT][ALL];
```

Where:

`connection_name` is the connection name specified in the `CONNECT` statement used to establish the connection. If you do not specify a connection name, the current connection is closed.

Include the `CURRENT` keyword to specify that ECPGPlus should close the most-recently used connection.

Include the DEFAULT keyword to specify that ECPGPlus should close the connection named DEFAULT. If you do not specify a name when opening a connection, ECPGPlus assigns the name, DEFAULT, to the connection.

Include the ALL keyword to instruct ECPGPlus to close all active connections.

The following example creates a connection (named hr_connection) that connects to the hr database, and then disconnects from the connection:

```
/* client.pgc*/
int main()
{
    EXEC SQL CONNECT TO hr AS connection_name;
    EXEC SQL DISCONNECT connection_name;
    return(0);
}
```

6.4.5.14 EXECUTE

Use the EXECUTE statement to execute a statement previously prepared using an EXEC SQL PREPARE statement. The syntax is:

```
EXEC SQL [FOR array_size] EXECUTE statement_name
  [USING {DESCRIPTOR SQLDA_descriptor
  |:host_variable [[INDICATOR] :indicator_variable]}];
```

Where:

array_size is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the FOR clause, the statement is executed once for each member of the array.

statement_name specifies the name assigned to the statement when the statement was created (using the EXEC SQL PREPARE statement).

Include the USING clause to supply values for parameters within the prepared statement:

Include the DESCRIPTOR *SQLDA_descriptor* clause to provide an SQLDA descriptor value for a parameter.

Use a *host_variable* (and an optional *indicator_variable*) to provide a user-specified value for a parameter.

The following example creates a prepared statement that inserts a record into the `emp` table:

```
EXEC SQL PREPARE add_emp (numeric, text, text, numeric) AS
    INSERT INTO emp VALUES($1, $2, $3, $4);
```

Each time you invoke the prepared statement, provide fresh parameter values for the statement:

```
EXEC SQL EXECUTE add_emp USING 8000, 'DAWSON', 'CLERK',
7788;
EXEC SQL EXECUTE add_emp USING 8001, 'EDWARDS', 'ANALYST',
7698;
```

6.4.5.15 EXECUTE DESCRIPTOR

Use the `EXECUTE` statement to execute a statement previously prepared by an `EXEC SQL PREPARE` statement, using an SQL descriptor. The syntax is:

```
EXEC SQL [FOR array_size] EXECUTE statement_identifier
  [USING [SQL] DESCRIPTOR descriptor_name]
  [INTO [SQL] DESCRIPTOR descriptor_name];
```

Where:

> *array_size* is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the `FOR` clause, the statement is executed once for each member of the array.
>
> *statement_identifier* specifies the identifier assigned to the statement with the `EXEC SQL PREPARE` statement.
>
> Include the `USING` clause to specify values for any input parameters required by the prepared statement.
>
> Include the `INTO` clause to specify a descriptor into which the `EXECUTE` statement will write the results returned by the prepared statement.
>
> *descriptor_name* specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

The following example executes the prepared statement, `give_raise`, using the values contained in the descriptor `stmtText`:

```
EXEC SQL PREPARE give_raise FROM :stmtText;
EXEC SQL EXECUTE give_raise USING DESCRIPTOR :stmtText;
```

6.4.5.16 EXECUTE...END EXEC

Use the `EXECUTE…END-EXEC` statement to embed an anonymous block into a client application. The syntax is:

```
EXEC SQL [AT database_name] EXECUTE anonymous_block END-EXEC;
```

Where:

database_name is the database identifier or a host variable that contains the database identifier. If you omit the `AT` clause, the statement will be executed on the current default database.

anonymous_block is an inline sequence of PL/pgSQL or SPL statements and declarations. You may include host variables and optional indicator variables within the block; each such variable is treated as an `IN/OUT` value.

The following example executes an anonymous block:

```
EXEC SQL EXECUTE
  BEGIN
    IF (current_user = :admin_user_name) THEN
      DBMS_OUTPUT.PUT_LINE('You are an administrator');
    END IF;
END-EXEC;
```

Please Note: the `EXECUTE…END EXEC` statement is supported only by Postgres Plus Advanced Server.

6.4.5.17 EXECUTE IMMEDIATE

Use the `EXECUTE IMMEDIATE` statement to execute a string that contains a SQL command. The syntax is:

```
EXEC SQL [AT database_name] EXECUTE IMMEDIATE command_text;
```

Where:

database_name is the database identifier or a host variable that contains the database identifier. If you omit the `AT` clause, the statement will be executed on the current default database.

command_text is the command executed by the `EXECUTE IMMEDIATE` statement.

This dynamic SQL statement is useful when you don't know the text of an SQL statement (i.e., when writing a client application). For example, a client application may prompt a (trusted) user for a statement to execute. After the user provides the text of the statement as a string value, the statement is then executed with an `EXECUTE IMMEDIATE` command.

The statement text may not contain references to host variables. If the statement may contain parameter markers or returns one or more values, you must use the `PREPARE` and `DESCRIBE` statements.

The following example executes the command contained in the `:command_text` host variable:

```
EXEC SQL EXECUTE IMMEDIATE :command_text;
```

6.4.5.18 FETCH

Use the `FETCH` statement to return rows from a cursor into an SQLDA descriptor or a target list of host variables. Before using a `FETCH` statement to retrieve information from a cursor, you must prepare the cursor using `DECLARE` and `OPEN` statements. The statement syntax is:

```
EXEC SQL [FOR array_size] FETCH cursor
  { USING DESCRIPTOR SQLDA_descriptor }|{ INTO target_list };
```

Where:

> *`array_size`* is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.
>
> *`cursor`* is the name of the cursor from which rows are being fetched, or a host variable that contains the name of the cursor.
>
> If you include a `USING` clause, the `FETCH` statement will populate the specified SQLDA descriptor with the values returned by the server.
>
> If you include an `INTO` clause, the `FETCH` statement will populate the host variables (and optional indicator variables) specified in the *`target_list`*.

The following code fragment declares a cursor named `employees` that retrieves the employee number, name and salary from the `emp` table:

```
EXEC SQL DECLARE employees CURSOR FOR
    SELECT empno, ename, esal FROM emp;
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp_cursor INTO :emp_no, :emp_name, :emp_sal;
```


6.4.5.19 FETCH DESCRIPTOR

Use the FETCH DESCRIPTOR statement to retrieve rows from a cursor into an SQL descriptor. The syntax is:

```
EXEC SQL [FOR array_size] FETCH cursor
  INTO [SQL] DESCRIPTOR descriptor_name;
```

Where:

> *array_size* is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.
>
> *cursor* is the name of the cursor from which rows are fetched, or a host variable that contains the name of the cursor. The client must DECLARE and OPEN the cursor before calling the FETCH DESCRIPTOR statement.
>
> Include the INTO clause to specify an SQL descriptor into which the EXECUTE statement will write the results returned by the prepared statement. *descriptor_name* specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor. Prior to use, the descriptor must be allocated using an ALLOCATE DESCRIPTOR statement.

The following example allocates a descriptor named row_desc that will hold the description and the values of a specific row in the result set. It then declares and opens a cursor for a prepared statement (my_cursor), before looping through the rows in result set, using a FETCH to retrieve the next row from the cursor into the descriptor:

```
EXEC SQL ALLOCATE DESCRIPTOR 'row_desc';
EXEC SQL DECLARE my_cursor CURSOR FOR query;
EXEC SQL OPEN my_cursor;

for( row = 0; ; row++ )
{
  EXEC SQL BEGIN DECLARE SECTION;
    int     col;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL FETCH my_cursor INTO SQL DESCRIPTOR 'row_desc';
```

6.4.5.20 GET DESCRIPTOR

Use the GET DESCRIPTOR statement to retrieve information from a descriptor. The GET DESCRIPTOR statement comes in two forms. The first form returns the number of values (or columns) in the descriptor.


```
EXEC SQL GET DESCRIPTOR descriptor_name
  :host_variable = COUNT;
```

The second form returns information about a specific value (specified by the `VALUE` *`column_number`* clause).

```
EXEC SQL [FOR array_size] GET DESCRIPTOR descriptor_name
  VALUE column_number {:host_variable = descriptor_item {,…}};
```

Where:

`array_size` is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you specify an *`array_size`*, the *`host_variable`* must be an array of that size; for example, if *`array_size`* is `10`, `:`*`host_variable`* must be a 10-member array of *`host_variables`*. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`descriptor_name` specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the `VALUE` clause to specify the information retrieved from the descriptor.

`column_number` identifies the position of the variable within the descriptor.

`host_variable` specifies the name of the host variable that will receive the value of the item.

`descriptor_item` specifies the type of the retrieved descriptor item.

ECPGPlus implements the following *`descriptor_item`* types:

- `TYPE`
- `LENGTH`
- `OCTET_LENGTH`
- `RETURNED_LENGTH`
- `RETURNED_OCTET_LENGTH`
- `PRECISION`
- `SCALE`
- `NULLABLE`
- `INDICATOR`
- `DATA`
- `NAME`

The following code fragment demonstrates using a GET DESCRIPTOR statement to obtain the number of columns entered in a user-provided string:

```
EXEC SQL ALLOCATE DESCRIPTOR parse_desc;
EXEC SQL PREPARE query FROM :stmt;
EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;
EXEC SQL GET DESCRIPTOR parse_desc :col_count = COUNT;
```

The example allocates an SQL descriptor (named parse_desc), before using a PREPARE statement to syntax check the string provided by the user (:stmt). A DESCRIBE statement moves the user-provided string into the descriptor, parse_desc. The call to EXEC SQL GET DESCRIPTOR interrogates the descriptor to discover the number of columns (:col_count) in the result set.

6.4.5.21 INSERT

Use the INSERT statement to add one or more rows to a table. The syntax for the ECPGPlus INSERT statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that a value is allowed. The syntax is:

```
[FOR exec_count] INSERT INTO table [(column [, ...])]
  {DEFAULT VALUES |
   VALUES ({expression | DEFAULT} [, ...])[, ...] | query}
  [RETURNING * | output_expression [[ AS ] output_name] [, ...]]
```

Where:

Include the FOR *exec_count* clause to specify the number of times the statement will execute; this clause is valid only if the VALUES clause references an array or a pointer to an array.

table specifies the (optionally schema-qualified) name of an existing table.

column is the name of a column in the table. The column name may be qualified with a subfield name or array subscript. Specify the DEFAULT VALUES clause to use default values for all columns.

expression is the expression, value, host variable or parameter marker that will be assigned to the corresponding column. Specify DEFAULT to fill the corresponding column with its default value.

query specifies a SELECT statement that supplies the row(s) to be inserted.

output_expression is an expression that will be computed and returned by the INSERT command after each row is inserted. The expression can refer to any column within the table. Specify * to return all columns of the inserted row(s).

output_name specifies a name to use for a returned column.

The following example adds a row to the employees table:

```
INSERT INTO emp (empno, ename, job, hiredate)
    VALUES ('8400', :ename, 'CLERK', '2011-10-31');
```

Note that the INSERT statement uses a host variable (:ename) to specify the value of the ename column.

For more information about using the INSERT statement, please see the PostgreSQL Core documentation, available from EnterpriseDB at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-insert.html

6.4.5.22 OPEN

Use the OPEN statement to open a cursor. The syntax is:

```
EXEC SQL [FOR array_size] OPEN cursor [USING parameters];
```

Where *parameters* is one of the following:

```
  DESCRIPTOR SQLDA_descriptor
or
  host_variable [ [ INDICATOR ] indicator_variable, … ]
```

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the FOR clause, the statement is executed once for each member of the array.

cursor is the name of the cursor being opened.

parameters is either DESCRIPTOR *SQLDA_descriptor* or a comma-separated list of host variables (and optional indicator variables) that initialize the cursor. If specifying an *SQLDA_descriptor*, the descriptor must be initialized with a DESCRIBE statement.

The OPEN statement initializes a cursor using the values provided in *parameters*. Once initialized, the cursor result set will remain unchanged unless the cursor is closed and re-opened. A cursor is automatically closed when an application terminates.

The following example declares a cursor named `employees`, that queries the `emp` table, returning the employee number, name, salary and commission of an employee whose name matches a user-supplied value (stored in the host variable, `:emp_name`).

```
  EXEC SQL DECLARE employees CURSOR FOR
    SELECT
      empno, ename, sal, comm
    FROM
      emp
    WHERE ename = :emp_name;
  EXEC SQL OPEN employees;
...
```

After declaring the cursor, the example uses an `OPEN` statement to make the contents of the cursor available to a client application.

6.4.5.23 OPEN DESCRIPTOR

Use the `OPEN DESCRIPTOR` statement to open a cursor with a SQL descriptor. The syntax is:

```
EXEC SQL [FOR array_size] OPEN cursor
  [USING [SQL] DESCRIPTOR descriptor_name]
  [INTO [SQL] DESCRIPTOR descriptor_name];
```

Where:

> *array_size* is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.
>
> *cursor* is the name of the cursor being opened.
>
> *descriptor_name* specifies the name of an SQL descriptor (in the form of a single-quoted string literal) or a host variable that contains the name of an SQL descriptor that contains the query that initializes the cursor.

For example, the following statement opens a cursor (named `emp_cursor`), using the host variable, `:employees`:

```
EXEC SQL OPEN emp_cursor USING DESCRIPTOR :employees;
```


6.4.5.24 PREPARE

Prepared statements are useful when a client application must perform a task multiple times; the statement is parsed, written and planned only once, rather than each time the statement is executed, saving repetitive processing time.

Use the `PREPARE` statement to prepare an SQL statement or PL/pgSQL block for execution. The statement is available in two forms; the first form is:

```
EXEC SQL [AT database_name] PREPARE statement_name
  FROM sql_statement;
```

The second form is:

```
EXEC SQL [AT database_name] PREPARE statement_name
  AS sql_statement;
```

Where:

> *database_name* is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the `AT` clause, the statement will execute against the current default database.
>
> *statement_name* is the identifier associated with a prepared SQL statement or PL/SQL block.
>
> *sql_statement* may take the form of a `SELECT` statement, a single-quoted string literal or host variable that contains the text of an SQL statement.

To include variables within a prepared statement, substitute placeholders (`$1`, `$2`, `$3`, etc.) for statement values that might change when you `PREPARE` the statement. When you `EXECUTE` the statement, provide a value for each parameter. The values must be provided in the order in which they will replace placeholders.

The following example creates a prepared statement (named `add_emp`) that inserts a record into the `emp` table:

```
EXEC SQL PREPARE add_emp (int, text, text, numeric) AS
    INSERT INTO emp VALUES($1, $2, $3, $4);
```

Each time you invoke the statement, provide fresh parameter values for the statement:

```
EXEC SQL EXECUTE add_emp(8003, 'Davis', 'CLERK', 2000.00);
EXEC SQL EXECUTE add_emp(8004, 'Myer', 'CLERK', 2000.00);
```

Please note: A client application must issue a PREPARE statement within each session in which a statement will be executed; prepared statements persist only for the duration of the current session.

6.4.5.25 ROLLBACK

Use the ROLLBACK statement to abort the current transaction, and discard any updates made by the transaction. The syntax is:

```
EXEC SQL [AT database_name] ROLLBACK [WORK]
  [ { TO [SAVEPOINT] savepoint } | RELEASE ]
```

Where:

> *database_name* is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the AT clause, the statement will execute against the current default database.
>
> Include the TO clause to abort any commands that were executed after the specified *savepoint*; use the SAVEPOINT statement to define the *savepoint*. If you omit the TO clause, the ROLLBACK statement will abort the transaction, discarding all updates.
>
> Include the RELEASE clause to cause the application to execute an EXEC SQL COMMIT RELEASE and close the connection.

Use the following statement to rollback a complete transaction:

```
EXEC SQL ROLLBACK;
```

Invoking this statement will abort the transaction, undoing all changes, erasing any savepoints, and releasing all transaction locks. If you include a savepoint (*my_savepoint* in the following example):

```
EXEC SQL ROLLBACK TO SAVEPOINT my_savepoint;
```

Only the portion of the transaction that occurred after the my_savepoint is rolled back; my_savepoint is retained, but any savepoints created after my_savepoint will be erased.

Rolling back to a specified savepoint releases all locks acquired after the savepoint.

6.4.5.26 SAVEPOINT

Use the `SAVEPOINT` statement to define a *savepoint*; a savepoint is a marker within a transaction. You can use a `ROLLBACK` statement to abort the current transaction, returning the state of the server to its condition prior to the specified savepoint. The syntax of a `SAVEPOINT` statement is:

```
EXEC SQL [AT database_name] SAVEPOINT savepoint_name
```

Where:

> *`database_name`* is the database identifier or a host variable that contains the database identifier against which the savepoint resides. If you omit the `AT` clause, the statement will execute against the current default database.
>
> *`savepoint_name`* is the name of the savepoint. If you re-use a *`savepoint_name`*, the original savepoint is discarded.

Savepoints can only be established within a transaction block. A transaction block may contain multiple savepoints.

To create a savepoint named `my_savepoint`, include the statement:

```
EXEC SQL SAVEPOINT my_savepoint;
```

6.4.5.27 SELECT

ECPGPlus extends support of the SQL `SELECT` statement by providing the `INTO` *`host_variables`* clause. The clause allows you to select specified information from an Advanced Server database into a host variable. The syntax for the `SELECT` statement is:

```
EXEC SQL [AT database_name]
SELECT
  [ hint ]
  [ ALL | DISTINCT [ ON(expression, ...) ]]
  select_list INTO host_variables

  [ FROM from_item [, from_item ]...]
  [ WHERE condition ]
  [ hierarchical_query_clause ]
  [ GROUP BY expression [, ...]]
  [ HAVING condition ]
  [ { UNION [ ALL ] | INTERSECT | MINUS } (subquery) ]
  [ ORDER BY expression [order_by_options]]
  [ LIMIT { count | ALL }]
  [ OFFSET start [ ROW | ROWS ] ]
  [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

```
[ FOR { UPDATE | SHARE } [OF table_name [, ...]][NOWAIT ][...]]
```

Where:

> *database_name* is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.
>
> *host_variables* is a list of host variables that will be populated by the `SELECT` statement. If the `SELECT` statement returns more than a single row, *host_variables* must be an array.

ECPGPlus provides support for the additional clauses of the SQL `SELECT` statement as documented in the PostgreSQL Core documentation, available from EnterpriseDB at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-select.html

To use the `INTO` *host_variables* clause, include the names of defined host variables when specifying the `SELECT` statement. For example, the following `SELECT` statement populates the `:emp_name` and `:emp_sal` host variables with a list of employee names and salaries:

```
EXEC SQL SELECT ename, sal
  INTO :emp_name, :emp_sal
  FROM emp
  WHERE empno = 7988;
```

The enhanced `SELECT` statement also allows you to include parameter markers (question marks) in any clause where a value would be permitted. For example, the following query contains a parameter marker in the `WHERE` clause:

```
SELECT * FROM emp WHERE dept_no = ?;
```

This `SELECT` statement allows you to provide a value at run-time for the `dept_no` parameter marker.

6.4.5.28 SET CONNECTION

There are (at least) three reasons you may need more than one connection in a given client application:

- You may want different privileges for different statements.
- You may need to interact with multiple databases within the same client.
- Multiple threads of execution (within a client application) cannot share a connection concurrently.

The syntax for the SET CONNECTION statement is:

```
EXEC SQL SET CONNECTION connection_name;
```

Where:

> *connection_name* is the name of the connection to the database.

To use the SET CONNECTION statement, you should open the connection to the database using the second form of the CONNECT statement; include the AS clause to specify a *connection_name*.

By default, the current thread uses the current connection; use the SET CONNECTION statement to specify a default connection for the current thread to use. The default connection is only used when you execute an EXEC SQL statement that does not explicitly specify a connection name. For example, the following statement will use the default connection because it does not include an AT *connection_name* clause:

```
EXEC SQL DELETE FROM emp;
```

This statement will not use the default connection because it specifies a connection name using the AT *connection_name* clause:

```
EXEC SQL AT acctg_conn DELETE FROM emp;
```

For example, a client application that creates and maintains multiple connections (such as):

```
EXEC SQL CONNECT TO edb AS acctg_conn
  USER 'alice' IDENTIFIED BY 'acctpwd';
```

and

```
EXEC SQL CONNECT TO edb AS hr_conn
  USER 'bob' IDENTIFIED BY 'hrpwd';
```

Can change between the connections with the SET CONNECTION statement:

```
SET CONNECTION acctg_conn;
```

or

```
SET CONNECTION hr_conn;
```

The server will use the privileges associated with the connection when determining the privileges available to the connecting client. When using the acctg_conn connection, the client will have the privileges associated with the role, `alice`; when connected using `hr_conn`, the client will have the privileges associated with `bob`.

6.4.5.29 SET DESCRIPTOR

Use the `SET DESCRIPTOR` statement to assign a value to a descriptor area using information provided by the client application in the form of a host variable or an integer value. The statement comes in two forms; the first form is:

```
EXEC SQL [FOR array_size] SET DESCRIPTOR descriptor_name
  VALUE column_number descriptor_item = host_variable;
```

The second form is:

```
EXEC SQL [FOR array_size] SET DESCRIPTOR descriptor_name
  COUNT = integer;
```

Where:

array_size is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.

descriptor_name specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the `VALUE` clause to describe the information stored in the descriptor.

column_number identifies the position of the variable within the descriptor.

descriptor_item specifies the type of the descriptor item.

host_variable specifies the name of the host variable that contains the value of the item.

ECPGPlus implements the following *descriptor_item* types:

- `TYPE`
- `LENGTH`
- `[REF] INDICATOR`
- `[REF] DATA`
- `[REF] RETURNED LENGTH`

For example, a client application might prompt a user for a dynamically created query:

```
query_text = promptUser("Enter a query");
```

To execute a dynamically created query, you must first *prepare* the query (parsing and validating the syntax of the query), and then *describe* the *input* parameters found in the query using the `EXEC SQL DESCRIBE INPUT` statement.

```
EXEC SQL ALLOCATE DESCRIPTOR query_params;
EXEC SQL PREPARE emp_query FROM :query_text;

EXEC SQL DESCRIBE INPUT emp_query
  USING SQL DESCRIPTOR 'query_params';
```

After describing the query, the `query_params` descriptor contains information about each parameter required by the query.

For this example, we'll assume that the user has entered:

```
SELECT ename FROM emp WHERE sal > ? AND job = ?;,
```

In this case, the descriptor describes two parameters:

- one for `sal > ?`
- one for `job = ?`

To discover the number of parameter markers (question marks) in the query (and therefore, the number of values you must provide before executing the query), use:

```
EXEC SQL GET DESCRIPTOR … :host_variable = COUNT;
```

Then, you can use `EXEC SQL GET DESCRIPTOR` to retrieve the name of each parameter. You can also use `EXEC SQL GET DESCRIPTOR` to retrieve the type of each parameter (along with the number of parameters) from the descriptor, or you can supply each *value* in the form of a character string and ECPG will convert that string into the required data type.

The data type of the first parameter is `numeric`; the type of the second parameter is `varchar`. The name of the first parameter is `sal`; the name of the second parameter is `job`.

Next, loop through each parameter, prompting the user for a value, and store those values in host variables. You can use `GET DESCRIPTOR … COUNT` to find the number of parameters in the query.


```
EXEC SQL GET DESCRIPTOR 'query_params'
  :param_count = COUNT;

for(param_number = 1;
    param_number <= param_count;
    param_number++)
{
```

Use GET DESCRIPTOR to copy the name of the parameter into the param_name host variable:

```
   EXEC SQL GET DESCRIPTOR 'query_params'
      VALUE :param_number :param_name = NAME;

   reply = promptUser(param_name);
   if (reply == NULL)
    reply_ind = 1;  /* NULL */
else
    reply_ind = 0;  /* NOT NULL */
```

To associate a *value* with each parameter, you use the EXEC SQL SET DESCRIPTOR statement. For example:

```
  EXEC SQL SET DESCRIPTOR 'query_params'
    VALUE :param_number DATA = :reply;
  EXEC SQL SET DESCRIPTOR 'query_params'
    VALUE :param_number INDICATOR = :reply_ind;
}
```

Now, you can use the EXEC SQL EXECUTE DESCRIPTOR statement to execute the prepared statement on the server.

6.4.5.30 UPDATE

Use an UPDATE statement to modify the data stored in a table. The syntax is:

```
EXEC SQL [AT database_name][FOR exec_count]
    UPDATE [ ONLY ] table [ [ AS ] alias ]
    SET {column = { expression | DEFAULT } |
        (column [, ...]) = ({ expression|DEFAULT } [, ...])} [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [[ AS ] output_name] [, ...] ]
```

Where:

database_name is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.

Include the `FOR` *exec_count* clause to specify the number of times the statement will execute; this clause is valid only if the `SET` or `WHERE` clause contains an array.

ECPGPlus provides support for the additional clauses of the SQL `UPDATE` statement as documented in the PostgreSQL Core documentation, available from EnterpriseDB at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-update.html

A host variable can be used in any clause that specifies a value. To use a host variable, simply substitute a defined variable for any value associated with any of the documented `UPDATE` clauses.

The following `UPDATE` statement changes the job description of an employee (identified by the `:ename` host variable) to the value contained in the `:new_job` host variable, and increases the employee's salary, by multiplying the current salary by the value in the `:increase` host variable:

```
EXEC SQL UPDATE emp
  SET job = :new_job, sal = sal * :increase
  WHERE ename = :ename;
```

The enhanced `UPDATE` statement also allows you to include parameter markers (question marks) in any clause where an input value would be permitted. For example, we can write the same update statement with a parameter marker in the `WHERE` clause:

```
EXEC SQL UPDATE emp
  SET job = ?, sal = sal * ?
  WHERE ename = :ename;
```

This `UPDATE` statement could allow you to prompt the user for a new value for the `job` column and provide the amount by which the `sal` column is incremented for the employee specified by `:ename`.

6.4.5.31 WHENEVER

Use the `WHENEVER` statement to specify the action taken by a client application when it encounters an SQL error or warning. The syntax is:

```
EXEC SQL WHENEVER condition action;
```

The following table describes the different conditions that might trigger an *action*:

Condition	Description
`NOT FOUND`	The server returns a `NOT FOUND` condition when it encounters a `SELECT` that returns no rows, or when a `FETCH` reaches the end of a result set.
`SQLERROR`	The server returns an `SQLERROR` condition when it encounters a serious error returned by an SQL statement.
`SQLWARNING`	The server returns an `SQLWARNING` condition when it encounters a non-fatal warning returned by an SQL statement.

The following table describes the actions that result from a client encountering a *`condition`*:

Action	Description
`CALL` *`function`*`([`*`args`*`])`	Instructs the client application to call the named *`function`*.
`CONTINUE`	Instructs the client application to proceed to the next statement.
`DO BREAK`	Instructs the client application to a C break statement. A break statement may appear in a `loop` or a `switch` statement. If executed, the break statement terminate the `loop` or the `switch` statement.
`DO CONTINUE`	Instructs the client application to emit a C `continue` statement. A `continue` statement may only exist within a loop, and if executed, will cause the flow of control to return to the top of the loop.
`DO` *`function`*`([`*`args`*`])`	Instructs the client application to call the named *`function`*.
`GOTO` *`label`* or `GO TO` *`label`*	Instructs the client application to proceed to the statement that contains the *`label`*.
`SQLPRINT`	Instructs the client application to print a message to standard error.
`STOP`	Instructs the client application to stop execution.

The following code fragment prints a message if the client application encounters a warning, and aborts the application if it encounters an error:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

Include the following code to specify that a client should continue processing after warning a user of a problem:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
```

Include the following code to call a function if a query returns no rows, or when a cursor reaches the end of a result set:

```
EXEC SQL WHENEVER NOT FOUND CALL error_handler(__LINE__);
```

7 Open Client Library

The Open Client Library provides application interoperability with the Oracle Call Interface – an application that was formerly "locked in" can now work with either a Postgres Plus Advanced Server or an Oracle database with minimal to no changes to the application code. The EnterpriseDB implementation of the Open Client Library is written in C.

Please note: EnterpriseDB does not support use of the Open Client Library with Oracle Real Application Clusters (RAC) and Oracle Exadata; the aforementioned Oracle products have not been evaluated nor certified with this EnterpriseDB product.

7.1 Comparison with Oracle Call Interface

The following diagram compares the Open Client Library and Oracle Call Interface application stacks.

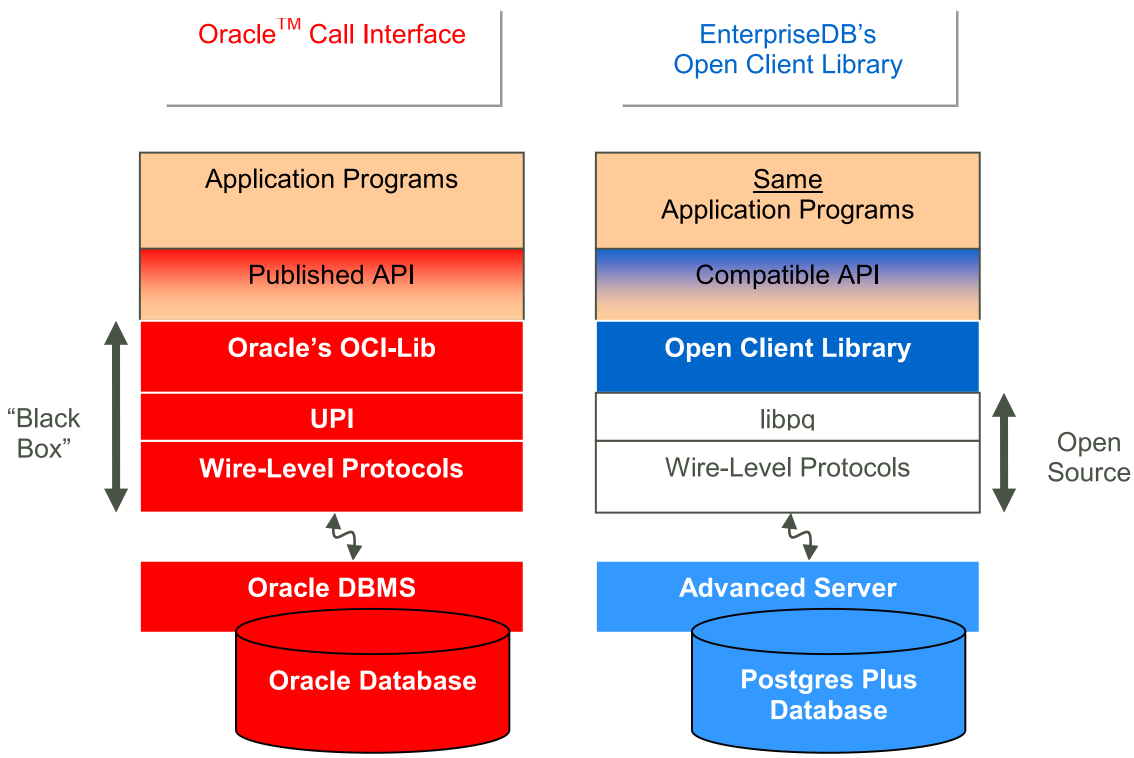


Figure 3 Open Client Library

7.2 Compiling and Linking a Program

The EnterpriseDB Open Client Library allows applications written using the Oracle Call Interface API to connect to and access an EnterpriseDB database with minimal changes to the C source code. The EnterpriseDB Open Client Library files are named:

On Linux:

```
libedboci.so
```

On Windows:

```
edboci.dll
```

The files are installed in the `connectors/edb-oci/lib` subdirectory.

Compiling and Linking a Sample Program

The following example compiles and links the sample program `edb_demo.c` in a Linux environment. The `edb_demo.c` is located in the `connectors/edb-oci/samples` subdirectory.

1. Set the `ORACLE_HOME` and `EDB_HOME` environment variables.

   Set `ORACLE_HOME` to the complete pathname of the Oracle home directory.

   For example:

   ```
   export
   ORACLE_HOME=/usr/lib/oracle/xe/app/oracle/product/10.2.0/serve
   r
   ```

   Set `EDB_HOME` to the complete pathname of the home directory.

   For example:

   ```
   export EDB_HOME=/opt/PostgresPlus
   ```

2. Set `LD_LIBRARY_PATH` to the complete path of `libpthread.so`. By default, `libpthread.so` is located in `/usr/lib`.

   ```
   export LD_LIBRARY_PATH=/usr/lib:$LD_LIBRARY_PATH
   ```

3. Set `LD_LIBRARY_PATH` to include the Advanced Server Open Client library. By default, `libiconv.so.2` is located in `$EDB_HOME/connectors/edb-oci/lib`.

```
export
LD_LIBRARY_PATH=$EDB_HOME/connectors/edb-oci:$EDB_HOME/
connectors/edb-oci/lib:$LD_LIBRARY_PATH
```

4. Then, compile and link the OCI API program.

```
cd $EDB_HOME/connectors/edb-oci/samples

make
```

7.3 Ref Cursor Support

The Advanced Server Open Client Library supports the use of Oracle-compatible REF CURSOR's as OUT parameters in PL/SQL procedures. Support is provided through the following API's:

- OCIBindByName
- OCIBindByPos
- OCIBindDynamic
- OCIStmtPrepare
- OCIStmtExecute
- OCIStmtFetch
- OCIAttrGet

OCL also supports the SQLT_RSET data type.

The following example demonstrates how to invoke a stored procedure that opens a cursor and returns a REF CURSOR as an output parameter. The code sample assumes that a PL/SQL procedure named openCursor (with an OUT parameter of type REF CURSOR) has been created on the database server, and that the required handles have been allocated:

```
char * openCursor =
  "begin \
     openCursor(:cmdRefCursor); \
   end;";
OCIStmt *stmtOpenRefCursor;
OCIStmt *stmtUseRefCursor;
```

Allocate handles for executing a stored procedure to open and use the REF CURSOR:

```
  /* Handle for the stored procedure to open the ref cursor */
  OCIHandleAlloc((dvoid *) envhp,
                 (dvoid **) &stmtOpenRefCursor,
                 OCI_HTYPE_STMT,
                 0,
                 (dvoid **) NULL));
```

```
  /* Handle for using the Ref Cursor */
  OCIHandleAlloc((dvoid *) envhp,
                 (dvoid **) &stmtUseRefCursor,
                 OCI_HTYPE_STMT,
                 0,
                 (dvoid **) NULL));
```

Then, prepare the PL/SQL block that is used to open the REF CURSOR:

```
  OCIStmtPrepare(stmtOpenRefCursor,
                  errhp,
                  (text *) openCursor,
                  (ub4) strlen(openCursor),
                  OCI_NTV_SYNTAX,
                  OCI_DEFAULT));
```

Bind the PL/SQL openCursor OUT parameter:

```
  OCIBindByPos(stmtOpenRefCursor,
                  &bndplrc1,
                  errhp,
                  1,
                  (dvoid*) &stmtUseRefCursor,
                          /* the returned ref cursor */
                  0,
                  SQLT_RSET,
                       /* SQLT_RSET type representing cursor
*/
                  (dvoid *) 0,
                  (ub2 *) 0,
                  (ub2) 0,
                  (ub4) 0,
                  (ub4 *) 0,
                  OCI_DEFAULT));
```

Use the stmtOpenRefCursor statement handle to call the openCursor procedure:

```
  OCIStmtExecute(svchp,
                  stmtOpenRefCursor,
                  errhp,
                  1,
                  0,
                  0,
                  0,
                  OCI_DEFAULT);
```

At this point, the stmtUseRefCursor statement handle contains the reference to the cursor. To obtain the information, define output variables for the ref cursor:

```
/* Define the output variables for the ref cursor */
  OCIDefineByPos(stmtUseRefCursor,
                  &defnEmpNo,
                  errhp,
                 (ub4) 1,
                 (dvoid *) &empNo,
                 (sb4) sizeof(empNo),
```

```
                     SQLT_INT,
                     (dvoid *) 0,
                     (ub2 *)0,
                     (ub2 *)0,
                     (ub4) OCI_DEFAULT));
```

Then, fetch the first row of the result set into the target variables:

```
/* Fetch the cursor data */
  OCIStmtFetch(stmtUseRefCursor,
                     errhp,
                     (ub4) 1,
                     (ub4) OCI_FETCH_NEXT,
                     (ub4) OCI_DEFAULT))
```

7.4 OCL Function Reference

The following tables list the functions supported in the Open Client Library. Note that any and all header files must be supplied by the user. Postgres Plus Advanced Server does not supply any such files.

7.4.1 Connect, Authorize and Initialize Functions

Table 9-7-1 Connect, Authorize, Terminate and Initialize Functions

Function	Description
`OCIBreak`	Aborts the specified OCI function.
`OCIEnvCreate`	Create an OCI environment.
`OCIEnvInit`	Initialize an OCI environment handle.
`OCIInitialize`	Initialize the OCI environment.
`OCILogoff`	Release a session.
`OCILogon`	Create a logon connection.
`OCILogon2`	Create a logon session in various modes.
`OCIReset`	Resets the current operation/protocol.
`OCIServerAttach`	Establish an access path to a data source. For information about using the `tnsnames.ora` file, see Section 9.6.
`OCIServerDetach`	Remove access to a data source.
`OCISessionBegin`	Create a user session.
`OCISessionEnd`	End a user session.
`OCISessionGet`	Get session from session pool.
`OCISessionRelease`	Release a session.
`OCITerminate`	Detach from shared memory subsystem.

7.4.1.1 Using the tnsnames.ora File

The `OCIServerAttach` method uses a connection descriptor specified in the `dblink` parameter of the `tnsnames.ora` file. Use the Oracle-compatible `tnsnames.ora` file to specify database connection addresses. Advanced Server searches the user's home directory for a file named `tnsnames.ora`. If Advanced Server doesn't find the `tnsnames.ora` file in the user's home directory, it searches the path specified by `TNS_ADMIN`.

The sample `tnsnames.ora` file contains:

```
 EDBX =
(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = localhost)(PORT = 5444))
  (CONNECT_DATA = (SERVER = DEDICATED)(SID = edb))
)
```

Any parameters not included in the sample, are ignored by the Open Client Library. In the sample, SID refers to the database named edb, in the cluster running on server 'localhost' at port 5444.

A C program call to OCIServerAttach that uses the tnsnames.ora file will look like:

```
static text *username = (text *) "enterprisedb";
static text *password = (text *) "edb";
static text *attach_str = "EDBX";
OCIServerAttach( srvhp, errhp, attach_str, strlen(attach_str),
0);
```

If you don't have a tnsnames.ora file, supply the connection string parameter in the form //localhost:5444/edbx.

7.4.2 Handle and Descriptor Functions

Table 9-7-2 Handle and Descriptor Functions

Function	Description
OCIAttrGet	Get handle attributes. Advanced server supports the following handle attributes: OCI_ATTR_USERNAME, OCI_ATTR_PASSWORD, OCI_ATTR_SERVER, OCI_ATTR_ENV, OCI_ATTR_SESSION, OCI_ATTR_ROW_COUNT, OCI_ATTR_CHARSET_FORM, OCI_ATTR_CHARSET_ID, EDB_ATTR_STMT_LEVEL_TX, OCI_ATTR_MODULE
OCIAttrSet	Set handle attributes. Advanced server supports the following handle attributes: OCI_ATTR_USERNAME, OCI_ATTR_PASSWORD, OCI_ATTR_SERVER, OCI_ATTR_ENV, OCI_ATTR_SESSION, OCI_ATTR_ROW_COUNT, OCI_ATTR_CHARSET_FORM, OCI_ATTR_CHARSET_ID, EDB_ATTR_STMT_LEVEL_TX, OCI_ATTR_MODULE
OCIDescriptorAlloc	Allocate and initialize a descriptor.
OCIDescriptorFree	Free an allocated descriptor.
OCIHandleAlloc	Allocate and initialize a handle.
OCIHandleFree	Free an allocated handle.
OCIParamGet	Get a parameter descriptor.
OCIParamSet	Set a parameter descriptor.

7.4.2.1 EDB_ATTR_EMPTY_STRINGS

By default, Advanced Server will treat an empty string as a NULL value. You can use the EDB_ATTR_EMPTY_STRINGS environment attribute to control the behavior of the OCL when mapping empty strings. To modify the mapping behavior, use the OCIAttrSet() function to set EDB_ATTR_EMPTY_STRINGS to one of the following:

Value	Description
OCI_DEFAULT	Treat an empty string as a NULL value.
EDB_EMPTY_STRINGS_NULL	Treat an empty string as a NULL value.
EDB_EMPTY_STRINGS_EMPTY	Treat an empty string as a string of zero length.

To find the value of EDB_ATTR_EMPTY_STRINGS, query OCIAttrGet().

7.4.2.2 EDB_ATTR_HOLDABLE

Advanced Server supports statements that execute as WITH HOLD cursors. The EDB_ATTR_HOLDABLE attribute specifies which statements execute as WITH HOLD cursors. The EDB_ATTR_HOLDABLE attribute can be set to any of the following three values:

- EDB_WITH_HOLD - execute as a WITH HOLD cursor
- EDB_WITHOUT_HOLD - execute using a protocol-level prepared statement
- OCI_DEFAULT - see the definition that follows

You can set the attribute in an OCIStmt handle or an OCIServer handle. When you create an OCIServer handle or an OCIStmt handle, the EDB_ATTR_HOLDABLE attribute for that handle is set to OCI_DEFAULT.

You can change the EDB_ATTR_HOLDABLE attribute for a handle by calling OCIAttrSet() and retrieve the attribute by calling OCIAttrGet().

When Advanced Server executes a SELECT statement, it examines the EDB_ATTR_HOLDABLE attribute in the OCIServer handle. If that attribute is set to EDB_WITH_HOLD, the query is executed as a WITH HOLD cursor.

If the EDB_ATTR_HOLDABLE attribute in the OCIServer handle is set to EDB_WITHOUT_HOLD, the query is executed as a normal prepared statement.

If the EDB_ATTR_HOLDABLE attribute in the OCIServer handle is set to OCI_DEFAULT, Advanced Server uses the value of the EDB_ATTR_HOLDABLE attribute in the OCIServer handle (if the EDB_ATTR_HOLDABLE attribute in the OCIServer is set to EDB_WITH_HOLD, the query executes as a WITH HOLD cursor, otherwise, the query executes as a protocol-prepared statement).

7.4.2.3 EDB_ATTR_STMT_LVL_TX

Unless otherwise instructed, the OCL library will `ROLLBACK` the current transaction whenever the server reports an error. If you choose, you can override the automatic `ROLLBACK` with the `edb_stmt_level_tx` parameter, which preserves modifications within a transaction, even if one (or several) statements raise an error within the transaction. For more information about `edb_stmt_level_tx`, see Section 1.3.4.

You can use the `OCIServer` attribute with `OCIAttrSet()` and `OCIAttrGet()` to enable or disable `EDB_ATTR_STMT_LEVEL_TX`. By default, `edb_stmt_level_tx` is disabled. To enable `edb_stmt_level_tx`, the client application must call `OCIAttrSet()`:

```
OCIServer *server  = myServer;
ub1        enabled = 1;

OCIAttrSet(server, OCI_HTYPE_SERVER, &enabled,
  sizeof(enabled), EDB_ATTR_STMT_LEVEL_TX, err);
```

To disable `edb_stmt_level_tx`:

```
OCIServer *server  = myServer;
ub1        enabled = 0;

OCIAttrSet(server, OCI_HTYPE_SERVER, &enabled,
  sizeof(enabled), EDB_ATTR_STMT_LEVEL_TX, err);
```

7.4.3 Bind, Define and Describe Functions

Table 9-7-3 Bind, Define, and Describe Functions

Function	Description
`OCIBindByName`	Bind by name.
`OCIBindByPos`	Bind by position.
`OCIBindDynamic`	Set additional attributes after bind.
`OCIBindArrayOfStruct`	Bind an array of structures for bulk operations.
`OCIDefineArrayOfStruct`	Specify the attributes of an array.
`OCIDefineByPos`	Define an output variable association.
`OCIDefineDynamic`	Set additional attributes for define.
`OCIDescribeAny`	Describe existing schema objects.
`OCIStmtGetBindInfo`	Get bind and indicator variable names and handle.
`OCIUserCallbackRegister`	Define a user-defined callback.

7.4.4 Statement Functions

Table 9-7-4 Statement Functions

Function	Description
`OCIStmtExecute`	Execute a prepared SQL statement.
`OCIStmtFetch`	Fetch rows of data (deprecated).
`OCIStmtFetch2`	Fetch rows of data.
`OCIStmtPrepare`	Prepare a SQL statement.
`OCIStmtPrepare2`	Prepare a SQL statement.
`OCIStmtRelease`	Release a statement handle.

7.4.5 Transaction Functions

Table 9-7-5 Transaction Functions

Function	Description
`OCITransCommit`	Commit a transaction.
`OCITransRollback`	Roll back a transaction.

7.4.6 XA Functions

Table 9-7-6 XA Functions

Function	Description
`xaoEnv`	Returns OCL environment handle.
`xaoSvcCtx`	Returns OCL service context.

7.4.6.1 xaoSvcCtx

In order to use the `xaoSvcCtx` function, extensions in the `xaoSvcCtx` or `xa_open` connection string format must be provided as follows:

```
Oracle_XA{+required_fields ...}
```

Where *`required_fields`* are the following:

`HostName=`*`host_ip_address`* specifies the IP address of the Advanced Server database.

`PortNumber=`*`host_port_number`* specifies the port number on which Advanced Server is running.

`SqlNet=`*`dbname`* specifies the database name.

`Acc=P/`*`username`*`/`*`password`* specifies the database username and password. *`password`* may be omitted in which case the field is specified as `Acc=P/username/`.

`AppName=app_id` specifies a number that identifies the application.

The following is an example of the connection string:

```
Oracle_XA+HostName=192.168.1.1+PortNumber=1533+SqlNet=XE+Acc=P/
user/password+AppName=1234
```

7.4.7 Date and Datetime Functions

Table 9-7-7 Date and Datetime Functions

Function	Description
`OCIDateAddDays`	Add or subtract a number of days.
`OCIDateAddMonths`	Add or subtract a number of months.
`OCIDateAssign`	Assign a date.
`OCIDateCheck`	Check if the given date is valid.
`OCIDateCompare`	Compare two dates.
`OCIDateDaysBetween`	Find the number of days between two dates.
`OCIDateFromText`	Convert a string to a date.
`OCIDateGetDate`	Get the date portion of a date.
`OCIDateGetTime`	Get the time portion of a date.
`OCIDateLastDay`	Get the date of the last day of the month.
`OCIDateNextDay`	Get the date of the next day.
`OCIDateSetDate`	Set the date portion of a date.
`OCIDateSetTime`	Set the time portion of a date.
`OCIDateSysDate`	Get the current system date and time.
`OCIDateToText`	Convert a date to a string.
`OCIDateTimeAssign`	Perform datetime assignment.
`OCIDateTimeCheck`	Check if the date is valid.
`OCIDateTimeCompare`	Compare two datetime values.
`OCIDateTimeConstruct`	Construct a datetime descriptor.
`OCIDateTimeConvert`	Convert one datetime type to another.
`OCIDateTimeFromArray`	Convert an array of size `OCI_DT_ARRAYLEN` to an `OCIDateTime` descriptor.
`OCIDateTimeFromText`	Convert the given string to Oracle datetime type in the `OCIDateTime` descriptor according to the specified format.
`OCIDateTimeGetDate`	Get the date portion of a datetime value.
`OCIDateTimeGetTime`	Get the time portion of a datetime value.
`OCIDateTimeGetTimeZoneName`	Get the time zone name portion of a datetime value.
`OCIDateTimeGetTimeZoneOffset`	Get the time zone (hour, minute) portion of a datetime value.
`OCIDateTimeSubtract`	Take two datetime values as input and return their difference as an interval.
`OCIDateTimeSysTimeStamp`	Get the system current date and time as a timestamp with time zone.
`OCIDateTimeToArray`	Convert an `OCIDateTime` descriptor to an array.
`OCIDateTimeToText`	Convert the given date to a string according to the specified format.

7.4.8 Interval Functions

Table 9-7-8 Interval Functions

Function	Description
OCIIntervalAdd	Adds two interval values.
OCIIntervalAssign	Copies one interval value into another interval value.
OCIIntervalCompare	Compares two interval values.
OCIIntervalGetDaySecond	Extracts days, hours, minutes, seconds and fractional seconds from an interval.
OCIIntervalSetDaySecond	Modifies days, hours, minutes, seconds and fractional seconds in an interval.
OCIIntervalGetYearMonth	Extracts year and month values from an interval.
OCIIntervalSetYearMonth	Modifies year and month values in an interval.
OCIIntervalDivide	Implements division of OCIInterval values by OCINumber values.
OCIIntervalMultiply	Implements multiplication of OCIInterval values by OCINumber values.
OCIIntervalSubtract	Subtracts one interval value from another interval value.
OCIIntervalToText	Extrapolates a character string from an interval.
OCIIntervalCheck	Verifies the validity of an interval value.
OCIIntervalToNumber	Converts an OCIInterval value into a OCINumber value.
OCIIntervalFromNumber	Converts a OCINumber value into an OCIInterval value.
OCIDateTimeIntervalAdd	Adds an OCIInterval value to an OCIDatetime value, resulting in an OCIDatetime value.
OCIDateTimeIntervalSub	Subtracts an OCIInterval value from an OCIDatetime value, resulting in an OCIDatetime value.
OCIIntervalFromText	Converts a text string into an interval.
OCIIntervalFromTZ	Converts a time zone specification into an interval value.

7.4.9 Number Functions

Table 9-7-9 Number Functions

Function	Description
OCINumberAbs	Compute the absolute value.
OCINumberAdd	Adds NUMBERs.
OCINumberArcCos	Compute the arc cosine.
OCINumberArcSin	Compute the arc sine.
OCINumberArcTan	Compute the arc tangent.
OCINumberArcTan2	Compute the arc tangent of two NUMBERs.
OCINumberAssign	Assign one NUMBER to another.
OCINumberCeil	Compute the ceiling of NUMBER.
OCINumberCmp	Compare NUMBERs.
OCINumberCos	Compute the cosine.
OCINumberDec	Decrement a NUMBER.
OCINumberDiv	Divide two NUMBERs.
OCINumberExp	Raise e to the specified NUMBER power.
OCINumberFloor	Compute the floor of a NUMBER.

Function	Description
OCINumberFromInt	Convert an integer to an Oracle NUMBER.
OCINumberFromReal	Convert a real to an Oracle NUMBER.
OCINumberFromText	Convert a string to an Oracle NUMBER.
OCINumberHypCos	Compute the hyperbolic cosine.
OCINumberHypSin	Compute the hyperbolic sine.
OCINumberHypTan	Compute the hyperbolic tangent.
OCINumberInc	Increments a NUMBER.
OCINumberIntPower	Raise a given base to an integer power.
OCINumberIsInt	Test if a NUMBER is an integer.
OCINumberIsZero	Test if a NUMBER is zero.
OCINumberLn	Compute the natural logarithm.
OCINumberLog	Compute the logarithm to an arbitrary base.
OCINumberMod	Modulo division.
OCINumberMul	Multiply NUMBERs.
OCINumberNeg	Negate a NUMBER.
OCINumberPower	Exponentiation to base e.
OCINumberPrec	Round a NUMBER to a specified number of decimal places.
OCINumberRound	Round a NUMBER to a specified decimal place.
OCINumberSetPi	Initialize a NUMBER to Pi.
OCINumberSetZero	Initialize a NUMBER to zero.
OCINumberShift	Multiply by 10, shifting specified number of decimal places.
OCINumberSign	Obtain the sign of a NUMBER.
OCINumberSin	Compute the sine.
OCINumberSqrt	Compute the square root of a NUMBER.
OCINumberSub	Subtract NUMBERs.
OCINumberTan	Compute the tangent.
OCINumberToInt	Convert a NUMBER to an integer.
OCINumberToReal	Convert a NUMBER to a real.
OCINumberToRealArray	Convert an array of NUMBER to a real array.
OCINumberToText	Converts a NUMBER to a string.
OCINumberTrunc	Truncate a NUMBER at a specified decimal place.

7.4.10 String Functions

Table 9-7-10 String Functions

Function	Description
OCIStringAllocSize	Get allocated size of string memory in bytes.
OCIStringAssign	Assign string to a string.
OCIStringAssignText	Assign text string to a string.
OCIStringPtr	Get string pointer.
OCIStringResize	Resize string memory.
OCIStringSize	Get string size.

7.4.11 Cartridge Services and File I/O Interface Functions

Table 9-7-11 Cartridge Services and File I/O Interface Functions

Function	Description
`OCIFileClose`	Close an open file.
`OCIFileExists`	Test to see if the file exists.
`OCIFileFlush`	Write buffered data to a file.
`OCIFileGetLength`	Get the length of a file.
`OCIFileInit`	Initialize the `OCIFile` package.
`OCIFileOpen`	Open a file.
`OCIFileRead`	Read from a file into a buffer.
`OCIFileSeek`	Change the current position in a file.
`OCIFileTerm`	Terminate the `OCIFile` package.
`OCIFileWrite`	Write `buflen` bytes into the file.

7.4.12 LOB Functions

Table 9-7-11 LOB Functions

Function	Description
`OCILobRead`	Returns a LOB value (or a portion of a LOB value).
`OCILOBWriteAppend`	Adds data to a LOB value.
`OCILobGetLength`	Returns the length of a LOB value.
`OCILobTrim`	Trims data from the end of a LOB value.
`OCILobOpen`	Opens a LOB value for use by other LOB functions.
`OCILobClose`	Closes a LOB value.

7.4.13 Miscellaneous Functions

Table 9-7-12 Miscellaneous Functions

Function	Description
`OCIClientVersion`	Return client library version.
`OCIErrorGet`	Return error message.
`OCIPGErrorGet`	Return native error messages reported by libpq or the server. The signature is: `sword OCIPGErrorGet(dvoid *hndlp, ub4 recordno, OraText *errcodep,ub4 errbufsiz, OraText *bufp, ub4 bufsiz, ub4 type)`
`OCIPasswordChange`	Change password.
`OCIPing`	Confirm that the connection and server are active.
`OCIServerVersion`	Get the Oracle version string.

7.4.14 Supported Data Types

Table 9-7-13 Supported Data Types

Function	Description
`ANSI_DATE`	ANSI date

SQLT_AFC	ANSI fixed character
SQLT_AVC	ANSI variable character
SQLT_BDOUBLE	Binary double
SQLT_BIN	Binary data
SQLT_BFLOAT	Binary float
SQLT_CHR	Character string
SQLT_DAT	Oracle date
SQLT_DATE	ANSI date
SQLT_FLT	Float
SQLT_INT	Integer
SQLT_LBI	Long binary
SQLT_LNG	Long
SQLT_LVB	Longer long binary
SQLT_LVC	Longer longs (character)
SQLT_NUM	Oracle numeric
SQLT_ODT	OCI date type
SQLT_STR	Zero-terminated string
SQLT_TIMESTAMP	Timestamp
SQLT_TIMESTAMP_TZ	Timestamp with time zone
SQLT_TIMESTAMP_LTZ	Timestamp with local time zone
SQLT_UIN	Unsigned integer
SQLT_VBI	VCS format binary
SQLT_VCS	Variable character
SQLT_VNU	Number with preceding length byte
SQLT_VST	OCI string type

7.5 Debugger

The Debugger is a tool that gives developers and DBAs the ability to test and debug Postgres Plus server-side programs using a graphical, dynamic environment. The types of programs that can be debugged are SPL stored procedures, functions, triggers, and packages as well as PL/pgSQL functions and triggers.

The Debugger is integrated with and invoked from the Postgres Enterprise Manager client. There are two basic ways the Debugger can be used to test programs:

- **Standalone Debugging.** The Debugger is used to start the program to be tested. You supply any input parameter values required by the program and you can immediately observe and step through the code of the program. Standalone debugging is the typical method used for new programs and for initial problem investigation.
- **In-Context Debugging.** The program to be tested is initiated by an application other than the Debugger. You first set a *global breakpoint* on the program to be tested. The application that makes the first call to the program encounters the global breakpoint. The application suspends execution at which point the Debugger takes control of the called program. You can then observe and step through the code of the called program as it runs within the context of the calling application. After you have completely stepped through the code of the called program in the Debugger, the suspended application resumes execution. In-context debugging is useful if it is difficult to reproduce a problem using standalone debugging due to complex interaction with the calling application.

The debugging tools and operations are the same whether using standalone or in-context debugging. The difference is in how the program to be debugged is invoked.

The following sections discuss the features and functionality of the Debugger using the standalone debugging method. The directions for starting the Debugger for in-context debugging are discussed in Section 7.5.5.3.

7.5.1 Configuring the Debugger

Before using the Debugger, edit the `postgresql.conf` file (located in the `data` subdirectory of your Postgres Plus Advanced Server home directory), adding `$libdir/plugin_debugger` to the libraries listed in the `shared_preload_libraries` configuration parameter:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/plugin_debugger'
```

After modifying the `shared_preload_libraries` parameter, you must restart the database server.

7.5.2 Starting the Debugger

You can use the Postgres Enterprise Manager (PEM) client to access the Debugger for standalone debugging. To open the Debugger, highlight the name of the stored procedure or function you wish to debug in the PEM `Object browser` panel. Then, navigate through the `Tools` menu to the `Debugging` menu and select `Debug` from the submenu as shown in Figure 7.1.

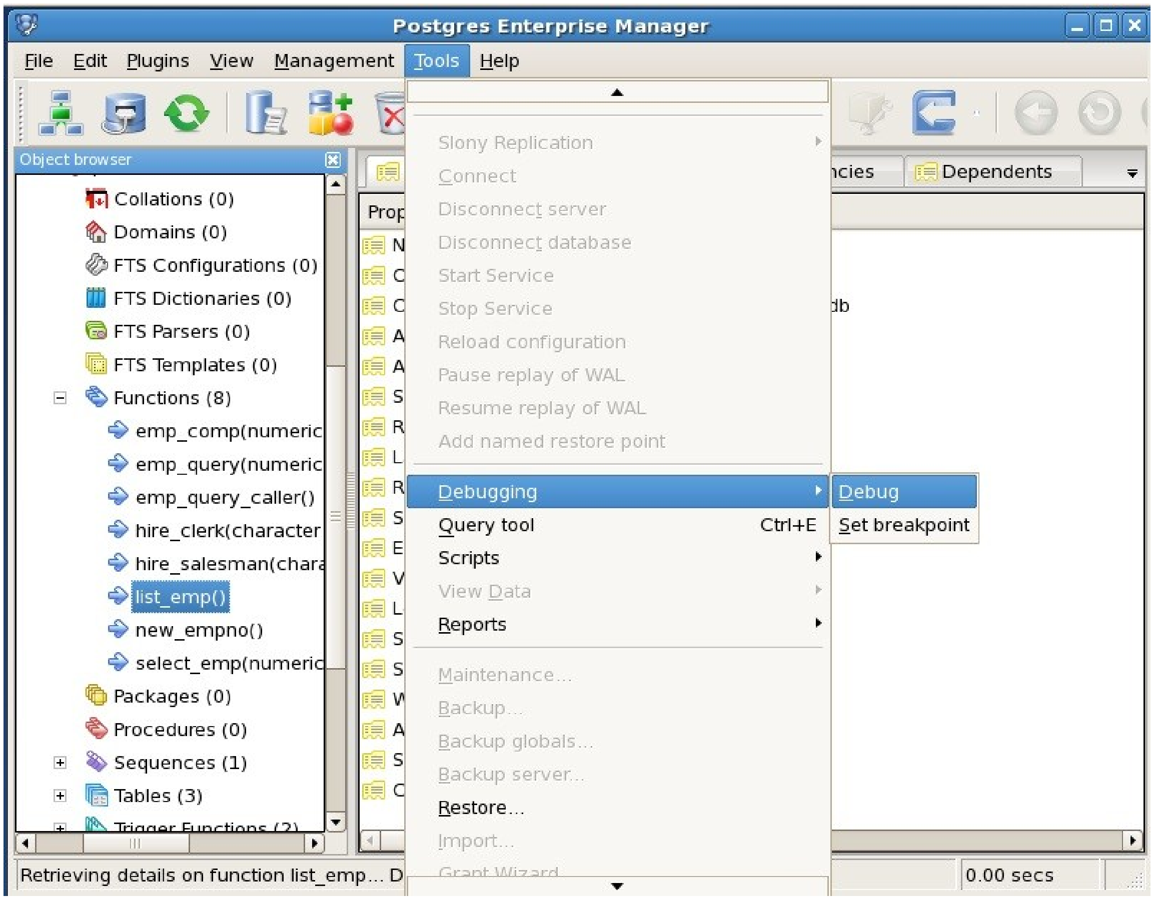


Figure 7.1 - Starting the Debugger from the Tools menu

You can also right-click on the name of the stored procedure or function in the PEM client `Object Browser`, and select `Debugging`, and the `Debug` from the context menu as shown in Figure 7.2.

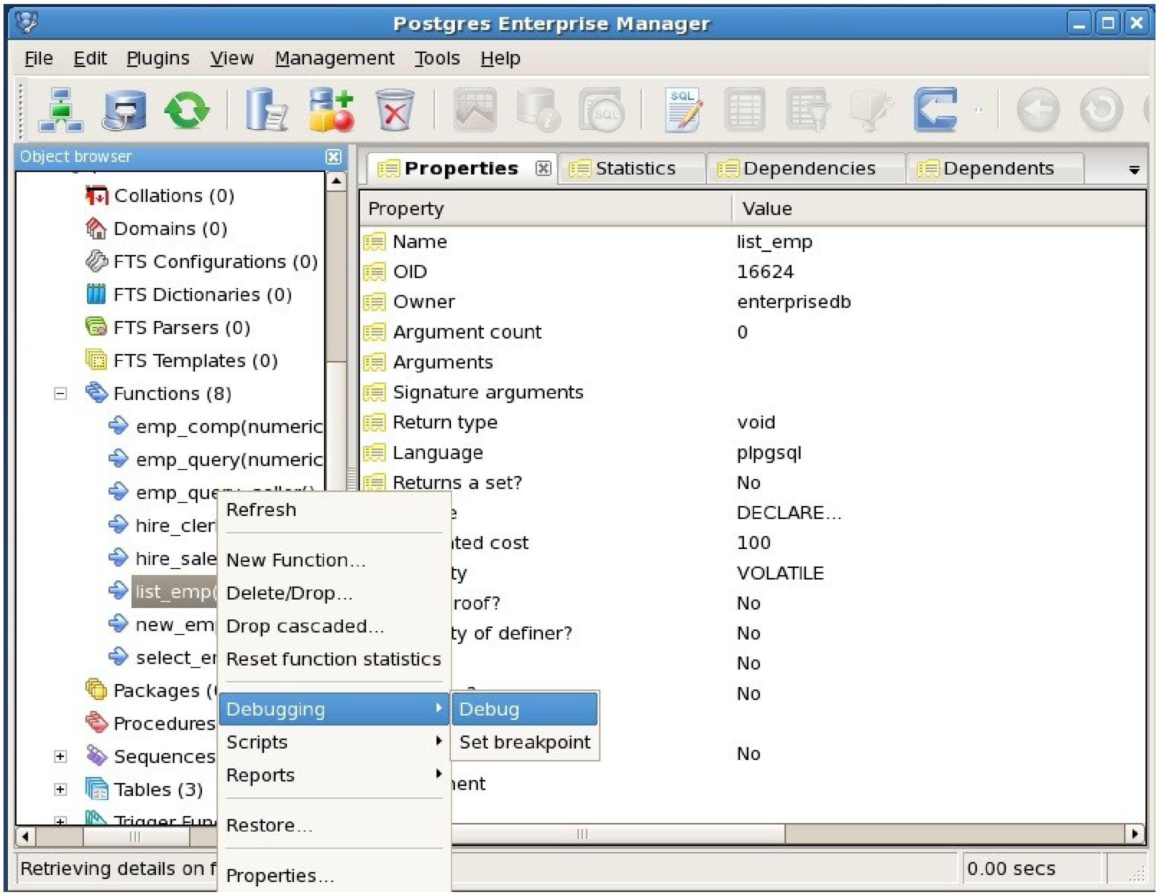


Figure 7.2 - Starting the Debugger from the object's context menu

Note that triggers cannot be debugged using standalone debugging. Triggers must be debugged using in-context debugging. See Section 7.5.5.3 for information on setting a global breakpoint for in-context debugging.

To debug a package, highlight the specific procedure or function under the package node of the package you wish to debug and follow the same directions as for stored procedures and functions.

7.5.3 The View Data Options Window

You can use the `View Data Options` window to pass parameter values when you are standalone-debugging a program that expects parameters. When you start the debugger, the `View Data Options` window opens automatically to display any `IN` or `IN OUT` parameters expected by the program. If the program declares no `IN` or `IN OUT` parameters, the `View Data Options` window does not open.

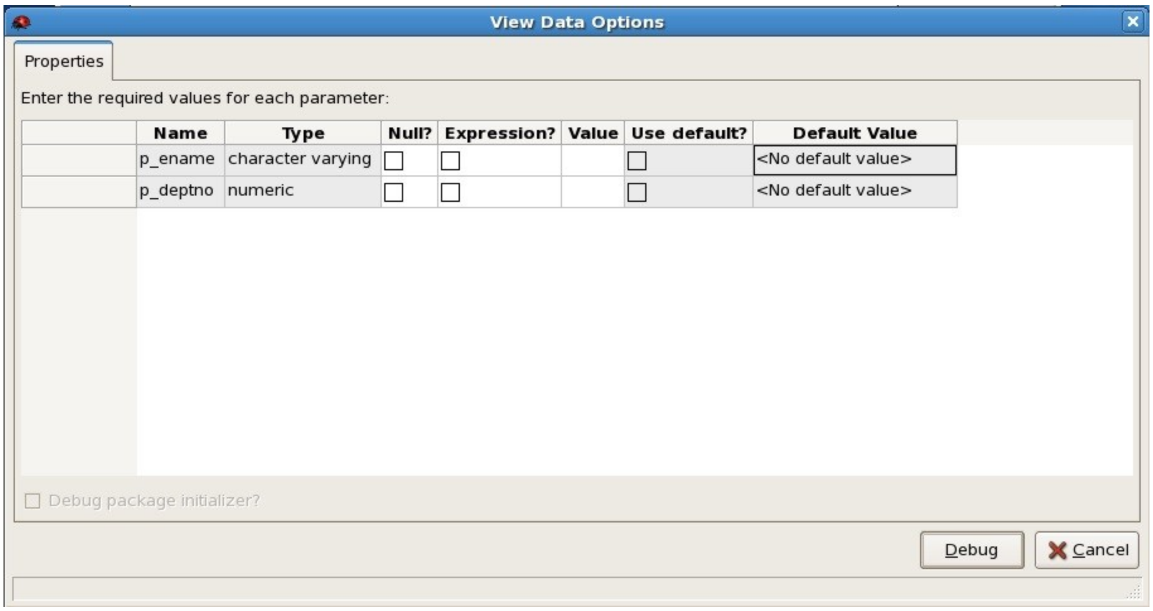


Figure 7.3 - The View Data Options window

Use the fields on the `View Data Options` window (shown in Figure 7.3) to provide a value for each parameter:

- The `Name` field contains the formal parameter name.
- The `Type` field contains the parameter data type.
- Check the `Null?` checkbox to indicate that the parameter is a `NULL` value.
- Check the `Expression` checkbox if the `Value` field contains an expression.
- The `Value` field contains the parameter value that will be passed to the program.
- Check the `Use default?` checkbox to indicate that the program should use the value in the `Default Value` field.
- The `Default Value` field contains the default value of the parameter.

Press the `Enter` key to select the next parameter in the list for data entry, or click on a `Value` field to select the parameter for data entry.

If you are debugging a procedure or function that is a member of a package that has an initialization section, check the `Debug Package Initializer` check box to instruct the Debugger to step into the package initialization section, allowing you to debug the initialization section code before debugging the procedure or function. If you do not select the check box, the Debugger executes the package initialization section without allowing you to see or step through the individual lines of code as they are executed.

After entering the desired parameter values, click the `OK` button to start the debugging process. Click the `Cancel` button to terminate the Debugger and return control to the PEM client.

Note: The `View Data Options` window does not open during in-context debugging. Instead, the application calling the program to be debugged must supply any required input parameter values.

When you have completed a full debugging cycle by stepping through the program code, the `View Data Options` window re-opens, allowing you to enter new parameter values and repeat the debugging cycle, or end the debugging session.

7.5.4 Main Debugger Window

The Main Debugger window (see Figure 7.4) contains three panes:

- the `Program Body` pane
- the `Stack` pane
- the `Output` pane

You can use the debugger menu bar or tool bar icons (located at the top of the debugger window) to access debugging functions.

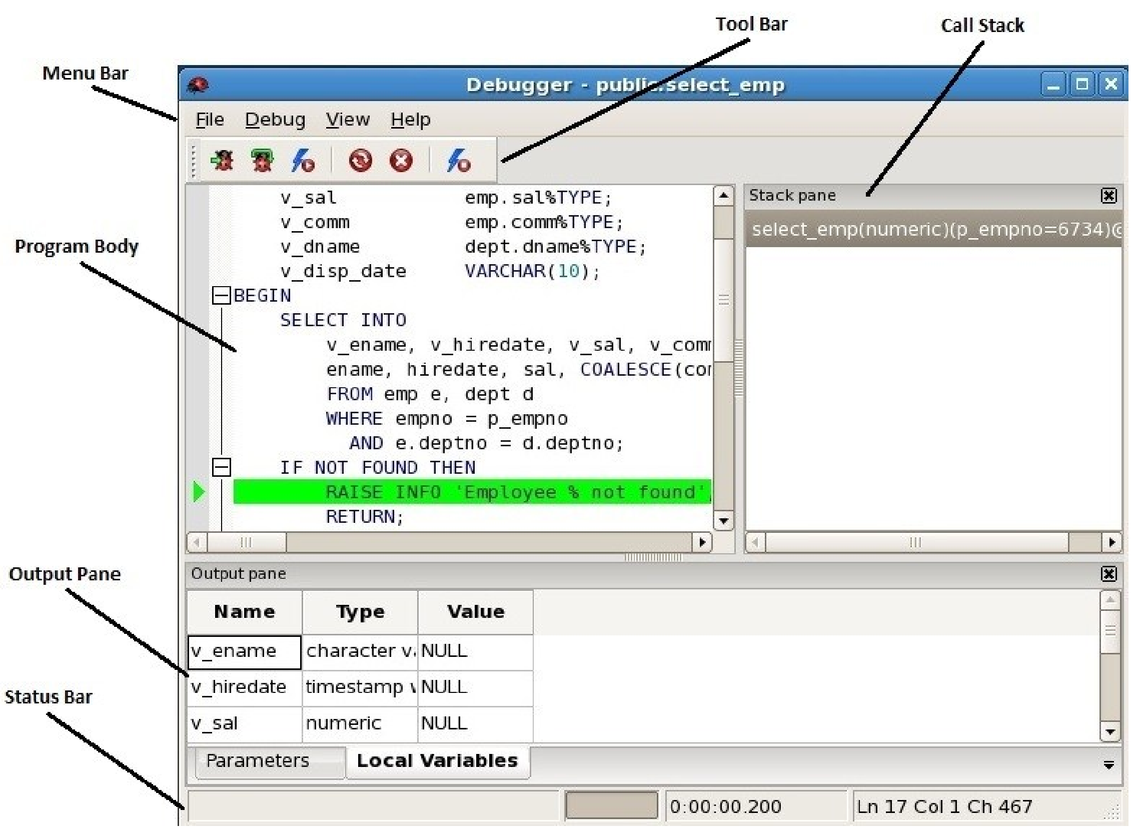


Figure 7.4 - The Main Debugger window

Status and error information is displayed in the status bar at the bottom of the Debugger window.

7.5.4.1 The Program Body Pane

The `Program Body` pane in the upper-left corner of the Debugger window displays the source code of the program that is being debugged.

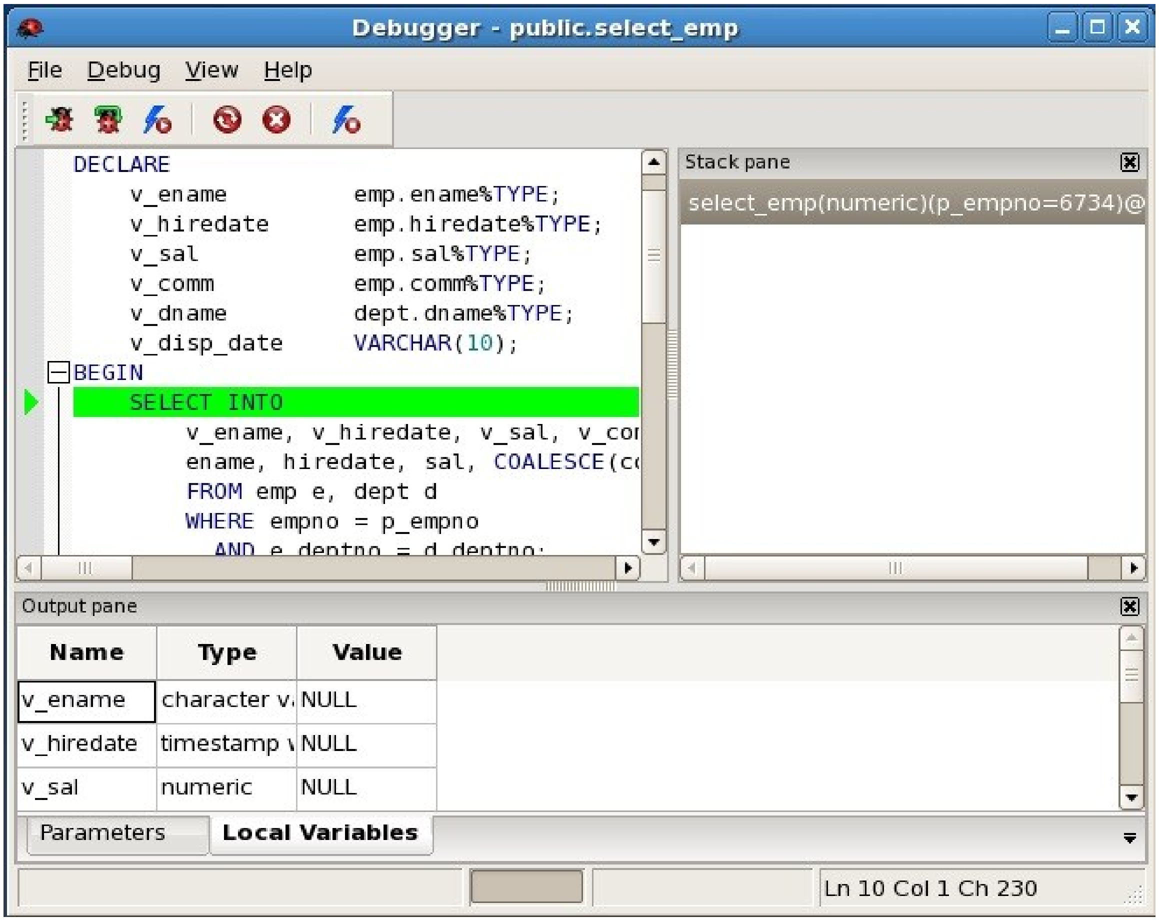


Figure 7.5 - The Program Body

Figure 7.5 shows that the Debugger is about to execute the `SELECT` statement. The green indicator in the program body highlights the next statement to execute.

7.5.4.2 The Stack Pane

The `Stack pane` displays a list of programs that are currently on the call stack (programs that have been invoked but which have not yet completed). When a program is called, the name of the program is added to the top of the list displayed in the `Stack pane`; when the program ends, its name is removed from the list.

The Stack pane also displays information about program calls. The information includes:

- The location of the call within the program
- The call arguments
- The name of the program being called

Reviewing the call stack can help you trace the course of execution through a series of nested programs.

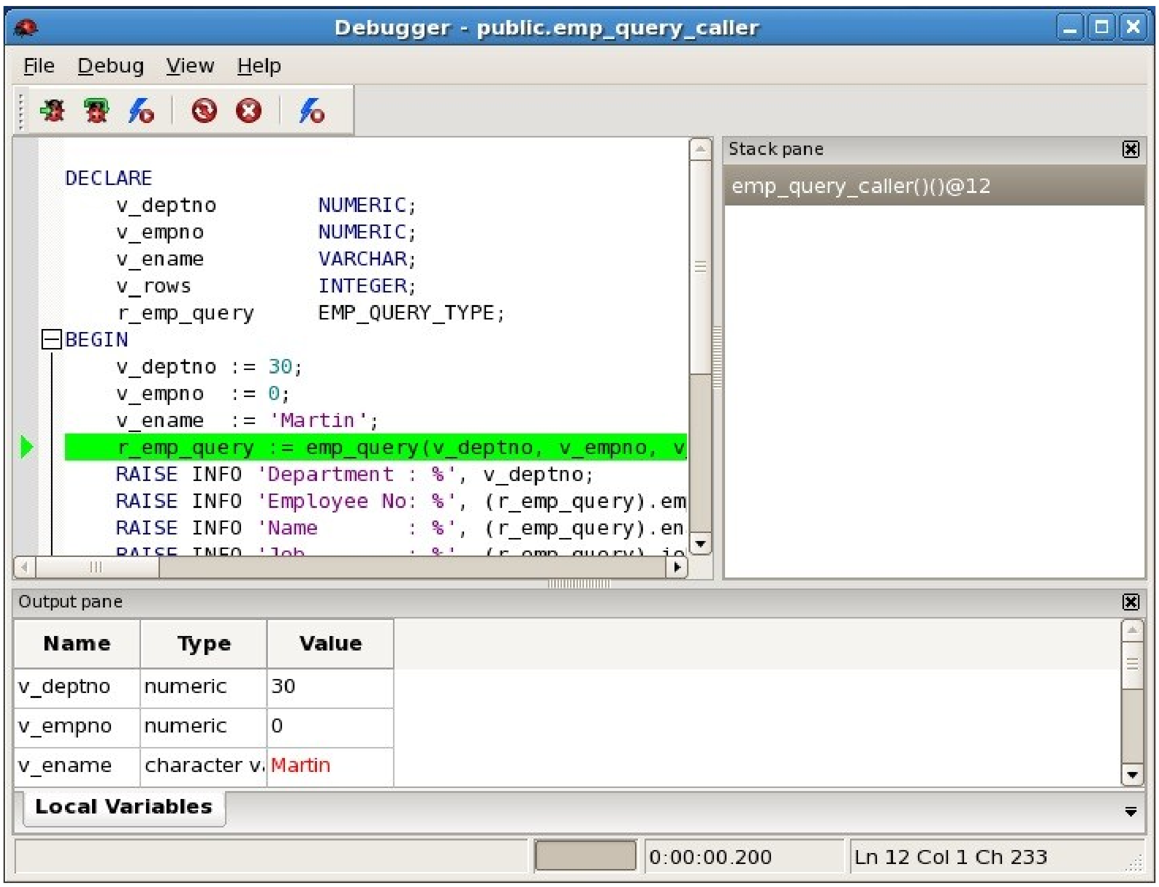


Figure 7.6 – A debugged program calling a subprogram

Figure 7.6 shows that `emp_query_caller` is about to call a subprogram named `emp_query`. `emp_query_caller` is currently at the top of the call stack.

After the call to `emp_query` executes, `emp_query` is displayed at the top of the `Stack pane`, and its code is displayed in the `Program Body` frame (see Figure 7.7).

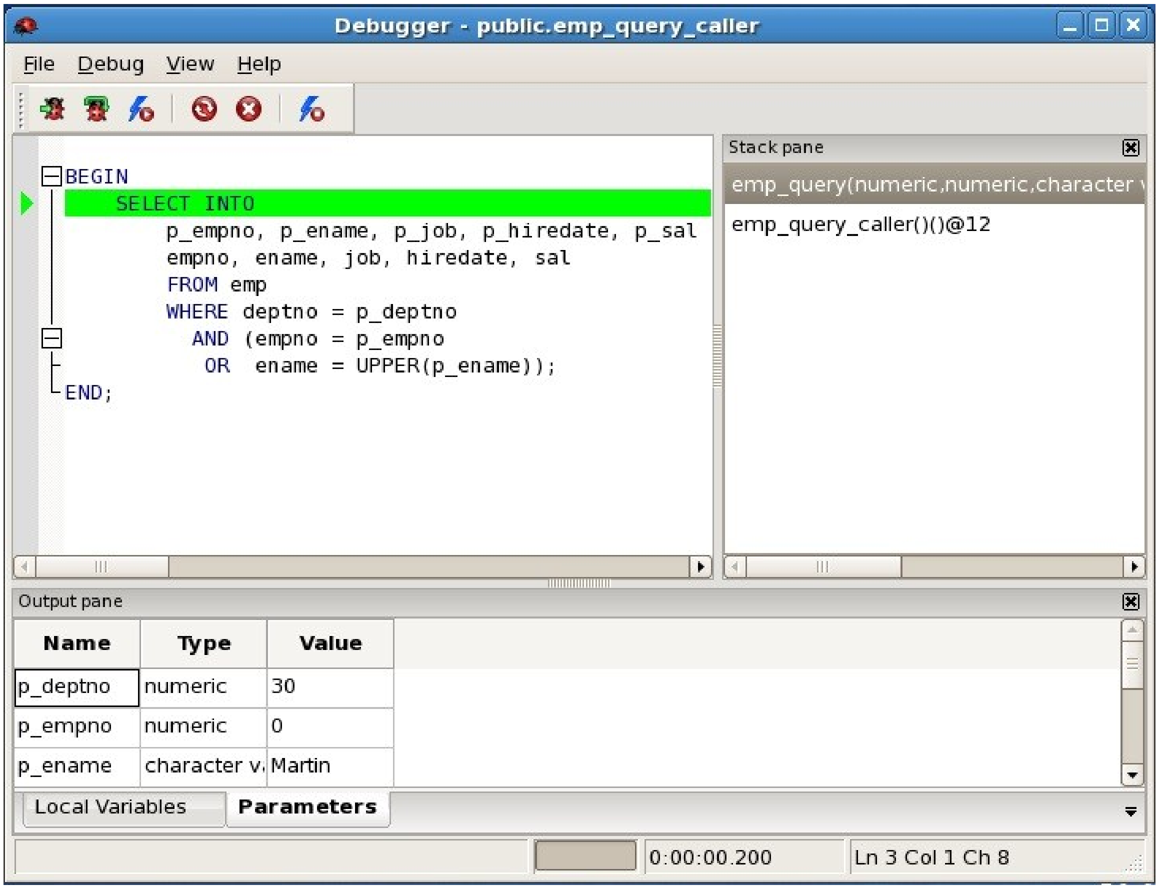


Figure 7.7 - Debugging the called subprogram

Upon completion of execution of the subprogram, control returns to the calling program (`public.emp_query_caller`), now displayed at the top of the `Stack pane` in Figure 7.8.

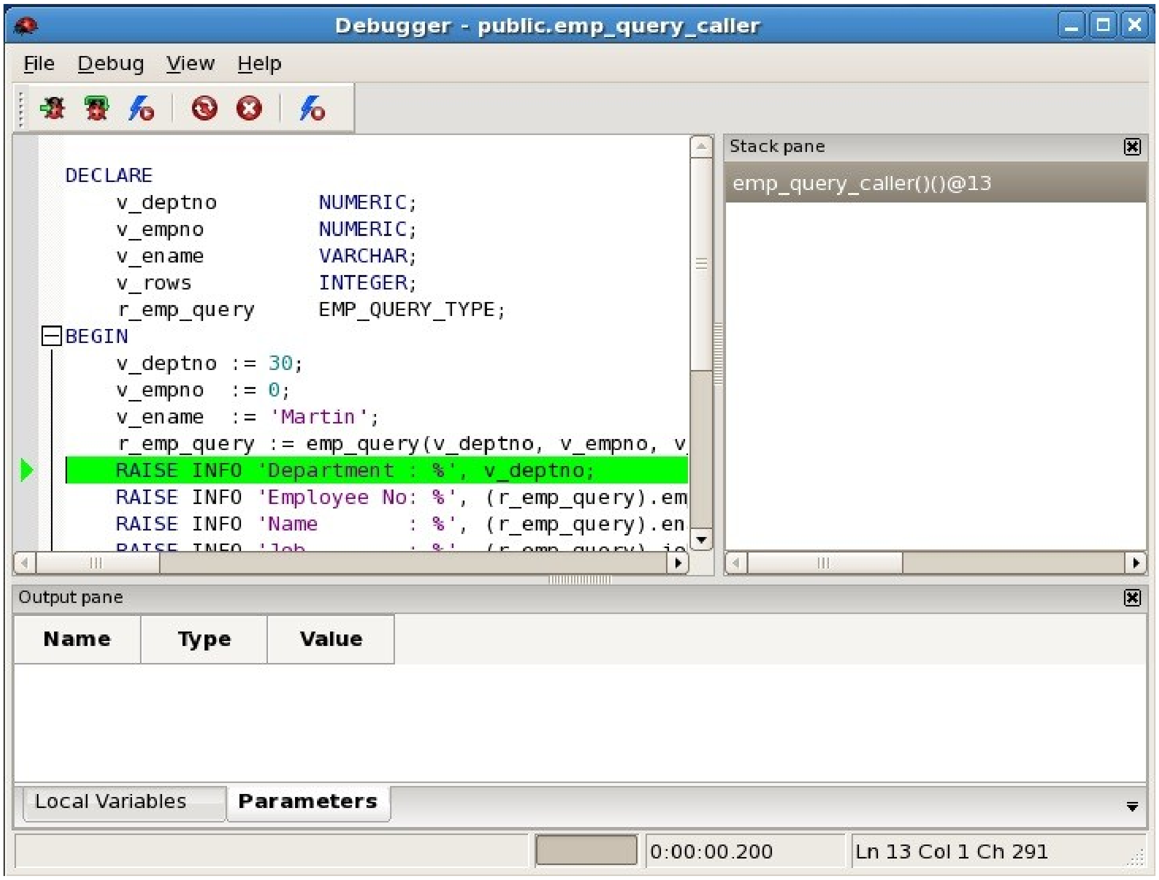


Figure 7.8 – Control returns from debugged subprogram

Highlight an entry in the call stack to review detailed information about the selected entry on the tabs in the `Output pane`. Using the call stack to navigate to another entry in the call stack will not alter the line that is currently executing.

7.5.4.3 The Output Pane

You can use tabs in the `Output pane` (see Figure 7.9) to view or modify parameter values or local variables, or to view messages generated by `RAISE INFO` and function results.

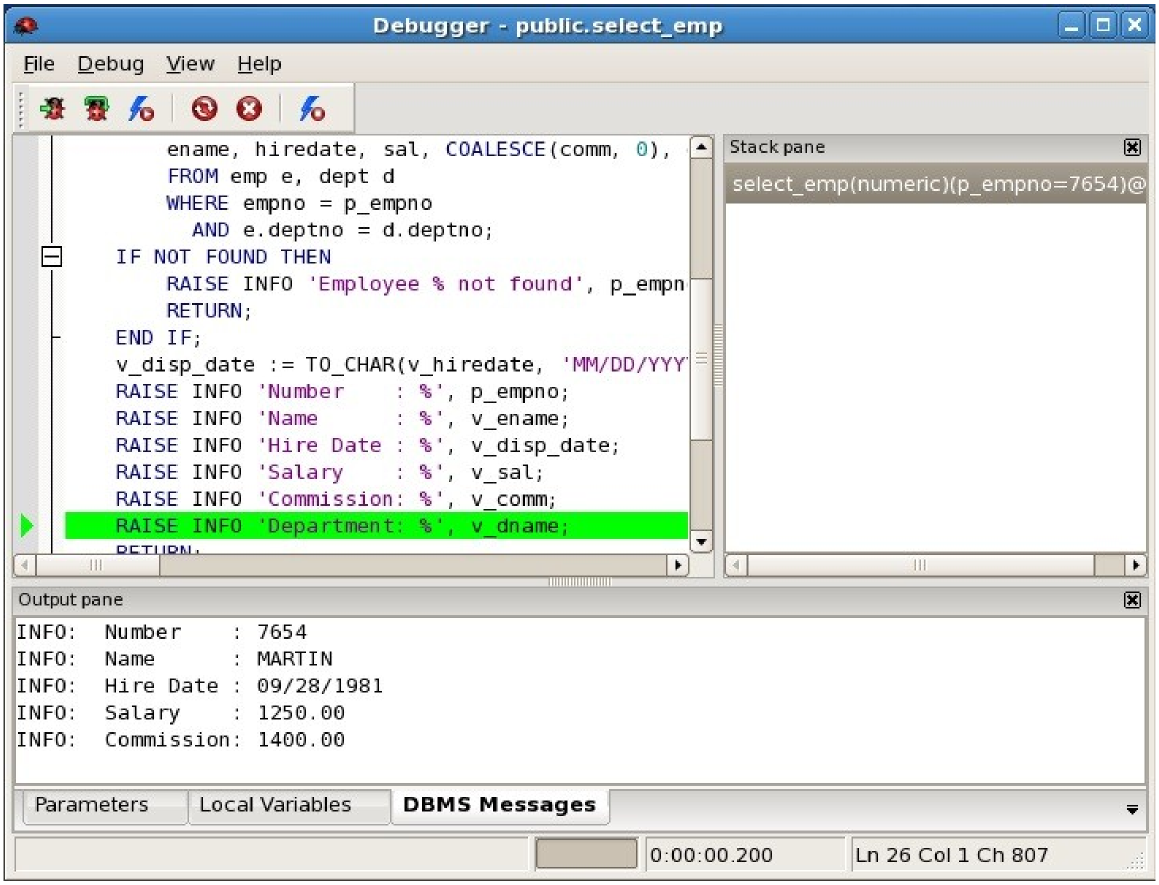


Figure 7.9 – The DBMS Messages tab of the Output pane.

Each tab contains a different type of information:

- The `Parameters` tab displays the current parameter values.
- The `Local Variables` tab displays the value of any variables declared within the program.
- The `DBMS Messages` tab displays any results returned by the program as it executes.
- The `Results` tab displays program results (if applicable).

7.5.4.4 The Status Bar

The status bar (see Figure 7.10) displays a message when the Debugger pauses, when a runtime error message is encountered, or when execution completes.

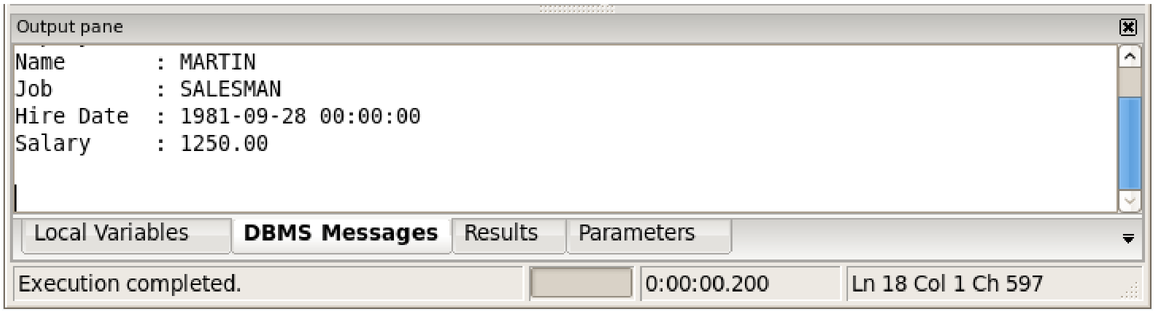


Figure 7.10 - The Status Bar, indicating Execution completed.

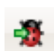
7.5.5 Debugging a Program

You can perform the following operations to debug a program:

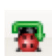
- Step through the program one line at a time
- Execute the program until you reach a breakpoint
- View and change local variable values within the program

7.5.5.1 Stepping Through the Code

Use the tool bar icons to step through a program with the Debugger:



Use the `Step Into` icon to execute the line of code currently highlighted by the green bar in the `Program Body` pane, and then pause execution. If the executed code line is a call to a subprogram, the called subprogram is brought into the `Program Body` pane, and the first executable line of code of the subprogram is highlighted as the Debugger waits for you to perform an operation on the subprogram.



Use the `Step Over` icon to execute a line of code, stepping over any subprograms invoked by that line of code. The subprogram is executed, but not debugged. If the subprogram contains a breakpoint, the debugger will stop at that breakpoint.

Use the `Continue` icon to execute the line of code highlighted by the green bar, and continue execution until either a breakpoint is encountered or the last line of the program has been executed.

Figure 7.11 shows the locations of the `Step Into`, `Step Over`, and `Continue` icons on the tool bar:

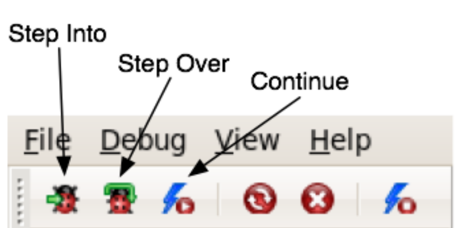


Figure 7.11 - The Step Into, Step Over, and Continue icons

The debugging operations are also accessible through the `Debug` menu, as shown in Figure 7.12.

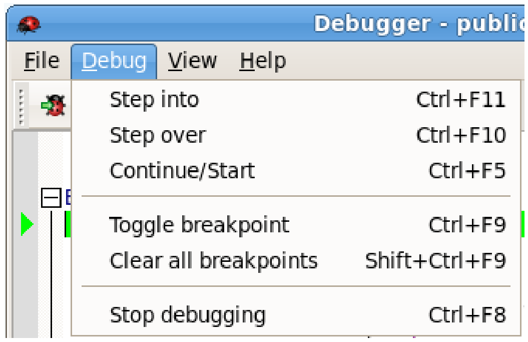


Figure 7.12 - Debug menu options

7.5.5.2 Using Breakpoints

As the Debugger executes a program, it pauses whenever it reaches a breakpoint. When the Debugger pauses, you can observe or change local variables, or navigate to an entry in the call stack to observe variables or set other breakpoints. The next step into, step over, or continue operation forces the debugger to resume execution with the next line of code following the breakpoint. There are two types of breakpoints:

Local Breakpoint - A local breakpoint can be set at any executable line of code within a program. The Debugger pauses execution when it reaches a line where a local breakpoint has been set.

Global Breakpoint - A global breakpoint will trigger when *any* session reaches that breakpoint. Set a global breakpoint if you want to perform in-context debugging of a program. When a global breakpoint is set on a program, the debugging session that set the global breakpoint waits until that program is invoked in another session. A global breakpoint can only be set by a superuser.

To create a local breakpoint, left-click in the grey shaded margin to the left of the line of code where you want the local breakpoint set. The Debugger displays a red dot in the margin, indicating a breakpoint has been set at the selected line of code (see Figure 7.13).

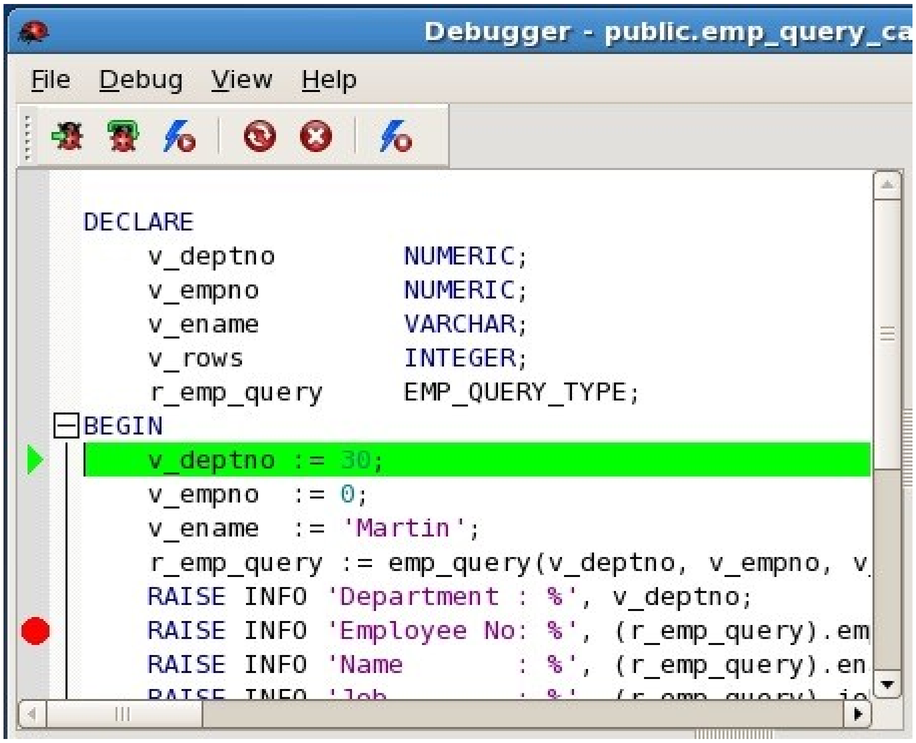


Figure 7.13 - Set a breakpoint by clicking in left-hand margin

You can also set a breakpoint by left-clicking in the `Program Body` to place your cursor, and selecting `Toggle Breakpoint` from `Debug` menu or by clicking the `Toggle Breakpoint` icon (see Figure 7.14). A red dot appears in the left-hand margin indicating a breakpoint has been set as the line of code.

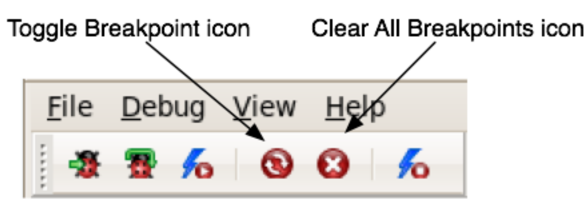


Figure 7.14 - The breakpoint control icons

You can set as many local breakpoints as desired. Local breakpoints remain in effect for the duration of a debugging session until they are removed.

Removing a Local Breakpoint

To remove a local breakpoint, you can:

- Left click the mouse on the red breakpoint indicator in the left margin of the `Program Body` pane. The red dot disappears, indicating that the breakpoint has been removed.
- Use your mouse to select the location of the breakpoint in the code body, and select `Toggle Breakpoint` from `Debug` menu, or click the `Toggle Breakpoint` icon.

You can remove all of the breakpoints from the program that currently appears in the `Program Body` frame by selecting `Clear all breakpoints` from the `Debug` menu (see Figure 7.15) or by clicking the `Clear All Breakpoints` icon.

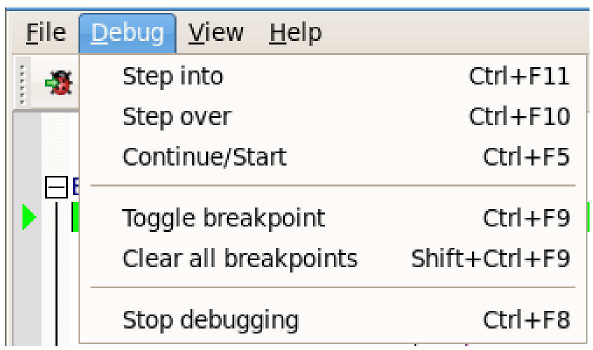


Figure 7.15 - The breakpoint menu options

Note: When you perform any of the preceding actions, only the breakpoints in the program that currently appears in the Program Body frame are removed. Breakpoints in called subprograms or breakpoints in programs that call the program currently appearing in the Program Body frame are not removed.

7.5.5.3 Setting a Global Breakpoint for In-Context Debugging

To set a global breakpoint for in-context debugging, highlight the stored procedure, function, or trigger on which you wish to set the breakpoint in the `Object browser` panel. Navigate through the `Tools` menu to select `Debugging`, and then `Set Breakpoint` (see Figure 7.16)

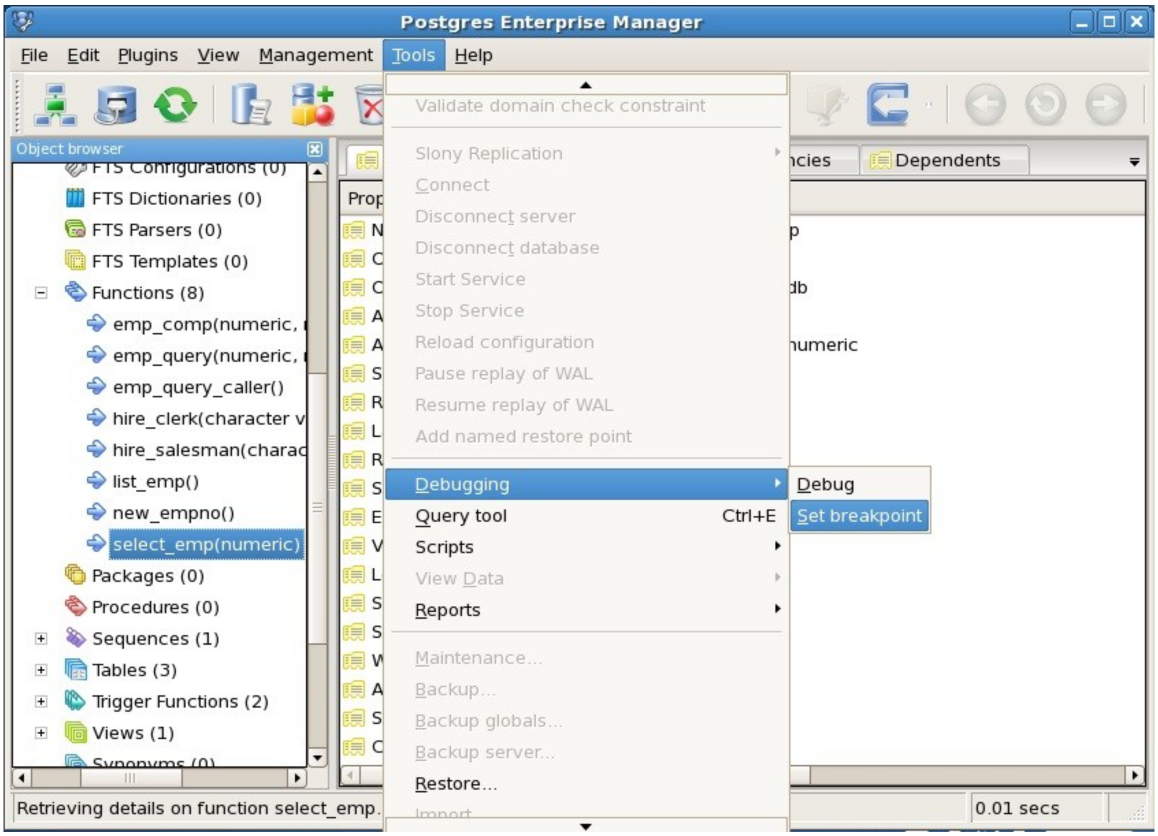


Figure 7.16 - Setting a global breakpoint from the Tools menu

Alternatively, you can right-click on the name of the stored procedure, function, or trigger on which you wish to set a global breakpoint and select `Debugging`, then `Set Breakpoint` from the context menu as shown in Figure 7.17.

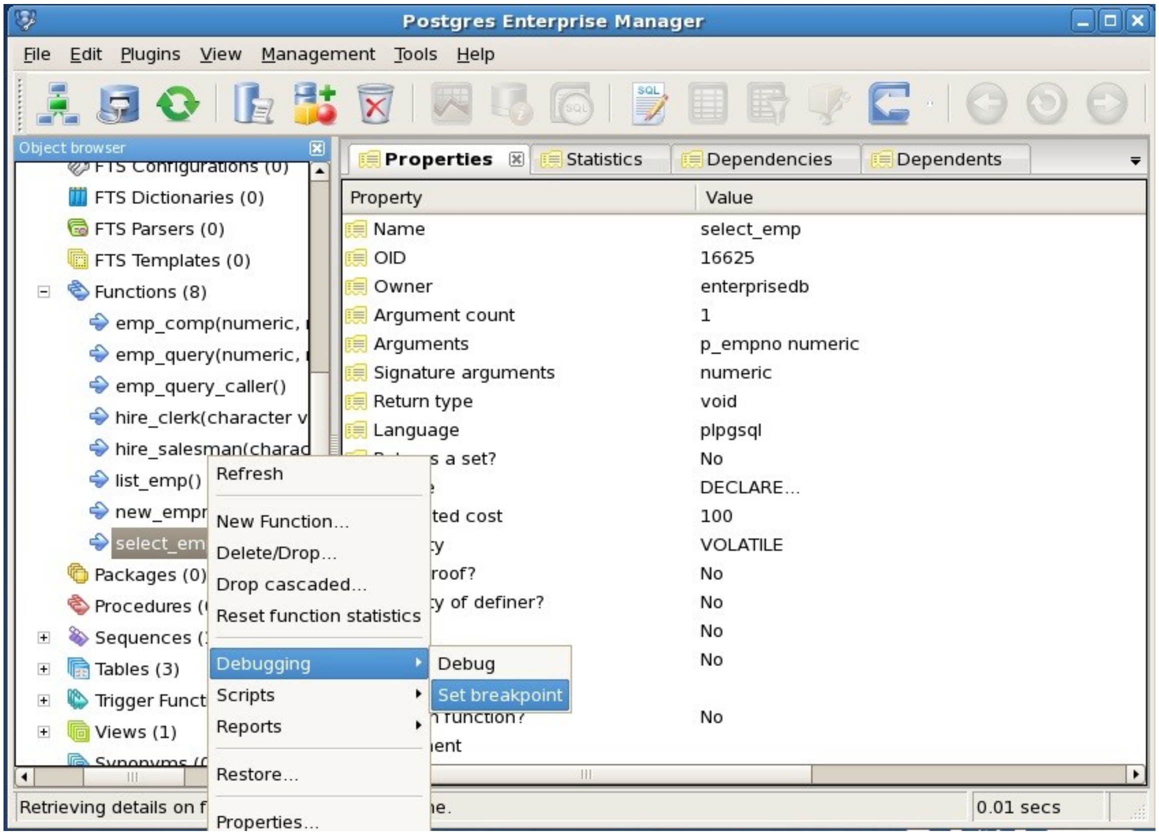


Figure 7.17 - Setting a global breakpoint from the object's context menu

To set a global breakpoint on a trigger, expand the table node that contains the trigger, highlight the specific trigger you wish to debug, and follow the same directions as for stored procedures and functions.

To set a global breakpoint in a package, highlight the specific procedure or function under the package node of the package you wish to debug and follow the same directions as for stored procedures and functions.

After you choose `Set Breakpoint`, the Debugger window opens and waits for an application to call the program to be debugged (see Figure 7.18).

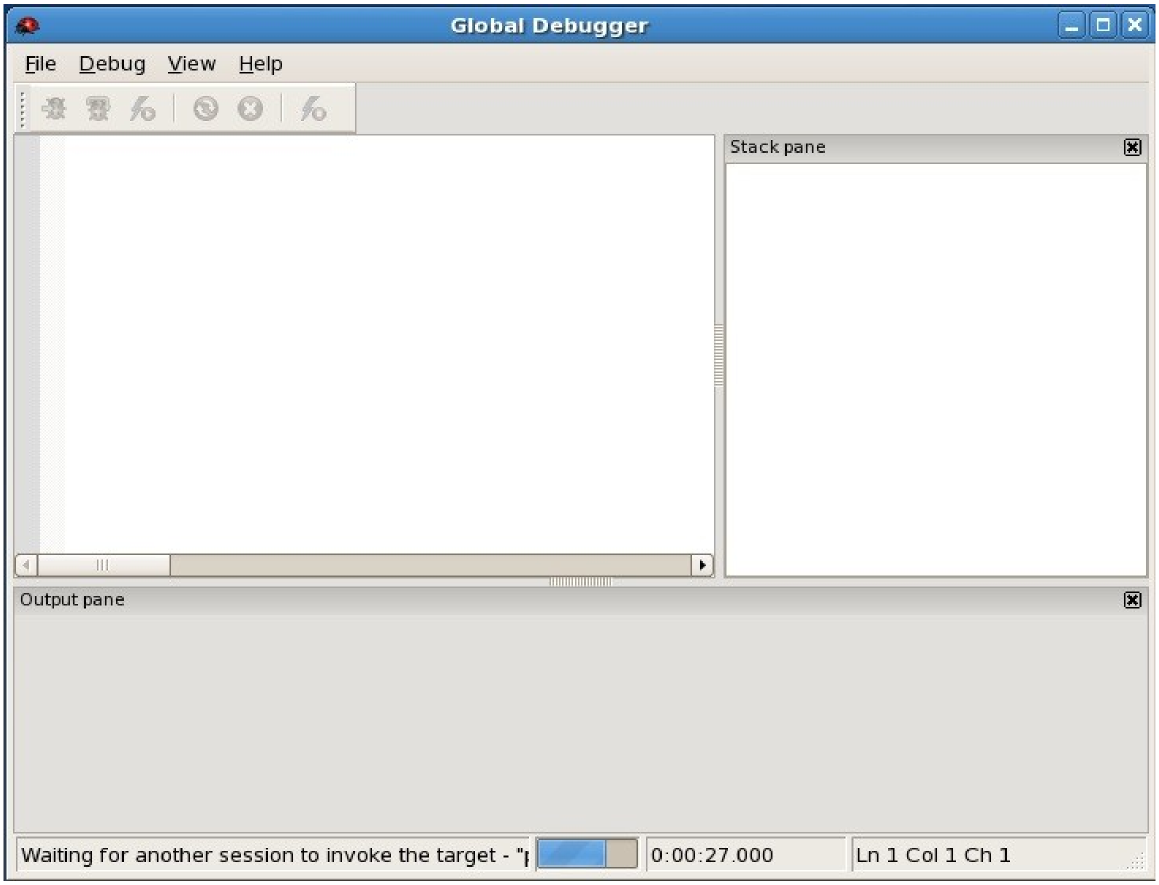


Figure 7.18 - Waiting for invocation of program to be debugged

In Figure 7.19, the EDB-PSQL client invokes the `select_emp` function (on which a global breakpoint has been set).

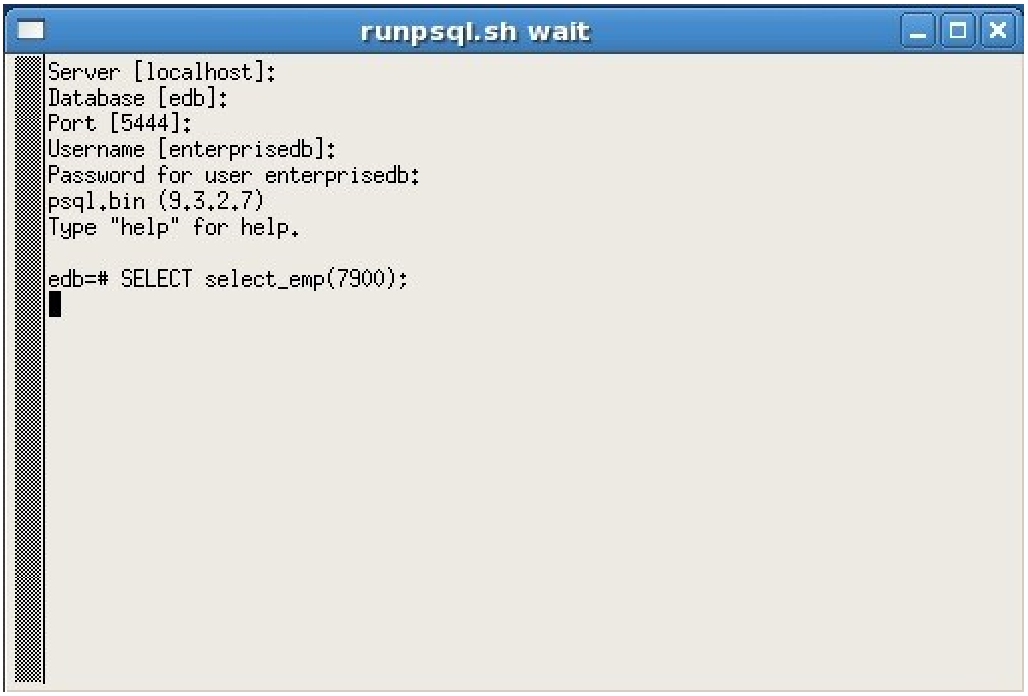


Figure 7.19 - Application invoking program with a global breakpoint

The `select_emp` function does not complete until you step through the program in the Debugger, which now appears as shown in Figure 7.20.

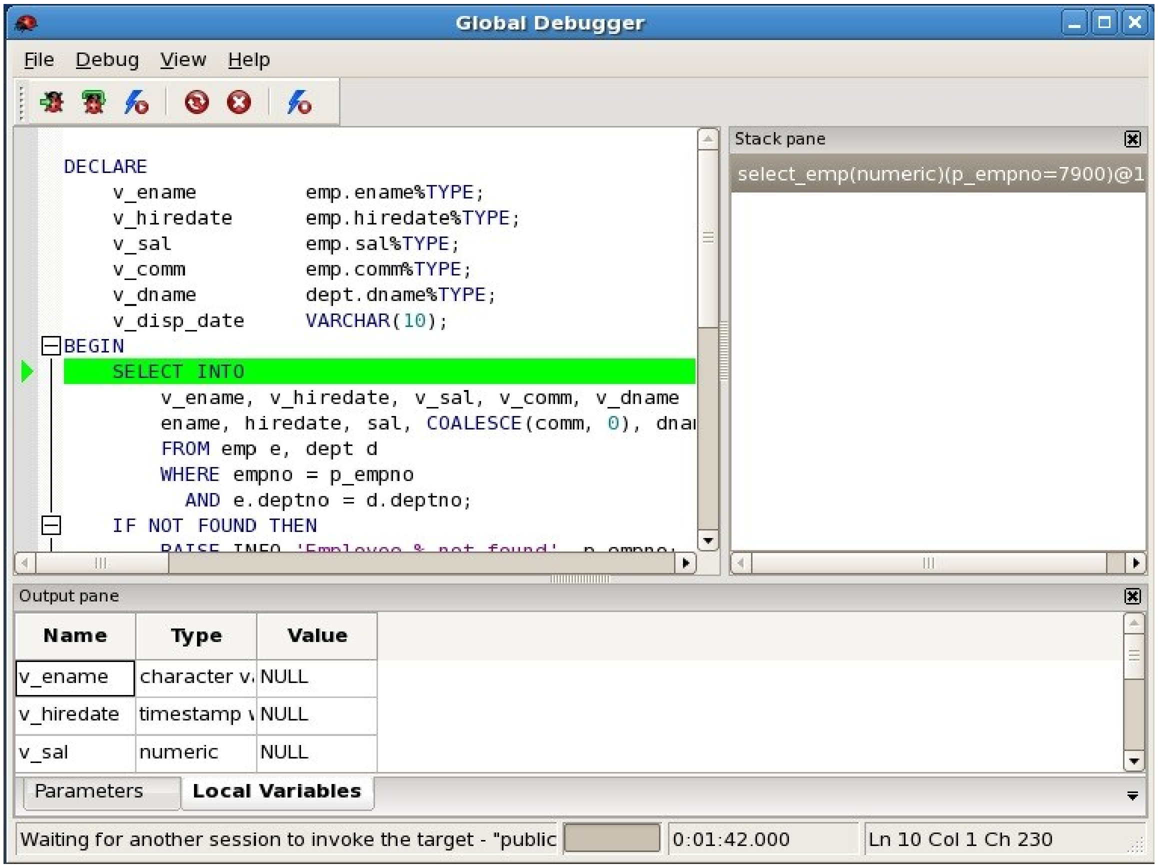


Figure 7.20 - Program on which a global breakpoint has been set

You can now debug the program using any of the previously discussed operations such as step into, step over, and continue, or set local breakpoints. When you have stepped through execution of the program, the calling application (EDB-PSQL) regains control as shown in Figure 7.21.

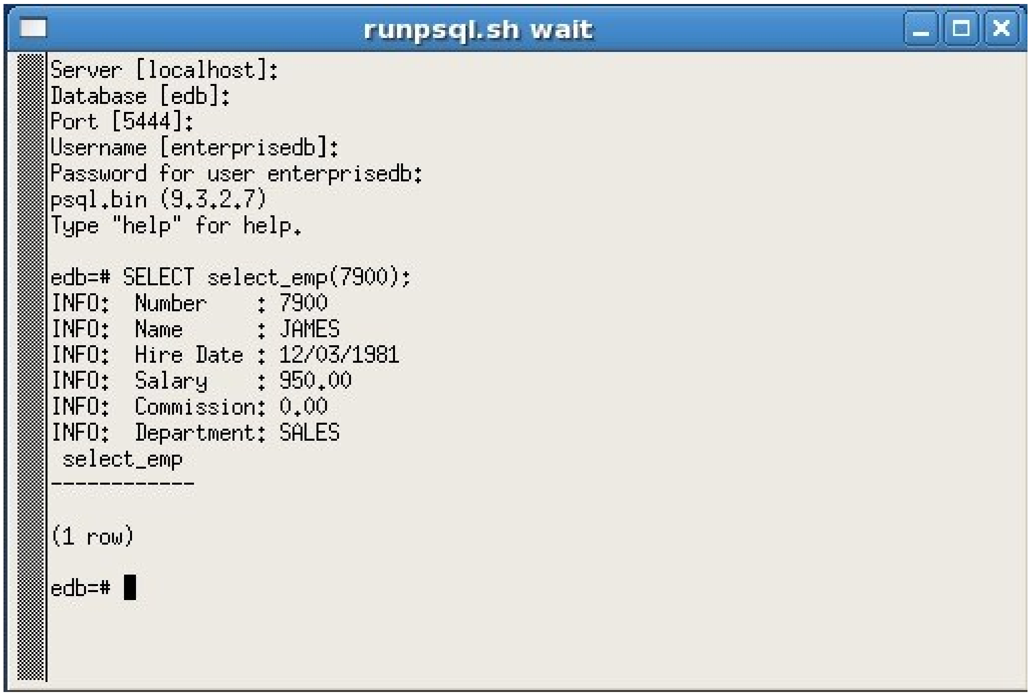


Figure 7.21 - Application after debugging

The `select_emp` function completes execution and its output is displayed.

At this point, you can end the Debugger session by choosing `Exit` from the `File` menu. If you do not end the Debugger session, the next application that invokes the program will encounter the global breakpoint and the debugging cycle will begin again.

7.5.5.4 Exiting the Debugger

To end a Debugger session and exit the Debugger, select `Exit` from `File` menu or press Alt-F4 as shown by the following:

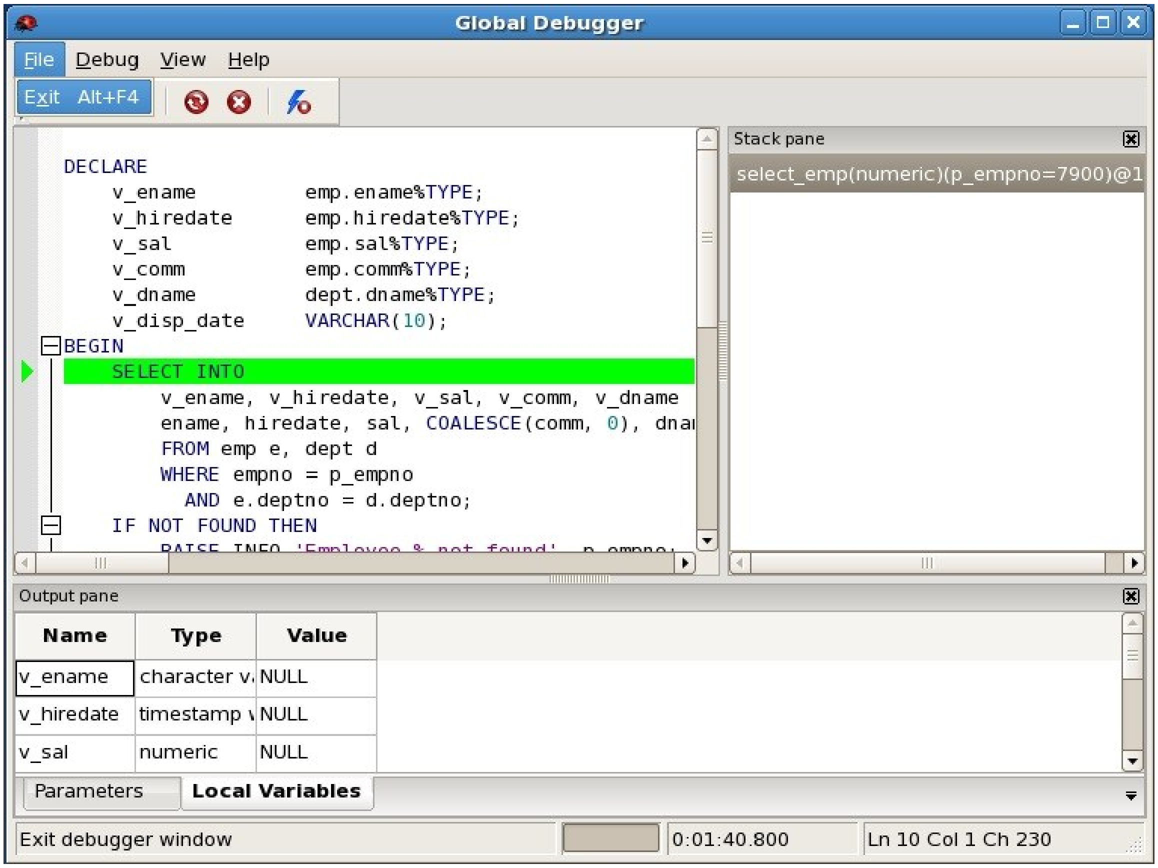


Figure 7.22 - Exiting from the Debugger

8 Performance Analysis and Tuning

Postgres Plus Advanced Server provides various tools for performance analysis and tuning. These features are described in this chapter.

8.1 Dynatune

Postgres Plus Advanced Server supports dynamic tuning of the database server to make the optimal usage of the system resources available on the host machine on which it is installed. The two parameters that control this functionality are located in the `postgresql.conf` file. These parameters are:

- `edb_dynatune`
- `edb_dynatune_profile`

8.1.1 edb_dynatune

`edb_dynatune` determines how much of the host system's resources are to be used by the database server based upon the host machine's total available resources and the intended usage of the host machine.

When Postgres Plus Advanced Server is initially installed, the `edb_dynatune` parameter is set in accordance with the selected usage of the host machine on which it was installed - i.e., development machine, mixed use machine, or dedicated server. For most purposes, there is no need for the database administrator to adjust the various configuration parameters in the `postgresql.conf` file in order to improve performance.

You can change the value of the `edb_dynatune` parameter after the initial installation of Postgres Plus Advanced Server by editing the `postgresql.conf` file. The postmaster must be restarted in order for the new configuration to take effect.

The `edb_dynatune` parameter can be set to any integer value between 0 and 100, inclusive. A value of 0, turns off the dynamic tuning feature thereby leaving the database server resource usage totally under the control of the other configuration parameters in the `postgresql.conf` file.

A low non-zero, value (e.g., 1 - 33) dedicates the least amount of the host machine's resources to the database server. This setting would be used for a development machine where many other applications are being used.

A value in the range of 34 - 66 dedicates a moderate amount of resources to the database server. This setting might be used for a dedicated application server that may have a fixed number of other applications running on the same machine as Postgres Plus Advanced Server.

The highest values (e.g., 67 - 100) dedicate most of the server's resources to the database server. This setting would be used for a host machine that is totally dedicated to running Postgres Plus Advanced Server.

Once a value of `edb_dynatune` is selected, database server performance can be further fine-tuned by adjusting the other configuration parameters in the `postgresql.conf` file. Any adjusted setting overrides the corresponding value chosen by `edb_dynatune`. You can change the value of a parameter by un-commenting the configuration parameter, specifying the desired value, and restarting the database server.

8.1.2 edb_dynatune_profile

The `edb_dynatune_profile` parameter is used to control tuning aspects based upon the expected workload profile on the database server. This parameter takes effect upon startup of the database server.

The possible values for `edb_dynatune_profile` are:

Value	Usage
`oltp`	Recommended when the database server is processing heavy online transaction processing workloads.
`reporting`	Recommended for database servers used for heavy data reporting.
`mixed`	Recommended for servers that provide a mix of transaction processing and data reporting.

8.2 Infinite Cache

Note: Infinite Cache has been deprecated and may be removed in a future release. Please contact your EnterpriseDB Account Manager or mailto:sales@enterprisedb.com for more information.

Database performance is typically governed by two competing factors:

- Memory access is fast; disk access is slow.
- Memory space is scarce; disk space is abundant.

Postgres Plus Advanced Server tries very hard to minimize disk I/O by keeping frequently used data in memory. When the first server process starts, it creates an in-memory data structure known as the *buffer cache*. The buffer cache is organized as a collection of 8K (8192 byte) pages: each page in the buffer cache corresponds to a page in some table or index. The buffer cache is shared between all processes servicing a given database.

When you select a row from a table, Advanced Server reads the page that contains the row into the shared buffer cache. If there isn't enough free space in the cache, Advanced Server *evicts* some other page from the cache. If Advanced Server evicts a page that has been modified, that data is written back out to disk; otherwise, it is simply discarded. Index pages are cached in the shared buffer cache as well.

Figure 1.1 demonstrates the flow of data in a typical Advanced Server session:

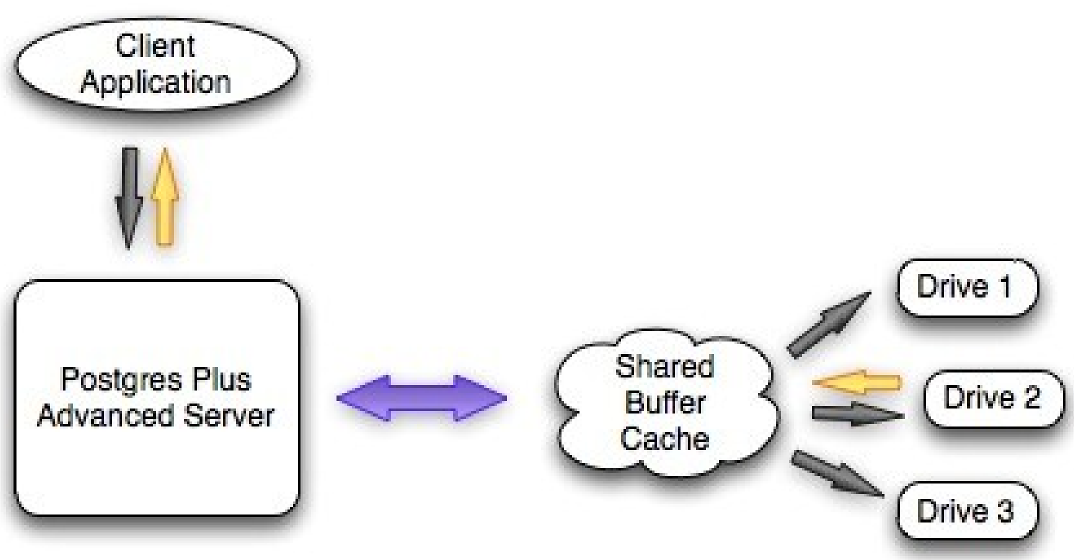


Figure 1.1 – Data Flow

A client application sends a query to the Postgres server and the server searches the shared buffer cache for the required data. If the requested data is found in the cache, the server immediately sends the data back to the client. If not, the server reads the page that holds the data into the shared buffer cache, evicting one or more pages if necessary. If the server decides to evict a page that has been modified, that page is written to disk.

As you can see, a query will execute much faster if the required data is found in the shared buffer cache.

One way to improve performance is to increase the amount of memory that you can devote to the shared buffer cache. However, most computers impose a strict limit on the amount of RAM that you can install. To help circumvent this limit, Infinite Cache lets you utilize memory from other computers connected to your network.

With Infinite Cache properly configured, Advanced Server will dedicate a portion of the memory installed on each *cache server* as a secondary memory cache. When a client application sends a query to the server, the server first searches the shared buffer cache for the required data; if the requested data is not found in the cache, the server searches for the necessary page in one of the cache servers.

Figure 1.2 shows the flow of data in an Advanced Server session with Infinite Cache:

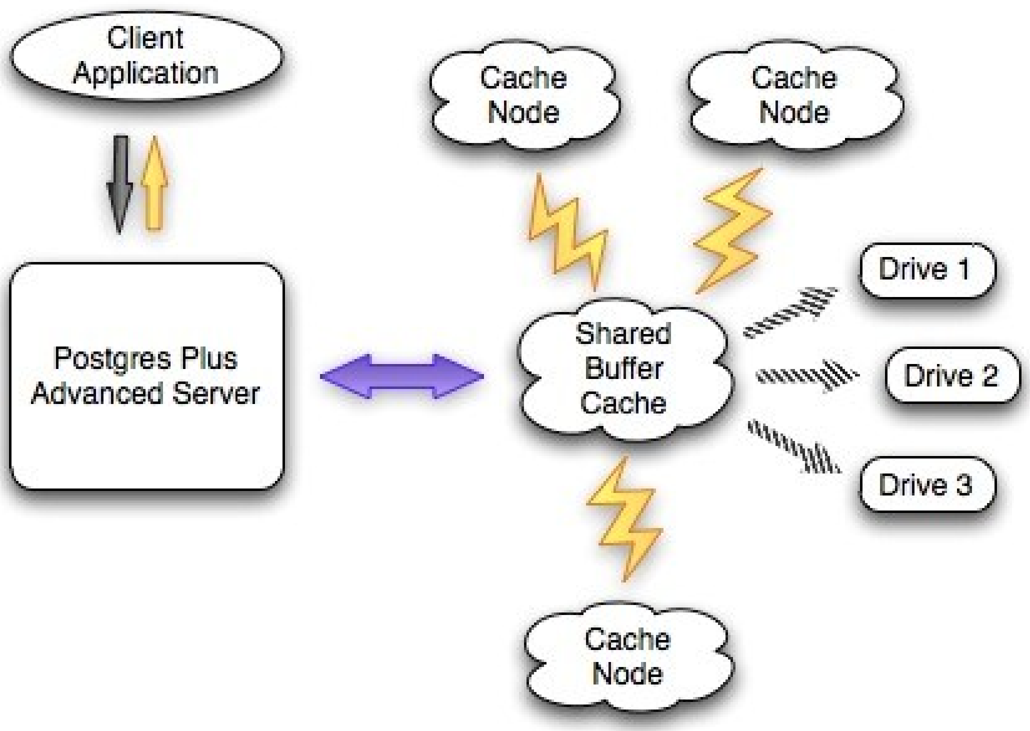


Figure 1.2 – Data flow with Infinite Cache

When a client application sends a query to the server, the server searches the shared buffer cache for the required data. If the requested data is found in the cache, the server immediately sends the data back to the client. If not, the server sends a request for the page to a specific cache server; if the cache server holds a copy of the page it sends the data back to the server and the server copies the page into the shared buffer cache. If the required page is not found in the primary cache (the shared buffer cache) or in the secondary cache (the cloud of cache servers), Advanced Server must read the page from disk. Infinite Cache improves performance by utilizing RAM from other computers on your network in order to avoid reading frequently accessed data from disk.

Updating the Cache Node Configuration

You can add or remove cache servers without restarting the database server by adding or deleting cache nodes from the list defined in the `edb_icache_servers` configuration parameter. For more information about changing the configuration parameter, see Section 8.2.2.2.

When you add one or more cache nodes, the server re-allocates the cache, dividing the cache evenly amongst the servers; each of the existing cache servers loses a percentage of the information that they have cached. You can calculate the percentage of the cache that remains valid with the following formula:

```
(existing_nodes * 100) / (existing_nodes + new_nodes)
```

For example, if an Advanced Server installation with three existing cache nodes adds an additional cache node, 75% of the existing cache remains valid after the reconfiguration.

If cache nodes are removed from a server, the data that has been stored on the remaining cache nodes is preserved. If one cache server is removed from a set of five cache servers, Advanced Server preserves the 80% of the distributed cache that is stored on the four remaining cache nodes.

When you change the cache server configuration (by adding or removing cache servers), the portion of the cache configuration that is preserved is not re-written unless the cache is completely re-warmed using the `edb_icache_warm()` function or `edb_icache_warm` utility. If you do not re-warm the cache servers, new cache servers will accrue cache data as queries are performed on the server.

Infinite Cache Offers a Second Performance Advantage: Compression.

Without Infinite Cache, Advanced Server will read each page from disk as an 8K chunk; when a page resides in the shared buffer cache, it consumes 8K of RAM. With Infinite Cache, Postgres can *compress* each page before sending it to a cache server. A compressed page can take significantly less room in the secondary cache, making more space available for other data and effectively increasing the size of the cache. A compressed page consumes less network bandwidth as well, decreasing the amount of time required to retrieve a page from the secondary cache.

The fact that Infinite Cache can compress each page may make it attractive to configure a secondary cache server on the same computer that runs your Postgres server. If, for example, your computer is configured with 6GB of RAM, you may want to allocate a smaller amount (say 1GB) for the primary cache (the shared buffer cache) and a larger amount (4GB) to the secondary cache (Infinite Cache), reserving 1GB for the operating system. Since the secondary cache resides on the same computer, there is very little overhead involved in moving data between the primary and secondary cache. All data stored in the Infinite Cache is compressed so the secondary cache can hold many more pages than would fit into the (uncompressed) shared buffer cache. If you had allocated 5GB to the shared buffer cache, the cache could hold no more than 65000 pages (approximately). By assigning 4GB of memory to Infinite Cache, the cache may be able to hold 130000 pages (at 2x compression), 195000 pages (at 3x compression) or more. The compression factor that you achieve is determined by the amount of redundancy in the data itself and the `edb_icache_compression_level` parameter.

To use Infinite Cache, you must specify a list of one or more cache servers (computers on your network) and start the `edb_icache` daemon on each of those servers.

Infinite Cache is supported on Linux, HPUX and Solaris systems only.

Please Note: Infinite Cache and the `effective_io_concurrency` parameter can potentially interfere with each other. You should disable asynchronous I/O requests (by setting the value of `effective_io_concurrency` to `0` in the `postgresql.conf` file) if you enable the Infinite Cache feature.

8.2.1 Installing Infinite Cache

Postgres Plus Advanced Server includes Infinite Cache functionality as part of a standard installation with either the graphical installer or the RPM installer. You can also optionally install only the Infinite Cache daemon on a supporting cache server.

For information about using the RPM packages to install Infinite Cache, please see the Postgres Plus Advanced Server Installation Guide, available at:

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

To use the graphical installer to install Advanced Server with Infinite Cache functionality, confirm that the box next to the `Database Server` option (located on the `Setup: Select Components` window, shown in Figure 8.3) is selected when running the installation wizard.

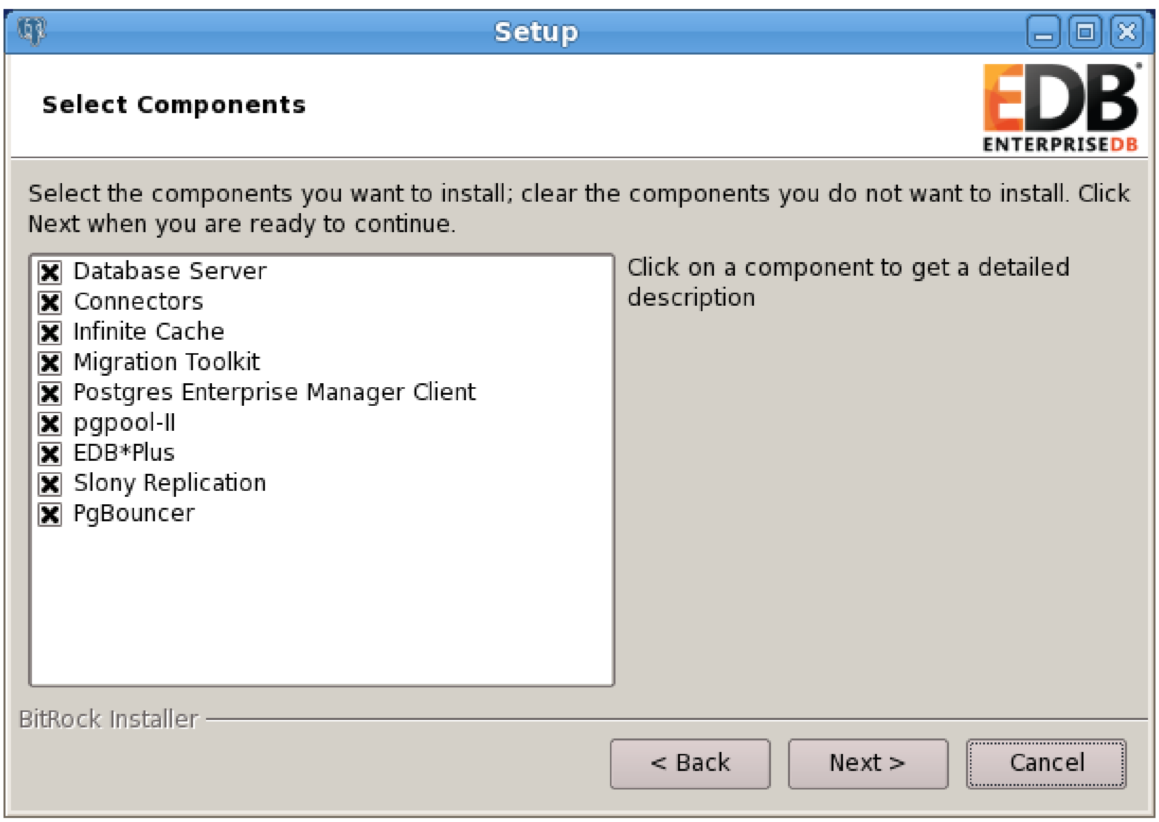


Figure 8.3: The `Setup: Select Components` window.

The `Database Server` option installs the following Infinite Cache components:

- The `ppas-infinitecache` service script.

- The Infinite Cache configuration file (`ppas-infinitecache`).
- A command line tool that allows you to pre-load the cache servers (`edb-icache-warm`).
- The `edb_icache` libraries (code libraries required by the `edb-icache` daemon).

The graphical installation wizard can selectively install only the Infinite Cache daemon on a cache server. To install the `edb-icache` daemon on a cache server, deploy the installation wizard on the machine hosting the cache; when the `Setup: Select Components` window opens, de-select all options except `Infinite Cache` (as shown in Figure 8.4).

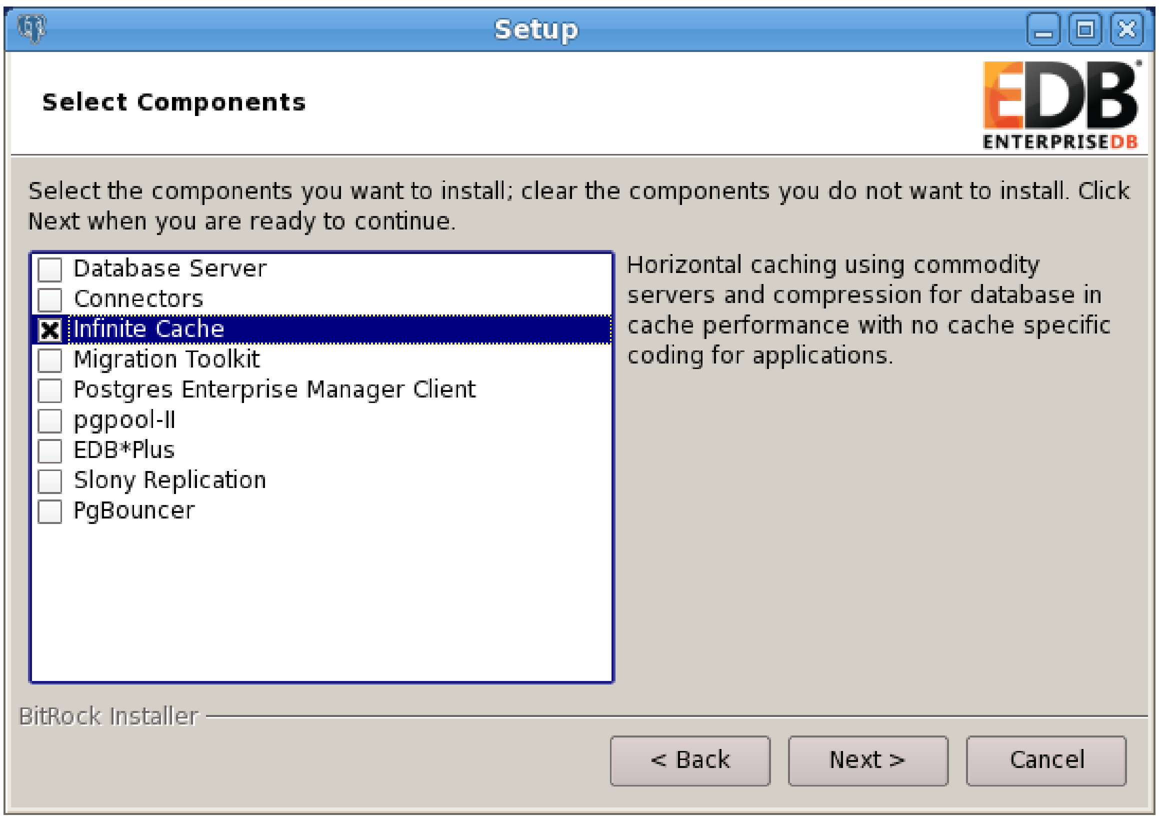


Figure 8.4: Installing only the `Infinite Cache Daemon`.

The `Infinite Cache Daemon` option installs the following:

- The `ppas-infinitecache` service script.
- The Infinite Cache configuration file (`ppas-infinitecache`).
- A command line tool that allows you to pre-load the cache servers (`edb-icache-warm`).
- The `edb_icache` libraries (code libraries required by the `edb-icache` daemon).

8.2.2 Configuring the Infinite Cache Server

Configuring Infinite Cache is a three-step process:

- Specify Infinite Cache server settings in the Infinite Cache configuration file.
- Modify the Advanced Server `postgresql.conf` file, enabling Infinite Cache, and specifying connection and compression settings.
- Start the Infinite Cache service.

8.2.2.1 Modifying Infinite Cache Settings

The Infinite Cache configuration file is named `ppas-infinitecache`, and contains two parameters and their associated values:

```
PORT=11211
CACHESIZE=500
```

To modify a parameter, open the `ppas-infinitecache` file (located in the `/opt/PostgresPlus/infinitecache/etc` directory) with your editor of choice, and modify the parameter values:

`PORT`

Use the `PORT` variable to specify the port where Infinite Cache will listen for connections from Advanced Server.

`CACHESIZE`

Use the `CACHESIZE` variable to specify the size of the cache (in MB).

8.2.2.2 Enabling Infinite Cache

The `postgresql.conf` file includes three configuration parameters that control the behavior of Infinite Cache. The `postgresql.conf` file is read each time you start the Advanced Server database server. To modify a parameter, open the `postgresql.conf` file (located in the `$PGDATA` directory) with your editor of choice, and edit the section of the configuration file shown below:

```
# - Infinite Cache
#edb_enable_icache = off
#edb_icache_servers = '' #'host1:port1,host2,ip3:port3,ip4'
#edb_icache_compression_level = 6
```

Lines that begin with a pound sign (#) are treated as comments; to enable a given parameter, remove the pound sign and specify a value for the parameter. When you've updated and saved the configuration file, restart the database server for the changes to take effect.

`edb_enable_icache`

Use the `edb_enable_icache` parameter to enable or disable Infinite Cache. When `edb_enable_icache` is set to `on`, Infinite Cache is enabled; if the parameter is set to `off`, Infinite Cache is disabled.

If you set `edb_enable_icache` to `on`, you must also specify a list of cache servers by setting the `edb_icache_servers` parameter (described in the next section).

The default value of `edb_enable_icache` is `off`.

`edb_icache_servers`

`The edb_icache_servers` parameter specifies a list of one or more servers with active edb-icache daemons. `edb_icache_servers` is a string value that takes the form of a comma-separated list of *hostname:port* pairs. You can specify each pair in any of the following forms:

- *`hostname`*
- *`IP-address`*
- *`hostname:portnumber`*
- *`IP-address:portnumber`*

If you do not specify a port number, Infinite Cache assumes that the cache server is listening at port `11211`. This configuration parameter will take effect only if `edb_enable_icache` is set to `on`. Use the `edb_icache_servers` parameter to specify a maximum of 128 cache nodes.

`edb_icache_compression_level`

`The edb_icache_compression_level` parameter controls the compression level that is applied to each page before storing it in the distributed Infinite Cache. This parameter must be an integer in the range `0` to `9`.

- A compression level of `0` disables compression; it uses no CPU time for compression, but requires more storage space and network resources to process.

- A compression level of 9 invokes the maximum amount of compression; it increases the load on the CPU, but less data flows across the network, so network demand is reduced. Each page takes less room on the Infinite Cache server, so memory requirements are reduced.

- A compression level of 5 or 6 is a reasonable compromise between the amount of compression received and the amount of CPU time invested.

By default, `edb_icache_compression_level` is set to 6.

When Advanced Server reads data from disk, it typically reads the data in 8K increments. If `edb_icache_compression_level` is set to 0, each time Advanced Server sends an 8K page to the Infinite Cache server that page is stored (uncompressed) in 8K of cache memory. If the `edb_icache_compression_level` parameter is set to 9, Advanced Server applies the maximum compression possible before sending it to the Infinite Cache server, so a page that previously took 8K of cached memory might take 2K of cached memory. Exact compression numbers are difficult to predict, as they are dependent on the nature of the data on each page.

The compression level must be set by the superuser and can be changed for the current session while the server is running. The following command disables the compression mechanism for the currently active session:

```
SET edb_icache_compression_level = 0
```

The following example shows a typical collection of Infinite Cache settings:

```
edb_enable_icache            = on
edb_icache_servers           = 'localhost,192.168.2.1:11200,192.168.2.2'
edb_icache_compression_level = 6
```

Please Note: Infinite Cache and the `effective_io_concurrency` parameter can potentially interfere with each other. You should disable asynchronous I/O requests (by setting the value of `effective_io_concurrency` to 0 in the `postgresql.conf` file) if you enable the Infinite Cache feature. By default, `effective_io_concurrency` is set to 1.

8.2.2.3 Controlling the Infinite Cache Server

Linux

On Linux, the Infinite Cache service script is named `ppas-infinitecache`. The service script resides in the `/etc/init.d` directory. You can control the Infinite Cache service, or check the status of the service with the following command:

```
/etc/init.d/ppas-infinitecache action
```

Where *`action`* specifies:

- `start` to start the service.
- `stop` to stop the service
- `restart` to stop and then start the service.
- `status` to return the status of the service.

8.2.3 Dynamically Modifying Infinite Cache Server Nodes

You can dynamically modify the Infinite Cache server nodes; to change the Infinite Cache server configuration, use the `edb_icache_servers` parameter in the `postgresql.conf` file to:

- specify additional cache information to add a server/s.
- delete server information to remove a server/s.
- specify additional server information and delete existing server information to both add and delete servers during the same reload operation.

After updating the `edb_icache_servers` parameter in the `postgresql.conf` file, you must reload the configuration parameters for the changes to take effect.

To reload the configuration parameters, navigate through the `Postgres Plus Advanced Server 9.4` menu to the `Expert Configuration` menu, and select the `Reload Configuration` option. If prompted, enter your password to reload the configuration parameters.

Alternatively, you can use the pg_ctl reload command to update the server's configuration parameters at the command line:

```
pg_ctl reload -D data_directory
```

Where *`data_directory`* specifies the complete path to the `data` directory.

Please Note: If the server detects a problem with the value specified for the `edb_icache_servers` parameter during a server `reload`, it will ignore changes to the parameter and use the last valid parameter value. If you are performing a server `restart`, and the parameter contains an invalid value, the server will return an error.

8.2.4 Controlling the edb-icache Daemons

`edb-icache` is a high-performance memory caching daemon that distributes and stores data in shared buffers. The server transparently interacts with `edb-icache` daemons to store and retrieve data.

Before starting the database server, the `edb-icache` daemons must be running on each server node. Log into each server and start the `edb-icache` server (on that host) by issuing the following command:

```
# edb-icache -u enterprisedb -d -m 1024
```

Where:

`-u`

> `-u` specifies the user name

`-m`

> `-m` specifies the amount of memory to be used by edb-icache. The default is 64MB.

`-d`

> `-d` designates that the service should run in the background

To gracefully kill an `edb-icache` daemon (close any in-use files, flush buffers, and exit), execute the command:

```
# killall -TERM edb-icache
```

If the `edb-icache` daemon refuses to die, you may need to use the following command:

```
# killall -KILL edb-icache
```

8.2.4.1 Command Line Options

To view the command line options for the `edb-icache` daemon, use the following command from the `edb_Infinite Cache` subdirectory, located in the Advanced Server installation directory:

```
# edb-icache -h
```

The command line options are:

Parameter	Description
`-p <port_number>`	The TCP port number the Infinite Cache daemon is listening on. The default is 11211.
`-U <UDP_number>`	The UDP port number the Infinite Cache daemon is listening on. The default is 0 (off).
`-s <pathname>`	The Unix socket pathname the Infinite Cache daemon is listening on. If included, the server limits access to the host on which the Infinite Cache daemon is running, and disables network support for Infinite Cache.
`-a <mask>`	The access mask for the Unix socket, in octal form. The default value is 0700.
`-l <ip_addr>`	Specifies the IP address that the daemon is listening on. If an individual address is not specified, the default value is INDRR_ANY; all IP addresses assigned to the resource are available to the daemon.
`-d`	Run as a daemon.
`-r`	Maximize core file limit.
`-u <username>`	Assume the identity of the specified user (when run as root).
`-m <numeric>`	Max memory to use for items in megabytes. Default is 64 MB.
`-M`	Return error on memory exhausted (rather than removing items).
`-c <numeric>`	Max simultaneous connections. Default is 1024.
`-k`	Lock down all paged memory. Note that there is a limit on how much memory you may lock. Trying to allocate more than that would fail, so be sure you set the limit correctly for the user you started the daemon with (not for -u <username> user; under sh this is done with 'ulimit -S -l NUM_KB').
`-v`	Verbose (print errors/warnings while in event loop).
`-vv`	Very verbose (include client commands and responses).
`-vvv`	Extremely verbose (also print internal state transitions).
`-h`	Print the help text and exit.
`-i`	Print memcached and libevent licenses.
`-P <file>`	Save PID in <file>, only used with -d option.
`-f <factor>`	Chunk size growth factor. Default value is 1.25.
`-n <bytes>`	Minimum space allocated for key+value+flags. Default is 48.
`-L`	Use large memory pages (if available). Increasing the memory page size could reduce the number of transition look-aside buffer misses and improve the performance. To get large pages from the OS, Infinite Cache will allocate the total item-cache in one large chunk.
`-D <char>`	Use <char> as the delimiter between key prefixes and IDs. This is used for per-prefix stats reporting. The default is":" (colon). If this option is specified, stats collection is enabled automatically; if not, then it may be enabled by sending the `stats detail on` command to the server.
`-t <num>`	Specifies the number of threads to use. Default is 4.
`-R`	Maximum number of requests per event; this parameter limits the number of requests process for a given connection to prevent starvation, default is 20.
`-C`	Disable use of CAS (check and set).
`-b`	Specifies the backlog queue limit, default is 1024.
`-B`	Specifies the binding protocol. Possible values are `ascii`, `binary` or `auto`; default value is `auto`.
`-I`	Override the size of each slab page. Specifies the max item size; default 1 MB, minimum size is 1 k, maximum is 128 MB).

8.2.4.2 edb-icache-tool

`edb-icache-tool` provides a command line interface that queries the `edb-icache` daemon to retrieve statistical information about a specific cache node. The syntax is:

```
edb-icache-tool <host[:port]> stats
```

`host` specifies the address of the host that you are querying.

`port` specifies the port that the daemon is listening on.

edb-icache-tool retrieves the statistics described in the following table:

Statistic	Description
`accepting_conns`	Will this server accept new connection(s)? 1 if yes, otherwise 0.
`auth_cmds`	Number of authentication commands handled by this server, success or failure.
`auth_errors`	Number of failed authentications.
`bytes`	Total number of bytes in use.
`bytes_read`	Total number of bytes received by this server (from the network).
`bytes_written`	Total number of bytes sent by this server (to the network).
`cas_badval`	Number of keys that have been compared and swapped by this server but the comparison (original) value did not match the supplied value.
`cas_hits`	Number of keys that have been compared and swapped by this server and found present.
`cas_misses`	Number of keys that have been compared and swapped by this server and not found.
`cmd_flush`	Cumulative number of flush requests sent to this server.
`cmd_get`	Cumulative number of read requests sent to this server.
`cmd_set`	Cumulative number of write requests sent to this server.
`conn_yields`	Number of times any connection yielded to another due to hitting the edb-icache -R limit.
`connection_structures`	Number of connection structures allocated by the server.
`curr_connections`	Number of open connections.
`curr_items`	Number of items currently stored by the server.
`decr_hits`	Number of decrement requests satisfied by this server.
`decr_misses`	Number of decrement requests not satisfied by this server.
`delete_hits`	Number of delete requests satisfied by this server.
`delete_misses`	Number of delete requests not satisfied by this server.
`evictions`	Number of valid items removed from cache to free memory for new items.
`get_hits`	Number of read requests satisfied by this server.
`get_misses`	Number of read requests not satisfied by this server.
`incr_hits`	Number of increment requests satisfied by this server.
`incr_misses`	Number of increment requests not satisfied by this server.
`limit_maxbytes`	Number of bytes allocated on this server for storage.
`listen_disabled_num`	Cumulative number of times this server has hit its connection limit.
`pid`	Process ID (on cache server).
`pointer_size`	Default pointer size on host OS (usually 32 or 64).
`reclaimed`	Number of times an entry was stored using memory from an expired entry.
`rusage_user`	Accumulated user time for this process (seconds.microseconds).
`rusage_system`	Accumulated system time for this process (seconds.microseconds).

threads	Number of worker threads requested.
total_time	Number of seconds since this server's base date (usually midnight, January 1, 1970, UTC).
total_connections	Total number of connections opened since the server started running.
total_items	Total number of items stored by this server (cumulative).
uptime	Amount of time that server has been active.
version	edb-icache version.

In the following example, edb-icache-tool retrieves statistical information about an Infinite Cache server located at the address, `192.168.23.85` and listening on port `11213`:

```
    # edb-icache-tool 192.168.23.85:11213 stats

Field                      Value
accepting_conns            1
auth_cmds                  0
auth_errors                0
bytes                      52901223
bytes_read                 188383848
bytes_written              60510385
cas_badval                 0
cas_hits                   0
cas_misses                 0
cmd_flush                  1
cmd_get                    53139
cmd_set                    229120
conn_yields                0
connection_structures      34
curr_connections           13
curr_items                 54953
decr_hits                  0
decr_misses                0
delete_hits                0
delete_misses              0
evictions                  0
get_hits                   52784
get_misses                 355
incr_hits                  0
incr_misses                0
limit_maxbytes             314572800
listen_disabled_num        0
pid                        7226
pointer_size               32
reclaimed                  0
rusage_system              10.676667
rusage_user                3.068191
threads                    4
time                       1320919080
total_connections          111
total_items                229120
uptime                     7649
version                    1.4.5
```


8.2.5 Warming the edb-icache Servers

When the server starts, the primary and secondary caches are empty. When Advanced Server processes a client request, the server reads the required data from disk and stores a copy in each cache. You can improve server performance by *warming* (or pre-loading) the data into the memory cache before a client asks for it.

There are two advantages to warming the cache. Advanced Server will find data in the cache the first time it is requested by a client application, instead of waiting for it to be read from disk. Also, manually warming the cache with the data that your applications are most likely to need saves time by avoiding future random disk reads. If you don't warm the cache at startup, Postgres Plus Advanced Server performance may not reach full speed until the client applications happen to load commonly used data into the cache.

There are several ways to load pages to warm the Infinite Cache server nodes. You can:

- Use the `edb_icache_warm` utility to warm the caches from the command line.
- Use the `edb_icache_warm()` function from within edb-psql.
- Use the `edb_icache_warm()` function via scripts to warm the cache.

While it is not necessary to re-warm the cache after making changes to an existing cache configuration, re-warming the cache can improve performance by bringing the new configuration of cache servers up-to-date.

8.2.5.1 The edb_icache_warm() Function

The `edb_icache_warm()` function comes in two variations; the first variation warms not only the table, but any indexes associated with the table. If you use the second variation, you must make additional calls to warm any associated indexes.

The first form of the `edb_icache_warm()` function warms the given table and any associated indexes into the cache. The signature is:

`edb_icache_warm(table_name)`

You may specify `table_name` as a table name, OID, or `regclass` value.

```
# edb-psql edb -c "select edb_icache_warm('accounts')"
```

When you call the first form of `edb_icache_warm()`, Advanced Server reads each page in the given table, compresses the page (if configured to do so), and then sends the compressed data to an Infinite Cache server. `edb_icache_warm()` also reads, compresses, and caches each page in each index defined for the given table.

The second form of the `edb_icache_warm()` function warms the pages that contain the specified range of bytes into the cache. The signature of the second form is:

```
edb_icache_warm(table-spec, startbyte, endbyte):
```

You must make subsequent calls to specify indexes separately when using this form of the `edb_icache_warm()` function.

```
# edb-psql edb -c "select edb_icache_warm('accounts', 1, 10000)"
```

The `edb_icache_warm()` function is typically called by a utility program (such as the `edb_icache_warm` utility) to spread the warming process among multiple processes that operate in parallel.

8.2.5.2 Using the edb_icache_warm Utility

You can use the `edb_icache_warm` command-line utility to load the cache servers with specified tables, allowing fast access to relevant data from the cache.

The syntax for `edb_icache_warm` is:

```
# edb_icache_warm -d database -t tablename
```

The only required parameter is *tablename*. *tablename* can be specified with or without the `-t` option. All other parameters are optional; if omitted, default values are inferred from Advanced Server environment variables.

The options for `edb_icache_warm` are:

Option	Variable	Description
-h	*Hostname*	The name of the host running Advanced Server. Include this parameter if you are running Advanced Server on a remote host. The default value is PGHOST.
-p	*Portname*	Port in use by Advanced Server. Default value is PGPORT.
-j	*process count*	Number of (parallel) processes used to warm the cache. The default value is 1.
-U	*Username*	The Advanced Server username. Unless specified, this defaults to PGUSER.
-d	*Database*	The name of database containing the tables to be warmed. Default value is PGDATABASE.
-t	*Tablename*	Name of table to be warmed. The index for the table is also warmed. Required.

8.2.6 Retrieving Statistics from Infinite Cache

8.2.6.1 Using edb_icache_stats()

You can view Infinite Cache statistics by using the `edb_icache_stats()` function at the `edb-psql` command line (or any other query tool).

The `edb_icache_stats()` function returns a result set that reflects the state of an Infinite Cache node or nodes and the related usage statistics. The result set includes:

Statistic	Description
hostname	Host name (or IP address) of server
Port	Port number at which edb-icache daemon is listening
State	Health of this server
write_failures	Number of write failures
Bytes	Total number of bytes in use
bytes_read	Total number of bytes received by this server (from the network)
bytes_written	Total number of bytes sent by this server (to the network)
cmd_get	Cumulative number of read requests sent to this server
cmd_set	Cumulative number of write requests sent to this server
connection_structures	Number of connection structures allocated by the server
curr_connections	Number of open connections
curr_items	Number of items currently stored by the server
Evictions	Number of valid items removed from cache to free memory for new items
get_hits	Number of read requests satisfied by this server
get_misses	Number of read requests not satisfied by this server
limit_maxbytes	Number of bytes allocated on this server for storage
Pid	Process ID (on cache server)
pointer_size	Default pointer size on host OS (usually 32 or 64)
rusage_user	Accumulated user time for this process (seconds.microseconds)
rusage_system	Accumulated system time for this process (seconds.microseconds)
Threads	Number of worker threads requested
total_time	Number of seconds since this server's base date (usually midnight, January 1, 1970, UTC)
total_connections	Total number of connections opened since the server started running
total_items	Total number of items stored by this server (cumulative)
Uptime	Amount of time that server has been active
Version	edb-icache version

You can use SQL queries to view Infinite Cache statistics. To view the server status of all Infinite Cache nodes:

```
SELECT hostname, port, state FROM edb_icache_stats()
```

```
 hostname      | port  | state
---------------+-------+--------
 192.168.23.85 | 11211 | UNHEALTHY
 192.168.23.85 | 11212 | ACTIVE
```



```
(2 rows)
```

Use the following command to view complete statistics (shown here using edb-psql's expanded display mode, `\x`) for a specified node:

```
SELECT * FROM edb_icache_stats() WHERE hostname = '192.168.23.85:11211'
```

```
-[RECORD 1]-----------+--------------
hostname              | 192.168.23.85
port                  | 11211
state                 | ACTIVE
write_failures        | 0
bytes                 | 225029460
bytes_read            | 225728252
bytes_written         | 192806774
cmd_get               | 23313
cmd_set               | 27088
connection_structures | 53
curr_connections      | 3
curr_items            | 27088
evictions             | 0
get_hits              | 23266
get_misses            | 47
limit_maxbytes        | 805306368
pid                   | 4240
pointer_size          | 32
rusage_user           | 0.481926
rusage_system         | 1.583759
threads               | 1
total_time            | 1242199782
total_connections     | 66
total_items           | 27088
uptime                | 714
version               | 1.2.6
```

8.2.6.2 edb_icache_server_list

The `edb_icache_server_list` view exposes information about the status and health of all Infinite Cache servers listed in the `edb_icache_servers` GUC. The `edb_icache_server_list` view is created using the `edb_icache stats()` API. The view exposes the following information for each server:

Statistic	Description
Hostname	Host name (or IP address) of server
Port	Port number at which edb-icache daemon is listening
State	Health of this server
write_failures	Number of write failures
total_memory	Number of bytes allocated to the cache on this server
memory_used	Number of bytes currently used by the cache
memory_free	Number of unused bytes remaining in the cache
hit_ratio	Percentage of cache hits

The `state` column will contain one of the following four values, reflecting the health of the given server:

Server State	Description
`Active`	The server is known to be up and running.
`Unhealthy`	An error occurred while interacting with the cache server. Postgres will attempt to re-establish the connection with the server.
`Offline`	Postgres can no longer contact the given server.
`Manual Offline`	You have taken the server offline with the edb_icache_server_enable() function.

Use the following `SELECT` statement to return the health of each node in the Infinite Cache server farm:

```
SELECT hostname, port, state FROM edb_icache_server_list
```

```
   hostname     | port   | state
----------------+--------+-------
 192.168.23.85 | 11211 | ACTIVE
 192.168.23.85 | 11212 | ACTIVE
(2 rows)
```

Use the following command to view complete details about a specific Infinite Cache node (shown here using edb-psql's `\x` expanded-view option):

```
SELECT * FROM edb_icache_server_list WHERE hostname = '192.168.23.85:11211'
```

```
-[RECORD 1]-----------+--------------
hostname              | 192.168.23.85
port                  | 11211
state                 | ACTIVE
write_failures        | 0
total_memory          | 805306368
memory_used           | 225029460
memory_free           | 580276908
hit_ratio             | 99.79
```


8.2.7 Retrieving Table Statistics

Advanced Server provides six system views that contain statistical information on a per-table basis. The views are:

- pg_statio_all_tables
- pg_statio_sys_tables
- pg_statio_user_tables
- pg_statio_all_indexes
- pg_statio_sys_indexes
- pg_statio_user_indexes

You can use standard SQL queries to view and compare the information stored in the views. The views contain information that will allow you to observe the effectiveness of the Advanced Server buffer cache and the icache servers.

8.2.7.1 pg_statio_all_tables

The pg_statio_all_tables view contains one row for each table in the database. The view contains the following information:

Column Name	Description
relid	The OID of the table.
schemaname	The name of the schema that the table resides in.
relname	The name of the table.
heap_blks_read	The number of heap blocks read.
heap_blks_hit	The number of heap blocks hit.
heap_blks_icache_hit	The number of heap blocks found on an icache server.
idx_blks_read	The number of index blocks read.
idx_blks_hit	The number of index blocks hit.
idx_blks_icache_hit	The number of index blocks found on an icache server.
toast_blks_read	The number of toast blocks read.
toast_blks_hit	The number of toast blocks hit.
toast_blks_icache_hit	The number of toast blocks found on an icache server.
tidx_blks_read	The number of index toast blocks read.
tidx_blks_hit	The number of index toast blocks hit.
tidx_blks_icache_hit	The number of index toast blocks found on an icache server.

You can execute a simple query to view performance statistics for a specific table:

```
SELECT * FROM pg_statio_all_tables WHERE relname='jobhist';

-[ RECORD 1 ]---------+---------
relid                 | 16402
schemaname            | public
relname               | jobhist
heap_blks_read        | 1
heap_blks_hit         | 51
```

```
heap_blks_icache_hit  | 0
idx_blks_read         | 2
idx_blks_hit          | 17
idx_blks_icache_hit   | 0
toast_blks_read       |
toast_blks_hit        |
toast_blks_icache_hit |
tidx_blks_read        |
tidx_blks_hit         |
tidx_blks_icache_hit  |
```

Or, you can view the statistics by activity level. The following example displays the statistics for the ten tables that have the greatest `heap_blks_icache_hit` activity:

```
SELECT * FROM pg_statio_all_tables ORDER BY heap_blks_icache_hit DESC LIMIT
10;

relid       schemaname                relname
   heap_blks_read    heap_blks_hit    heap_blks_icache_hit
   idx_blks_read     idx_blks_hit     idx_blks_icache_hit
   toast_blks_read   toast_blks_hit   toast_blks_icache_hit
   tidx_blks_read    tidx_blks_hit    tidx_blks_icache_hit
-----------------------------------------------------------------------------
16390       public                    pgbench_accounts
   264105            71150            81498
   13171             282541           18053

1259        pg_catalog                pg_class
   22                2904             18
   14                3449             11

1249        pg_catalog                pg_attribute
   49                1619             16
   17                2841             13

1255        pg_catalog                pg_proc
   38                276              11
   33                682              16
   0                 0                0
   0                 0                0

2619        pg_catalog                pg_statistic
   20                295              8
   4                 436              4
   0                 0                0
   0                 0                0

2617        pg_catalog                pg_operator
   20                293              8
   19                791              10

2602        pg_catalog                pg_amop
   10                721              6
   13                1154             13

2610        pg_catalog                pg_index
   10                633              6
   8                 719              8

1247        pg_catalog                pg_type
   17                235              5
```

```
   12              433              4

2615       pg_catalog               pg_namespace
   4               260              4
   6               330              4
   0               0                0
   0               0                0
 (10 rows)
```

8.2.7.2 pg_statio_sys_tables

The `pg_statio_sys_tables` view contains one row for each table in a system-defined schema. The statistical information included in this view is the same as for `pg_statio_all_tables`.

8.2.7.3 pg_statio_user_tables

The `pg_statio_user_tables` view contains one row for each table in a user-defined schema. The statistical information in this view is the same as for `pg_statio_all_tables`.

8.2.7.4 pg_statio_all_indexes

The `pg_statio_all_indexes` view contains one row for each index in the current database. The view contains the following information:

Column Name	Description
`relid`	The OID of the indexed table
`indexrelid`	The OID of the index.
`schemaname`	The name of the schema that the table resides in.
`relname`	The name of the table.
`indexrelname`	The name of the index
`idx_blks_read`	The number of index blocks read.
`idx_blks_hit`	The number of index blocks hit.
`idx_blks_icache_hit`	The number of index blocks found on an icache server.

You can execute a simple query to view performance statistics for the indexes on a specific table:

```
SELECT * FROM pg_statio_all_indexes WHERE relname='pg_attribute';

-[ RECORD 1 ]---------+---------
relid                 | 1249
indexrelid            | 2658
schemaname            | pg_catalog
relname               | pg_attribute
indexrelname          | pg_attribute_relid_attnam_index
idx_blks_read         | 10
idx_blks_hit          | 1200
idx_blks_icache_hit   | 0
-[ RECORD 2 ]---------+---------
```

```
relid                  | 1249
indexrelid             | 2659
schemaname             | pg_catalog
relname                | pg_attribute
indexrelname           | pg_attribute_relid_attnum_index
idx_blks_read          | 12
idx_blks_hit           | 3917
idx_blks_icache_hit    | 0
```

The result set from the query includes the statistical information for two indexes; the `pg_attribute` table has two indexes.

You can also view the statistics by activity level. The following example displays the statistics for the ten indexes that have the greatest `idx_blks_icache_hit` activity:

```
SELECT * FROM pg_statio_all_indexes ORDER BY idx_blks_icache_hit DESC LIMIT
10;

relid  indexrelid  schemaname   relname
indexrelname                 idx_blks_read  idx_blks_hit  idx_blks_icache_hit
-------------------------------------------------------------------------------
16390  16401       public       pgbench_accounts
pgbench_accounts_pkey        13171          282541        18053

1249   2659        pg_catalog   pg_attribute
pg_attr_relid_attnum_index   14             2749          13

1255   2690        pg_catalog   proc
pg_proc_oid_index            16             580           12

1259   2663        pg_catalog   pg_class
pg_class_relname_nsp_index   10             2019          7

2602   2654        pg_catalog   pg_amop
pg_amop_opr_fam_index        7              453           7

2603   2655        pg_catalog   pg_amproc
pg_amproc_fam_proc_index     6              605           6

2617   2688        pg_catalog   pg_operator
pg_operator_oid_index        7              452           6

2602   2653        pg_catalog   pg_amop
pg_amop_fam_strat_index      6              701           6

2615   2684        pg_catalog   pg_namespace
pg_namespace_nspname_index   4              328           4

1262   2672        pg_catalog   pg_database
pg_database_oid_index        4              254           4
```

8.2.7.5 pg_statio_sys_indexes

The pg_statio_sys_indexes view contains one row for each index on the system tables. The statistical information in this view is the same as in pg_statio_all_indexes.

8.2.7.6 pg_statio_user_indexes

The pg_statio_user_indexes view contains one row for each index on a table that resides in a user-defined schema. The statistical information in this view is the same as in pg_statio_all_indexes.

8.2.8 edb_icache_server_enable()

You can use the `edb_icache_server_enable()` function to take the Infinite Cache server offline for maintenance or other planned downtime. The syntax is:

```
void edb_icache_server_enable(host TEXT, port INTEGER, online BOOL)
```

host specifies the host that you want to disable. The host name may be specified by name or numeric address.

port specifies the port number that the Infinite Cache server is listening on.

online specifies the state of the Infinite Cache server. The value of online must be true or false.

To take a server offline, specify the host that you want to disable, the port number that the Infinite Cache server is listening on, and `false`. To bring the Infinite Cache server back online, specify the host name and port number, and pass a value of `true`.

The state of a server taken offline with the `edb_icache_server_enable()` function is `MANUAL OFFLINE`. Postgres Plus Advanced Server will not automatically reconnect to an Infinite Cache server that you have taken offline with `edb_icache_server_enable(..., false)`; you must bring the server back online by calling `edb_icache_server_enable(..., true)`.

8.2.9 Infinite Cache Log Entries

When you start Advanced Server, a message that includes Infinite Cache status, cache node count and cache node size is written to the server log. The following example shows the server log for an active Infinite Cache installation with two 750 MB cache servers:

```
** EnterpriseDB Dynamic Tuning Agent*************************************
*       System Utilization: 66 %                                        *
*       Autovacuum Naptime: 60   Seconds                                *
*       Infinite Cache: on                                               *
*       Infinite Cache Servers: 2                                        *
*       Infinite Cache Size: 1.500  GB                                   *
*************************************************************************
```

8.2.10 Allocating Memory to the Cache Servers

As mentioned earlier in this document, each computer imposes a limit on the amount of *physical* memory that you can install. However, modern operating systems typically simulate a larger *address* space so that programs can transparently access more memory than is actually installed. This "virtual memory" allows a computer to run multiple programs that may simultaneously require more memory than is physically available. For example, you may run an e-mail client, a web browser, and a database server which each require 1GB of memory on a machine that contains only 2GB of physical RAM. When the operating system runs out of physical memory, it starts swapping bits and pieces of the currently running programs to disk to make room to satisfy your current demand for memory.

This can bring your system to a grinding halt.

Since the primary goal of Infinite Cache is to improve performance by limiting disk I/O, you should avoid dedicating so much memory to Infinite Cache that the operating system must start swapping data to disk. If the operating system begins to swap to disk, you lose the benefits offered by Infinite Cache.

The overall demand for physical memory can vary throughout the day; if the server is frequently idle, you may never encounter swapping. If you have dedicated a large portion of physical memory to the cache, and system usage increases, the operating system may start swapping. To get the best performance and avoid disk swapping, dedicate a server node to Infinite Cache so other applications on that computer will not compete for physical memory.

8.3 Index Advisor

The Index Advisor utility helps determine which columns you should index to improve performance in a given workload. Index Advisor considers B-tree (single-column or composite) index types, and does not identify other index types (GIN, GiST, Hash) that may improve performance. Index Advisor is installed with Postgres Plus Advanced Server.

Index Advisor works with Advanced Server's query planner by creating *hypothetical indexes* that the query planner uses to calculate execution costs as if such indexes were available. Index Advisor identifies the indexes by analyzing SQL queries supplied in the workload.

There are three ways to use Index Advisor to analyze SQL queries:

- Invoke the Index Advisor utility program, supplying a text file containing the SQL queries that you wish to analyze; Index Advisor will generate a text file with `CREATE INDEX` statements for the recommended indexes.
- Provide queries at the EDB-PSQL command line that you want Index Advisor to analyze.
- Access Index Advisor through the Postgres Enterprise Manager client. When accessed via the PEM client, Index Advisor works with SQL Profiler, providing indexing recommendations on code captured in SQL traces. For more information about using SQL Profiler and Index Advisor with PEM, please see Section 8.4 of the *PEM Getting Started Guide*, available from the EnterpriseDB website at:

  http://www.enterprisedb.com/products-services-training/products/postgres-enterprise-manager

Index Advisor will attempt to make indexing recommendations on `INSERT`, `UPDATE`, `DELETE` and `SELECT` statements. When invoking Index Advisor, you supply the workload in the form of a set of queries (if you are providing the command in an SQL file) or an `EXPLAIN` statement (if you are specifying the SQL statement at the psql command line). Index Advisor displays the query plan and estimated execution cost for the supplied query, but does not actually execute the query.

During the analysis, Index Advisor compares the query execution costs with and without hypothetical indexes. If the execution cost using a hypothetical index is less than the execution cost without it, both plans are reported in the `EXPLAIN` statement output, metrics that quantify the improvement are calculated, and Index Advisor generates the `CREATE INDEX` statement needed to create the index.

If no hypothetical index can be found that reduces the execution cost, Index Advisor displays only the original query plan output of the `EXPLAIN` statement.

Index Advisor does not actually create indexes on the tables. Use the CREATE INDEX statements supplied by Index Advisor to add any recommended indexes to your tables.

A script supplied with Advanced Server creates the table in which Index Advisor stores the indexing recommendations generated by the analysis; the script also creates a function and a view of the table to simplify the retrieval and interpretation of the results.

If you choose to forego running the script, Index Advisor will log recommendations in a temporary table that is available only for the duration of the Index Advisor session.

8.3.1 Index Advisor Components

The Index Advisor shared library interacts with the query planner to make indexing recommendations. The Postgres Plus Advanced Server installer creates the following shared library in the `libdir` subdirectory of your Postgres Plus Advanced Server home directory:

On Linux:

```
index_advisor.so
```

On Windows:

```
index_advisor.dll
```

Please note that libraries in the `libdir` directory can only be loaded by a superuser. A database administrator can allow a non-superuser to use Index Advisor by manually copying the Index Advisor file from the `libdir` directory into the `libdir/plugins` directory (under your Advanced Server home directory). Only a trusted non-superuser should be allowed access to the plugin; this is an unsafe practice in a production environment.

The installer also creates the Index Advisor utility program and setup script:

`pg_advise_index`

> `pg_advise_index` is a utility program that reads a user-supplied input file containing SQL queries and produces a text file containing `CREATE INDEX` statements that can be used to create the indexes recommended by the Index Advisor. The `pg_advise_index` program is located in the `bin` subdirectory of the Postgres Plus Advanced Server home directory.

`index_advisor.sql`

> `index_advisor.sql` is a script that creates a permanent Index Advisor log table along with a function and view to facilitate reporting of recommendations

from the log table. The script is located in the `share/contrib` subdirectory of the Postgres Plus Advanced Server directory.

The `index_advisor.sql` script creates the `index_advisor_log` table, the `show_index_recommendations()` function and the `index_recommendations` view. These database objects must be created in a schema that is accessible by, and included in the search path of the role that will invoke Index Advisor.

`index_advisor_log`

Index Advisor logs indexing recommendations in the `index_advisor_log` table. If Index Advisor does not find the `index_advisor_log` table in the user's search path, Index Advisor will store any indexing recommendations in a temporary table of the same name. The temporary table exists only for the duration of the current session.

`show_index_recommendations()`

`show_index_recommendations()` is a PL/pgSQL function that interprets and displays the recommendations made during a specific Index Advisor session (as identified by its backend process ID).

`index_recommendations`

Index Advisor creates the `index_recommendations` view based on information stored in the `index_advisor_log` table during a query analysis. The view produces output in the same format as the `show_index_recommendations()` function, but contains Index Advisor recommendations for all stored sessions, while the result set returned by the `show_index_recommendations()` function are limited to a specified session.

8.3.2 Index Advisor Configuration

Index Advisor does not require any configuration to generate recommendations that are available only for the duration of the current session; to store the results of multiple sessions, you must create the `index_advisor_log` table (where Advanced Server will store Index Advisor recommendations). To create the `index_advisor_log` table , you must run the `index_advisor.sql` script.

When selecting a storage schema for the Index Advisor table, function and view, keep in mind that all users that invoke Index Advisor (and query the result set) must have `USAGE` privileges on the schema. The schema must be in the search path of all users that are interacting with the Index Advisor.

1. Place the selected schema at the start of your `search_path` parameter. For example, if your search path is currently:

   `search_path=public, accounting`
   and you want the Index Advisor objects to be created in a schema named `advisor`, use the command:
   `SET search_path = advisor, public, accounting;`

2. Run the `index_advisor.sql` script to create the database objects. If you are running the psql client, you can use the command:

   `\i` *`full_pathname`*`/index_advisor.sql`
   Specify the pathname to the `index_advisor.sql` script in place of *`full_pathname`*.

3. Grant privileges on the `index_advisor_log` table to all Index Advisor users; this step is not necessary if the Index Advisor user is a superuser, or the owner of these database objects.

   - Grant `SELECT` and `INSERT` privileges on the `index_advisor_log` table to allow a user to invoke Index Advisor.

   - Grant `DELETE` privileges on the `index_advisor_log` table to allow the specified user to delete the table contents.

   - Grant `SELECT` privilege on the `index_recommendations` view.

The following example demonstrates the creation of the Index Advisor database objects in a schema named `ia`, which will then be accessible to an Index Advisor user with user name *`ia_user`*:

```
$ edb-psql -d edb -U enterprisedb
edb-psql (9.4.0.0)
Type "help" for help.

edb=# CREATE SCHEMA ia;
CREATE SCHEMA
edb=# SET search_path TO ia;
SET
edb=# \i /opt/PostgresPlus/9.4AS/share/contrib/index_advisor.sql
CREATE TABLE
CREATE INDEX
CREATE INDEX
CREATE FUNCTION
CREATE FUNCTION
CREATE VIEW
edb=# GRANT USAGE ON SCHEMA ia TO ia_user;
GRANT
edb=# GRANT SELECT, INSERT, DELETE ON index_advisor_log TO ia_user;
GRANT
```



```
edb=# GRANT SELECT ON index_recommendations TO ia_user;
GRANT
```

While using Index Advisor, the specified schema (`ia`) must be included in *`ia_user`*'s `search_path` parameter.

8.3.3 Using Index Advisor

When you invoke Index Advisor, you must supply a workload; the workload is either a query (specified at the command line), or a file that contains a set of queries (executed by the `pg_advise_index()` function). After analyzing the workload, Index Advisor will either store the result set in a temporary table, or in a permanent table. You can review the indexing recommendations generated by Index Advisor and use the `CREATE INDEX` statements generated by Index Advisor to create the recommended indexes.

Note: You should not run Index Advisor in read-only transactions.

The following examples assume that superuser `enterprisedb` is the Index Advisor user, and the Index Advisor database objects have been created in a schema in the `search_path` of superuser `enterprisedb`.

The examples in the following sections use the table created with the statement shown below:

```
CREATE TABLE t( a INT, b INT );
INSERT INTO t SELECT s, 99999 - s FROM generate_series(0,99999) AS s;
ANALYZE t;
```

The resulting table contains the following rows:

```
   a   |   b
-------+-------
     0 | 99999
     1 | 99998
     2 | 99997
     3 | 99996
       .
       .
       .
 99997 |     2
 99998 |     1
 99999 |     0
```

8.3.3.1 Using the pg_advise_index Utility

When invoking the `pg_advise_index` utility, you must include the name of a file that contains the queries that will be executed by `pg_advise_index`; the queries may be on the same line, or on separate lines, but each query must be terminated by a semicolon. Queries within the file should not begin with the `EXPLAIN` keyword.

The following example shows the contents of a sample `workload.sql` file:

```
SELECT * FROM t WHERE a = 500;
```

```
SELECT * FROM t WHERE b < 1000;
```

Run the `pg_advise_index` program as shown in the code sample below:

```
$ pg_advise_index -d edb -h localhost -U enterprisedb -s 100M -o advisory.sql
workload.sql
poolsize = 102400 KB
load workload from file 'workload.sql'
Analyzing queries .. done.
size = 2184 KB, benefit = 1684.720000
size = 2184 KB, benefit = 1655.520000
/* 1. t(a): size=2184 KB, benefit=1684.72 */
/* 2. t(b): size=2184 KB, benefit=1655.52 */
/* Total size = 4368KB */
```

In the code sample, the `-d`, `-h`, and `-U` options are psql connection options.

`-s`

> `-s` is an optional parameter that limits the maximum size of the indexes recommended by Index Advisor. If Index Advisor does not return a result set, `-s` may be set too low.

`-o`

> The recommended indexes are written to the file specified after the `-o` option.

The information displayed by the `pg_advise_index` program is logged in the `index_advisor_log` table. In response to the command shown in the example, Index Advisor writes the following `CREATE INDEX` statements to the `advisory.sql` output file

```
create index idx_t_1 on t (a);
create index idx_t_2 on t (b);
```

You can create the recommended indexes at the psql command line with the `CREATE INDEX` statements in the file, or create the indexes by executing the `advisory.sql` script.

```
$ edb-psql -d edb -h localhost -U enterprisedb -e -f advisory.sql
create index idx_t_1 on t (a);
CREATE INDEX
create index idx_t_2 on t (b);
CREATE INDEX
```

8.3.3.2 Using Index Advisor at the psql Command Line

You can use Index Advisor to analyze SQL statements entered at the edb-psql (or psql) command line; the following steps detail loading the Index Advisor plugin and using Index Advisor:

1. Connect to the server with the `edb-psql` command line utility, and load the Index Advisor plugin:

```
$ edb-psql -d edb -U enterprisedb
…
edb=# LOAD 'index_advisor';
LOAD
```

2. Use the `edb-psql` command line to invoke each SQL command that you would like Index Advisor to analyze. Index Advisor stores any recommendations for the queries in the `index_advisor_log` table. If the `index_advisor_log` table does not exist in the user's `search_path`, a temporary table is created with the same name. This temporary table exists only for the duration of the user's session.

After loading the Index Advisor plugin, Index Advisor will analyze all SQL statements and log any indexing recommendations for the duration of the session.

If you would like Index Advisor to analyze a query (and make indexing recommendations) without actually executing the query, preface the SQL statement with the `EXPLAIN` keyword.

If you do not preface the statement with the `EXPLAIN` keyword, Index Advisor will analyze the statement while the statement executes, writing the indexing recommendations to the `index_advisor_log` table for later review.

In the example that follows, the `EXPLAIN` statement displays the normal query plan, followed by the query plan of the same query, if the query were using the recommended hypothetical index:

```
edb=# EXPLAIN SELECT * FROM t WHERE a < 10000;
                                QUERY PLAN
-----------------------------------------------------------------------------
Seq Scan on t  (cost=0.00..1693.00 rows=10105 width=8)
  Filter: (a < 10000)
Result  (cost=0.00..337.10 rows=10105 width=8)
  One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text
  ->  Index Scan using "<hypothetical-index>:1" on t
      (cost=0.00..337.10 rows=10105 width=8)
        Index Cond: (a < 10000)
(6 rows)


edb=# EXPLAIN SELECT * FROM t WHERE a = 100;
                                QUERY PLAN
```



```
----------------------------------------------------------------------------
Seq Scan on t  (cost=0.00..1693.00 rows=1 width=8)
  Filter: (a = 100)
Result  (cost=0.00..8.28 rows=1 width=8)
  One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text
  ->  Index Scan using "<hypothetical-index>:3" on t
      (cost=0.00..8.28 rows=1 width=8)
        Index Cond: (a = 100)
(6 rows)
```

For information about reviewing the recommended queries, see Section 8.3.4.

After loading the Index Advisor plugin, the default value of `index_advisor.enabled` is `on`. The Index Advisor plugin must be loaded to use a `SET` or `SHOW` command to display the current value of `index_advisor.enabled`.

You can use the `index_advisor.enabled` parameter to temporarily disable Index Advisor without interrupting the psql session:

```
edb=# SET index_advisor.enabled TO off;
SET
```

To enable Index Advisor, set the parameter to `on`:

```
edb=# SET index_advisor.enabled TO on;
SET
```

8.3.4 Reviewing the Index Advisor Recommendations

There are several ways to review the index recommendations generated by Index Advisor. You can:

- Query the `index_advisor_log` table.
- Run the `show_index_recommendations` function.
- Query the `index_recommendations` view.

8.3.4.1 Using the show_index_recommendations() Function

To review the recommendations of the Index Advisor utility using the `show_index_recommendations()` function, call the function, specifying the process ID of the session:

```
SELECT show_index_recommendations( pid );
```

Where *`pid`* is the process ID of the current session. If you do not know the process ID of your current session, passing a value of `NULL` will also return a result set for the current session.

The following code fragment shows an example of a row in a result set:

```
edb=# SELECT show_index_recommendations(null);
                      show_index_recommendations
---------------------------------------------------------------------
 create index idx_t_a on t(a);/* size: 2184 KB, benefit: 3040.62,
 gain: 1.39222666981456 */
(1 row)
```

In the example, `create index idx_t_a on t(a)` is the SQL statement needed to create the index suggested by Index Advisor. Each row in the result set shows:

- The command required to create the recommended index.
- The maximum estimated size of the index.
- The calculated benefit of using the index.
- The estimated gain that will result from implementing the index.

You can display the results of all Index Advisor sessions from the following view:

```
SELECT * FROM index_recommendations;
```

8.3.4.2 Querying the index_advisor_log Table

Index Advisor stores indexing recommendations in a table named `index_advisor_log`. Each row in the `index_advisor_log` table contains the result of a query where Index Advisor determines it can recommend a hypothetical index to reduce the execution cost of that query.

Column	Type	Description
`reloid`	`oid`	OID of the base table for the index
`relname`	`name`	Name of the base table for the index
`attrs`	`integer[]`	Recommended index columns (identified by column number)
`benefit`	`real`	Calculated benefit of the index for this query
`index_size`	`integer`	Estimated index size in disk-pages
`backend_pid`	`integer`	Process ID of the process generating this recommendation
`timestamp`	`timestamp`	Date/Time when the recommendation was generated

You can query the `index_advisor_log` table at the psql command line. The following example shows the `index_advisor_log` table entries resulting from two Index Advisor sessions. Each session contains two queries, and can be identified (in the table below) by a different `backend_pid` value. For each session, Index Advisor generated two index recommendations.

```
  edb=# SELECT * FROM index_advisor_log;
   reloid | relname | attrs | benefit | index_size | backend_pid |
timestamp
  --------+---------+-------+---------+------------+-------------+-----------
-----------------------
    16651 | t       | {1}   | 1684.72 |       2184 |        3442 | 22-MAR-11
16:44:32.712638 -04:00
    16651 | t       | {2}   | 1655.52 |       2184 |        3442 | 22-MAR-11
16:44:32.759436 -04:00
    16651 | t       | {1}   | 1355.9  |       2184 |        3506 | 22-MAR-11
16:48:28.317016 -04:00
    16651 | t       | {1}   | 1684.72 |       2184 |        3506 | 22-MAR-11
16:51:45.927906 -04:00
  (4 rows)
```

Index Advisor added the first two rows to the table after analyzing the following two queries executed by the `pg_advise_index` utility:

```
  SELECT * FROM t WHERE a = 500;
  SELECT * FROM t WHERE b < 1000;
```

The value of `3442` in column `backend_pid` identifies these results as coming from the session with process ID `3442`.

The value of `1` in column `attrs` in the first row indicates that the hypothetical index is on the first column of the table (column `a` of table `t`).

The value of 2 in column attrs in the second row indicates that the hypothetical index is on the second column of the table (column b of table t).

Index Advisor added the last two rows to the table after analyzing the following two queries (executed at the psql command line):

```
edb=# EXPLAIN SELECT * FROM t WHERE a < 10000;
                                          QUERY PLAN
--------------------------------------------------------------------------
--------------------
 Seq Scan on t  (cost=0.00..1693.00 rows=10105 width=8)
   Filter: (a < 10000)
 Result  (cost=0.00..337.10 rows=10105 width=8)
   One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text
   ->  Index Scan using "<hypothetical-index>:1" on t  (cost=0.00..337.10
rows=10105 width=8)
         Index Cond: (a < 10000)
(6 rows)

edb=# EXPLAIN SELECT * FROM t WHERE a = 100;
                                       QUERY PLAN
--------------------------------------------------------------------------
--------------
 Seq Scan on t  (cost=0.00..1693.00 rows=1 width=8)
   Filter: (a = 100)
 Result  (cost=0.00..8.28 rows=1 width=8)
   One-Time Filter: '===[ HYPOTHETICAL PLAN ]==='::text
   ->  Index Scan using "<hypothetical-index>:3" on t  (cost=0.00..8.28
rows=1 width=8)
         Index Cond: (a = 100)
(6 rows)
```

The values in the benefit column of the index_advisor_log table are calculated using the following formula:

benefit = (normal execution cost) - (execution cost with hypothetical index)

The value of the benefit column for the last row of the index_advisor_log table (shown in the example) is calculated using the query plan for the following SQL statement:

```
EXPLAIN SELECT * FROM t WHERE a = 100;
```

The execution costs of the different execution plans are evaluated and compared:

```
benefit = (Seq Scan on t cost) - (Index Scan using
<hypothetical-index>)
```

and the benefit is added to the table:

```
benefit = 1693.00 - 8.28
benefit = 1684.72
```


You can delete rows from the `index_advisor_log` table when you no longer have the need to review the results of the queries stored in the row.

8.3.4.3 Querying the index_recommendations View

The `index_recommendations` view contains the calculated metrics and the `CREATE INDEX` statements to create the recommended indexes for all sessions whose results are currently in the `index_advisor_log` table. You can display the results of all stored Index Advisor sessions by querying the `index_recommendations` view as shown below:

```
SELECT * FROM index_recommendations;
```

Using the example shown in the previous section (*Querying the index_advisor_log Table*), the `index_recommendations` view displays the following:

```
  edb=# SELECT * FROM index_recommendations;
   backend_pid |                                 show_index_recommendations
  -------------+-------------------------------------------------------------
--------------------------------
          3442 | create index idx_t_a on t(a);/* size: 2184 KB, benefit:
1684.72, gain: 0.771392654586624 */
          3442 | create index idx_t_b on t(b);/* size: 2184 KB, benefit:
1655.52, gain: 0.758021539820856 */
          3506 | create index idx_t_a on t(a);/* size: 2184 KB, benefit:
3040.62, gain: 1.39222666981456 */
  (3 rows)
```

Within each session, the results of all queries that benefit from the same recommended index are combined to produce one set of metrics per recommended index, reflected in the fields named `benefit` and `gain`.

The formulas for the fields are as follows:

```
size    = MAX(index size of all queries)
benefit = SUM(benefit of each query)
gain    = SUM(benefit of each query) / MAX(index size of all
queries)
```

So for example, using the following query results from the process with a `backend_pid` of `3506`:

```
   reloid | relname | attrs | benefit | index_size | backend_pid |
timestamp
  --------+---------+-------+---------+------------+-------------+-----------
-----------------------
    16651 | t       | {1}   |  1355.9 |       2184 |        3506 | 22-MAR-11
16:48:28.317016 -04:00
    16651 | t       | {1}   | 1684.72 |       2184 |        3506 | 22-MAR-11
16:51:45.927906 -04:00
```

The metrics displayed from the `index_recommendations` view for `backend_pid 3506` are:

```
  backend_pid |                                   show_index_recommendations
 -------------+----------------------------------------------------------------
--------------------------------
         3506 | create index idx_t_a on t(a);/* size: 2184 KB, benefit:
3040.62, gain: 1.39222666981456 */
```

The metrics from the view are calculated as follows:

```
benefit = (benefit from 1st query) + (benefit from 2nd query)
benefit = 1355.9 + 1684.72
benefit = 3040.62
```

and

```
gain = ((benefit from 1st query) + (benefit from 2nd query))
/ MAX(index size of all queries)
gain = (1355.9 + 1684.72) / MAX(2184, 2184)
gain = 3040.62 / 2184
gain = 1.39223
```

The gain metric is useful when comparing the relative advantage of the different recommended indexes derived during a given session. The larger the gain value, the better the cost effectiveness derived from the index weighed against the possible disk space consumption of the index.

8.3.5 Limitations

Index Advisor does not consider Index Only scans; it does consider Index scans when making recommendations.

Index Advisor ignores any computations found in the `WHERE` clause. Effectively, the index field in the recommendations will not be any kind of expression; the field will be a simple column name.

Index Advisor does not consider inheritance when recommending hypothetical indexes. If a query references a parent table, Index Advisor does not make any index recommendations on child tables.

Restoration of a `pg_dump` backup file that includes the `index_advisor_log` table or any tables for which indexing recommendations were made and stored in the `index_advisor_log` table, may result in "broken links" between the `index_advisor_log` table and the restored tables referenced by rows in the `index_advisor_log` table because of changes in object identifiers (OIDs).

If it is necessary to display the recommendations made prior to the backup, you can replace the old OIDs in the `reloid` column of the `index_advisor_log` table with the new OIDs of the referenced tables using the SQL `UPDATE` statement:

```
UPDATE index_advisor_log SET reloid = new_oid WHERE reloid =
old_oid;
```


8.4 SQL Profiler

Inefficient SQL code is one of, if not the leading cause of database performance problems. The challenge for database administrators and developers is locating and then optimizing this code in large, complex systems.

SQL Profiler helps you locate and optimize poorly running SQL code.

Specific features and benefits of SQL Profiler include the following:

- **On-Demand Traces.** You can capture SQL traces at any time by manually setting up your parameters and starting the trace.
- **Scheduled Traces.** For inconvenient times, you can also specify your trace parameters and schedule them to run at some later time.
- **Save Traces.** Execute your traces and save them for later review.
- **Trace Filters.** Selectively filter SQL captures by database and by user, or capture every SQL statement sent by all users against all databases.
- **Trace Output Analyzer.** A graphical table lets you quickly sort and filter queries by duration or statement, and a graphical or text based `EXPLAIN` plan lays out your query paths and joins.
- **Index Advisor Integration.** Once you have found your slow queries and optimized them, you can also let the Index Advisor recommend the creation of underlying table indices to further improve performance.

For more information about SQL Profiler and Postgres Enterprise Manager, visit the EnterpriseDB website at:

http://www.enterprisedb.com/postgres-enterprise-manager

8.5 Query Optimization Hints

When you invoke a `DELETE`, `INSERT`, `SELECT` or `UPDATE` command, the server generates a set of execution plans; after analyzing those execution plans, the server selects a plan that will (generally) return the result set in the least amount of time. The server's choice of plan is dependent upon several factors:

- The estimated execution cost of data handling operations.
- Parameter values assigned to parameters in the `Query Tuning` section of the `postgresql.conf` file.
- Column statistics that have been gathered by the `ANALYZE` command.

As a rule, the query planner will select the least expensive plan. You can use an *optimizer hint* to influence the server as it selects a query plan. An optimizer hint is a directive (or multiple directives) embedded in a comment-like syntax that immediately follows a `DELETE`, `INSERT`, `SELECT` or `UPDATE` command. Keywords in the comment instruct the server to employ or avoid a specific plan when producing the result set.

Synopsis

```
{ DELETE | INSERT | SELECT | UPDATE } /*+ { hint [ comment ] }
[...] */
  statement_body

{ DELETE | INSERT | SELECT | UPDATE } --+ { hint [ comment ] }
[...]
  statement_body
```

Optimizer hints may be included in either of the forms shown above. Note that in both forms, a plus sign (+) must immediately follow the `/*` or `--` opening comment symbols, with no intervening space, or the server will not interpret the following tokens as hints.

If you are using the first form, the hint and optional comment may span multiple lines. The second form requires all hints and comments to occupy a single line; the remainder of the statement must start on a new line.

Description

Please Note:

- The database server will always try to use the specified hints if at all possible.
- If a planner method parameter is set so as to disable a certain plan type, then this plan will not be used even if it is specified in a hint, unless there are no other possible options for the planner. Examples of planner method parameters are

enable_indexscan, enable_seqscan, enable_hashjoin, enable_mergejoin, and enable_nestloop. These are all Boolean parameters.

- Remember that the hint is embedded within a comment. As a consequence, if the hint is misspelled or if any parameter to a hint such as view, table, or column name is misspelled, or non-existent in the SQL command, there will be no indication that any sort of error has occurred. No syntax error will be given and the entire hint is simply ignored.
- If an alias is used for a table or view name in the SQL command, then the alias name, not the original object name, must be used in the hint. For example, in the command, `SELECT /*+ FULL(acct) */ * FROM accounts acct ...`, `acct`, the alias for `accounts`, must be specified in the `FULL` hint, not the table name, `accounts`.
- Use the `EXPLAIN` command to ensure that the hint is correctly formed and the planner is using the hint. See the *Postgres Plus* documentation set for information on the `EXPLAIN` command.
- In general, optimizer hints should not be used in production applications. Typically, the table data changes throughout the life of the application. By ensuring that the more dynamic columns are `ANALYZE`d frequently, the column statistics will be updated to reflect value changes and the planner will use such information to produce the least cost plan for any given command execution. Use of optimizer hints defeats the purpose of this process and will result in the same plan regardless of how the table data changes.

Parameters

hint

An optimizer hint directive.

comment

A string with additional information. Note that there are restrictions as to what characters may be included in the comment. Generally, *comment* may only consist of alphabetic, numeric, the underscore, dollar sign, number sign and space characters. These must also conform to the syntax of an identifier. Any subsequent hint will be ignored if the comment is not in this form.

statement_body

The remainder of the `DELETE`, `INSERT`, `SELECT`, or `UPDATE` command.

The following sections describe the optimizer hint directives in more detail.

8.5.1 Default Optimization Modes

There are a number of optimization modes that can be chosen as the default setting for a Postgres Plus Advanced Server database cluster. This setting can also be changed on a per session basis by using the ALTER SESSION command as well as in individual DELETE, SELECT, and UPDATE commands within an optimizer hint. The configuration parameter that controls these default modes is named OPTIMIZER_MODE. The following table shows the possible values.

Table 3-8-1 Default Optimization Modes

Hint	Description
ALL_ROWS	Optimizes for retrieval of all rows of the result set.
CHOOSE	Does no default optimization based on assumed number of rows to be retrieved from the result set. This is the default.
FIRST_ROWS	Optimizes for retrieval of only the first row of the result set.
FIRST_ROWS_10	Optimizes for retrieval of the first 10 rows of the results set.
FIRST_ROWS_100	Optimizes for retrieval of the first 100 rows of the result set.
FIRST_ROWS_1000	Optimizes for retrieval of the first 1000 rows of the result set.
FIRST_ROWS(*n*)	Optimizes for retrieval of the first *n* rows of the result set. This form may not be used as the object of the ALTER SESSION SET OPTIMIZER_MODE command. It may only be used in the form of a hint in a SQL command.

These optimization modes are based upon the assumption that the client submitting the SQL command is interested in viewing only the first "n" rows of the result set and will then abandon the remainder of the result set. Resources allocated to the query are adjusted as such.

Examples

Alter the current session to optimize for retrieval of the first 10 rows of the result set.

```
ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS_10;
```

The current value of the OPTIMIZER_MODE parameter can be shown by using the SHOW command. Note that this command is a utility dependent command. In PSQL, the SHOW command is used as follows:

```
SHOW OPTIMIZER_MODE;

optimizer_mode
-----------------
 first_rows_10
(1 row)
```

The SHOW command has the following syntax:

```
SHOW PARAMETER OPTIMIZER_MODE;

NAME
--------------------------------------------------
VALUE
--------------------------------------------------
optimizer_mode
first_rows_10
```

The following example shows an optimization mode used in a `SELECT` command as a hint:

```
SELECT /*+ FIRST_ROWS(7) */ * FROM emp;

 empno | ename  |    job    | mgr  |      hiredate      |   sal   |  comm   | deptno
-------+--------+-----------+------+--------------------+---------+---------+--------
  7369 | SMITH  | CLERK     | 7902 | 17-DEC-80 00:00:00 |  800.00 |         |     20
  7499 | ALLEN  | SALESMAN  | 7698 | 20-FEB-81 00:00:00 | 1600.00 |  300.00 |     30
  7521 | WARD   | SALESMAN  | 7698 | 22-FEB-81 00:00:00 | 1250.00 |  500.00 |     30
  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)
```

8.5.2 Access Method Hints

The following hints influence how the optimizer accesses relations to create the result set.

Table 3-8-2 Access Method Hints

Hint	Description
`FULL(table)`	Perform a full sequential scan on *table*.
`INDEX(table [ index ] [...])`	Use *index* on *table* to access the relation.
`NO_INDEX(table [ index ] [...])`	Do not use *index* on *table* to access the relation.

In addition, the `ALL_ROWS`, `FIRST_ROWS`, and `FIRST_ROWS(n)` hints of Table 3-8-1 can be used.

Examples

The sample application does not have sufficient data to illustrate the effects of optimizer hints so the remainder of the examples in this section will use a banking database created by the `pgbench` application located in the `PostgresPlus\9.4AS\bin` subdirectory.

The following steps create a database named, `bank`, populated by the tables, `accounts`, `branches`, `tellers`, and `history`. The `-s 5` option specifies a scaling factor of five which results in the creation of five branches, each with 100,000 accounts, resulting in a total of 500,000 rows in the `accounts` table and five rows in the `branches` table. Ten tellers are assigned to each branch resulting in a total of 50 rows in the `tellers` table.

Note, if using Linux use the `export` command instead of the `SET PATH` command as shown below.

```
export PATH=/opt/PostgresPlus/9.4AS/bin:$PATH
```

The following example was run in Windows.

```
SET PATH=C:\PostgresPlus\9.4AS\bin;%PATH%

createdb -U enterprisedb bank
CREATE DATABASE

pgbench -i -s 5 -U enterprisedb -d bank

creating tables...
10000 tuples done.
20000 tuples done.
30000 tuples done.
        .
        .
        .
470000 tuples done.
```



```
480000 tuples done.
490000 tuples done.
500000 tuples done.
set primary key...
vacuum...done.
```

Ten transactions per client are then processed for eight clients for a total of 80 transactions. This will populate the `history` table with 80 rows.

```
pgbench -U enterprisedb -d bank -c 8 -t 10
        .
        .
        .
transaction type: TPC-B (sort of)
scaling factor: 5
number of clients: 8
number of transactions per client: 10
number of transactions actually processed: 80/80
tps = 6.023189 (including connections establishing)
tps = 7.140944 (excluding connections establishing)
```

The table definitions are shown below:

```
\d accounts

       Table "public.accounts"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 aid      | integer       | not null
 bid      | integer       |
 abalance | integer       |
 filler   | character(84) |
Indexes:
    "accounts_pkey" PRIMARY KEY, btree (aid)

\d branches

       Table "public.branches"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 bid      | integer       | not null
 bbalance | integer       |
 filler   | character(88) |
Indexes:
    "branches_pkey" PRIMARY KEY, btree (bid)

\d tellers

        Table "public.tellers"
  Column  |     Type      | Modifiers
----------+---------------+-----------
 tid      | integer       | not null
 bid      | integer       |
 tbalance | integer       |
 filler   | character(84) |
Indexes:
    "tellers_pkey" PRIMARY KEY, btree (tid)

\d history
```

```
               Table "public.history"
 Column |            Type             | Modifiers
--------+-----------------------------+-----------
 tid    | integer                     |
 bid    | integer                     |
 aid    | integer                     |
 delta  | integer                     |
 mtime  | timestamp without time zone |
 filler | character(22)               |
```

The EXPLAIN command shows the plan selected by the query planner. In the following example, aid is the primary key column, so an indexed search is used on index, accounts_pkey.

```
EXPLAIN SELECT * FROM accounts WHERE aid = 100;

                                  QUERY PLAN
-----------------------------------------------------------------------------
--
 Index Scan using accounts_pkey on accounts  (cost=0.00..8.32 rows=1
width=97)
   Index Cond: (aid = 100)
(2 rows)
```

The FULL hint is used to force a full sequential scan instead of using the index as shown below:

```
EXPLAIN SELECT /*+ FULL(accounts) */ * FROM accounts WHERE aid = 100;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on accounts  (cost=0.00..14461.10 rows=1 width=97)
   Filter: (aid = 100)
(2 rows)
```

The NO_INDEX hint also forces a sequential scan as shown below:

```
EXPLAIN SELECT /*+ NO_INDEX(accounts accounts_pkey) */ * FROM accounts WHERE
aid = 100;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on accounts  (cost=0.00..14461.10 rows=1 width=97)
   Filter: (aid = 100)
(2 rows)
```

In addition to using the EXPLAIN command as shown in the prior examples, more detailed information regarding whether or not a hint was used by the planner can be obtained by setting the client_min_messages and trace_hints configuration parameters as follows:

```
SET client_min_messages TO info;
SET trace_hints TO true;
```

The SELECT command with the NO_INDEX hint is repeated below to illustrate the additional information produced when the aforementioned configuration parameters are set.

```
EXPLAIN SELECT /*+ NO_INDEX(accounts accounts_pkey) */ * FROM accounts WHERE
aid = 100;

INFO:  [HINTS] Index Scan of [accounts].[accounts_pkey] rejected because of
NO_INDEX hint.

INFO:  [HINTS] Bitmap Heap Scan of [accounts].[accounts_pkey] rejected
because of NO_INDEX hint.
                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on accounts  (cost=0.00..14461.10 rows=1 width=97)
   Filter: (aid = 100)
(2 rows)
```

Note that if a hint is ignored, the INFO: [HINTS] line will not appear. This may be an indication that there was a syntax error or some other misspelling in the hint as shown in the following example where the index name is misspelled.

```
EXPLAIN SELECT /*+ NO_INDEX(accounts accounts_xxx) */ * FROM accounts WHERE
aid = 100;

                                  QUERY PLAN
-----------------------------------------------------------------------------
--
 Index Scan using accounts_pkey on accounts  (cost=0.00..8.32 rows=1
 width=97)
   Index Cond: (aid = 100)
(2 rows
```

8.5.3 Specifying a Join Order

Include the `ORDERED` directive to instruct the query optimizer to join tables in the order in which they are listed in the `FROM` clause. If you do not include the `ORDERED` keyword, the query optimizer will choose the order in which to join the tables.

For example, the following command allows the optimizer to choose the order in which to join the tables listed in the `FROM` clause:

```
SELECT e.ename, d.dname, h.startdate
  FROM emp e, dept d, jobhist h
  WHERE d.deptno = e.deptno
  AND h.empno = e.empno;
```

The following command instructs the optimizer to join the tables in the ordered specified:

```
SELECT /*+ ORDERED */ e.ename, d.dname, h.startdate
  FROM emp e, dept d, jobhist h
  WHERE d.deptno = e.deptno
  AND h.empno = e.empno;
```

In the `ORDERED` version of the command, Advanced Server will first join `emp e` with `dept d` before joining the results with `jobhist h`. Without the `ORDERED` directive, the join order is selected by the query optimizer.

Please note: the `ORDERED` directive does not work for outer joins that contain a '+' sign.

8.5.4 Joining Relations Hints

When two tables are to be joined, there are three possible plans that may be used to perform the join.

- *Nested Loop Join* – The right table is scanned once for every row in the left table.
- *Merge Sort Join* – Each table is sorted on the join attributes before the join starts. The two tables are then scanned in parallel and the matching rows are combined to form the join rows.
- *Hash Join* – The right table is scanned and its join attributes are loaded into a hash table using its join attributes as hash keys. The left table is then scanned and its join attributes are used as hash keys to locate the matching rows from the right table.

The following table lists the optimizer hints that can be used to influence the planner to use one type of join plan over another.

Table 3-8-3 Join Hints

Hint	Description
`USE_HASH(`*`table`*` [...])`	Use a hash join with a hash table created from the join attributes of *`table`*.
`NO_USE_HASH(`*`table`*` [...])`	Do not use a hash join created from the join attributes of *`table`*.
`USE_MERGE(`*`table`*` [...])`	Use a merge sort join for *`table`*.
`NO_USE_MERGE(`*`table`*` [...])`	Do not use a merge sort join for *`table`*.
`USE_NL(`*`table`*` [...])`	Use a nested loop join for *`table`*.
`NO_USE_NL(`*`table`*` [...])`	Do not use a nested loop join for *`table`*.

Examples

In the following example, a join is performed on the `branches` and `accounts` tables. The query plan shows that a hash join is used by creating a hash table from the join attribute of the `branches` table.

```
EXPLAIN SELECT b.bid, a.aid, abalance FROM branches b, accounts a WHERE b.bid
= a.bid;

                                QUERY PLAN
--------------------------------------------------------------------------
 Hash Join  (cost=1.11..20092.70 rows=500488 width=12)
   Hash Cond: (a.bid = b.bid)
   ->  Seq Scan on accounts a  (cost=0.00..13209.88 rows=500488 width=12)
   ->  Hash  (cost=1.05..1.05 rows=5 width=4)
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
(5 rows)
```

By using the `USE_HASH(a)` hint, the planner is forced to create the hash table from the `accounts` join attribute instead of from the `branches` table. Note the use of the alias, `a`, for the `accounts` table in the `USE_HASH` hint.

```
EXPLAIN SELECT /*+ USE_HASH(a) */ b.bid, a.aid, abalance FROM branches b,
accounts a WHERE b.bid = a.bid;

                                   QUERY PLAN
-----------------------------------------------------------------------------
---
 Hash Join  (cost=21909.98..30011.52 rows=500488 width=12)
   Hash Cond: (b.bid = a.bid)
   ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
   ->  Hash  (cost=13209.88..13209.88 rows=500488 width=12)
         ->  Seq Scan on accounts a  (cost=0.00..13209.88 rows=500488
width=12)
(5 rows)
```

Next, the `NO_USE_HASH(a b)` hint forces the planner to use an approach other than hash tables. The result is a nested loop.

```
EXPLAIN SELECT /*+ NO_USE_HASH(a b) */ b.bid, a.aid, abalance FROM branches
b, accounts a WHERE b.bid = a.bid;

                                QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop  (cost=1.05..69515.84 rows=500488 width=12)
   Join Filter: (b.bid = a.bid)
   ->  Seq Scan on accounts a  (cost=0.00..13209.88 rows=500488 width=12)
   ->  Materialize  (cost=1.05..1.11 rows=5 width=4)
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
(5 rows)
```

Finally, the `USE_MERGE` hint forces the planner to use a merge join.

```
EXPLAIN SELECT /*+ USE_MERGE(a) */ b.bid, a.aid, abalance FROM branches b,
accounts a WHERE b.bid = a.bid;

                                   QUERY PLAN
-----------------------------------------------------------------------------
---
 Merge Join  (cost=69143.62..76650.97 rows=500488 width=12)
   Merge Cond: (b.bid = a.bid)
   ->  Sort  (cost=1.11..1.12 rows=5 width=4)
         Sort Key: b.bid
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
   ->  Sort  (cost=69142.52..70393.74 rows=500488 width=12)
         Sort Key: a.bid
         ->  Seq Scan on accounts a  (cost=0.00..13209.88 rows=500488
width=12)
(8 rows)
```

In this three-table join example, the planner first performs a hash join on the `branches` and `history` tables, then finally performs a nested loop join of the result with the `accounts_pkey` index of the `accounts` table.


```
EXPLAIN SELECT h.mtime, h.delta, b.bid, a.aid FROM history h, branches b,
accounts a WHERE h.bid = b.bid AND h.aid = a.aid;

                                      QUERY PLAN
-----------------------------------------------------------------------------
---------
 Nested Loop  (cost=1.11..207.95 rows=26 width=20)
   ->  Hash Join  (cost=1.11..25.40 rows=26 width=20)
         Hash Cond: (h.bid = b.bid)
         ->  Seq Scan on history h  (cost=0.00..20.20 rows=1020 width=20)
         ->  Hash  (cost=1.05..1.05 rows=5 width=4)
               ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
   ->  Index Scan using accounts_pkey on accounts a  (cost=0.00..7.01 rows=1
       width=4)
         Index Cond: (h.aid = a.aid)
(8 rows)
```

This plan is altered by using hints to force a combination of a merge sort join and a hash join.

```
EXPLAIN SELECT /*+ USE_MERGE(h b) USE_HASH(a) */ h.mtime, h.delta, b.bid,
a.aid FROM history h, branches b, accounts a WHERE h.bid = b.bid AND h.aid =
a.aid;

                                        QUERY PLAN
-----------------------------------------------------------------------------
--------------
 Merge Join  (cost=23480.11..23485.60 rows=26 width=20)
   Merge Cond: (h.bid = b.bid)
   ->  Sort  (cost=23479.00..23481.55 rows=1020 width=20)
         Sort Key: h.bid
         ->  Hash Join  (cost=21421.98..23428.03 rows=1020 width=20)
               Hash Cond: (h.aid = a.aid)
               ->  Seq Scan on history h  (cost=0.00..20.20 rows=1020
                   width=20)
               ->  Hash  (cost=13209.88..13209.88 rows=500488 width=4)
                     ->  Seq Scan on accounts a  (cost=0.00..13209.88
                         rows=500488 width=4)
   ->  Sort  (cost=1.11..1.12 rows=5 width=4)
         Sort Key: b.bid
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
(12 rows)
```

8.5.5 Global Hints

Thus far, hints have been applied directly to tables that are referenced in the SQL command. It is also possible to apply hints to tables that appear in a view when the view is referenced in the SQL command. The hint does not appear in the view, itself, but rather in the SQL command that references the view.

When specifying a hint that is to apply to a table within a view, the view and table names are given in dot notation within the hint argument list.

Synopsis

hint(*view*.*table*)

Parameters

hint

Any of the hints in Table 3-8-2 or Table 3-8-3.

view

The name of the view containing *table*.

table

The table on which the hint is to be applied.

Examples

A view named, `tx`, is created from the three-table join of `history`, `branches`, and `accounts` shown in the final example of Section 8.5.3.

```
CREATE VIEW tx AS SELECT h.mtime, h.delta, b.bid, a.aid FROM history h,
branches b, accounts a WHERE h.bid = b.bid AND h.aid = a.aid;
```

The query plan produced by selecting from this view is show below:

```
EXPLAIN SELECT * FROM tx;

                                      QUERY PLAN
-----------------------------------------------------------------------------
---------
 Nested Loop  (cost=1.11..207.95 rows=26 width=20)
   ->  Hash Join  (cost=1.11..25.40 rows=26 width=20)
         Hash Cond: (h.bid = b.bid)
         ->  Seq Scan on history h  (cost=0.00..20.20 rows=1020 width=20)
```

```
         ->  Hash  (cost=1.05..1.05 rows=5 width=4)
               ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
   ->  Index Scan using accounts_pkey on accounts a  (cost=0.00..7.01 rows=1
       width=4)
         Index Cond: (h.aid = a.aid)
(8 rows)
```

The same hints that were applied to this join at the end of Section 8.5.3 can be applied to the view as follows:

```
EXPLAIN SELECT /*+ USE_MERGE(tx.h tx.b) USE_HASH(tx.a) */ * FROM tx;

                                        QUERY PLAN

-----------------------------------------------------------------------------
-------------
-
 Merge Join  (cost=23480.11..23485.60 rows=26 width=20)
   Merge Cond: (h.bid = b.bid)
   ->  Sort  (cost=23479.00..23481.55 rows=1020 width=20)
         Sort Key: h.bid
         ->  Hash Join  (cost=21421.98..23428.03 rows=1020 width=20)
               Hash Cond: (h.aid = a.aid)
               ->  Seq Scan on history h  (cost=0.00..20.20 rows=1020
                   width=20)
               ->  Hash  (cost=13209.88..13209.88 rows=500488 width=4)
                     ->  Seq Scan on accounts a  (cost=0.00..13209.88
                         rows=500488 width=4)
   ->  Sort  (cost=1.11..1.12 rows=5 width=4)
         Sort Key: b.bid
         ->  Seq Scan on branches b  (cost=0.00..1.05 rows=5 width=4)
(12 rows)
```

In addition to applying hints to tables within stored views, hints can be applied to tables within subqueries as illustrated by the following example. In this query on the sample application `emp` table, employees and their managers are listed by joining the `emp` table with a subquery of the `emp` table identified by the alias, `b`.

```
SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename" FROM emp a,
(SELECT * FROM emp) b WHERE a.mgr = b.empno;

empno | ename  | mgr empno | mgr ename
-------+--------+-----------+-----------
  7902 | FORD   |      7566 | JONES
  7788 | SCOTT  |      7566 | JONES
  7521 | WARD   |      7698 | BLAKE
  7844 | TURNER |      7698 | BLAKE
  7654 | MARTIN |      7698 | BLAKE
  7900 | JAMES  |      7698 | BLAKE
  7499 | ALLEN  |      7698 | BLAKE
  7934 | MILLER |      7782 | CLARK
  7876 | ADAMS  |      7788 | SCOTT
  7782 | CLARK  |      7839 | KING
  7698 | BLAKE  |      7839 | KING
  7566 | JONES  |      7839 | KING
  7369 | SMITH  |      7902 | FORD
(13 rows)
```

The plan chosen by the query planner is shown below:

```
EXPLAIN SELECT a.empno, a.ename, b.empno "mgr empno", b.ename "mgr ename"
FROM emp a, (SELECT * FROM emp) b WHERE a.mgr = b.empno;

                            QUERY PLAN
-------------------------------------------------------------------
 Merge Join  (cost=2.81..3.08 rows=13 width=26)
   Merge Cond: (a.mgr = emp.empno)
   ->  Sort  (cost=1.41..1.44 rows=14 width=20)
         Sort Key: a.mgr
         ->  Seq Scan on emp a  (cost=0.00..1.14 rows=14 width=20)
   ->  Sort  (cost=1.41..1.44 rows=14 width=13)
         Sort Key: emp.empno
         ->  Seq Scan on emp  (cost=0.00..1.14 rows=14 width=13)
(8 rows)
```

A hint can be applied to the `emp` table within the subquery to perform an index scan on index, `emp_pk`, instead of a table scan. Note the difference in the query plans.

```
EXPLAIN SELECT /*+ INDEX(b.emp emp_pk) */ a.empno, a.ename, b.empno "mgr
empno", b.ename "mgr ename" FROM emp a, (SELECT * FROM emp) b WHERE a.mgr =
b.empno;

                                QUERY PLAN
---------------------------------------------------------------------------
 Merge Join  (cost=1.41..13.21 rows=13 width=26)
   Merge Cond: (a.mgr = emp.empno)
   ->  Sort  (cost=1.41..1.44 rows=14 width=20)
         Sort Key: a.mgr
         ->  Seq Scan on emp a  (cost=0.00..1.14 rows=14 width=20)
   ->  Index Scan using emp_pk on emp  (cost=0.00..12.46 rows=14 width=13)
(6 rows)
```

8.5.6 Using the APPEND Optimizer Hint

By default, Advanced Server will add new data into the first available free-space in a table (vacated by vacuumed records). Include the `APPEND` directive after an `INSERT` or `SELECT` command to instruct the server to bypass mid-table free space, and affix new rows to the end of the table. This optimizer hint can be particularly useful when bulk loading data.

The syntax is:

```
/*+APPEND*/
```

For example, the following command instructs the server to append the data in the `INSERT` statement to the end of the `sales` table:

```
INSERT /*+APPEND*/ INTO sales VALUES
(10, 10, '01-Mar-2011', 10, 'OR');
```

Note that Advanced Server supports the `APPEND` hint when adding multiple rows in a single `INSERT` statement:

```
INSERT /*+APPEND*/ INTO sales VALUES
(20, 20, '01-Aug-2011', 20, 'NY'),
(30, 30, '01-Feb-2011', 30, 'FL'),
(40, 40, '01-Nov-2011', 40, 'TX');
```

The `APPEND` hint can also be included in the `SELECT` clause of an `INSERT INTO` statement:

```
INSERT INTO sales_history SELECT /*+APPEND*/ FROM sales;
```

8.5.7 Conflicting Hints

If a command includes two or more conflicting hints, the server will ignore the contradictory hints. The following table lists hints that are contradictory to each other.

Table 3-8-4 Conflicting Hints

Hint	Conflicting Hint
`ALL_ROWS`	`FIRST_ROWS` - all formats
`FULL(`*table*`)`	`INDEX(`*table* `[` *index* `])`
`INDEX(`*table*`)`	`FULL(`*table*`)` `NO_INDEX(`*table*`)`
`INDEX(`*table index*`)`	`FULL(`*table*`)` `NO_INDEX(`*table index*`)`
`USE_HASH(`*table*`)`	`NO_USE_HASH(`*table*`)`
`USE_MERGE(`*table*`)`	`NO_USE_MERGE(`*table*`)`
`USE_NL(`*table*`)`	`NO_USE_NL(`*table*`)`

8.6 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.

8.6.1 Querying the DBMS_PROFILER Tables and View

The following step-by-step example uses DBMS_PROFILER to retrieve performance information for procedures, functions, and triggers included in the sample data distributed with Advanced Server.

1. Open the EDB-PSQL command line, and establish a connection to the Advanced Server database. Use an `EXEC` statement to start the profiling session:

```
acctg=# EXEC dbms_profiler.start_profiler('profile list_emp');

EDB-SPL Procedure successfully completed
```

   (Note: the call to `start_profiler()` includes a comment that DBMS_PROFILER associates with the profiler session).

2. Then call the `list_emp` function:

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

(1 row)
```

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

```
acctg=# EXEC dbms_profiler.stop_profiler;

EDB-SPL Procedure successfully completed
```

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

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

EDB-SPL Procedure successfully completed
```

5. Invoke the `get_dept_name` function:

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

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

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

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

```
acctg=# EXEC dbms_profiler.stop_profiler;

EDB-SPL Procedure successfully completed
```

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

```
acctg=# SELECT * FROM plsql_profiler_runs;
 runid | related_run |  run_owner   |         run_date          |               run_comment
| run_total_time | run_system_info | run_comment1 | spare1
-------+-------------+--------------+---------------------------+----------------------------
-----------+----------------+-----------------+--------------+--------
     1 |             | enterprisedb | 04-FEB-14 09:32:48.874315 | profile list_emp
|           4154 |                 |              |
     2 |             | enterprisedb | 04-FEB-14 09:41:30.546503 | profile get_dept_name and
emp_sal_trig |           2088 |                 |              |
(2 rows)
```

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

```
acctg=# SELECT * FROM plsql_profiler_units;
 runid | unit_number | unit_type |  unit_owner  |            unit_name            |
unit_timestamp | total_time | spare1 | spare2
-------+-------------+-----------+--------------+---------------------------------+-----------
-----+------------+--------+--------
```

```
     1 |       16999 | FUNCTION   | enterprisedb | list_emp()                     |
|         4 |         |
     2 |       17002 | FUNCTION   | enterprisedb | user_audit_trig()              |
|         1 |         |
     2 |       17000 | FUNCTION   | enterprisedb | get_dept_name(p_deptno numeric) |
|         1 |         |
     2 |       17004 | FUNCTION   | enterprisedb | emp_sal_trig()                 |
|         1 |         |
(4 rows)
```

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

```
acctg=# SELECT runid, sourcecode, func_oid, line_number, exec_count, tuples_returned,
time_total FROM plsql_profiler_rawdata;
runid |                            sourcecode                            | func_oid |
line_number | exec_count | tuples_returned | time_total
-------+------------------------------------------------------------------+----------+--------
-----+------------+-----------------+------------
     1 | DECLARE                                                          |    16999 |
1 |         0 |                 0 |           0
     1 |     v_empno         NUMERIC(4);                                  |    16999 |
2 |         0 |                 0 |           0
     1 |     v_ename         VARCHAR(10);                                 |    16999 |
3 |         0 |                 0 |           0
     1 |     emp_cur CURSOR FOR                                           |    16999 |
4 |         0 |                 0 |           0
     1 |         SELECT empno, ename FROM memp ORDER BY empno;            |    16999 |
5 |         0 |                 0 |           0
     1 | BEGIN                                                            |    16999 |
6 |         0 |                 0 |           0
     1 |     OPEN emp_cur;                                                |    16999 |
7 |         0 |                 0 |           0
     1 |     RAISE INFO 'EMPNO    ENAME';                                 |    16999 |
8 |         1 |                 0 |    0.001621
     1 |     RAISE INFO '-----    -------';                               |    16999 |
9 |         1 |                 0 |    0.000301
     1 |     LOOP                                                         |    16999 |
10 |         1 |                 0 |     4.6e-05
     1 |         FETCH emp_cur INTO v_empno, v_ename;                     |    16999 |
11 |         1 |                 0 |    0.001114
     1 |         EXIT WHEN NOT FOUND;                                     |    16999 |
12 |        15 |                 0 |    0.000206
     1 |         RAISE INFO '%      %', v_empno, v_ename;                 |    16999 |
13 |        15 |                 0 |     8.3e-05
     1 |     END LOOP;                                                    |    16999 |
14 |        14 |                 0 |    0.000773
     1 |     CLOSE emp_cur;                                               |    16999 |
15 |         0 |                 0 |           0
     1 |     RETURN;                                                      |    16999 |
16 |         1 |                 0 |       1e-05
     1 | END;                                                             |    16999 |
17 |         1 |                 0 |           0
     1 |                                                                  |    16999 |
18 |         0 |                 0 |           0
     2 | DECLARE                                                          |    17002 |
1 |         0 |                 0 |           0
     2 |     v_action        VARCHAR(24);                                 |    17002 |
2 |         0 |                 0 |           0
     2 |     v_text          TEXT;                                        |    17002 |
3 |         0 |                 0 |           0
     2 | BEGIN                                                            |    17002 |
4 |         0 |                 0 |           0
     2 |     IF TG_OP = 'INSERT' THEN                                     |    17002 |
5 |         0 |                 0 |           0
     2 |         v_action := ' added employee(s) on ';                    |    17002 |
6 |         1 |                 0 |    0.000143
```



```
     2 |     ELSIF TG_OP = 'UPDATE' THEN                                  |    17002 |
7 |          0 |                 0 |           0
     2 |         v_action := ' updated employee(s) on ';                  |    17002 |
8 |          0 |                 0 |           0
     2 |     ELSIF TG_OP = 'DELETE' THEN                                  |    17002 |
9 |          1 |                 0 |     3.2e-05
     2 |         v_action := ' deleted employee(s) on ';                  |    17002 |
10 |          0 |                 0 |           0
     2 |     END IF;                                                      |    17002 |
11 |          0 |                 0 |           0
     2 |     v_text := 'User ' || USER || v_action || CURRENT_DATE;       |    17002 |
12 |          0 |                 0 |           0
     2 |     RAISE INFO ' %', v_text;                                     |    17002 |
13 |          1 |                 0 |    0.000383
     2 |     RETURN NULL;                                                 |    17002 |
14 |          1 |                 0 |     6.3e-05
     2 | END;                                                             |    17002 |
15 |          1 |                 0 |     3.6e-05
     2 |                                                                  |    17002 |
16 |          0 |                 0 |           0
     2 | DECLARE                                                          |    17000 |
1 |          0 |                 0 |           0
     2 |     v_dname         VARCHAR(14);                                 |    17000 |
2 |          0 |                 0 |           0
     2 | BEGIN                                                            |    17000 |
3 |          0 |                 0 |           0
     2 |     SELECT INTO v_dname dname FROM dept WHERE deptno = p_deptno; |    17000 |
4 |          0 |                 0 |           0
     2 |     RETURN v_dname;                                              |    17000 |
5 |          1 |                 0 |    0.000647
     2 |     IF NOT FOUND THEN                                            |    17000 |
6 |          1 |                 0 |     2.6e-05
     2 |         RAISE INFO 'Invalid department number %', p_deptno;      |    17000 |
7 |          0 |                 0 |           0
     2 |         RETURN '';                                               |    17000 |
8 |          0 |                 0 |           0
     2 |     END IF;                                                      |    17000 |
9 |          0 |                 0 |           0
     2 | END;                                                             |    17000 |
10 |          0 |                 0 |           0
     2 |                                                                  |    17000 |
11 |          0 |                 0 |           0
     2 | DECLARE                                                          |    17004 |
1 |          0 |                 0 |           0
     2 |     sal_diff       NUMERIC(7,2);                                 |    17004 |
2 |          0 |                 0 |           0
     2 | BEGIN                                                            |    17004 |
3 |          0 |                 0 |           0
     2 |     IF TG_OP = 'INSERT' THEN                                     |    17004 |
4 |          0 |                 0 |           0
     2 |         RAISE INFO 'Inserting employee %', NEW.empno;            |    17004 |
5 |          1 |                 0 |     8.4e-05
     2 |         RAISE INFO '..New salary: %', NEW.sal;                   |    17004 |
6 |          0 |                 0 |           0
     2 |         RETURN NEW;                                              |    17004 |
7 |          0 |                 0 |           0
     2 |     END IF;                                                      |    17004 |
8 |          0 |                 0 |           0
     2 |     IF TG_OP = 'UPDATE' THEN                                     |    17004 |
9 |          0 |                 0 |           0
     2 |         sal_diff := NEW.sal - OLD.sal;                           |    17004 |
10 |          1 |                 0 |    0.000355
     2 |         RAISE INFO 'Updating employee %', OLD.empno;             |    17004 |
11 |          1 |                 0 |    0.000177
     2 |         RAISE INFO '..Old salary: %', OLD.sal;                   |    17004 |
12 |          1 |                 0 |     5.5e-05
     2 |         RAISE INFO '..New salary: %', NEW.sal;                   |    17004 |
13 |          1 |                 0 |     3.1e-05
     2 |         RAISE INFO '..Raise     : %', sal_diff;                  |    17004 |
14 |          1 |                 0 |     2.8e-05
```

```
     2 |         RETURN NEW;                                            |    17004 |
15 |         1 |                 0 |     2.7e-05
     2 |     END IF;                                                    |    17004 |
16 |         1 |                 0 |       1e-06
     2 |     IF TG_OP = 'DELETE' THEN                                   |    17004 |
17 |         0 |                 0 |           0
     2 |         RAISE INFO 'Deleting employee %', OLD.empno;           |    17004 |
18 |         0 |                 0 |           0
     2 |         RAISE INFO '..Old salary: %', OLD.sal;                 |    17004 |
19 |         0 |                 0 |           0
     2 |         RETURN OLD;                                            |    17004 |
20 |         0 |                 0 |           0
     2 |     END IF;                                                    |    17004 |
21 |         0 |                 0 |           0
     2 | END;                                                           |    17004 |
22 |         0 |                 0 |           0
     2 |                                                                |    17004 |
23 |         0 |                 0 |           0
(68 rows)
```

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

```
acctg=# SELECT * FROM plsql_profiler_data;
runid | unit_number | line# | total_occur | total_time | min_time | max_time | spare1 | spare2
| spare3 | spare4
-------+-------------+-------+-------------+------------+----------+----------+--------+------
--+--------+--------
     1 |       16999 |     1 |           0 |          0 |        0 |        0 |        |
|        |
     1 |       16999 |     2 |           0 |          0 |        0 |        0 |        |
|        |
     1 |       16999 |     3 |           0 |          0 |        0 |        0 |        |
|        |
     1 |       16999 |     4 |           0 |          0 |        0 |        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)
```

8.6.2 DBMS_PROFILER Functions and Procedures

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

Table 8-5 DBMS_PROFILER Functions/Procedures

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

Return Values

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.

8.6.2.1 FLUSH_DATA

The `FLUSH_DATA` procedure or function flushes the data collected in the current session without terminating the profiler session. The data is flushed to the tables listed in Section 6.3 of the Postgres Plus Advanced Server Performance Features Guide. The signature of the `FLUSH_DATA` function is:

```
DBMS_PROFILER.FLUSH_DATA
    RETURN INTEGER;
```

The signature of the FLUSH_DATA procedure is:

```
DBMS_PROFILER.FLUSH_DATA;
```

8.6.2.2 GET_VERSION

The GET_VERSION procedure returns the version of DBMS_PROFILER. The procedure signature is:

```
DBMS_PROFILER.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.

8.6.2.3 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:

```
DBMS_PROFILER.INTERNAL_VERSION_CHECK
    RETURN INTEGER;
```

8.6.2.4 PAUSE_PROFILER

The PAUSE_PROFILER function or procedure pauses a profiling session. The function signature is:

```
DBMS_PROFILER.PAUSE_PROFILER
    RETURN INTEGER;
```

The signature of the PAUSE_PROFILER procedure is:

```
DBMS_PROFILER.PAUSE_PROFILER;
```

8.6.2.5 RESUME_PROFILER

The RESUME_PROFILER function or procedure resumes a paused profiling session. The function signature is:

```
DBMS_PROFILER.RESUME_PROFILER
    RETURN INTEGER;
```

The signature of the RESUME_PROFILER procedure is:

```
DBMS_PROFILER.RESUME_PROFILER;
```

8.6.2.6 START_PROFILER

The START_PROFILER function or procedure starts a data collection session. The START_PROFILER function has two forms:

```
DBMS_PROFILER.START_PROFILER(
    run_comment IN TEXT := sysdate,
    run_comment1 IN TEXT := '',
    run_number OUT INTEGER)
  RETURN INTEGER;
DBMS_PROFILER.START_PROFILER(
    run_comment IN TEXT := sysdate,
    run_comment1 IN TEXT := '')
  RETURN INTEGER;
```

The START_PROFILER procedure has two forms:

```
DBMS_PROFILER.START_PROFILER (
    run_comment IN TEXT := sysdate,
    run_comment1 IN TEXT := '');
DBMS_PROFILER.START_PROFILER (
    run_comment IN TEXT := sysdate,
    run_comment1 IN 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.

8.6.2.7 STOP_PROFILER

The `STOP_PROFILER` function or procedure stops a profiling session and flushes the performance information to the DBMS_PROFILER tables and view. The `STOP_PROFILER` function signature is:

```
DBMS_PROFILER.STOP_PROFILER;
  RETURN INTEGER;
```

The signature of the `START_PROFILER` procedure is:

```
DBMS_PROFILER.STOP_PROFILER;
```

8.6.3 DBMS_PROFILER - Reference

The Advanced Server installer creates the following tables and views that you can query to review PL/SQL performance profile information:

Table Name	Description
PLSQL_PROFILER_RUNS	Table containing information about all profiler runs, organized by runid.
PLSQL_PROFILER_UNITS	Table containing information about all profiler runs, organized by unit.
PLSQL_PROFILER_DATA	View containing performance statistics.
PLSQL_PROFILER_RAWDATA	Table containing the performance statistics *and* the extended performance statistics for DRITA counters and timers.

8.6.3.1 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 nanoseconds)
run_system_info	TEXT	Currently Unused
run_comment1	TEXT	Additional user comments
spare1	TEXT	Currently Unused

8.6.3.2 PLSQL_PROFILER_UNITS

The PLSQL_PROFILER_UNITS table contains the following columns:

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

Column	Data Type	Description
total_time	BIGINT	Time spent within the unit (in nanoseconds)
spare1	BIGINT	Currently Unused
spare2	BIGINT	Currently Unused

8.6.3.3 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.
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

8.6.3.4 PLSQL_PROFILER_RAWDATA

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

Column	Data Type	Description
runid	INTEGER	The run identifier (plsql_profiler_runnumber).
sourcecode	TEXT	The individual line of profiled code.
func_oid	OID	Object ID of the unit that contains the current line.
line_number	INTEGER	Current line number of the profiled workload.
exec_count	BIGINT	The number of times that the line was executed.
time_total	DOUBLE PRECISION	The amount of time spent executing the line.
time_shortest	DOUBLE PRECISION	The minimum execution time for the line.
time_longest	DOUBLE PRECISION	The maximum execution time for the line.
tuples_returned	BIGINT	Currently Unused
num_scans	BIGINT	Currently Unused
tuples_fetched	BIGINT	Currently Unused
tuples_inserted	BIGINT	Currently Unused
tuples_updated	BIGINT	Currently Unused
tuples_deleted	BIGINT	Currently Unused
blocks_fetched	BIGINT	Currently Unused

Column	Data Type	Description
blocks_hit	BIGINT	Currently Unused
wal_write	BIGINT	The server has waited for a write to the write-ahead log buffer (expect this value to be high).
wal_flush	BIGINT	The server has waited for the write-ahead log to flush to disk.
wal_file_sync	BIGINT	The server has waited for the write-ahead log to sync to disk (related to the wal_sync_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open_sync).
buffer_free_list_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory).
shmem_index_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the shared-memory map.
oid_gen_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the next available OID (object ID).
xid_gen_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the next available transaction ID.
proc_array_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the process array
sinval_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the cache invalidation state.
freespace_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the freespace map.
wal_insert_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small.
wal_write_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes write-ahead log flushes.
control_file_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number).
checkpoint_lock_acquire	BIGINT	A server process has waited for the short-term lock that prevents simultaneous checkpoints.
clog_control_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the commit log.
subtrans_control_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the subtransaction log.
multi_xact_gen_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes).
multi_xact_offset_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes).
multi_xact_member_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes).

Column	Data Type	Description
rel_cache_init_lock_acquire	BIGINT	The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.
bgwriter_communication_lock_acquire	BIGINT	The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.
two_phase_state_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the list of prepared transactions.
tablespace_create_lock_acquire	BIGINT	The server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands.
btree_vacuum_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.
add_in_shmem_lock_acquire	BIGINT	Currently Unused
autovacuum_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the shared autovacuum state.
autovacuum_schedule_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the autovacuum schedule.
syncscan_lock_acquire	BIGINT	The server has waited for the short-term lock that coordinates synchronous scans.
icache_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to InfiniteCache state
breakpoint_lock_acquire	BIGINT	The server has waited for the short-term lock that synchronizes access to the debugger breakpoint list.
lwlock_acquire	BIGINT	The server has waited for a short-term lock that has not been described elsewhere in this section.
db_file_read	BIGINT	A server process has waited for the completion of a read (from disk).
db_file_write	BIGINT	A server process has waited for the completion of a write (to disk).
db_file_sync	BIGINT	A server process has waited for the operating system to flush all changes to disk.
db_file_extend	BIGINT	A server process has waited for the operating system while adding a new page to the end of a file.
sql_parse	BIGINT	Currently Unused
query_plan	BIGINT	The server has generated a query plan.
infinitecache_read	BIGINT	The server has waited for an Infinite Cache read request.
infinitecache_write	BIGINT	The server has waited for an Infinite Cache write request.
wal_write_time	BIGINT	The amount of time that the server has waited for a write to the write-ahead log buffer (expect this value to be high).
wal_flush_time	BIGINT	The amount of time that the server has waited for the write-ahead log to flush to disk.
wal_file_sync_time	BIGINT	The amount of time that the server has waited for the write-ahead log to sync to disk (related to the wal_sync_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open_sync).
buffer_free_list_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the list of

Column	Data Type	Description
		free buffers (in shared memory).
shmem_index_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the shared-memory map.
oid_gen_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the next available OID (object ID).
xid_gen_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the next available transaction ID.
proc_array_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the process array.
sinval_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the cache invalidation state.
freespace_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the freespace map.
wal_insert_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small.
wal_write_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes write-ahead log flushes.
control_file_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number).
checkpoint_lock_acquire_time	BIGINT	The amount of time that the server process has waited for the short-term lock that prevents simultaneous checkpoints.
clog_control_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the commit log.
subtrans_control_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the subtransaction log.
multi_xact_gen_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes).
multi_xact_offset_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes).
multi_xact_member_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes).
rel_cache_init_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.
bgwriter_communication_loc	BIGINT	The amount of time that the bgwriter (background

Column	Data Type	Description
k_acquire_time		writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.
two_phase_state_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the list of prepared transactions.
tablespace_create_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that prevents simultaneous CREATE TABLESPACE or DROP TABLESPACE commands.
btree_vacuum_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.
add_in_shmem_lock_acquire_time	BIGINT	Obsolete/unused
autovacuum_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the shared autovacuum state.
autovacuum_schedule_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the autovacuum schedule.
syncscan_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that coordinates synchronous scans.
icache_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to InfiniteCache state
breakpoint_lock_acquire_time	BIGINT	The amount of time that the server has waited for the short-term lock that synchronizes access to the debugger breakpoint list.
lwlock_acquire_time	BIGINT	The amount of time that the server has waited for a short-term lock that has not been described elsewhere in this section.
db_file_read_time	BIGINT	The amount of time that the server process has waited for the completion of a read (from disk).
db_file_write_time	BIGINT	The amount of time that the server process has waited for the completion of a write (to disk).
db_file_sync_time	BIGINT	The amount of time that the server process has waited for the operating system to flush all changes to disk.
db_file_extend_time	BIGINT	The amount of time that the server process has waited for the operating system while adding a new page to the end of a file.
sql_parse_time	BIGINT	The amount of time that the server has parsed a SQL statement.
query_plan_time	BIGINT	The amount of time that the server has computed the execution plan for a SQL statement.
infinitecache_read_time	BIGINT	The amount of time that the server has waited for an Infinite Cache read request.
infinitecache_write_time	BIGINT	The amount of time that the server has waited for an Infinite Cache write request.
totalwaits	BIGINT	The total number of event waits.
totalwaittime	BIGINT	The total time spent waiting for an event.

8.7 Dynamic Runtime Instrumentation Tools Architecture (DRITA)

The Dynamic Runtime Instrumentation Tools Architecture (DRITA) allows a DBA to query catalog views to determine the *wait events* that affect the performance of individual sessions or the system as a whole. DRITA records the number of times each event occurs as well as the time spent waiting; you can use this information to diagnose performance problems. DRITA offers this functionality, while consuming minimal system resources.

DRITA compares *snapshots* to evaluate the performance of a system. A snapshot is a saved set of system performance data at a given point in time. Each snapshot is identified by a unique ID number; you can use snapshot ID numbers with DRITA reporting functions to return system performance statistics.

8.7.1 Configuring and Using DRITA

Advanced Server's `postgresql.conf` file includes a configuration parameter named `timed_statistics` that controls the collection of timing data. The valid parameter values are `TRUE` or `FALSE`; the default value is `FALSE`.

This is a dynamic parameter which can be modified in the `postgresql.conf` file, or while a session is in progress. To enable DRITA, you must either:

Modify the `postgresql.conf` file, setting the `timed_statistics` parameter to `TRUE`.

or

Connect to the server with the EDB-PSQL client, and invoke the command:

```
SET timed_statistics = TRUE
```

After modifying the `timed_statistics` parameter, take a starting snapshot. A snapshot captures the current state of each timer and event counter. The server will compare the starting snapshot to a later snapshot to gauge system performance.

Use the `edbsnap()` function to take the beginning snapshot:

```
edb=# SELECT * FROM edbsnap();
       edbsnap
----------------------
 Statement processed.
(1 row)
```

Then, run the workload that you would like to evaluate; when the workload has completed (or at a strategic point during the workload), take another snapshot:

```
edb=# SELECT * FROM edbsnap();
       edbsnap
----------------------
 Statement processed.
(1 row)
```

You can capture multiple snapshots during a session. Then, use the DRITA functions and reports to manage and compare the snapshots to evaluate performance information.

8.8 DRITA Functions

You can use DRITA functions to gather wait information and manage snapshots. DRITA functions are fully supported by Postgres Plus Advanced Server 9.5 when installed in either Oracle-compatible or PostgreSQL-compatible mode.

8.8.1 get_snaps()

The `get_snaps()` function returns a list of the current snapshots. The signature is:

```
get_snaps()
```

The following example demonstrates using the `get_snaps()` function to display a list of snapshots:

```
edb=# SELECT * FROM get_snaps();
          get_snaps
------------------------------
 1  11-FEB-10 10:41:05.668852
 2  11-FEB-10 10:42:27.26154
 3  11-FEB-10 10:45:48.999992
 4  11-FEB-10 11:01:58.345163
 5  11-FEB-10 11:05:14.092683
 6  11-FEB-10 11:06:33.151002
 7  11-FEB-10 11:11:16.405664
 8  11-FEB-10 11:13:29.458405
 9  11-FEB-10 11:23:57.595916
 10 11-FEB-10 11:29:02.214014
 11 11-FEB-10 11:31:44.244038
(11 rows)
```

The first column in the result list displays the snapshot identifier; the second column displays the date and time that the snapshot was captured.

8.8.2 sys_rpt()

The `sys_rpt()` function returns system wait information. The signature is:

`sys_rpt(`*`beginning_id, ending_id, top_n`*`)`

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

top_n

top_n represents the number of rows to return

This example demonstrates a call to the `sys_rpt()` function:

```
edb=# SELECT * FROM sys_rpt(9, 10, 10);
                                   sys_rpt
-----------------------------------------------------------------------------
 WAIT NAME                                 COUNT      WAIT TIME        % WAIT
 ---------------------------------------------------------------------------
 wal write                                 21250      104.723772       36.31
 db file read                              121407     72.143274        25.01
 wal flush                                 84185      51.652495        17.91
 wal file sync                             712        29.482206        10.22
 infinitecache write                       84178      15.814444        5.48
 db file write                             84177      14.447718        5.01
 infinitecache read                        672        0.098691         0.03
 db file extend                            190        0.040386         0.01
 query plan                                52         0.024400         0.01
 wal insert lock acquire                   4          0.000837         0.00
(12 rows)
```

The information displayed in the result set includes:

Column Name	Description
WAIT NAME	The name of the wait.
COUNT	The number of times that the wait event occurred.
WAIT TIME	The time of the wait event in milliseconds.
% WAIT	The percentage of the total wait time used by this wait for this session.

8.8.3 sess_rpt()

The `sess_rpt()` function returns session wait information. The signature is:

`sess_rpt(beginning_id, ending_id, top_n)`

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

top_n

top_n represents the number of rows to return

The following example demonstrates a call to the `sess_rpt()` function:

```
SELECT * FROM sess_rpt(18, 19, 10);

                              sess_rpt
-----------------------------------------------------------------------------
ID    USER       WAIT NAME              COUNT TIME(ms)   %WAIT SES  %WAIT ALL
 ----------------------------------------------------------------------------

 17373 enterprise db file read            30    0.175713   85.24      85.24
 17373 enterprise query plan              18    0.014930   7.24       7.24
 17373 enterprise wal flush               6     0.004067   1.97       1.97
 17373 enterprise wal write               1     0.004063   1.97       1.97
 17373 enterprise wal file sync           1     0.003664   1.78       1.78
 17373 enterprise infinitecache read      38    0.003076   1.49       1.49
 17373 enterprise infinitecache write     5     0.000548   0.27       0.27
 17373 enterprise db file extend          190   0.04.386   0.03       0.03
 17373 enterprise db file write           5     0.000082   0.04       0.04
 (11 rows)
```

The information displayed in the result set includes:

Column Name	Description
`ID`	The processID of the session.
`USER`	The name of the user incurring the wait.
`WAIT NAME`	The name of the wait event.
`COUNT`	The number of times that the wait event occurred.
`TIME (ms)`	The length of the wait event in milliseconds.
`% WAIT SES`	The percentage of the total wait time used by this wait for this session.
`% WAIT ALL`	The percentage of the total wait time used by this wait (for all sessions).

8.8.4 sessid_rpt()

The `sessid_rpt()` function returns session ID information for a specified backend. The signature is:

`sessid_rpt(`*beginning_id*`,` *ending_id*`,` *backend_id*`)`

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

backend_id

backend_id is an integer value that represents the backend identifier.

The following code sample demonstrates a call to `sessid_rpt()`:

```
SELECT * FROM sessid_rpt(18, 19, 17373);

                                sessid_rpt
-----------------------------------------------------------------------------
 ID    USER       WAIT NAME                COUNT TIME(ms)  %WAIT SES   %WAIT ALL
 ----------------------------------------------------------------------------
 17373 enterprise db file read              30   0.175713  85.24       85.24
 17373 enterprise query plan                18   0.014930  7.24        7.24
 17373 enterprise wal flush                 6    0.004067  1.97        1.97
 17373 enterprise wal write                 1    0.004063  1.97        1.97
 17373 enterprise wal file sync             1    0.003664  1.78        1.78
 17373 enterprise infinitecache read        38   0.003076  1.49        1.49
 17373 enterprise infinitecache write       5    0.000548  0.27        0.27
 17373 enterprise db file extend            190  0.040386  0.03        0.03
 17373 enterprise db file write             5    0.000082  0.04        0.04
(11 rows)
```

The information displayed in the result set includes:

Column Name	Description
`ID`	The process ID of the wait.
`USER`	The name of the user that owns the session.
`WAIT NAME`	The name of the wait event.
`COUNT`	The number of times that the wait event occurred.
`TIME (ms)`	The length of the wait in milliseconds.
`% WAIT SES`	The percentage of the total wait time used by this wait for this session.
`% WAIT ALL`	The percentage of the total wait time used by this wait (for all sessions).

8.8.5 sesshist_rpt()

The `sesshist_rpt()` function returns session wait information for a specified backend. The signature is:

```
sesshist_rpt(snapshot_id, session_id)
```

Parameters

snapshot_id

snapshot_id is an integer value that identifies the snapshot.

session_id

session_id is an integer value that represents the session.

The following example demonstrates a call to the `sesshist_rpt()` function:

```
edb=# SELECT * FROM sesshist_rpt (9, 5531);
                              sesshist_rpt
----------------------------------------------------------------------------
 ID    USER       SEQ  WAIT NAME
   ELAPSED(ms)    File  Name                    # of Blk   Sum of Blks
 ----------------------------------------------------------------------------
 5531 enterprise 1     db file read
   18546         14309  session_waits_pk        1          1
 5531 enterprise 2     infinitecache read
   125           14309  session_waits_pk        1          1
 5531 enterprise 3     db file read
   376           14304  edb$session_waits       0          1
 5531 enterprise 4     infinitecache read
   166           14304  edb$session_waits       0          1
 5531 enterprise 5     db file read
   7978          1260   pg_authid               0          1
 5531 enterprise 6     infinitecache read
   154           1260   pg_authid               0          1
 5531 enterprise 7     db file read
   628           14302  system_waits_pk         1          1
 5531 enterprise 8     infinitecache read
   463           14302  system_waits_pk         1          1
 5531 enterprise 9     db file read
   3446          14297  edb$system_waits        0          1
 5531 enterprise 10    infinitecache read
   187           14297  edb$system_waits        0          1
 5531 enterprise 11    db file read
   14750         14295  snap_pk                 1          1
 5531 enterprise 12    infinitecache read
   416           14295  snap_pk                 1          1
 5531 enterprise 13    db file read
   7139          14290  edb$snap                0          1
 5531 enterprise 14    infinitecache read
   158           14290  edb$snap                0          1
 5531 enterprise 15    db file read
   27287         14288  snapshot_num_seq        0          1
 5531 enterprise 16    infinitecache read
(17 rows)
```

The information displayed in the result set includes:

Column Name	Description
`ID`	The system-assigned identifier of the wait.
`USER`	The name of the user that incurred the wait.
`SEQ`	The sequence number of the wait event.
`WAIT NAME`	The name of the wait event.
`ELAPSED (ms)`	The length of the wait event in milliseconds.
`File`	The relfilenode number of the file.
`Name`	If available, the name of the file name related to the wait event.
`# of Blk`	The block number read or written for a specific instance of the event .
`Sum of Blks`	The number of blocks read.

8.8.6 purgesnap()

The `purgesnap()` function purges a range of snapshots from the snapshot tables. The signature is:

> `purgesnap(`*`beginning_id`*`, `*`ending_id`*`)`

Parameters

`beginning_id`

> *`beginning_id`* is an integer value that represents the beginning session identifier.

`ending_id`

> *`ending_id`* is an integer value that represents the ending session identifier.

`purgesnap()` removes all snapshots between *`beginning_id`* and *`ending_id`* (inclusive):

```
SELECT * FROM purgesnap(6, 9);

             purgesnap
------------------------------------
 Snapshots in range 6 to 9 deleted.
(1 row)
```

A call to the `get_snaps()` function after executing the example shows that snapshots `6` through `9` have been purged from the snapshot tables:

```
edb=# SELECT * FROM get_snaps();
          get_snaps
```

```
------------------------------
 1  11-FEB-10 10:41:05.668852
 2  11-FEB-10 10:42:27.26154
 3  11-FEB-10 10:45:48.999992
 4  11-FEB-10 11:01:58.345163
 5  11-FEB-10 11:05:14.092683
 10 11-FEB-10 11:29:02.214014
 11 11-FEB-10 11:31:44.244038
(7 rows)
```

8.8.7 truncsnap()

Use the `truncsnap()` function to delete all records from the snapshot table. The signature is:

```
truncsnap()
```

For example:

```
SELECT * FROM truncsnap();

      truncsnap
----------------------
 Snapshots truncated.
(1 row)
```

A call to the `get_snaps()` function after calling the `truncsnap()` function shows that all records have been removed from the snapshot tables:

```
SELECT * FROM get_snaps();
 get_snaps
-----------
(0 rows)
```

8.9 Simulating Statspack AWR Reports

The functions described in this section return information comparable to the information contained in an Oracle Statspack/AWR (Automatic Workload Repository) report. When taking a snapshot, performance data from system catalog tables is saved into history tables. The reporting functions listed below report on the differences between two given snapshots.

- `stat_db_rpt()`
- `stat_tables_rpt()`
- `statio_tables_rpt()`
- `stat_indexes_rpt()`
- `statio_indexes_rpt()`

The reporting functions can be executed individually or you can execute all five functions by calling the `edbreport()` function.

8.9.1 edbreport()

The `edbreport()` function includes data from the other reporting functions, plus additional system information. The signature is:

```
edbreport(beginning_id, ending_id)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

The call to the `edbreport()` function returns a composite report that contains system information and the reports returned by the other statspack functions. :

```
edb=# SELECT * FROM edbreport(9, 10);

edbreport
-----------------------------------------------------------------------------
    EnterpriseDB Report for database edb        23-AUG-15
 Version: EnterpriseDB 9.5.0.0 on i686-pc-linux-gnu
      Begin snapshot: 9 at 23-AUG-15 13:45:07.165123
      End snapshot:   10 at 23-AUG-15 13:45:35.653036
```



```
Size of database edb is 155 MB
     Tablespace: pg_default Size: 179 MB Owner: enterprisedb
     Tablespace: pg_global  Size: 435 kB Owner: enterprisedb

Schema: pg_toast_temp_1         Size: 0 bytes    Owner: enterprisedb
Schema: public                  Size: 0 bytes    Owner: enterprisedb
Schema: enterprisedb            Size: 143 MB     Owner: enterprisedb
Schema: pgagent                 Size: 192 kB     Owner: enterprisedb
Schema: dbms_job_procedure      Size: 0 bytes    Owner: enterprisedb
```

The information displayed in the report introduction includes the database name and version, the current date, the beginning and ending snapshot date and times, database and tablespace details and schema information.

```
               Top 10 Relations by pages

TABLE                                        RELPAGES
----------------------------------------------------------------------------
pgbench_accounts                              15874
pg_proc                                       102
edb$statio_all_indexes                        73
edb$stat_all_indexes                          73
pg_attribute                                  67
pg_depend                                     58
edb$statio_all_tables                         49
edb$stat_all_tables                           47
pgbench_tellers                               37
pg_description                                32
```

The information displayed in the `Top 10 Relations by pages` section includes:

Column Name	Description
`TABLE`	The name of the table.
`RELPAGES`	The number of pages in the table.

```
               Top 10 Indexes by pages

INDEX                                        RELPAGES
----------------------------------------------------------------------------
pgbench_accounts_pkey                         2198
pg_depend_depender_index                      32
pg_depend_reference_index                     31
pg_proc_proname_args_nsp_index                30
pg_attribute_relid_attnam_index               23
pg_attribute_relid_attnum_index               17
pg_description_o_c_o_index                    15
edb$statio_idx_pk                             11
edb$stat_idx_pk                               11
pg_proc_oid_index                             9
```

The information displayed in the `Top 10 Indexes by pages` section includes:

Column Name	Description
INDEX	The name of the index.
RELPAGES	The number of pages in the index.

```
                Top 10 Relations by DML

SCHEMA          RELATION                       UPDATES   DELETES   INSERTS
---------------------------------------------------------------------------
enterprisedb    pgbench_accounts               10400     0         1000000
enterprisedb    pgbench_tellers                10400     0         100
enterprisedb    pgbench_branches               10400     0         10
enterprisedb    pgbench_history                0         0         10400
pgagent         pga_jobclass                   0         0         6
pgagent         pga_exception                  0         0         0
pgagent         pga_job                        0         0         0
pgagent         pga_jobagent                   0         0         0
pgagent         pga_joblog                     0         0         0
pgagent         pga_jobstep                    0         0         0
```

The information displayed in the `Top 10 Relations by DML` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
UPDATES	The number of UPDATES performed on the table.
DELETES	The number of DELETES performed on the table.
INSERTS	The number of INSERTS performed on the table.

```
   DATA from pg_stat_database

 DATABASE   NUMBACKENDS  XACT COMMIT  XACT ROLLBACK   BLKS READ  BLKS HIT
BLKS ICACHE HIT      HIT RATIO  ICACHE HIT RATIO
 ----------------------------------------------------------------------------
 edb        0            142          0               78         10446
    0                 99.26      0.00

   DATA from pg_buffercache not included because pg_buffercache is not
installed
```

The information displayed in the `DATA from pg_stat_database` section of the report includes:

Column Name	Description
DATABASE	The name of the database.
NUMBACKENDS	Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset.
XACT COMMIT	Number of transactions in this database that have been committed.
XACT ROLLBACK	Number of transactions in this database that have been rolled back.
BLKS READ	Number of disk blocks read in this database.
BLKS HIT	Number of times disk blocks were found already in the buffer cache (when a read was not necessary).
BLKS ICACHE HIT	The number of blocks found in Infinite Cache.

Column Name	Description
HIT RATIO	The percentage of times that a block was found in the shared buffer cache.
ICACHE HIT RATIO	The percentage of times that a block was found in Infinite Cache.

```
   DATA from pg_stat_all_tables ordered by seq scan

 SCHEMA                    RELATION                              SEQ SCAN    REL TUP READ
IDX SCAN    IDX TUP READ INS     UPD     DEL
 ----------------------------------------------------------------------------
 pg_catalog                pg_class                              16          7162
546         319            0       1       0
 pg_catalog                pg_am                                 13          13
0           0              0       0       0
 pg_catalog                pg_database                           4           16
42          42             0       0       0
 pg_catalog                pg_index                              4           660
145         149            0       0       0
 pg_catalog                pg_namespace                          4           100
49          49             0       0       0
 sys                       edb$snap                              1           9
0           0              1       0       0
 pg_catalog                pg_authid                             1           1
25          25             0       0       0
 sys                       edb$session_wait_history              0           0
0           0              50      0       0
 sys                       edb$session_waits                     0           0
0           0              2       0       0
 sys                       edb$stat_all_indexes                  0           0
0           0              165     0       0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by seq scan` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
SEQ SCAN	The number of sequential scans initiated on this table..
REL TUP READ	The number of tuples read in the table.
IDX SCAN	The number of index scans initiated on the table.
IDX TUP READ	The number of index tuples read.
INS	The number of rows inserted.
UPD	The number of rows updated.
DEL	The number of rows deleted.

```
   DATA from pg_stat_all_tables ordered by rel tup read

 SCHEMA                    RELATION                              SEQ SCAN    REL TUP READ
IDX SCAN    IDX TUP READ INS     UPD     DEL
 ----------------------------------------------------------------------------
 pg_catalog                pg_class                              16          7162
546         319            0       1       0
 pg_catalog                pg_index                              4           660
145         149            0       0       0
```

```
 pg_catalog          pg_namespace                  4          100
49          49         0      0      0
 pg_catalog          pg_database                   4          16
42          42         0      0      0
 pg_catalog          pg_am                         13         13
0           0          0      0      0
 sys                 edb$snap                      1          9
0           0          1      0      0
 pg_catalog          pg_authid                     1          1
25          25         0      0      0
 sys                 edb$session_wait_history      0          0
0           0          50     0      0
 sys                 edb$session_waits             0          0
0           0          2      0      0
 sys                 edb$stat_all_indexes          0          0
0           0          165    0      0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by rel tup read` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`SEQ SCAN`	The number of sequential scans performed on the table.
`REL TUP READ`	The number of tuples read from the table.
`IDX SCAN`	The number of index scans performed on the table.
`IDX TUP READ`	The number of index tuples read.
`INS`	The number of rows inserted.
`UPD`	The number of rows updated.
`DEL`	The number of rows deleted.

```
   DATA from pg_statio_all_tables

 SCHEMA      RELATION             HEAP     HEAP     HEAP     IDX      IDX
                                  READ     HIT      ICACHE   READ     HIT
                                                    HIT

             IDX      TOAST    TOAST    TOAST    TIDX     TIDX    TIDX
             ICACHE   READ     HIT      ICACHE   READ     HIT     ICACHE
             HIT                        HIT                       HIT
-----------------------------------------------------------------------------
 public      pgbench_accounts     92766    67215    288       59      32126
             9        0        0        0        0        0       0
 pg_catalog  pg_class             0        296      0        3        16
             0        0        0        0        0        0       0
 sys         edb$stat_all_indexes 8        125      0        4        233
             0        0        0        0        0        0       0
 sys         edb$statio_all_index 8        125      0        4        233
             0        0        0        0        0        0       0
 sys         edb$stat_all_tables  6        91       0        2        174
             0        0        0        0        0        0       0
 sys         edb$statio_all_table 6        91       0        2        174
             0        0        0        0        0        0       0
 pg_catalog  pg_namespace         3        72       0        0        0
             0        0        0        0        0        0       0
 sys         edb$session_wait_his 1        24       0        4        47
             0        0        0        0        0        0       0
```

```
 pg_catalog  pg_opclass          3        13       0        2        0
             0        0        0        0        0        0        0
 pg_catalog  pg_trigger          0        12       0        1        15
             0        0        0        0        0        0        0
```

The information displayed in the `Data from pg_statio_all_tables` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`HEAP READ`	The number of heap blocks read.
`HEAP HIT`	The number of heap blocks hit.
`HEAP ICACHE HIT`	The number of heap blocks in Infinite Cache.
`IDX READ`	The number of index blocks read.
`IDX HIT`	The number of index blocks hit.
`IDX ICACHE HIT`	The number of index blocks in Infinite Cache.
`TOAST READ`	The number of toast blocks read.
`TOAST HIT`	The number of toast blocks hit.
`TOAST ICACHE HIT`	The number of toast blocks in Infinite Cache.
`TIDX READ`	The number of toast index blocks read.
`TIDX HIT`	The number of toast index blocks hit.
`TIDX ICACHE HIT`	The number of toast index blocks in Infinite Cache.

```
   DATA from pg_stat_all_indexes

 SCHEMA                 RELATION                   INDEX
IDX SCAN   IDX TUP READ IDX TUP FETCH
 ----------------------------------------------------------------------------
 pg_catalog             pg_attribute
pg_attribute_relid_attnum_index     427        907          907
 pg_catalog             pg_class                   pg_class_relname_nsp_index
289        62             62
 pg_catalog             pg_class                   pg_class_oid_index
257        257            257
 pg_catalog             pg_statistic
pg_statistic_relid_att_inh_index    207        196          196
 enterprisedb           pgbench_accounts           pgbench_accounts_pkey
200        255            200
 pg_catalog             pg_cast                    pg_cast_source_target_index
199        50             50
 pg_catalog             pg_proc                    pg_proc_oid_index
116        116            116
 pg_catalog             edb_partition              edb_partition_partrelid_index
112        0              0
 pg_catalog             edb_policy                 edb_policy_object_name_index
112        0              0
 enterprisedb           pgbench_branches           pgbench_branches_pkey
101        110            0
```

The information displayed in the `DATA from pg_stat_all_indexes` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the index resides.
RELATION	The name of the table on which the index is defined.
INDEX	The name of the index.
IDX SCAN	The number of indexes scans initiated on this index.
IDX TUP READ	Number of index entries returned by scans on this index
IDX TUP FETCH	Number of live table rows fetched by simple index scans using this index.

```
   DATA from pg_statio_all_indexes

 SCHEMA                    RELATION                   INDEX
IDX BLKS READ   IDX BLKS HIT    IDX BLKS ICACHE HIT
 ----------------------------------------------------------------------------
 pg_catalog                pg_attribute
pg_attribute_relid_attnum_index     0               867             0
 enterprisedb              pgbench_accounts           pgbench_accounts_pkey
1               778             0
 pg_catalog                pg_class                   pg_class_relname_nsp_index
0               590             0
 pg_catalog                pg_class                   pg_class_oid_index
0               527             0
 pg_catalog                pg_statistic
pg_statistic_relid_att_inh_index    0               441             0
 sys                       edb$stat_all_indexes       edb$stat_idx_pk
1               332             0
 sys                       edb$statio_all_indexes     edb$statio_idx_pk
1               332             0
 pg_catalog                pg_proc                    pg_proc_oid_index
0               244             0
 sys                       edb$stat_all_tables        edb$stat_tab_pk
0               241             0
 sys                       edb$statio_all_tables      edb$statio_tab_pk
0               241             0
```

The information displayed in the `DATA from pg_statio_all_indexes` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the index resides.
RELATION	The name of the table on which the index is defined.
INDEX	The name of the index.
IDX BLKS READ	The number of index blocks read.
IDX BLKS HIT	The number of index blocks hit.
IDX BLKS ICACHE HIT	The number of index blocks in Infinite Cache that were hit.

```
    System Wait Information

 WAIT NAME                                 COUNT      WAIT TIME        % WAIT
 ---------------------------------------------------------------------------
 query plan                                0          0.000407         100.00
 db file read                              0          0.000000         0.00
```

The information displayed in the `System Wait Information` section includes:

Column Name	Description
WAIT NAME	The name of the wait.
COUNT	The number of times that the wait event occurred.
WAIT TIME	The length of the wait time in milliseconds.
% WAIT	The percentage of the total wait time used by this wait for this session.

```
    Database Parameters from postgresql.conf

 PARAMETER                         SETTING
CONTEXT     MINVAL       MAXVAL
 ----------------------------------------------------------------------------
 allow_system_table_mods            off
postmaster
 application_name                   psql
user
 archive_command                    (disabled)
sighup
 archive_mode                       off
postmaster
 archive_timeout                    0
sighup      0            2147483647
 array_nulls                        on
user
 authentication_timeout             60
sighup      1            600
 autovacuum                         on
sighup
 autovacuum_analyze_scale_factor    0.1
sighup      0            100
 autovacuum_analyze_threshold       50
sighup      0            2147483647
 autovacuum_freeze_max_age          200000000
postmaster  100000000    2000000000
 autovacuum_max_workers             3
postmaster  1            8388607
 autovacuum_naptime                 60
sighup      1            2147483
 autovacuum_vacuum_cost_delay       20
...
```

The information displayed in the `Database Parameters from postgresql.conf` section includes:

Column Name	Description
PARAMETER	The name of the parameter.
SETTING	The current value assigned to the parameter.
CONTEXT	The context required to set the parameter value.
MINVAL	The minimum value allowed for the parameter.
MAXVAL	The maximum value allowed for the parameter.

8.9.2 stat_db_rpt()

The signature is:

```
stat_db_rpt(beginning_id, ending_id)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

The following example demonstrates the `stat_db_rpt()` function:

```
SELECT * FROM stat_db_rpt(9, 10);
                              stat_db_rpt
-----------------------------------------------------------------------------
   DATA from pg_stat_database

 DATABASE    NUMBACKENDS  XACT COMMIT  XACT ROLLBACK   BLKS READ  BLKS HIT
        BLKS ICACHE HIT       HIT RATIO       ICACHE HIT RATIO
-----------------------------------------------------------------------------
 edb         1            21           0               92928      101217
        301                   52.05          0.15
```

The information displayed in the `DATA from pg_stat_database` section of the report includes:

Column Name	Description
DATABASE	The name of the database.
NUMBACKENDS	Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset.
XACT COMMIT	The number of transactions in this database that have been committed.
XACT ROLLBACK	The number of transactions in this database that have been rolled back.
BLKS READ	The number of blocks read.
BLKS HIT	The number of blocks hit.
BLKS ICACHE HIT	The number of blocks in Infinite Cache that were hit.
HIT RATIO	The percentage of times that a block was found in the shared buffer cache.
ICACHE HIT RATIO	The percentage of times that a block was found in Infinite Cache.

8.9.3 stat_tables_rpt()

The signature is:

```
function_name(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should return information about all tables.

The `stat_tables_rpt()` function returns a two-part report. The first portion of the report contains:

```
SELECT * FROM stat_tables_rpt(18, 19, 10, 'ALL');

stat_tables_rpt
-------------------------------------------------------------------------------
DATA from pg_stat_all_tables ordered by seq scan

SCHEMA         RELATION
    SEQ SCAN    REL TUP READ IDX SCAN    IDX TUP READ   INS     UPD     DEL
-------------------------------------------------------------------------------
pg_catalog     pg_class
```

```
    8           2952         78         65             0      0      0
pg_catalog    pg_index
    4           448          23         28             0      0      0
pg_catalog    pg_namespace
    4           76           1          1              0      0      0
pg_catalog    pg_database
    3           6            0          0              0      0      0
pg_catalog    pg_authid
    2           1            0          0              0      0      0
sys           edb$snap
    1           15           0          0              1      0      0
public        accounts
    0           0            0          0              0      0      0
public        branches
    0           0            0          0              0      0      0
sys           edb$session_wait_history
    0           0            0          0              25     0      0
sys           edb$session_waits
    0           0            0          0              10     0      0
```

The information displayed in the `DATA from pg_stat_all_tables ordered by seq scan` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the table resides.
`RELATION`	The name of the table.
`SEQ SCAN`	The number of sequential scans on the table.
`REL TUP READ`	The number of tuples read from the table.
`IDX SCAN`	The number of index scans performed on the table.
`IDX TUP READ`	The number of index tuples read from the table.
`INS`	The number of rows inserted.
`UPD`	The number of rows updated.
`DEL`	The number of rows deleted.

The second portion of the report contains:

```
DATA from pg_stat_all_tables ordered by rel tup read

SCHEMA       RELATION
    SEQ SCAN   REL TUP READ IDX SCAN   IDX TUP READ INS    UPD    DEL
-----------------------------------------------------------------------------
pg_catalog   pg_class
    8          2952         78         65           0      0      0
pg_catalog   pg_index
    4          448          23         28           0      0      0
pg_catalog   pg_namespace
    4          76           1          1            0      0      0
sys          edb$snap
    1          15           0          0            1      0      0
pg_catalog   pg_database
    3          6            0          0            0      0      0
pg_catalog   pg_authid
    2          1            0          0            0      0      0
public       accounts
    0          0            0          0            0      0      0
```



```
public        branches
    0          0            0          0            0       0       0
sys           edb$session_wait_history
    0          0            0          0            25      0       0
sys           edb$session_waits
    0          0            0          0            10      0       0
(29 rows)
```

The information displayed in the `DATA from pg_stat_all_tables ordered by rel tup read` section includes:

Column Name	Description
SCHEMA	The name of the schema in which the table resides.
RELATION	The name of the table.
SEQ SCAN	The number of sequential scans performed on the table.
REL TUP READ	The number of tuples read from the table.
IDX SCAN	The number of index scans performed on the table.
IDX TUP READ	The number of live rows fetched by index scans.
INS	The number of rows inserted.
UPD	The number of rows updated.
DEL	The number of rows deleted.

8.9.4 statio_tables_rpt()

The signature is:

`statio_tables_rpt(`*`beginning_id, ending_id, top_n, scope`*`)`

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.

- `USER` indicates that the function should return information about user-defined tables.

- `ALL` specifies that the function should return information about all tables.

The `statio_tables_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM statio_tables_rpt(9, 10, 10, 'SYS');

                               statio_tables_rpt
-----------------------------------------------------------------------------
   DATA from pg_statio_all_tables

 SCHEMA      RELATION                HEAP     HEAP     HEAP     IDX      IDX
                                     READ     HIT      ICACHE   READ     HIT
                                                       HIT

             IDX      TOAST    TOAST    TOAST    TIDX     TIDX    TIDX
             ICACHE   READ     HIT      ICACHE   READ     HIT     ICACHE
             HIT                    HIT                         HIT
-----------------------------------------------------------------------------
 public      pgbench_accounts        92766    67215    288      59       32126
             9        0        0        0        0        0        0
 pg_catalog  pg_class                0        296      0        3        16
             0        0        0        0        0        0        0
 sys         edb$stat_all_indexes 8          125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$statio_all_index 8          125      0        4        233
             0        0        0        0        0        0        0
 sys         edb$stat_all_tables  6          91       0        2        174
             0        0        0        0        0        0        0
 sys         edb$statio_all_table 6          91       0        2        174
             0        0        0        0        0        0        0
 pg_catalog  pg_namespace            3        72       0        0        0
             0        0        0        0        0        0        0
 sys         edb$session_wait_his 1          24       0        4        47
             0        0        0        0        0        0        0
 pg_catalog  pg_opclass              3        13       0        2        0
             0        0        0        0        0        0        0
 pg_catalog  pg_trigger              0        12       0        1        15
             0        0        0        0        0        0        0
(16 rows)
```

The information displayed in the `Data from pg_statio_all_tables` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the relation.
`HEAP READ`	The number of heap blocks read.

Column Name	Description
HEAP HIT	The number of heap blocks hit.
HEAP ICACHE HIT	The number of heap blocks in Infinite Cache.
IDX READ	The number of index blocks read.
IDX HIT	The number of index blocks hit.
IDX ICACHE HIT	The number of index blocks in Infinite Cache.
TOAST READ	The number of toast blocks read.
TOAST HIT	The number of toast blocks hit.
TOAST ICACHE HIT	The number of toast blocks in Infinite Cache.
TIDX READ	The number of toast index blocks read.
TIDX HIT	The number of toast index blocks hit.
TIDX ICACHE HIT	The number of toast index blocks in Infinite Cache.

8.9.5 stat_indexes_rpt()

The signature is:

```
stat_indexes_rpt(beginning_id, ending_id, top_n, scope)
```

Parameters

beginning_id

beginning_id is an integer value that represents the beginning session identifier.

ending_id

ending_id is an integer value that represents the ending session identifier.

top_n

top_n represents the number of rows to return

scope

scope determines which tables the function returns statistics about. Specify SYS, USER or ALL:

- SYS indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: pg_catalog, information_schema, sys, or dbo.
- USER indicates that the function should return information about user-defined tables.

- `ALL` specifies that the function should return information about all tables.

The `stat_indexes_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM stat_indexes_rpt(9, 10, 10, 'ALL');

                            stat_indexes_rpt
-----------------------------------------------------------------------------
   DATA from pg_stat_all_indexes

 SCHEMA        RELATION        INDEX
                          IDX SCAN    IDX TUP READ    IDX TUP FETCH
-----------------------------------------------------------------------------
 pg_catalog    pg_cast         pg_cast_source_target_index
                          30          7               7
 pg_catalog    pg_class        pg_class_oid_index
                          15          15              15
 pg_catalog    pg_trigger      pg_trigger_tgrelid_tgname_index
                          12          12              12
 pg_catalog    pg_attribute    pg_attribute_relid_attnum_index
                          7           31              31
 pg_catalog    pg_statistic    pg_statistic_relid_att_index
                          7           0               0
 pg_catalog    pg_database     pg_database_oid_index
                          5           5               5
 pg_catalog    pg_proc         pg_proc_oid_index
                          5           5               5
 pg_catalog    pg_operator     pg_operator_oprname_l_r_n_index
                          3           1               1
 pg_catalog    pg_type         pg_type_typname_nsp_index
                          3           1               1
 pg_catalog    pg_amop         pg_amop_opr_fam_index
                          2           3               3
(14 rows)
```

The information displayed in the `DATA from pg_stat_all_indexes` section includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the relation.
`INDEX`	The name of the index.
`IDX SCAN`	The number of indexes scanned.
`IDX TUP READ`	The number of index tuples read.
`IDX TUP FETCH`	The number of index tuples fetched.

8.9.6 statio_indexes_rpt()

The signature is:

```
statio_indexes_rpt(beginning_id, ending_id, top_n, scope)
```

Parameters

`beginning_id`

`beginning_id` is an integer value that represents the beginning session identifier.

`ending_id`

`ending_id` is an integer value that represents the ending session identifier.

`top_n`

`top_n` represents the number of rows to return

`scope`

`scope` determines which tables the function returns statistics about. Specify `SYS`, `USER` or `ALL`:

- `SYS` indicates that the function should return information about system defined tables. A table is considered a system table if it is stored in one of the following schemas: `pg_catalog`, `information_schema`, `sys`, or `dbo`.
- `USER` indicates that the function should return information about user-defined tables.
- `ALL` specifies that the function should return information about all tables.

The `statio_indexes_rpt()` function returns a report that contains:

```
edb=# SELECT * FROM statio_indexes_rpt(9, 10, 10, 'SYS');

                            statio_indexes_rpt
-----------------------------------------------------------------------------
   DATA from pg_statio_all_indexes

 SCHEMA      RELATION        INDEX
                        IDX BLKS READ   IDX BLKS HIT    IDX BLKS ICACHE HIT
-----------------------------------------------------------------------------
public                  pgbench_accounts            pgbench_accounts_pkey
                        59                32126             9
 sys                     edb$stat_all_indexes       edb$stat_idx_pk
```

```
                          4                233               0
 sys                    edb$statio_all_indexes    edb$statio_idx_pk
                          4                233               0
 sys                    edb$stat_all_tables       edb$stat_tab_pk
                          2                174               0
 sys                    edb$statio_all_tables     edb$statio_tab_pk
                          2                174               0
 sys                    edb$session_wait_history  session_waits_hist_pk
                          4                47                0
 pg_catalog             pg_cast                   pg_cast_source_target_index
                          1                29                0
 pg_catalog             pg_trigger                pg_trig_tgrelid_tgname_index
                          1                15                0
 pg_catalog             pg_class                  pg_class_oid_index
                          1                14                0
 pg_catalog             pg_statistic              pg_statistic_relid_att_index
                          2                12                0
(14 rows)
```

The information displayed in the `DATA from pg_statio_all_indexes` report includes:

Column Name	Description
`SCHEMA`	The name of the schema in which the relation resides.
`RELATION`	The name of the table on which the index is defined.
`INDEX`	The name of the index.
`IDX BLKS READ`	The number of index blocks read.
`IDX BLKS HIT`	The number of index blocks hit.
`IDX BLKS ICACHE HIT`	The number of index blocks in Infinite Cache that were hit.

8.10 Performance Tuning Recommendations

To use DRITA reports for performance tuning, review the top five events in a given report, looking for any event that takes a disproportionately large percentage of resources. In a streamlined system, user I/O will probably make up the largest number of waits. Waits should be evaluated in the context of CPU usage and total time; an event may not be significant if it takes 2 minutes out of a total measurement interval of 2 hours, if the rest of the time is consumed by CPU time. The component of response time (CPU "work" time or other "wait" time) that consumes the highest percentage of overall time should be evaluated.

When evaluating events, watch for:

Event type	Description
Checkpoint waits	Checkpoint waits may indicate that checkpoint parameters need to be adjusted.
WAL-related waits	WAL-related waits may indicate `wal_buffers` are under-sized.
SQL Parse waits	If the number of waits is high, try to use prepared statements.
db file random reads	If high, check that appropriate indexes and statistics exist.
db file random writes	If high, may need to decrease `bgwriter_delay`.
btree random lock acquires	May indicate indexes are being rebuilt. Schedule index builds during less active time.

Performance reviews should also include careful scrutiny of the hardware, the operating system, the network and the application SQL statements.

8.11 Event Descriptions

Event Name	Description
add in shmem lock acquire	Obsolete/unused
bgwriter communication lock acquire	The bgwriter (background writer) process has waited for the short-term lock that synchronizes messages between the bgwriter and a backend process.
btree vacuum lock acquire	The server has waited for the short-term lock that synchronizes access to the next available vacuum cycle ID.
buffer free list lock acquire	The server has waited for the short-term lock that synchronizes access to the list of free buffers (in shared memory).
checkpoint lock acquire:	A server process has waited for the short-term lock that prevents simultaneous checkpoints.
checkpoint start lock acquire	The server has waited for the short-term lock that synchronizes access to the bgwriter checkpoint schedule.
clog control lock acquire	The server has waited for the short-term lock that synchronizes access to the commit log.
control file lock acquire	The server has waited for the short-term lock that synchronizes write access to the control file (this should usually be a low number).
db file extend	A server process has waited for the operating system while adding a new page to the end of a file.
db file read	A server process has waited for the completion of a read (from disk).
db file write	A server process has waited for the completion of a write (to disk).
db file sync	A server process has waited for the operating system to flush all changes to disk.
first buf mapping lock acquire	The server has waited for a short-term lock that synchronizes access to the shared-buffer mapping table.
freespace lock acquire	The server has waited for the short-term lock that synchronizes access to the freespace map.
Infinite Cache read	The server has waited for an Infinite Cache read request.
Infinite Cache write	The server has waited for an Infinite Cache write request.
lwlock acquire	The server has waited for a short-term lock that has not been described elsewhere in this section.
multi xact gen lock acquire	The server has waited for the short-term lock that synchronizes access to the next available multi-transaction ID (when a SELECT...FOR SHARE statement executes).
multi xact member lock acquire	The server has waited for the short-term lock that synchronizes access to the multi-transaction member file (when a SELECT...FOR SHARE statement executes).
multi xact offset lock acquire	The server has waited for the short-term lock that synchronizes access to the multi-transaction offset file (when a SELECT...FOR SHARE statement executes).
oid gen lock acquire	The server has waited for the short-term lock that synchronizes access to the next available OID (object ID).
query plan	The server has computed the execution plan for a SQL statement.
rel cache init lock acquire	The server has waited for the short-term lock that prevents simultaneous relation-cache loads/unloads.
shmem index lock acquire	The server has waited for the short-term lock that synchronizes access to the shared-memory map.
sinval lock acquire	The server has waited for the short-term lock that synchronizes access to the cache invalidation state.

`sql parse`	The server has parsed a SQL statement.
`subtrans control lock acquire`	The server has waited for the short-term lock that synchronizes access to the subtransaction log.
`tablespace create lock acquire`	The server has waited for the short-term lock that prevents simultaneous `CREATE TABLESPACE` or `DROP TABLESPACE` commands.
`two phase state lock acquire`	The server has waited for the short-term lock that synchronizes access to the list of prepared transactions.
`wal insert lock acquire`	The server has waited for the short-term lock that synchronizes write access to the write-ahead log. A high number may indicate that WAL buffers are sized too small.
`wal write lock acquire`	The server has waited for the short-term lock that synchronizes write-ahead log flushes.
`wal file sync`	The server has waited for the write-ahead log to sync to disk (related to the wal_sync_method parameter which, by default, is 'fsync' - better performance can be gained by changing this parameter to open_sync).
`wal flush`	The server has waited for the write-ahead log to flush to disk.
`wal write`	The server has waited for a write to the write-ahead log buffer (expect this value to be high).
`xid gen lock acquire`	The server has waited for the short-term lock that synchronizes access to the next available transaction ID.

8.12 Catalog Views

The following DRITA catalog views provide access to performance information relating to system waits.

8.12.1 edb$system_waits

The `edb$system_waits` view summarizes the number of waits and the total wait time per session for each wait named. It also displays the average and max wait times. The following example shows the result of a `SELECT` statement on the `edb$system_waits` view:

```
select * from sys.edb$system_waits;

 edb_id | dbname |wait_name   | wait_count |avg_wait | max_wait | totalwait
--------+--------+-----------+------------+---------+----------+----------
      1 | edb    |db fileread|        301 |0.011516 | 0.629986 | 2.742500
      1 | edb    |wal flush  |         26 |0.010364 | 0.085380 | 0.269452
      1 | edb    |wal write  |         26 |0.010355 | 0.085371 | 0.269232
      1 | edb    |query plan |        277 |0.001367 | 0.049425 | 0.192442
      2 | edb    |wal flush  |         28 |0.040443 | 0.095150 | 0.431984
      2 | edb    |wal write  |         28 |0.040434 | 0.095093 | 0.431698
      2 | edb    |query plan |        299 |0.001479 | 0.049425 | 0.262596
```

`edb$system_waits` summarizes the following information:

Column Name	Type	Description
`edb_id`	`BIGINT`	Wait identifier.
`dbname`	`NAME`	Name of the database in which the wait occurs.
`wait_name`	`TEXT`	Name of the wait event.
`wait_count`	`BIGINT`	Number of times the wait event has occurred.
`avg_wait`	`NUMERIC`	Average wait time in milliseconds.
`max_wait`	`NUMERIC(50,6)`	Maximum wait time in milliseconds.
`totalwait`	`NUMERIC(50,6)`	Total wait time in milliseconds.

8.12.2 edb$session_waits

The `edb$session_waits` view summarizes the number of waits and the total wait time per session for each wait named and identified by backend ID. It also displays the average and max wait times. The following code sample shows the result of a `SELECT` statement on the `edb$session_waits` view:

```
SELECT * FROM sys.edb$session_waits;

 edb_id | dbname | backend_id |  wait_name    | wait_count | avg_wait_time |
max_wait_time| total_wait_time |  usename      |  current_query
--------+--------+------------+---------------+------------+---------------+-
-------------+----------------+--------------+--------------------------
      1 | edb    |      22935 | db file read  |        175 |      0.008399 |
    0.629986 |       1.469887 | enterprisedb | <IDLE>
      1 | edb    |      22988 | db file read  |        116 |      0.009556 |
    0.040627 |       1.108438 | enterprisedb | select * from edbsnap();
      1 | edb    |      22988 | wal flush     |        26  |      0.010364 |
    0.085380 |       0.269452 | enterprisedb | select * from edbsnap();
(3 rows)
```

`edb$session_waits` summarizes the following information:

Column Name	Type	Description
`edb_id`	`BIGINT`	Wait identifier.
`dbname`	`NAME`	Name of the database in which the wait occurs.
`backend_id`	`BIGINT`	The backend ID of the process.
`wait_name`	`TEXT`	Name of the wait event.
`wait_count`	`BIGINT`	Number of times the wait event has occurred.
`avg_wait_time`	`NUMERIC(50,6)`	Average wait time in milliseconds.
`max_wait_time`	`NUMERIC`	Maximum wait time in milliseconds.
`total_wait_time`	`NUMERIC`	Total wait time in milliseconds.
`use_name`	`NAME`	The name of the user invoking the query.
`current_query`	`TEXT`	The query that is currently executing.

8.12.3 edb$session_wait_history

The `edb$session_wait_history` view contains the last 25 wait events for each backend ID active during the session. The following code sample shows the result of a `SELECT` statement on the `edb$session_wait_history` view:

```
SELECT * FROM sys.edb$session_wait_history;

 edb_id | dbname | backend_id | seq |   wait_name    | elapsed | p1 | p2 | p3
--------+--------+------------+-----+----------------+---------+----+----+----
      1 | edb    |      22935 |   1 | query plan     |      54 |   0 |   0 |   0
      1 | edb    |      22935 |   2 | db file read   |    1116 |2689|   8 |   1
      1 | edb    |      22935 |   3 | db file read   |     983 |1255|  32 |   1
      1 | edb    |      22935 |   4 | db file read   |   13717 |2691|  19 |   1
      1 | edb    |      22935 |   5 | query plan     |      75 |    0|   0 |   0
      1 | edb    |      22935 |   6 | db file read   |   11053 |1255|   7 |   1
      1 | edb    |      22935 |   7 | db file read   |     404 |2689|   4 |   1
 (7 rows)
```

The `edb$session_wait_history` view includes the following information:

Column Name	Type	Description
edb_id	BIGINT	Wait identifier.
dbname	TEXT	Name of the database in which the wait occurs.
backend_id	BIGINT	The session identifier of the process in which the wait occurs.
seq	BIGINT	The sequence number of the event (value 1 through 25).
wait_name	TEXT	Name of the wait event.
elapsed	BIGINT	Elapsed time in milliseconds.
p1	BIGINT	Wait specific – see table below.
p2	BIGINT	Wait specific – see table below.
p3	BIGINT	Wait specific – see table below.

The values contained in the `p1`, `p2`, and `p3` columns are wait-specific. The following waits include information in those columns:

Wait Name	p1	p2	p3
wal file sync	0 means Fsync 1 means Fdatasync 2 means open 3 means Fsync writethrough 4 means open dsync For more information, please see the documentation for WAL_SYNC_METHOD	unused	unused
Infinite Cache write	The Infinite Cache node ID that was written	The file ID from pg_class.relfilenode	The block number that was written
Infinite Cache read	The file ID from pg_class.relfilenode	The block number that was written	unused
db file extend	The file ID from pg_class.relfilenode	The block number that was extended	Skip Fsync; 1 if True, 0 if False
db file read	The file ID from	The block number that was	unused

	pg_class.relfilenode	read	
db file write	The file ID from pg_class.relfilenode	The block number that was written	unused

For all other event types, the `p1`, `p2`, and `p3` columns are unused.

9 Built-In Utility Packages

This chapter describes the built-in packages that are provided with Postgres Plus 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. See the GRANT command for granting privileges.

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;
```

Note: When executing a built-in package procedure that has no IN OUT or OUT parameters from within a PL/pgSQL function or trigger, the PERFORM statement must be used as illustrated by the following example:

```
PERFORM DBMS_ALERT.SIGNAL('dept_alert', 'Alert message');
```

This differs from the manner in which a procedure is executed from within an SPL anonymous block, procedure, function, or trigger as shown by the following example:

```
DBMS_ALERT.SIGNAL('dept_alert', 'Alert message');
```

Within an SPL program, the package-qualified procedure name is specified without the PERFORM statement.

When executing a built-in package procedure that has a single IN OUT or OUT parameter from within a PL/pgSQL function or trigger, a variable with a data type compatible with the IN OUT or OUT parameter must be assigned the result of the evaluated function as illustrated by the following example:

```
DECLARE
    v_item          VARCHAR(100);
BEGIN
        .
        .
        .
    v_item := DBMS_PIPE.UNPACK_MESSAGE(v_item);
```

When executing a built-in package procedure that has more than one IN OUT or OUT parameters from within a PL/pgSQL function or trigger, a variable of type RECORD must be assigned the result of the evaluated function. The returned values of the individual IN OUT and OUT parameters can then be referenced from the individual fields of the record in the form, *record_name.parameter_name* where *record_name* is the RECORD type variable name and *parameter_name* is the name of an IN OUT or OUT parameter declared in the built-in package procedure parameter declaration as illustrated by the following example:

```
DECLARE
    v_name            VARCHAR2(30);
    v_msg             VARCHAR2(80);
    v_status          INTEGER;
    v_timeout         NUMBER(3) := 120;
    v_waitany         RECORD;
BEGIN
        .
        .
        .
    v_waitany := DBMS_ALERT.WAITANY(v_name,v_msg,v_status,v_timeout);
    RAISE INFO 'Alert name   : %', v_waitany.name;
    RAISE INFO 'Alert msg    : %', v_waitany.message;
    RAISE INFO 'Alert status : %', v_waitany.status;
```

See Section 9.1.5 for the parameter declarations of the `DBMS_ALERT.WAITANY` procedure.

9.1 DBMS_ALERT

The DBMS_ALERT package provides the capability to register for, send, and receive alerts.

Table 7-9-1 DBMS_ALERT Functions/Procedures

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

Advanced Server 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/9.4AS/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.

9.1.1 REGISTER

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

REGISTER(*name* VARCHAR2)

Parameters

name

Name of the alert to be registered.

Examples

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

```
DECLARE
    v_name          VARCHAR2(30) := 'alert_test';
    v_msg           VARCHAR2(80);
    v_status        INTEGER;
    v_timeout       NUMBER(3) := 120;
BEGIN
    DBMS_ALERT.REGISTER(v_name);
    DBMS_OUTPUT.PUT_LINE('Registered for alert ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
    DBMS_ALERT.WAITONE(v_name,v_msg,v_status,v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.REMOVE(v_name);
END;

Registered for alert alert_test
Waiting for signal...
```

9.1.2 REMOVE

The REMOVE procedure unregisters the session for the named alert.

REMOVE(*name* VARCHAR2)

Parameters

name

Name of the alert to be unregistered.

9.1.3 REMOVEALL

The `REMOVEALL` procedure unregisters the session for all alerts.

```
REMOVEALL
```

9.1.4 SIGNAL

The `SIGNAL` procedure signals the occurrence of the named alert.

```
SIGNAL(name VARCHAR2, message VARCHAR2)
```

Parameters

name

Name of the alert.

message

Information to pass with this alert.

Examples

The following anonymous block signals an alert for `alert_test`.

```
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
```

9.1.5 WAITANY

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

```
WAITANY(name OUT VARCHAR2, message OUT VARCHAR2,
  status OUT INTEGER, timeout NUMBER)
```

Parameters

name

Variable receiving the name of the alert.

message

Variable receiving the message sent by the `SIGNAL` procedure.

status

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

timeout

Time to wait for an alert in seconds.

Examples

The following anonymous block uses the `WAITANY` procedure to receive an alert named, `alert_test` or `any_alert`:

```
DECLARE
    v_name          VARCHAR2(30);
    v_msg           VARCHAR2(80);
    v_status        INTEGER;
    v_timeout       NUMBER(3) := 120;
BEGIN
    DBMS_ALERT.REGISTER('alert_test');
    DBMS_ALERT.REGISTER('any_alert');
    DBMS_OUTPUT.PUT_LINE('Registered for alert alert_test and any_alert');
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
    DBMS_ALERT.WAITANY(v_name,v_msg,v_status,v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.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
```

9.1.6 WAITONE

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

```
WAITONE(name VARCHAR2, message OUT VARCHAR2,
  status OUT INTEGER, timeout NUMBER)
```

Parameters

name

Name of the alert.

message

Variable receiving the message sent by the `SIGNAL` procedure.

status

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

timeout

Time to wait for an alert in seconds.

Examples

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

```
DECLARE
    v_name          VARCHAR2(30) := 'alert_test';
    v_msg           VARCHAR2(80);
    v_status        INTEGER;
    v_timeout       NUMBER(3) := 120;
BEGIN
    DBMS_ALERT.REGISTER(v_name);
    DBMS_OUTPUT.PUT_LINE('Registered for alert ' || v_name);
```

```
    DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
    DBMS_ALERT.WAITONE(v_name,v_msg,v_status,v_timeout);
    DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
    DBMS_OUTPUT.PUT_LINE('Alert msg    : ' || v_msg);
    DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
    DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
    DBMS_ALERT.REMOVE(v_name);
END;

Registered for alert alert_test
Waiting for signal...
```

Signal sent for `alert_test` sent by an anonymous block in a second session:

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

Issued alert for alert_test
```

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

```
Registered for alert alert_test
Waiting for signal...
Alert name   : alert_test
Alert msg    : This is the message from alert_test
Alert status : 0
Alert timeout: 120 seconds
```

9.1.7 Comprehensive Example

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

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

```
CREATE OR REPLACE FUNCTION dept_alert_trig() RETURNS TRIGGER
AS $$
DECLARE
    v_action        VARCHAR(25);
BEGIN
    IF TG_OP = 'INSERT' THEN
        v_action := ' added department(s) ';
    ELSIF TG_OP = 'UPDATE' THEN
        v_action := ' updated department(s) ';
    ELSIF TG_OP = 'DELETE' THEN
        v_action := ' deleted department(s) ';
    END IF;
    PERFORM DBMS_ALERT.SIGNAL('dept_alert',USER || v_action || 'on ' ||
```



```
        TO_CHAR(CURRENT_TIMESTAMP, 'DD-MON-YY HH24:MI:SS'));
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER dept_alert_trig
    AFTER INSERT OR UPDATE OR DELETE ON dept
    FOR EACH STATEMENT EXECUTE PROCEDURE dept_alert_trig();

CREATE OR REPLACE FUNCTION emp_alert_trig() RETURNS TRIGGER
AS $$
DECLARE
    v_action        VARCHAR(25);
BEGIN
    IF TG_OP = 'INSERT' THEN
        v_action := ' added employee(s) ';
    ELSIF TG_OP = 'UPDATE' THEN
        v_action := ' updated employee(s) ';
    ELSIF TG_OP = 'DELETE' THEN
        v_action := ' deleted employee(s) ';
    END IF;
    PERFORM DBMS_ALERT.SIGNAL('emp_alert',USER || v_action || 'on ' ||
         TO_CHAR(CURRENT_TIMESTAMP, 'DD-MON-YY HH24:MI:SS'));
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER emp_alert_trig
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH STATEMENT EXECUTE PROCEDURE emp_alert_trig();
```

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...
```


Note: In the following sessions with users mary and john, PSQL is executed in `AUTOCOMMIT off` mode. This affects the number of alerts displayed by multiple SQL statements against the same table by the same user. If the PSQL default `AUTCOMMIT on` mode were used instead, two alerts for user mary on the `emp` table would be displayed instead of one since there would be two `INSERT` statements in two separate transactions.

```
\set AUTOCOMMIT off
```

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);
COMMIT;
```

The following change is made by user, john:

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

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 05-FEB-14 14:45:16
Alert status : 0
-------------------------------------------------------------
Alert name   : emp_alert
Alert msg    : mary added employee(s) on 05-FEB-14 14:45:16
Alert status : 0
-------------------------------------------------------------
Alert name   : dept_alert
Alert msg    : john added department(s) on 05-FEB-14 14:45:31
Alert status : 0
-------------------------------------------------------------
Alert status : 1

EDB-SPL Procedure successfully completed
```

9.2 DBMS_CRYPTO

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

Table 7.7.2 DBMS_CRYPTO Functions and Procedures

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

DBMS_CRYPTO functions and procedures support the following error messages:

```
ORA-28239 - DBMS_CRYPTO.KeyNull
ORA-28829 - DBMS_CRYPTO.CipherSuiteNull
ORA-28827 - DBMS_CRYPTO.CipherSuiteInvalid
```

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.

9.2.1 DECRYPT

The `DECRYPT` function or procedure decrypts data using a user-specified cipher algorithm, key and optional initialization vector. The signature of the `DECRYPT` function is:

```
DECRYPT
  (src IN RAW, typ IN INTEGER, key IN RAW, iv IN RAW
   DEFAULT NULL) RETURN RAW
```

The signature of the `DECRYPT` procedure is:

```
DECRYPT
  (dst INOUT BLOB, src IN BLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

or

```
DECRYPT
  (dst INOUT CLOB, src IN CLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

When invoked as a procedure, `DECRYPT` returns `BLOB` or `CLOB` data to a user-specified `BLOB`.

Parameters

dst

> *dst* specifies the name of a `BLOB` to which the output of the `DECRYPT` procedure will be written. The `DECRYPT` procedure will overwrite any existing data currently in *dst*.

src

> *src* specifies the source data that will be decrypted. If you are invoking `DECRYPT` as a function, specify `RAW` data; if invoking `DECRYPT` as a procedure, specify `BLOB` or `CLOB` data.

typ

> *typ* specifies the block cipher type and any modifiers. This should match the type specified when the *src* was encrypted. Advanced Server supports the following block cipher algorithms, modifiers and cipher suites:

Block Cipher Algorithms	
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       VARCHAR(90) PRIMARY KEY,    -- username
    ciphertext      RAW(9)                      -- encrypted password
);

CREATE OR REPLACE FUNCTION get_password (
    username        VARCHAR2
) RETURNS RAW
AS $$
DECLARE
    typ             INTEGER := 4353;            -- 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;
```



```
$$ LANGUAGE 'plpgsql';
```

Note that when calling `DECRYPT`, you must pass the same cipher type, key value and initialization vector that was used when `ENCRYPTING` the target.

9.2.2 ENCRYPT

The `ENCRYPT` function or procedure uses a user-specified algorithm, key, and optional initialization vector to encrypt `RAW`, `BLOB` or `CLOB` data. The signature of the `ENCRYPT` function is:

```
ENCRYPT
  (src IN RAW, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL) RETURN RAW
```

The signature of the `ENCRYPT` procedure is:

```
ENCRYPT
  (dst INOUT BLOB, src IN BLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

or

```
ENCRYPT
  (dst INOUT BLOB, src IN CLOB, typ IN INTEGER, key IN RAW,
   iv IN RAW DEFAULT NULL)
```

When invoked as a procedure, `ENCRYPT` returns `BLOB` or `CLOB` data to a user-specified `BLOB`.

Parameters

dst

> *dst* specifies the name of a `BLOB` to which the output of the `ENCRYPT` procedure will be written. The `ENCRYPT` procedure will overwrite any existing data currently in *dst*.

src

> *src* specifies the source data that will be encrypted. If you are invoking `ENCRYPT` as a function, specify `RAW` data; if invoking `ENCRYPT` as a procedure, specify `BLOB` or `CLOB` data.

typ

typ specifies the block cipher type that will be used by `ENCRYPT`, and any modifiers. Advanced Server supports the block cipher algorithms, modifiers and cipher suites listed below:

Block Cipher Algorithms	
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       VARCHAR(90) PRIMARY KEY,    -- username
    ciphertext      RAW(9)                      -- encrypted password
);

CREATE OR REPLACE FUNCTION set_password (
    username        VARCHAR,
    cleartext       RAW
) RETURNS VOID
AS $$
DECLARE
    typ             INTEGER := 4353;            -- 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;
    RETURN;
END;
$$ LANGUAGE 'plpgsql';
```

`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`.

9.2.3 HASH

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

```
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;
```

9.2.4 MAC

The `MAC` function uses a user-specified `MAC` function to return the hashed `MAC` value of a `RAW` or `CLOB` value. The `MAC` function is available in three forms:

```
MAC
  (src IN RAW, typ IN INTEGER, key IN RAW) RETURN RAW

MAC
  (src IN CLOB, typ IN INTEGER, key IN RAW) RETURN RAW
```

Parameters

src

> *src* specifies the value for which the `MAC` value will be generated. Specify a `RAW`, `BLOB`, or `CLOB` value.

typ

> *typ* specifies the `MAC` function used. Advanced Server supports the `MAC` functions listed below.

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.

9.2.5 RANDOMBYTES

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

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

Parameters

number_bytes

number_bytes specifies the number of random bytes to be returned

Examples

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

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

9.2.6 RANDOMINTEGER

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

```
RANDOMINTEGER() RETURNS INTEGER
```

Examples

The following example uses the RANDOMINTEGER function to return a cryptographically strong random INTEGER value:

```
DECLARE
    result          INTEGER;
BEGIN
    result := DBMS_CRYPTO.RANDOMINTEGER();
    DBMS_OUTPUT.PUT_LINE(result);
```

```
END;
```

9.2.7 RANDOMNUMBER

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

```
RANDOMNUMBER() RETURNS NUMBER
```

Examples

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

```
DECLARE
    result          NUMBER;
BEGIN
    result := DBMS_CRYPTO.RANDOMNUMBER();
    DBMS_OUTPUT.PUT_LINE(result);
END;
```

9.3 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 Postgres Plus Advanced Server installer installs `pgAgent`, but you must start the `pgAgent` service manually prior to using `DBMS_JOB`. See the readme file, `README-pgagent.txt`, located in the *`POSTGRES_PLUS_HOME`*`/doc` directory for information on starting `pgAgent`. 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.

Table 9-2 DBMS_JOB Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
`BROKEN(`*`job`*`, `*`broken`*` [, `*`next_date`*` ])`	Procedure	n/a	Specify that a given job is either broken or not broken.
`CHANGE(`*`job`*`, `*`what`*`, `*`next_date`*`, `*`interval`*`, `*`instance`*`, `*`force`*`)`	Procedure	n/a	Change the job's parameters.
`INTERVAL(`*`job`*`, `*`interval`*`)`	Procedure	n/a	Set the execution frequency by means of a date function that is recalculated each time the job is run. This value becomes the next date/time for execution.
`NEXT_DATE(`*`job`*`, `*`next_date`*`)`	Procedure	n/a	Set the next date/time the job is to be run.
`REMOVE(`*`job`*`)`	Procedure	n/a	Delete the job definition from the database.
`RUN(`*`job`*`)`	Procedure	n/a	Forces execution of a job even if it is marked broken.
`SUBMIT(`*`job`*` OUT, `*`what`*` [, `*`next_date`*` [, `*`interval`*` [, `*`no_parse`*` ]]])`	Procedure	n/a	Creates a job and stores its definition in the database.
`WHAT(`*`job`*`, `*`what`*`)`	Procedure	n/a	Change the stored procedure run by a job.

When and how often a job is run is dependent upon two interacting parameters – *`next_date`* and *`interval`*. The *`next_date`* parameter is a date/time value that specifies the next date/time when the job is to be executed. The *`interval`* parameter is a string that contains a date function that evaluates to a date/time value.

Just prior to any execution of the job, the expression in the *`interval`* parameter is evaluated. The resulting value replaces the *`next_date`* value stored with the job. The job is then executed. In this manner, the expression in *`interval`* is repeatedly re-evaluated prior to each job execution, supplying the *next_date* date/time for the next execution.

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;
```

9.3.1 BROKEN

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

BROKEN(*job* BINARY_INTEGER, *broken* BOOLEAN [, *next_date* DATE])

Parameters

job

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

broken

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

next_date

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

Examples

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

```
BEGIN
    DBMS_JOB.BROKEN(104,true);
END;
```

Change the state back to not broken:


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

9.3.2 CHANGE

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

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

Parameters

job

Identifier of the job to modify.

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;
```

9.3.3 INTERVAL

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

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

Parameters

`job`

Identifier of the job to modify.

`interval`

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

Examples

Change the job to run once a week:

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

9.3.4 NEXT_DATE

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

`NEXT_DATE(`*`job`*` BINARY_INTEGER, `*`next_date`*` DATE)`

Parameters

`job`

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

`next_date`

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

Examples

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

```
BEGIN
    DBMS_JOB.NEXT_DATE(104, TO_DATE('14-DEC-07','DD-MON-YY'));
END;
```

9.3.5 REMOVE

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

`REMOVE(job BINARY_INTEGER)`

Parameters

job

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

Examples

Remove a job from the database:

```
BEGIN
    DBMS_JOB.REMOVE(104);
END;
```

9.3.6 RUN

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

`RUN(job BINARY_INTEGER)`

Parameters

job

Identifier of the job to be run.

Examples

Force a job to be run.

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

9.3.7 SUBMIT

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

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

Parameters

job

Identifier assigned to the job.

what

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

next_date

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

interval

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

no_parse

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)
```

9.3.8 WHAT

The `WHAT` procedure changes the stored procedure that the job will execute.

`WHAT(`*`job`*` BINARY_INTEGER, `*`what`*` VARCHAR2)`

Parameters

`job`

Identifier of the job for which the stored procedure is to be changed.

`what`

Name of the stored procedure to be executed.

Examples

Change the job to run the `list_emp` procedure:

```
BEGIN
    DBMS_JOB.WHAT(104,'list_emp;');
END;
```

9.4 DBMS_LOB

The DBMS_LOB package provides the capability to operate on large objects.

Table 9-3 DBMS_LOB Functions/Procedures

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

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

Table 9-4 DBMS_LOB Public Variables

Public Variables	Data Type	Value
compress off	INTEGER	0
compress_on	INTEGER	1
deduplicate_off	INTEGER	0
deduplicate_on	INTEGER	4

Public Variables	Data Type	Value
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 `BLOB`s. Lengths and offsets are measured in characters if the large objects are `CLOB`s.

9.4.1 APPEND

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

```
APPEND(dest_lob IN OUT { BLOB | CLOB }, src_lob { BLOB | CLOB })
```

Parameters

dest_lob

Large object locator for the destination object. Must be the same data type as *src_lob*.

src_lob

Large object locator for the source object. Must be the same data type as *dest_lob*.

9.4.2 COMPARE

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

```
status INTEGER COMPARE(lob_1 { BLOB | CLOB },
  lob_2 { BLOB | CLOB }
```

```
  [, amount INTEGER [, offset_1 INTEGER [, offset_2 INTEGER ]]])
```

Parameters

lob_1

Large object locator of the first large object to be compared. Must be the same data type as `lob_2`.

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.

9.4.3 CONVERTTOBLOB

The `CONVERTTOBLOB` procedure provides the capability to convert character data to binary.

```
CONVERTTOBLOB(dest_lob IN OUT BLOB, src_clob CLOB,
  amount INTEGER, dest_offset IN OUT INTEGER,
  src_offset IN OUT INTEGER, blob_csid NUMBER,
```

```
  lang_context IN OUT INTEGER, warning OUT INTEGER)
```

Parameters

dest_lob

> BLOB large object locator to which the character data is to be converted.

src_clob

> CLOB large object locator of the character data to be converted.

amount

> Number of characters of *src_clob* to be converted.

dest_offset IN

> Position in bytes in the destination BLOB where writing of the source CLOB should begin. The first byte is offset 1.

dest_offset OUT

> Position in bytes in the destination BLOB after the write operation completes. The first byte is offset 1.

src_offset IN

> Position in characters in the source CLOB where conversion to the destination BLOB should begin. The first character is offset 1.

src_offset OUT

> Position in characters in the source CLOB after the conversion operation completes. The first character is offset 1.

blob_csid

> Character set ID of the converted, destination BLOB.

lang_context IN

> Language context for the conversion. The default value of 0 is typically used for this setting.

lang_context OUT

Language context after the conversion completes.

warning

0 if the conversion was successful, 1 if an inconvertible character was encountered.

9.4.4 CONVERTTOCLOB

The `CONVERTTOCLOB` procedure provides the capability to convert binary data to character.

```
CONVERTTOCLOB(dest_lob IN OUT CLOB, src_blob BLOB,
  amount INTEGER, dest_offset IN OUT INTEGER,
  src_offset IN OUT INTEGER, blob_csid NUMBER,
  lang_context IN OUT INTEGER, warning OUT INTEGER)
```

Parameters

dest_lob

`CLOB` large object locator to which the binary data is to be converted.

src_blob

`BLOB` large object locator of the binary data to be converted.

amount

Number of bytes of *src_blob* to be converted.

dest_offset IN

Position in characters in the destination `CLOB` where writing of the source `BLOB` should begin. The first character is offset 1.

dest_offset OUT

Position in characters in the destination `CLOB` after the write operation completes. The first character is offset 1.

src_offset IN

Position in bytes in the source BLOB where conversion to the destination CLOB should begin. The first byte is offset 1.

src_offset OUT

Position in bytes in the source BLOB after the conversion operation completes. The first byte is offset 1.

blob_csid

Character set ID of the converted, destination CLOB.

lang_context IN

Language context for the conversion. The default value of 0 is typically used for this setting.

lang_context OUT

Language context after the conversion completes.

warning

0 if the conversion was successful, 1 if an inconvertible character was encountered.

9.4.5 COPY

The COPY procedure provides the capability to copy one large object to another. The source and destination large objects must be the same data type.

```
COPY(dest_lob IN OUT { BLOB | CLOB }, src_lob { BLOB | CLOB },
  amount INTEGER
  [, dest_offset INTEGER [, src_offset INTEGER ]])
```

Parameters

dest_lob

Large object locator of the large object to which *src_lob* is to be copied. Must be the same data type as *src_lob*.

src_lob

Large object locator of the large object to be copied to *dest_lob*. Must be the same data type as *dest_lob*.

amount

Number of bytes/characters of *src_lob* to be copied.

dest_offset

Position in the destination large object where writing of the source large object should begin. The first position is offset 1. The default is 1.

src_offset

Position in the source large object where copying to the destination large object should begin. The first position is offset 1. The default is 1.

9.4.6 ERASE

The ERASE procedure provides the capability to erase a portion of a large object. To erase a large object means to replace the specified portion with zero-byte fillers for BLOBs or with spaces for CLOBs. The actual size of the large object is not altered.

```
ERASE(lob_loc IN OUT { BLOB | CLOB }, amount IN OUT INTEGER
  [, offset INTEGER ])
```

Parameters

lob_loc

Large object locator of the large object to be erased.

amount IN

Number of bytes/characters to be erased.

amount OUT

Number of bytes/characters actually erased. This value can be smaller than the input value if the end of the large object is reached before *amount* bytes/characters have been erased.

offset

Position in the large object where erasing is to begin. The first byte/character is position 1. The default is 1.

9.4.7 GET_STORAGE_LIMIT

The `GET_STORAGE_LIMIT` function returns the limit on the largest allowable large object.

```
size INTEGER GET_STORAGE_LIMIT(lob_loc BLOB)
```

```
size INTEGER GET_STORAGE_LIMIT(lob_loc CLOB)
```

Parameters

size

Maximum allowable size of a large object in this database.

lob_loc

This parameter is ignored, but is included for compatibility.

9.4.8 GETLENGTH

The `GETLENGTH` function returns the length of a large object.

```
amount INTEGER GETLENGTH(lob_loc BLOB)
```

```
amount INTEGER GETLENGTH(lob_loc CLOB)
```

Parameters

lob_loc

Large object locator of the large object whose length is to be obtained.

amount

Length of the large object in bytes for `BLOB`s or characters for `CLOB`s.

9.4.9 INSTR

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

```
position INTEGER INSTR(lob_loc { BLOB | CLOB },
  pattern { RAW | VARCHAR2 } [, offset INTEGER [, nth INTEGER ]])
```

Parameters

lob_loc

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

pattern

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

offset

Position within *lob_loc* to start search for *pattern*. The first byte/character is position 1. The default is 1.

nth

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*.

9.4.10 READ

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

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

Parameters

lob_loc

Large object locator of the large object to be read.

amount IN

Number of bytes/characters to read.

amount OUT

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

offset

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

buffer

Variable to receive the large object. If *lob_loc* is a BLOB, then *buffer* must be RAW. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR2.

9.4.11 SUBSTR

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

```
data { RAW | VARCHAR2 } SUBSTR(lob_loc { BLOB | CLOB }
  [, amount INTEGER [, offset INTEGER ]])
```

Parameters

lob_loc

Large object locator of the large object to be read.

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.

9.4.12 TRIM

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

```
TRIM(lob_loc IN OUT { BLOB | CLOB }, newlen INTEGER)
```

Parameters

lob_loc

Large object locator of the large object to be trimmed.

newlen

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

9.4.13 WRITE

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

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

Parameters

lob_loc

Large object locator of the large object to be written.

amount

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

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.

9.4.14 WRITEAPPEND

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

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

Parameters

lob_loc

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

amount

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

buffer

Data to be appended to the large object. If *lob_loc* is a BLOB, then *buffer* must be RAW. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR2.

9.5 DBMS_LOCK

Advanced Server provides support for the `DBMS_LOCK.SLEEP` procedure.

Table 7.7.2 DBMS_LOCK Procedure

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

9.5.1 SLEEP

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

```
SLEEP(seconds NUMBER)
```

Parameters

seconds

> *seconds* specifies the number of seconds for which you wish to suspend the session. *seconds* can be a fractional value; for example, enter `1.75` to specify one and three-fourths of a second.

9.6 DBMS_MVIEW

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

Table 7.7.2 DBMS_MVIEW Procedures

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

9.6.1 GET_MV_DEPENDENCIES

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

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

Parameters

list

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

deplist

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

Examples

The following example:

```
DECLARE
  deplist VARCHAR2(1000);
BEGIN
  DBMS_MVIEW.GET_MV_DEPENDENCIES('public.emp_view', deplist);
  DBMS_OUTPUT.PUT_LINE('deplist: ' || deplist);
END;
```

Displays a list of the dependencies on a materialized view named `public.emp_view`.

9.6.2 REFRESH

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

```
REFRESH(
  list IN VARCHAR2,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  push_deferred_rpc IN BOOLEAN DEFAULT TRUE,
```

```
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  purge_option IN NUMBER DEFAULT 1,
  parallelism IN NUMBER DEFAULT 0,
  heap_size IN NUMBER DEFAULT 0,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

Use the second form to specify view names in a table of `DBMS_UTILITY.UNCL_ARRAY` values:

```
REFRESH(
  tab IN OUT DBMS_UTILITY.UNCL_ARRAY,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  push_deferred_rpc IN BOOLEAN DEFAULT TRUE,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  purge_option IN NUMBER DEFAULT 1,
  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');
```

9.6.3 REFRESH_ALL_MVIEWS

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

```
REFRESH_ALL_MVIEWS(
  number_of_failures OUT BINARY_INTEGER,
  method IN VARCHAR2 DEFAULT NULL,
```

```
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  atomic_refresh IN BOOLEAN DEFAULT TRUE);
```

Parameters

number_of_failures

number_of_failures is a `BINARY_INTEGER` that specifies the number of failures that occurred during the refresh operation.

method

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

rollback_seg

rollback_seg is accepted for compatibility and ignored. The default is `NULL`.

refresh_after_errors

refresh_after_errors is accepted for compatibility and ignored. The default is `FALSE`.

atomic_refresh

atomic_refresh is accepted for compatibility and ignored. The default is `TRUE`.

Examples

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

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

Upon completion, `errors` contains the number of failures.

9.6.4 REFRESH_DEPENDENT

Use the `REFRESH_DEPENDENT` procedure to refresh all material views that are dependent on the views specified in the call to the procedure. You can specify a comma-separated list or provide the view names in a table of `DBMS_UTILITY.UNCL_ARRAY` values.

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

```
REFRESH_DEPENDENT(
  number_of_failures OUT BINARY_INTEGER,
  list IN VARCHAR2,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

Use the second form of the procedure to refresh all material views that are dependent on the views specified in a table of `DBMS_UTILITY.UNCL_ARRAY` values:

```
REFRESH_DEPENDENT(
  number_of_failures OUT BINARY_INTEGER,
  tab IN DBMS_UTILITY.UNCL_ARRAY,
  method IN VARCHAR2 DEFAULT NULL,
  rollback_seg IN VARCHAR2 DEFAULT NULL,
  refresh_after_errors IN BOOLEAN DEFAULT FALSE,
  atomic_refresh IN BOOLEAN DEFAULT TRUE,
  nested IN BOOLEAN DEFAULT FALSE);
```

Parameters

number_of_failures

> *number_of_failures* is a `BINARY_INTEGER` that contains the number of failures that occurred during the refresh operation.

list

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

tab

> *tab* is a table of `DBMS_UTILITY.UNCL_ARRAY` values that specify the name (or names) of a materialized view.

method

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

rollback_seg

rollback_seg is accepted for compatibility and ignored. The default is NULL.

refresh_after_errors

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.

9.7 DBMS_OUTPUT

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

The procedures and functions available in the DBMS_OUTPUT package are listed in the following table.

Table 7-9-5 DBMS_OUTPUT Functions/Procedures

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

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

Table 7-9-6 DBMS_OUTPUT Public Variables

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

9.7.1 CHARARR

The CHARARR is for storing multiple message lines.

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


9.7.2 DISABLE

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

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

```
DISABLE
```

Examples

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

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

9.7.3 ENABLE

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

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

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

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

Parameters

buffer_size

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

Examples

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

```
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;
```

9.7.4 GET_LINE

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

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)
```

9.7.5 GET_LINES

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

`GET_LINES(`*`lines`*` OUT CHARARR, `*`numlines`*` IN OUT INTEGER)`

Parameters

`lines`

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

`numlines`` IN`

> Number of lines to be retrieved from the message buffer.

`numlines`` OUT`

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

Examples

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

```
EXEC DBMS_OUTPUT.SERVEROUTPUT(FALSE);

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

```
        DBMS_OUTPUT.PUT_LINE(v_emprec);
    END LOOP;
END;

DECLARE
    v_lines         DBMS_OUTPUT.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)
```

9.7.6 NEW_LINE

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

```
NEW_LINE
```

Parameters

The `NEW_LINE` procedure expects no parameters.

9.7.7 PUT

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

`PUT(item VARCHAR2)`

Parameters

item

Text written to the message buffer.

Examples

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

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

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
```


9.7.8 PUT_LINE

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

```
PUT_LINE(item VARCHAR2)
```

Parameters

item

Text to be written to the message buffer.

Examples

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

```
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
```

9.7.9 SERVEROUTPUT

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

In PSQL, `SERVEROUTPUT(TRUE)` is the default setting.

`SERVEROUTPUT(`*`stdout`*` BOOLEAN)`

Parameters

`stdout`

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

Examples

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

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

This message goes to the command line
```

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

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

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

9.8 DBMS_PIPE

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

The procedures and functions available in the DBMS_PIPE package are listed in the following table.

Table 7-9-7 DBMS_PIPE Functions/Procedures

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

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

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

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

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

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

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

9.8.1 CREATE_PIPE

The `CREATE_PIPE` function creates an explicit public pipe or an explicit private pipe with a specified name.

```
status INTEGER CREATE_PIPE(pipename VARCHAR2
  [, maxpipesize INTEGER ] [, private BOOLEAN ])
```

Parameters

pipename

Name of the pipe.

maxpipesize

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

private

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

status

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

Examples

The following example creates a private pipe named `messages`:

```
DECLARE
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.CREATE_PIPE('messages');
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END;
CREATE_PIPE status: 0
```

The following example creates a public pipe named `mailbox`:

```
DECLARE
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.CREATE_PIPE('mailbox',8192,FALSE);
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END;
CREATE_PIPE status: 0
```

9.8.2 NEXT_ITEM_TYPE

The `NEXT_ITEM_TYPE` function returns an integer code identifying the data type of the next data item in a message that has been retrieved into the session's local message buffer. As each item is moved off of the local message buffer with the `UNPACK_MESSAGE` procedure, the `NEXT_ITEM_TYPE` function will return the data type code for the next available item. A code of 0 is returned when there are no more items left in the message.

`typecode` `INTEGER NEXT_ITEM_TYPE`

Parameters

`typecode`

Code identifying the data type of the next data item as shown in Table 7-9-8.

Table 7-9-8 NEXT_ITEM_TYPE Data Type Codes

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

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
```

9.8.3 PACK_MESSAGE

The `PACK_MESSAGE` procedure places an item of data in the session's local message buffer. `PACK_MESSAGE` must be executed at least once before issuing a `SEND_MESSAGE` call.

```
PACK_MESSAGE(item { DATE | NUMBER | VARCHAR2 | RAW })
```

Use the `UNPACK_MESSAGE` procedure to obtain data items once the message is retrieved using a `RECEIVE_MESSAGE` call.

Parameters

item

> An expression evaluating to any of the acceptable parameter data types. The value is added to the session's local message buffer.

9.8.4 PURGE

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

PURGE(*pipename* VARCHAR2)

Use the REMOVE_PIPE function to delete an explicit pipe.

Parameters

pipename

Name of the pipe.

Examples

Two messages are sent on a pipe:

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

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

SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

Receive the first message and unpack it:

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

9.8.5 RECEIVE_MESSAGE

The RECEIVE_MESSAGE function obtains a message from a specified pipe.

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

Parameters

pipename

Name of the pipe.

timeout

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

status

Status code returned by the operation.

The possible status codes are:

Table 7-9-9 RECEIVE_MESSAGE Status Codes

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

9.8.6 REMOVE_PIPE

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

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

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

Parameters

pipename

Name of the pipe.

status

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

Examples

Two messages are sent on a pipe:

```
DECLARE
    v_status        INTEGER;
BEGIN
    v_status := DBMS_PIPE.CREATE_PIPE('pipe');
    DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status : ' || v_status);

    DBMS_PIPE.PACK_MESSAGE('Message #1');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);

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

CREATE_PIPE status : 0
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

Receive the first message and unpack it:

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

RECEIVE_MESSAGE status: 0
Item: Message #1
```

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
```

9.8.7 RESET_BUFFER

The RESET_BUFFER procedure resets a "pointer" to the session's local message buffer back to the beginning of the buffer. This has the effect of causing subsequent PACK_MESSAGE calls to overwrite any data items that existed in the message buffer prior to the RESET_BUFFER call.

```
RESET_BUFFER
```

Examples

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

```
DECLARE
    v_status        INTEGER;
BEGIN
    DBMS_PIPE.PACK_MESSAGE('Hi, John');
    DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?');
    DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?');
    DBMS_PIPE.RESET_BUFFER;
    DBMS_PIPE.PACK_MESSAGE('Hi, Bob');
    DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 9:30, tomorrow?');
    v_status := DBMS_PIPE.SEND_MESSAGE('pipe');
    DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END;

SEND_MESSAGE status: 0
```

The message to Bob is in the received message.

```
DECLARE
```

```
    v_item          VARCHAR2(80);
    v_status        INTEGER;
BEGIN
    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?
```

9.8.8 SEND_MESSAGE

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

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

Parameters

pipename

Name of the pipe.

timeout

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

maxpipesize

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

status

Status code returned by the operation.

The possible status codes are:

Table 7-9-10 SEND_MESSAGE Status Codes

Status Code	Description
0	Success
1	Time out
3	Function interrupted

9.8.9 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
```

9.8.10 UNPACK_MESSAGE

The `UNPACK_MESSAGE` procedure copies the data items of a message from the local message buffer to a specified program variable. The message must be placed in the local message buffer with the `RECEIVE_MESSAGE` function before using `UNPACK_MESSAGE`.

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

Parameters

item

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

9.8.11 Comprehensive Example

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

```
CREATE OR REPLACE PACKAGE mailbox
IS
    PROCEDURE create_mailbox;
    PROCEDURE add_message (
        p_mailbox   VARCHAR2,
        p_item_1    VARCHAR2,
        p_item_2    VARCHAR2 DEFAULT 'END',
        p_item_3    VARCHAR2 DEFAULT 'END'
    );
    PROCEDURE empty_mailbox (
        p_mailbox   VARCHAR2,
        p_waittime  INTEGER DEFAULT 10
    );
END mailbox;

CREATE OR REPLACE PACKAGE BODY mailbox
IS
    PROCEDURE create_mailbox
    IS
        v_mailbox   VARCHAR2(30);
        v_status    INTEGER;
    BEGIN
        v_mailbox := DBMS_PIPE.UNIQUE_SESSION_NAME;
        v_status := DBMS_PIPE.CREATE_PIPE(v_mailbox,1000,FALSE);
        IF v_status = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Created mailbox: ' || v_mailbox);
        ELSE
            DBMS_OUTPUT.PUT_LINE('CREATE_PIPE failed - status: ' ||
                v_status);
        END IF;
    END create_mailbox;

    PROCEDURE add_message (
        p_mailbox   VARCHAR2,
        p_item_1    VARCHAR2,
        p_item_2    VARCHAR2 DEFAULT 'END',
        p_item_3    VARCHAR2 DEFAULT 'END'
    )
    IS
        v_item_cnt  INTEGER := 0;
        v_status    INTEGER;
    BEGIN
        DBMS_PIPE.PACK_MESSAGE(p_item_1);
        v_item_cnt := 1;
        IF p_item_2 != 'END' THEN
            DBMS_PIPE.PACK_MESSAGE(p_item_2);
            v_item_cnt := v_item_cnt + 1;
        END IF;
        IF p_item_3 != 'END' THEN
            DBMS_PIPE.PACK_MESSAGE(p_item_3);
            v_item_cnt := v_item_cnt + 1;
        END IF;
        v_status := DBMS_PIPE.SEND_MESSAGE(p_mailbox);
        IF v_status = 0 THEN
```

```
            DBMS_OUTPUT.PUT_LINE('Added message with ' || v_item_cnt ||
                ' item(s) to mailbox ' || p_mailbox);
        ELSE
            DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE in add_message failed - ' ||
                'status: ' || v_status);
        END IF;
    END add_message;

    PROCEDURE empty_mailbox (
        p_mailbox   VARCHAR2,
        p_waittime  INTEGER DEFAULT 10
    )
    IS
        v_msgno     INTEGER DEFAULT 0;
        v_itemno    INTEGER DEFAULT 0;
        v_item      VARCHAR2(100);
        v_status    INTEGER;
    BEGIN
        v_status := DBMS_PIPE.RECEIVE_MESSAGE(p_mailbox,p_waittime);
        WHILE v_status = 0 LOOP
            v_msgno := v_msgno + 1;
            DBMS_OUTPUT.PUT_LINE('****** Start message #' || v_msgno ||
                ' ******');
            BEGIN
                LOOP
                    v_status := DBMS_PIPE.NEXT_ITEM_TYPE;
                    EXIT WHEN v_status = 0;
                    DBMS_PIPE.UNPACK_MESSAGE(v_item);
                    v_itemno := v_itemno + 1;
                    DBMS_OUTPUT.PUT_LINE('Item #' || v_itemno || ': ' ||
                        v_item);
                END LOOP;
                DBMS_OUTPUT.PUT_LINE('******* End message #' || v_msgno ||
                    ' *******');
                DBMS_OUTPUT.PUT_LINE('*');
                v_itemno := 0;
                v_status := DBMS_PIPE.RECEIVE_MESSAGE(p_mailbox,1);
            END;
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('Number of messages received: ' || v_msgno);
        v_status := DBMS_PIPE.REMOVE_PIPE(p_mailbox);
        IF v_status = 0 THEN
            DBMS_OUTPUT.PUT_LINE('Deleted mailbox ' || p_mailbox);
        ELSE
            DBMS_OUTPUT.PUT_LINE('Could not delete mailbox - status: '
                || v_status);
        END IF;
    END empty_mailbox;
END mailbox;
```

The following demonstrates the execution of the procedures in `mailbox`. The first procedure creates a public pipe using a name generated by the `UNIQUE_SESSION_NAME` function.

```
EXEC mailbox.create_mailbox;

Created mailbox: PG$PIPE$13$3940
```


Using the mailbox name, any user in the same database with access to the `mailbox` package and `DBMS_PIPE` package can add messages:

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

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

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

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

Finally, the contents of the mailbox can be emptied:

```
EXEC mailbox.empty_mailbox('PG$PIPE$13$3940');

****** Start message #1 ******
Item #1: Hi, John
Item #2: Can you attend a meeting at 3:00, today?
Item #3: -- Mary
******* End message #1 *******
*
****** Start message #2 ******
Item #1: Don't forget to submit your report
Item #2: Thanks,
Item #3: Joe
******* End message #2 *******
*
Number of messages received: 2
Deleted mailbox PG$PIPE$13$3940
```

9.9 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.

For more information about the DBMS_PROFILER built-in package (including usage examples and a reference guide to the DBMS_PROFILER tables and views), see Section 9.9.

Table 9-11 DBMS_PROFILER Functions/Procedures

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

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

Table 9-12 DBMS_PROFILER Status Codes and Exceptions

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.

9.9.1 FLUSH_DATA

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

```
status INTEGER FLUSH_DATA
```

```
FLUSH_DATA
```

Parameters

status

Status code returned by the operation.

9.9.2 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`.

9.9.3 INTERNAL_VERSION_CHECK

The `INTERNAL_VERSION_CHECK` function confirms that the current version of `DBMS_PROFILER` will work with the current database. The function signature is:

```
status INTEGER INTERNAL_VERSION_CHECK
```

Parameters

status

Status code returned by the operation.

9.9.4 PAUSE_PROFILER

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

status `INTEGER PAUSE_PROFILER`

`PAUSE_PROFILER`

Parameters

status

Status code returned by the operation.

9.9.5 RESUME_PROFILER

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

status `INTEGER RESUME_PROFILER`

`RESUME_PROFILER`

Parameters

status

Status code returned by the operation.

9.9.6 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.

9.9.7 STOP_PROFILER

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

```
status INTEGER STOP_PROFILER

STOP_PROFILER
```

Parameters

status

Status code returned by the operation.

9.10 DBMS_RANDOM

The DBMS_RANDOM package provides a number of methods to generate random values. The procedures and functions available in the DBMS_RANDOM package are listed in the following table.

Table 7. DBMS_RANDOM Functions/Procedures

Function/Procedure	Return Type	Description
INITIALIZE(*val*)	n/a	Initializes the DBMS_RANDOM package with the specified seed *value*. Deprecated, but supported for backward compatibility.
NORMAL()	NUMBER	Returns a random NUMBER.
RANDOM	INTEGER	Returns a random INTEGER with a value greater than or equal to -2^31 and less than 2^31. Deprecated, but supported for backward compatibility.
SEED(*val*)	n/a	Resets the seed with the specified value.
SEED(*val*)	n/a	Resets the seed with the specified value.
STRING(*opt*, *len*)	VARCHAR2	Returns a random string.
TERMINATE	n/a	TERMINATE has no effect. Deprecated, but supported for backward compatibility.
VALUE	NUMBER	Returns a random number with a value greater than or equal to 0 and less than 1, with 38 digit precision.
VALUE(*low*, *high*)	NUMBER	Returns a random number with a value greater than or equal to *low* and less than *high*.

9.10.1 INITIALIZE

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

```
INITIALIZE(val IN INTEGER)
```

This procedure should be considered deprecated; it is included for backward compatibility only.

Parameters

val

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

Example

The following code snippet demonstrates a call to the `INITIALIZE` procedure that initializes the `DBMS_RANDOM` package with the seed value, `6475`.

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

9.10.2 NORMAL

The `NORMAL` function returns a random number of type `NUMBER`. The signature is:

`result` `NUMBER NORMAL()`

Parameters

`result`

`result` is a random value of type `NUMBER`.

Example

The following code snippet demonstrates a call to the `NORMAL` function:

```
x:= DBMS_RANDOM.NORMAL();
```

9.10.3 RANDOM

The `RANDOM` function returns a random `INTEGER` value that is greater than or equal to -2 ^31 and less than 2 ^31. The signature is:

`result` `INTEGER RANDOM()`

This function should be considered deprecated; it is included for backward compatibility only.

Parameters

`result`

`result` is a random value of type `INTEGER`.

Example

The following code snippet demonstrates a call to the RANDOM function. The call returns a random number:

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

9.10.4 SEED

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

SEED(*val* IN INTEGER)

Parameters

val

> *val* is the seed value used by the DBMS_RANDOM package algorithm.

Example

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

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

9.10.5 SEED

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

SEED(*val* IN VARCHAR2)

Parameters

val

> *val* is the seed value used by the DBMS_RANDOM package algorithm.

Example

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

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

9.10.6 STRING

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

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

Parameters

opt

Formatting option for the returned string. *option* may be:

Option	Specifies Formatting Option
`u` or `U`	Uppercase alpha string
`l` or `L`	Lowercase alpha string
`a` or `A`	Mixed case string
`x` or `X`	Uppercase alpha-numeric string
`p` or `P`	Any printable characters

len

The length of the returned string.

result

result is a random value of type `VARCHAR2`.

Example

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

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


9.10.7 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.

9.10.8 VALUE

The `VALUE` function returns a random `NUMBER` that is greater than or equal to 0, and less than 1, with 38 digit precision. The `VALUE` function has two forms; the signature of the first form is:

```
result NUMBER VALUE()
```

Parameters

`result`

`result` is a random value of type `NUMBER`.

Example

The following code snippet demonstrates a call to the `VALUE` function. The call returns a random `NUMBER`:

```
x := DBMS_RANDOM.VALUE();
```

9.10.9 VALUE

The `VALUE` function returns a random `NUMBER` with a value that is between user-specified boundaries. The `VALUE` function has two forms; the signature of the second form is:

```
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);
```

9.11 DBMS_RLS

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

Table 9-13 DBMS_RLS Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
ADD_POLICY(*object_schema*, *object_name*, *policy_name*, *function_schema*, *policy_function* [, *statement_types* [, *update_check* [, *enable* [, *static_policy* [, *policy_type* [, *long_predicate* [, *sec_relevant_cols* [, *sec_relevant_cols_opt*]]]]]]]])	Procedure	n/a	Add a security policy to a database object.
DROP_POLICY(*object_schema*, *object_name*, *policy_name*)	Procedure	n/a	Remove a security policy from a database object.
ENABLE_POLICY(*object_schema*, *object_name*, *policy_name*, *enable*)	Procedure	n/a	Enable or disable a security policy.

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 and PL/pgSQL in addition to SPL.

Note: The database objects currently supported by Advanced Server Virtual Private Database are tables. Policies cannot be applied to views or synonyms.

The advantages of using Virtual Private Database are the following:

- Provides a fine-grained level of security. Database object level privileges given by the `GRANT` command determine access privileges to the entire instance of a database object, while Virtual Private Database provides access control for the individual rows of a database object instance.
- A different security policy can be applied depending upon the type of SQL command (`INSERT`, `UPDATE`, `DELETE`, or `SELECT`).

- The security policy can vary dynamically for each applicable SQL command affecting the database object depending upon factors such as the session user of the application accessing the database object.
- Invocation of the security policy is transparent to all applications that access the database object and thus, individual applications do not have to be modified to apply the security policy.
- Once a security policy is enabled, it is not possible for any application (including new applications) to circumvent the security policy except by the system privilege noted by the following.
- Even superusers cannot circumvent the security policy except by the system privilege noted by the following.

Note: The only way security policies can be circumvented is if the `EXEMPT ACCESS POLICY` system privilege has been granted to a user. The `EXEMPT ACCESS POLICY` privilege should be granted with extreme care as a user with this privilege is exempted from all policies in the database.

The `DBMS_RLS` package provides procedures to create policies, remove policies, enable policies, and disable policies.

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 `ALL_POLICIES` (see Section 10.11), `DBA_POLICIES` (see Section 10.35), or `USER_POLICIES` (see Section 10.62).

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 Postgres Plus 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;
```

9.11.1 ADD_POLICY

The ADD_POLICY procedure creates a new policy by associating a policy function with a database object.

You must be a superuser to execute this procedure.

```
ADD_POLICY(object_schema VARCHAR2, object_name VARCHAR2,
  policy_name VARCHAR2, function_schema VARCHAR2,
  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

The intended purpose of this parameter is 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: 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`

Its intended purpose is to determine 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`

Its intended purpose is to allow 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`

Its intended purpose is when *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. 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. Postgres Plus 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)
```

9.11.2 DROP_POLICY

The DROP_POLICY procedure deletes an existing policy. The policy function and database object associated with the policy are not deleted by the DROP_POLICY procedure.

You must be a superuser to execute this procedure.

```
DROP_POLICY(object_schema VARCHAR2, object_name VARCHAR2,
  policy_name VARCHAR2)
```

Parameters

object_schema

Name of the schema containing the database object to which the policy applies.

object_name

Name of the database object to which the policy applies.

policy_name

Name of the policy to be deleted.

Examples

The following example deletes policy secure_update on table public.vpemp:

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_update';
BEGIN
    DBMS_RLS.DROP_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name
    );
END;
```

9.11.3 ENABLE_POLICY

The ENABLE_POLICY procedure enables or disables an existing policy on the specified database object.

You must be a superuser to execute this procedure.

```
ENABLE_POLICY(object_schema VARCHAR2, object_name VARCHAR2,
  policy_name VARCHAR2, enable BOOLEAN)
```

Parameters

object_schema

Name of the schema containing the database object to which the policy applies.

object_name

Name of the database object to which the policy applies.

policy_name

Name of the policy to be enabled or disabled.

enable

When set to `TRUE`, the policy is enabled. When set to `FALSE`, the policy is disabled.

Examples

The following example disables policy `secure_update` on table `public.vpemp`:

```
DECLARE
    v_object_schema         VARCHAR2(30) := 'public';
    v_object_name           VARCHAR2(30) := 'vpemp';
    v_policy_name           VARCHAR2(30) := 'secure_update';
    v_enable                BOOLEAN := FALSE;
BEGIN
    DBMS_RLS.ENABLE_POLICY(
        v_object_schema,
        v_object_name,
        v_policy_name,
        v_enable
    );
END;
```


9.12 DBMS_SCHEDULER

The DBMS_SCHEDULER package provides a way to create and manage jobs, programs and job schedules.

Table 7.7.2 DBMS_SCHEDULER Functions and Procedures

Function/Procedure	Return Type	Description
CREATE_JOB(*job_name, job_type, job_action, number_of_arguments, start_date, repeat_interval, end_date, job_class, enabled, auto_drop, comments*)	n/a	Use the first form of the CREATE_JOB procedure to create a job, specifying program and schedule details by means of parameters.
CREATE_JOB(*job_name, program_name, schedule_name, job_class, enabled, auto_drop, comments*)	n/a	Use the second form of CREATE_JOB to create a job that uses a named program and named schedule.
CREATE_PROGRAM(*program_name, program_type, program_action, number_of_arguments, enabled, comments*)	n/a	Use CREATE_PROGRAM to create a program.
CREATE_SCHEDULE(*schedule_name, start_date, repeat_interval, end_date, comments*)	n/a	Use the CREATE_SCHEDULE procedure to create a schedule.
DEFINE_PROGRAM_ARGUMENT(*program_name, argument_position, argument_name, argument_type, default_value, out_argument*)	n/a	Use the first form of the DEFINE_PROGRAM_ARGUMENT procedure to define a program argument that has a default value.
DEFINE_PROGRAM_ARGUMENT(*program_name, argument_position, argument_name, argument_type, out_argument*)	n/a	Use the first form of the DEFINE_PROGRAM_ARGUMENT procedure to define a program argument that does not have a default value.
DISABLE(*name, force, commit_semantics*)	n/a	Use the DISABLE procedure to disable a job or program.
DROP_JOB(*job_name, force, defer, commit_semantics*)	n/a	Use the DROP_JOB procedure to drop a job.
DROP_PROGRAM(*program_name, force*)	n/a	Use the DROP_PROGRAM procedure to drop a program.
DROP_PROGRAM_ARGUMENT(*program_name, argument_position*)	n/a	Use the first form of DROP_PROGRAM_ARGUMENT to drop a program argument by specifying the argument position.
DROP_PROGRAM_ARGUMENT(*program_name, argument_name*)	n/a	Use the second form of DROP_PROGRAM_ARGUMENT to drop a program argument by specifying the argument name.
DROP_SCHEDULE(*schedule_name, force*)	n/a	Use the DROP SCHEDULE procedure to drop a schedule.
ENABLE(*name, commit_semantics*)	n/a	Use the ENABLE command to enable a program or job.

Function/Procedure	Return Type	Description
`EVALUATE_CALENDAR_STRING(` *`calendar_string`*`,` *`start_date`*`,` *`return_date_after`*`,` *`next_run_date`*`)`	`n/a`	Use `EVALUATE_CALENDAR_STRING` to review the execution date described by a user-defined calendar schedule.
`RUN_JOB(`*`job_name`*`,` *`use_current_session`*`,` *`manually`*`)`	`n/a`	Use the `RUN_JOB` procedure to execute a job immediately.
`SET_JOB_ARGUMENT_VALUE(` *`job_name`*`,` *`argument_position`*`,` *`argument_value`*`)`	`n/a`	Use the first form of `SET_JOB_ARGUMENT` value to set the value of a job argument described by the argument's position.
`SET_JOB_ARGUMENT_VALUE(` *`job_name`*`,` *`argument_name`*`,` *`argument_value`*`)`	`n/a`	Use the second form of `SET_JOB_ARGUMENT` value to set the value of a job argument described by the argument's name.

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.

9.12.1 Using Calendar Syntax to Specify a Repeating Interval

The `CREATE_JOB` and `CREATE_SCHEDULE` procedures use a 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:

<table>
<tr><th>Token type</th><th>Syntax</th><th>Valid Values</th></tr>
<tr><td>FREQ</td><td>FREQ=predefined_interval</td><td>Where predefined_interval is one of the following: YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY. The SECONDLY keyword is not supported.</td></tr>
<tr><td>BYMONTH</td><td>BYMONTH=month(, month)...</td><td>Where month is the three-letter abbreviation of the month name: JAN | FEB | MAR | APR | MAY | JUN | JUL | AUG | SEP | OCT | NOV | DEC</td></tr>
<tr><td>BYMONTH</td><td>BYMONTH=month(, month)...</td><td>Where month is the numeric value representing the month: 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12</td></tr>
<tr><td>BYMONTHDAY</td><td>BYMONTHDAY=day_of_month</td><td>Where day_of_month is a value from 1 through 31</td></tr>
<tr><td>BYDAY</td><td>BYDAY=weekday</td><td>Where weekday is a three-letter abbreviation or single-digit value representing the day of the week.
<table>
<tr><td>Monday</td><td>MON</td><td>1</td></tr>
<tr><td>Tuesday</td><td>TUE</td><td>2</td></tr>
<tr><td>Wednesday</td><td>WED</td><td>3</td></tr>
<tr><td>Thursday</td><td>THU</td><td>4</td></tr>
<tr><td>Friday</td><td>FRI</td><td>5</td></tr>
<tr><td>Saturday</td><td>SAT</td><td>6</td></tr>
<tr><td>Sunday</td><td>SUN</td><td>7</td></tr>
</table></td></tr>
<tr><td>BYDATE</td><td>BYDATE=date(, date)...</td><td>Where date is YYYYMMDD.
YYYY is a four-digit year representation of the year,
MM is a two-digit representation of the month,
and DD is a two-digit day representation of the day.</td></tr>
<tr><td>BYDATE</td><td>BYDATE=date(, date)...</td><td>Where date is MMDD.
MM is a two-digit representation of the month,
and DD is a two-digit day representation of the day</td></tr>
<tr><td>BYHOUR</td><td>BYHOUR=hour</td><td>Where hour is a value from 0 through 23.</td></tr>
<tr><td>BYMINUTE</td><td>BYMINUTE=minute</td><td>Where minute is a value from 0 through 59.</td></tr>
</table>

9.12.2 CREATE_JOB

Use the `CREATE_JOB` procedure to create a job. The procedure comes in two forms; the first form of the procedure specifies a schedule within the job definition, as well as a job action that will be invoked when the job executes:

```
CREATE_JOB(
  job_name IN VARCHAR2,
  job_type IN VARCHAR2,
  job_action IN VARCHAR2,
  number_of_arguments IN PLS_INTEGER DEFAULT 0,
  start_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
  repeat_interval IN VARCHAR2 DEFAULT NULL,
  end_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL,
  job_class IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
  enabled IN BOOLEAN DEFAULT FALSE,
  auto_drop IN BOOLEAN DEFAULT TRUE,
  comments IN VARCHAR2 DEFAULT NULL)
```

The second form uses a job schedule to specify the schedule on which the job will execute, and specifies the name of a program that will execute when the job runs:

```
CREATE_JOB(
  job_name IN VARCHAR2,
  program_name IN VARCHAR2,
  schedule_name IN VARCHAR2,
  job_class IN VARCHAR2 DEFAULT 'DEFAULT_JOB_CLASS',
  enabled IN BOOLEAN DEFAULT FALSE,
  auto_drop IN BOOLEAN DEFAULT TRUE,
  comments IN VARCHAR2 DEFAULT NULL)
```

Parameters

job_name

> *job_name* specifies the optionally schema-qualified name of the job being created.

job_type

> *job_type* specifies the type of job. The current implementation of `CREATE_JOB` supports a job type of `PLSQL_BLOCK` or `STORED_PROCEDURE`.

job_action

> If *job_type* is `PLSQL_BLOCK`, *job_action* specifies the content of the PL/SQL block that will be invoked when the job executes. The block must be terminated with a semi-colon (`;`).

If *job_type* is STORED_PROCEDURE, *job_action* specifies the optionally schema-qualified name of the procedure.

number_of_arguments

number_of_arguments is an INTEGER value that specifies the number of arguments expected by the job. The default is 0.

start_date

start_date is a TIMESTAMP WITH TIME ZONE value that specifies the first time that the job is scheduled to execute. The default value is NULL, indicating that the job should be scheduled to execute when the job is enabled.

repeat_interval

repeat_interval is a VARCHAR2 value that specifies how often the job will repeat. If a *repeat_interval* is not specified, the job will execute only once. The default value is NULL.

For information about defining a repeating schedule for a job, see Section 9.12.1.

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.

9.12.3 CREATE_PROGRAM

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

```
CREATE_PROGRAM(
  program_name IN VARCHAR2,
  program_type IN VARCHAR2,
  program_action IN VARCHAR2,
  number_of_arguments IN PLS_INTEGER DEFAULT 0,
  enabled IN BOOLEAN DEFAULT FALSE,
  comments IN VARCHAR2 DEFAULT NULL)
```

Parameters

program_name

program_name specifies the name of the program that is being created.

program_type

program_type specifies the type of program. The current implementation of CREATE_PROGRAM supports a *program_type* of PLSQL_BLOCK or PROCEDURE.

program_action

If *program_type* is PLSQL_BLOCK, *program_action* contains the PL/SQL block that will execute when the program is invoked. The PL/SQL block must be terminated with a semi-colon (;).

If *program_type* is PROCEDURE, *program_action* contains the name of the stored procedure.

number_of_arguments

If *program_type* is PLSQL_BLOCK, this argument is ignored.

If *program_type* is PROCEDURE, *number_of_arguments* specifies the number of arguments required by the procedure. The default value is 0.

enabled

enabled specifies if the program is created enabled or disabled:

- If *enabled* is TRUE, the program is created enabled.
- If *enabled* is FALSE, the program is created disabled; use the DBMS_SCHEDULER.ENABLE program to enable a disabled program.

The default value is FALSE.

comments

Use the *comments* parameter to specify a comment about the program; by default, this parameter is NULL.

Example

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.

9.12.4 CREATE_SCHEDULE

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

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

Parameters

schedule_name

schedule_name specifies the name of the schedule.

start_date

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

repeat_interval

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

For information about defining a repeating schedule for a job, see Section 9.12.1.

Please note: you must provide a value for either *start_date* or *repeat_interval*; if both *start_date* and *repeat_interval* are NULL, the server will return an error.

end_date IN TIMESTAMP WITH TIME ZONE DEFAULT NULL

end_date is a TIMESTAMP WITH TIME ZONE value that specifies a time after which the schedule will no longer execute. If a date is specified, the *end_date* must be after the *start_date*. The default value is NULL.

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

comments IN VARCHAR2 DEFAULT NULL)

Use the *comments* parameter to specify a comment about the schedule; by default, this parameter is NULL.

Example

The following code fragment calls CREATE_SCHEDULE to create a schedule named weeknights_at_5:

```
EXEC
  DBMS_SCHEDULER.CREATE_SCHEDULE (
    schedule_name    => 'weeknights_at_5',
    start_date       => '01-JUN-13 09:00:00.000000'
    repeat_interval  => 'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',
    comments         => 'This schedule executes each weeknight at 5:00');
```

The schedule executes each weeknight, at 5:00 pm, effective after June 1, 2013. Since no end_date is specified, the schedule will execute indefinitely until it is disabled with DBMS_SCHEDULER.DISABLE.

9.12.5 DEFINE_PROGRAM_ARGUMENT

Use the DEFINE_PROGRAM_ARGUMENT procedure to define a program argument. The DEFINE_PROGRAM_ARGUMENT procedure comes in two forms; the first form defines an argument with a default value:

```
DEFINE_PROGRAM_ARGUMENT(
  program_name IN VARCHAR2,
  argument_position IN PLS_INTEGER,
  argument_name IN VARCHAR2 DEFAULT NULL,
  argument_type IN VARCHAR2,
```



```
  default_value IN VARCHAR2,
  out_argument IN BOOLEAN DEFAULT FALSE)
```

The second form defines an argument without a default value:

```
DEFINE_PROGRAM_ARGUMENT(
  program_name IN VARCHAR2,
  argument_position IN PLS_INTEGER,
  argument_name IN VARCHAR2 DEFAULT NULL,
  argument_type IN VARCHAR2,
  out_argument IN BOOLEAN DEFAULT FALSE)
```

Parameters

program_name

> *program_name* is the name of the program to which the arguments belong.

argument_position

> *argument_position* specifies the position of the argument as it is passed to the program.

argument_name

> *argument_name* specifies the optional name of the argument. By default, *argument_name* is NULL.

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.

9.12.6 DISABLE

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

```
DISABLE(
  name IN VARCHAR2,
  force IN BOOLEAN DEFAULT FALSE,
  commit_semantics IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```

Parameters

name

name specifies the name of the program or job that is being disabled.

force

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');
```

9.12.7 DROP_JOB

Use the DROP_JOB procedure to DROP a job, DROP any arguments that belong to the job, and eliminate any future job executions. The signature of the procedure is:

```
DROP_JOB(
  job_name IN VARCHAR2,
  force IN BOOLEAN DEFAULT FALSE,
  defer IN BOOLEAN DEFAULT FALSE,
  commit_semantics IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```

Parameters

job_name

job_name specifies the name of the job that is being dropped.

force

force is accepted for compatibility, and ignored.

defer

defer is accepted for compatibility, and ignored.

commit_semantics

commit_semantics instructs the server how to handle an error encountered while dropping a program or job. By default, *commit_semantics* is set to STOP_ON_FIRST_ERROR, instructing the server to stop when it encounters an error.

The TRANSACTIONAL and ABSORB_ERRORS keywords are accepted for compatibility, and ignored.

Example

The following call to DROP_JOB drops a job named update_log:

```
DBMS_SCHEDULER.DROP_JOB('update_log');
```

9.12.8 DROP_PROGRAM

The DROP_PROGRAM procedure

The signature of the DROP_PROGRAM procedure is:

```
DROP_PROGRAM(
  program_name IN VARCHAR2,
  force IN BOOLEAN DEFAULT FALSE)
```

Parameters

program_name

program_name specifies the name of the program that is being dropped.

force

force is a BOOLEAN value that instructs the server how to handle programs with dependent jobs.

Specify FALSE to instruct the server to return an error if the program is referenced by a job.

Specify TRUE to instruct the server to disable any jobs that reference the program before dropping the program.

The default value is FALSE.

Example

The following call to DROP_PROGRAM drops a job named update_emp:

```
DBMS_SCHEDULER.DROP_PROGRAM('update_emp');
```

9.12.9 DROP_PROGRAM_ARGUMENT

Use the DROP_PROGRAM_ARGUMENT procedure to drop a program argument. The DROP_PROGRAM_ARGUMENT procedure comes in two forms; the first form uses an argument position to specify which argument to drop:

```
DROP_PROGRAM_ARGUMENT(
  program_name IN VARCHAR2,
  argument_position IN PLS_INTEGER)
```

The second form takes the argument name:

```
DROP_PROGRAM_ARGUMENT(
  program_name IN VARCHAR2,
  argument_name IN VARCHAR2)
```

Parameters

program_name

program_name specifies the name of the program that is being modified.

argument_position

argument_position specifies the position of the argument that is being dropped.

argument_name

argument_name specifies the name of the argument that is being dropped.

Examples

The following call to DROP_PROGRAM_ARGUMENT drops the first argument in the update_emp program:

```
DBMS_SCHEDULER.DROP_PROGRAM_ARGUMENT('update_emp', 1);
```

The following call to DROP_PROGRAM_ARGUMENT drops an argument named emp_name:

```
DBMS_SCHEDULER.DROP_PROGRAM_ARGUMENT(update_emp', 'emp_name');
```


9.12.10 DROP_SCHEDULE

Use the DROP_SCHEDULE procedure to drop a schedule. The signature is:

```
DROP_SCHEDULE(
  schedule_name IN VARCHAR2,
  force IN BOOLEAN DEFAULT FALSE)
```

Parameters

schedule_name

schedule_name specifies the name of the schedule that is being dropped.

force

force specifies the behavior of the server if the specified schedule is referenced by any job:

- Specify FALSE to instruct the server to return an error if the specified schedule is referenced by a job. This is the default behavior.
- Specify TRUE to instruct the server to disable to any jobs that use the specified schedule before dropping the schedule. Any running jobs will be allowed to complete before the schedule is dropped.

Example

The following call to DROP_SCHEDULE drops a schedule named weeknights_at_5:

```
DBMS_SCHEDULER.DROP_SCHEDULE('weeknights_at_5', TRUE);
```

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

9.12.11 ENABLE

Use the ENABLE procedure to enable a disabled program or job.

The signature of the ENABLE procedure is:

```
ENABLE(
  name IN VARCHAR2,
  commit_semantics IN VARCHAR2 DEFAULT 'STOP_ON_FIRST_ERROR')
```

Parameters

name

> *name* specifies the name of the program or job that is being enabled.

commit_semantics

> *commit_semantics* instructs the server how to handle an error encountered while enabling a program or job. By default, *commit_semantics* is set to STOP_ON_FIRST_ERROR, instructing the server to stop when it encounters an error.
>
> The TRANSACTIONAL and ABSORB_ERRORS keywords are accepted for compatibility, and ignored.

Example

The following call to DBMS_SCHEDULER.ENABLE enables the update_emp program:

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

9.12.12 EVALUATE_CALENDAR_STRING

Use the EVALUATE_CALENDAR_STRING procedure to evaluate the *repeat_interval* value specified when creating a schedule with the CREATE_SCHEDULE procedure. The EVALUATE_CALENDAR_STRING procedure will return the date and time that a specified schedule will execute without actually scheduling the job.

The signature of the EVALUATE_CALENDAR_STRING procedure is:

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

Parameters

calendar_string

> *calendar_string* is the calendar string that describes a *repeat_interval* (see Section 9.12.1) that is being evaluated.

start_date IN TIMESTAMP WITH TIME ZONE

> *start_date* is the date and time after which the *repeat_interval* will become valid.

return_date_after

> Use the *return_date_after* parameter to specify the date and time that EVALUATE_CALENDAR_STRING should use as a starting date when evaluating the *repeat_interval*.
>
> For example, if you specify a *return_date_after* value of 01-APR-13 09.00.00.000000, EVALUATE_CALENDAR_STRING will return the date and time of the first iteration of the schedule after April 1st, 2013.

next_run_date OUT TIMESTAMP WITH TIME ZONE

> *next_run_date* is an OUT parameter that will contain the first occurrence of the schedule after the date specified by the *return_date_after* parameter.

Example

The following example evaluates a calendar string and returns the first date and time that the schedule will be executed after June 15, 2013:

```
DECLARE
  result     TIMESTAMP;
BEGIN

  DBMS_SCHEDULER.EVALUATE_CALENDAR_STRING
  (
    'FREQ=DAILY;BYDAY=MON,TUE,WED,THU,FRI;BYHOUR=17;',
    '15-JUN-2013', NULL, result
  );

    DBMS_OUTPUT.PUT_LINE('next_run_date: ' || result);
END;
/

next_run_date: 17-JUN-13 05.00.00.000000 PM
```

June 15, 2013 is a Saturday; the schedule will not execute until Monday, June 17, 2013 at 5:00 pm.

9.12.13 RUN_JOB

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

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

Parameters

job_name

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.

9.12.14 SET_JOB_ARGUMENT_VALUE

Use the SET_JOB_ARGUMENT_VALUE procedure to specify a value for an argument. The SET_JOB_ARGUMENT_VALUE procedure comes in two forms; the first form specifies which argument should be modified by position:

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

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

```
SET_JOB_ARGUMENT_VALUE(
  job_name IN VARCHAR2,
```



```
  argument_name IN VARCHAR2,
  argument_value IN VARCHAR2)
```

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

Parameters

job_name

job_name specifies the name of the job to which the modified argument belongs.

argument_position

Use *argument_position* to specify the argument position for which the value will be set.

argument_name

Use *argument_name* to specify the argument by name for which the value will be set.

argument_value

argument_value specifies the new value of the argument.

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');
```

9.13 DBMS_SESSION

Advanced Server provides support for the DBMS_SESSION.SET_ROLE procedure.

Table 7.7.2 DBMS_SESSION Procedure

Function/Procedure	Return Type	Description
SET_ROLE(*role_cmd*)	n/a	Executes a *SET ROLE* statement followed by the string value specified in *role_cmd*.

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

9.13.1 SET_ROLE

The SET_ROLE procedure sets the current session user to the role specified in *role_cmd*. After invoking the SET_ROLE procedure, the current session will use the permissions assigned to the specified role. The signature of the procedure is:

```
SET_ROLE(role_cmd)
```

The SET_ROLE procedure appends the value specified for *role_cmd* to the SET ROLE statement, and then invokes the statement.

Parameters

role_cmd

role_cmd specifies a role name in the form of a string value.

Example

The following call to the SET_ROLE procedure invokes the SET ROLE command to set the identity of the current session user to manager:

```
edb=# exec DBMS_SESSION.SET_ROLE('manager');
```

9.14 DBMS_SQL

The DBMS_SQL package provides an application interface 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 without modifying your application.

DBMS_SQL assumes the privileges of the current user when executing dynamic SQL statements.

Table 9-14 DBMS_SQL Functions/Procedures

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

The following table lists the public variable available in the DBMS_SQL package.

Table 9-15 DBMS_SQL Public Variables

Public Variables	Data Type	Value	Description
native	INTEGER	1	See DBMS_SQL.PARSE for more information.
V6	INTEGER	2	See DBMS_SQL.PARSE for more information.
V7	INTEGER	3	See DBMS_SQL.PARSE for more information

9.14.1 BIND_VARIABLE

The `BIND_VARIABLE` procedure provides the capability to associate a value with an `IN` or `IN OUT` bind variable in a SQL command.

```
BIND_VARIABLE(c INTEGER, name VARCHAR2,
  value { BLOB | CLOB | DATE | FLOAT | INTEGER | NUMBER |
          TIMESTAMP | VARCHAR2 }
  [, out_value_size INTEGER ])
```

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value to be assigned.

out_value_size

If *name* is an `IN OUT` variable, defines the maximum length of the output value. If not specified, the length of *value* is assumed.

Examples

The following anonymous block uses bind variables to insert a row into the `emp` table.

```
DECLARE
    curid           INTEGER;
    v_sql           VARCHAR2(150) := 'INSERT INTO emp VALUES ' ||
                        '(:p_empno, :p_ename, :p_job, :p_mgr, ' ||
                        ':p_hiredate, :p_sal, :p_comm, :p_deptno)';
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
```



```
    v_job           emp.job%TYPE;
    v_mgr           emp.mgr%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_status        INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(curid,v_sql,DBMS_SQL.native);
    v_empno    := 9001;
    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
```

9.14.2 BIND_VARIABLE_CHAR

The `BIND_VARIABLE_CHAR` procedure provides the capability to associate a `CHAR` value with an `IN` or `IN OUT` bind variable in a SQL command.

```
BIND_VARIABLE_CHAR(c INTEGER, name VARCHAR2, value CHAR
  [, out_value_size INTEGER ])
```

Parameters

c

Cursor ID of the cursor for the SQL command with bind variables.

name

Name of the bind variable in the SQL command.

value

Value of type `CHAR` to be assigned.

out_value_size

If *name* is an `IN OUT` variable, defines the maximum length of the output value. If not specified, the length of *value* is assumed.

9.14.3 BIND VARIABLE RAW

The `BIND_VARIABLE_RAW` procedure provides the capability to associate a `RAW` value with an `IN` or `IN OUT` bind variable in a SQL command.

```
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.

9.14.4 CLOSE_CURSOR

The `CLOSE_CURSOR` procedure closes an open cursor. The resources allocated to the cursor are released and it can no longer be used.

```
CLOSE_CURSOR(c IN OUT INTEGER)
```

Parameters

c

Cursor ID of the cursor to be closed.

Examples

The following example closes a previously opened cursor:

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
            .
            .
            .
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

9.14.5 COLUMN_VALUE

The `COLUMN_VALUE` procedure defines a variable to receive a value from a cursor.

```
COLUMN_VALUE(c INTEGER, position INTEGER, value OUT { BLOB |
  CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }
  [, column_error OUT NUMBER [, actual_length OUT INTEGER ]])
```

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

position

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

value

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

column_error

Error number associated with the column, if any.

actual_length

Actual length of the data prior to any truncation.

Examples

The following example shows the portion of an anonymous block that receives the values from a cursor using the `COLUMN_VALUE` procedure.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_sql           VARCHAR2(50) := 'SELECT empno, ename, hiredate, sal, ' ||
                                    'comm FROM emp';
    v_status        INTEGER;
BEGIN
            .
            .
            .
    LOOP
        v_status := DBMS_SQL.FETCH_ROWS(curid);
        EXIT WHEN v_status = 0;
        DBMS_SQL.COLUMN_VALUE(curid,1,v_empno);
        DBMS_SQL.COLUMN_VALUE(curid,2,v_ename);
        DBMS_SQL.COLUMN_VALUE(curid,3,v_hiredate);
        DBMS_SQL.COLUMN_VALUE(curid,4,v_sal);
        DBMS_SQL.COLUMN_VALUE(curid,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;
```

9.14.6 COLUMN_VALUE_CHAR

The `COLUMN_VALUE_CHAR` procedure defines a variable to receive a `CHAR` value from a cursor.

```
COLUMN_VALUE_CHAR(c INTEGER, position INTEGER, value OUT CHAR
  [, column_error OUT NUMBER [, actual_length OUT INTEGER ]])
```

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

position

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

value

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

column_error

Error number associated with the column, if any.

actual_length

Actual length of the data prior to any truncation.

9.14.7 COLUMN VALUE RAW

The `COLUMN_VALUE_RAW` procedure defines a variable to receive a `RAW` value from a cursor.

```
COLUMN_VALUE_RAW(c INTEGER, position INTEGER, value OUT RAW
  [, column_error OUT NUMBER [, actual_length OUT INTEGER ]])
```

Parameters

c

Cursor id of the cursor returning data to the variable being defined.

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.

9.14.8 DEFINE_COLUMN

The DEFINE_COLUMN procedure defines a column or expression in the SELECT list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN(c INTEGER, position INTEGER, column { BLOB |
  CLOB | DATE | FLOAT | INTEGER | NUMBER | TIMESTAMP | VARCHAR2 }
  [, column_size INTEGER ])
```

Parameters

c

Cursor id of the cursor associated with the SELECT command.

position

Position of the column or expression in the SELECT list that is being defined.

column

A variable that is of the same data type as the column or expression in position *position* of the SELECT list.

column_size

The maximum length of the returned data. *column_size* must be specified only if *column* is VARCHAR2. Returned data exceeding *column_size* is truncated to *column_size* characters.

Examples

The following shows how the empno, ename, hiredate, sal, and comm columns of the emp table are defined with the DEFINE_COLUMN procedure.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
    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;
```

9.14.9 DEFINE_COLUMN_CHAR

The `DEFINE_COLUMN_CHAR` procedure defines a `CHAR` column or expression in the `SELECT` list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN_CHAR(c INTEGER, position INTEGER, column CHAR,
column_size INTEGER)
```

Parameters

c

Cursor id of the cursor associated with the `SELECT` command.

position

Position of the column or expression in the `SELECT` list that is being defined.

column

A `CHAR` variable.

column_size

The maximum length of the returned data. Returned data exceeding *column_size* is truncated to *column_size* characters.

9.14.10 DEFINE COLUMN RAW

The `DEFINE_COLUMN_RAW` procedure defines a `RAW` column or expression in the `SELECT` list that is to be returned and retrieved in a cursor.

```
DEFINE_COLUMN_RAW(c INTEGER, position INTEGER, column RAW,
  column_size INTEGER)
```

Parameters

c

Cursor id of the cursor associated with the `SELECT` command.

position

Position of the column or expression in the `SELECT` list that is being defined.

column

A `RAW` variable.

column_size

The maximum length of the returned data. Returned data exceeding *column_size* is truncated to *column_size* characters.

9.14.11 DESCRIBE COLUMNS

The `DESCRIBE_COLUMNS` procedure describes the columns returned by a cursor.

```
DESCRIBE_COLUMNS(c INTEGER, col_cnt OUT INTEGER, desc_t OUT
  DESC_TAB);
```

Parameters

c

The cursor ID of the cursor.

col_cnt

The number of columns in cursor result set.

desc_tab

The table that contains a description of each column returned by the cursor. The descriptions are of type `DESC_REC`, and contain the following values:

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

9.14.12 EXECUTE

The `EXECUTE` function executes a parsed SQL command or SPL block.

```
status INTEGER EXECUTE(c INTEGER)
```

Parameters

c

Cursor ID of the parsed SQL command or SPL block to be executed.

status

Number of rows processed if the SQL command was DELETE, INSERT, or UPDATE. *status* is meaningless for all other commands.

Examples

The following anonymous block inserts a row into the dept table.

```
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;
```

9.14.13 EXECUTE_AND_FETCH

Function EXECUTE_AND_FETCH executes a parsed SELECT command and fetches one row.

```
status INTEGER EXECUTE_AND_FETCH(c INTEGER
  [, exact BOOLEAN ])
```

Parameters

c

Cursor id of the cursor for the SELECT command to be executed.

exact

If set to TRUE, an exception is thrown if the number of rows in the result set is not exactly equal to 1. If set to FALSE, no exception is thrown. The default is FALSE. A NO_DATA_FOUND exception is thrown if *exact* is TRUE and there are no rows in the result set. A `TOO_MANY_ROWS` exception is thrown if *exact* is `TRUE` and there is more than one row in the result set.

status

Returns 1 if a row was successfully fetched, 0 if no rows to fetch. If an exception is thrown, no value is returned.

Examples

The following stored procedure uses the `EXECUTE_AND_FETCH` function to retrieve one employee using the employee's name. An exception will be thrown if the employee is not found, or there is more than one employee with the same name.

```
CREATE OR REPLACE PROCEDURE select_by_name(
    p_ename         emp.ename%TYPE
)
IS
    curid           INTEGER;
    v_empno         emp.empno%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR2(10);
    v_sql           VARCHAR2(120) := 'SELECT empno, hiredate, sal, ' ||
                                     'NVL(comm, 0), dname ' ||
                                     'FROM emp e, dept d ' ||
                                     '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
```

9.14.14 FETCH_ROWS

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

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

Parameters

c

Cursor ID of the cursor from which to fetch a row.

status

Returns `1` if a row was successfully fetched, `0` if no more rows to fetch.

Examples

The following example 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
```

9.14.15 IS_OPEN

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

```
status BOOLEAN IS_OPEN(c INTEGER)
```

Parameters

c

Cursor ID of the cursor to be tested.

status

Set to `TRUE` if the cursor is open, set to `FALSE` if the cursor is not open.

9.14.16 LAST_ROW_COUNT

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

```
rowcnt INTEGER LAST_ROW_COUNT
```

Parameters

rowcnt

Number of row fetched thus far.

Examples

The following example uses the `LAST_ROW_COUNT` function to display the total number of rows fetched in the query.

```
DECLARE
    curid           INTEGER;
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_hiredate      DATE;
    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
```

9.14.17 OPEN_CURSOR

The `OPEN_CURSOR` function creates a new cursor. A cursor must be used to parse and execute any dynamic SQL statement. Once a cursor has been opened, it can be re-used with the same or different SQL statements. The cursor does not have to be closed and re-opened in order to be re-used.

c `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;
```

9.14.18 PARSE

The PARSE procedure parses a SQL command or SPL block. If the SQL command is a DDL command, it is immediately executed and does not require running the EXECUTE function.

```
PARSE(c INTEGER, statement VARCHAR2, language_flag INTEGER)
```

Parameters

c

Cursor ID of an open cursor.

statement

SQL command or SPL block to be parsed. A SQL command must not end with the semi-colon terminator, however an SPL block does require the semi-colon terminator.

language_flag

Use DBMS_SQL.V6, DBMS_SQL.V7 or DBMS_SQL.native. This flag is ignored, and all syntax is assumed to be in 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;
```

9.15 DBMS_UTILITY

The `DBMS_UTILITY` package provides various utility programs.

Table 9-16 DBMS_UTILITY Functions/Procedures

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

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

Table 9-17 DBMS_UTILITY Public Variables

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.

9.15.1 LNAME_ARRAY

The LNAME_ARRAY is for storing lists of long names including fully-qualified names.

```
TYPE lname_array IS TABLE OF VARCHAR2(4000) INDEX BY BINARY_INTEGER;
```

9.15.2 UNCL_ARRAY

The UNCL_ARRAY is for storing lists of users and names.

```
TYPE uncl_array IS TABLE OF VARCHAR2(227) INDEX BY BINARY_INTEGER;
```

9.15.3 ANALYZE_DATABASE, ANALYZE SCHEMA and ANALYZE PART_OBJECT

The ANALYZE_DATABASE(), ANALYZE_SCHEMA() and ANALYZE_PART_OBJECT() procedures provide the capability to gather statistics on tables in the database. When you execute the ANALYZE statement, Postgres samples the data in a table and records distribution statistics in the pg_statistics system table.

ANALYZE_DATABASE, ANALYZE_SCHEMA, and ANALYZE_PART_OBJECT differ primarily in the number of tables that are processed:

- ANALYZE_DATABASE analyzes all tables in all schemas within the current database.
- ANALYZE_SCHEMA analyzes all tables in a given schema (within the current database).
- ANALYZE_PART_OBJECT analyzes a single table.

The syntax for the ANALYZE commands are:

```
ANALYZE_DATABASE(method VARCHAR2 [, estimate_rows NUMBER
  [, estimate_percent NUMBER [, method_opt VARCHAR2 ]]])

ANALYZE_SCHEMA(schema VARCHAR2, method VARCHAR2
  [, estimate_rows NUMBER [, estimate_percent NUMBER
  [, method_opt VARCHAR2 ]]])

ANALYZE_PART_OBJECT(schema VARCHAR2, object_name VARCHAR2
  [, 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.

9.15.4 CANONICALIZE

The CANONICALIZE procedure performs the following operations on an input string:

- If the string is not double-quoted, verifies that it uses the characters of a legal identifier. If not, an exception is thrown. If the string is double-quoted, all characters are allowed.
- If the string is not double-quoted and does not contain periods, uppercases all alphabetic characters and eliminates leading and trailing spaces.
- If the string is double-quoted and does not contain periods, strips off the double quotes.
- If the string contains periods and no portion of the string is double-quoted, uppercases each portion of the string and encloses each portion in double quotes.
- If the string contains periods and portions of the string are double-quoted, returns the double-quoted portions unchanged including the double quotes and returns the non-double-quoted portions uppercased and enclosed in double quotes.

```
CANONICALIZE(name VARCHAR2, canon_name OUT VARCHAR2,
  canon_len BINARY_INTEGER)
```

Parameters

name

String to be canonicalized.

canon_name

The canonicalized string.

canon_len

Number of bytes in *name* to canonicalize starting from the first character.

Examples

The following procedure applies the CANONICALIZE procedure on its input parameter and displays the results.

```
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
```

9.15.5 COMMA_TO_TABLE

The `COMMA_TO_TABLE` procedure converts a comma-delimited list of names into a table of names. Each entry in the list becomes a table entry. The names must be formatted as valid identifiers.

```
COMMA_TO_TABLE(list VARCHAR2, tablen OUT BINARY_INTEGER,
  tab OUT { LNAME_ARRAY | UNCL_ARRAY })
```

Parameters

list

Comma-delimited list of names.

tablen

Number of entries in *tab*.

tab

Table containing the individual names in *list*.

LNAME_ARRAY

A DBMS_UTILITY LNAME_ARRAY (as described in Section 9.15.1).

UNCL_ARRAY

A DBMS_UTILITY UNCL_ARRAY (as described in Section 9.15.2).

Examples

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

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

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

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

9.15.6 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 9.4.0.0 on i686-pc-linux-gnu, compiled by GCC gcc (GCC)
4.1.2 20080704 (Red Hat 4.1.2-48), 32-bit
Compatibility: EnterpriseDB 9.4.0.0 on i686-pc-linux-gnu, compiled by GCC gcc
(GCC) 4.1.220080704 (Red Hat 4.1.2-48), 32-bit
```

9.15.7 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
```

9.15.8 FORMAT_CALL_STACK

The `FORMAT_CALL_STACK` function returns the formatted contents of the current call stack.

```
DBMS_UTILITY.FORMAT_CALL_STACK
return VARCHAR2
```

This function can be used in a stored procedure, function or package to return the current call stack in a readable format. This function is useful for debugging purposes.

9.15.9 GET_CPU_TIME

The `GET_CPU_TIME` function returns the CPU time in hundredths of a second from some arbitrary point in time.

```
cputime NUMBER GET_CPU_TIME
```

Parameters

cputime

Number of hundredths of a second of CPU time.

Examples

The following `SELECT` command retrieves the current CPU time, which is 603 hundredths of a second or .0603 seconds.

```
SELECT DBMS_UTILITY.GET_CPU_TIME FROM DUAL;

get_cpu_time
--------------
          603
```

9.15.10 GET_DEPENDENCY

The `GET_DEPENDENCY` procedure provides the capability to list the objects that are dependent upon the specified object. `GET_DEPENDENCY` does not show dependencies for functions or procedures.

```
GET_DEPENDENCY(type VARCHAR2, schema VARCHAR2,
  name VARCHAR2)
```

Parameters

type

The object type of *name*. Valid values are `INDEX`, `PACKAGE`, `PACKAGE BODY`, `SEQUENCE`, `TABLE`, `TRIGGER`, `TYPE` and `VIEW`.

schema

Name of the schema in which *name* exists.

name

Name of the object for which dependencies are to be obtained.

Examples

The following anonymous block finds dependencies on the `EMP` table.

```
BEGIN
    DBMS_UTILITY.GET_DEPENDENCY('TABLE','public','EMP');
END;

DEPENDENCIES ON public.EMP
------------------------------------------------------------------
*TABLE public.EMP()
*   CONSTRAINT c public.emp()
*   CONSTRAINT f public.emp()
*   CONSTRAINT p public.emp()
*   TYPE public.emp()
*   CONSTRAINT c public.emp()
*   CONSTRAINT f public.jobhist()
*   VIEW .empname_view()
```


9.15.11 GET_HASH_VALUE

The `GET_HASH_VALUE` function provides the capability to compute a hash value for a given string.

```
hash NUMBER GET_HASH_VALUE(name VARCHAR2, base NUMBER,
  hash_size NUMBER)
```

Parameters

name

The string for which a hash value is to be computed.

base

Starting value at which hash values are to be generated.

hash_size

The number of hash values for the desired hash table.

hash

The generated hash value.

Examples

The following anonymous block creates a table of hash values using the `ename` column of the `emp` table and then displays the key along with the hash value. The hash values start at 100 with a maximum of 1024 distinct values.

```
DECLARE
    v_hash          NUMBER;
    TYPE hash_tab IS TABLE OF NUMBER INDEX BY VARCHAR2(10);
    r_hash          HASH_TAB;
    CURSOR emp_cur IS SELECT ename FROM emp;
BEGIN
    FOR r_emp IN emp_cur LOOP
        r_hash(r_emp.ename) :=
            DBMS_UTILITY.GET_HASH_VALUE(r_emp.ename,100,1024);
    END LOOP;
    FOR r_emp IN emp_cur LOOP
        DBMS_OUTPUT.PUT_LINE(RPAD(r_emp.ename,10) || ' ' ||
            r_hash(r_emp.ename));
    END LOOP;
END;

SMITH      377
ALLEN      740
WARD       718
```

```
JONES       131
MARTIN      176
BLAKE       568
CLARK       621
SCOTT       1097
KING        235
TURNER      850
ADAMS       156
JAMES       942
FORD        775
MILLER      148
```

9.15.12 GET_PARAMETER_VALUE

The `GET_PARAMETER_VALUE` procedure provides the capability to retrieve database initialization parameter settings.

```
status BINARY_INTEGER GET_PARAMETER_VALUE(parnam VARCHAR2,
  intval OUT INTEGER, strval OUT VARCHAR2)
```

Parameters

parnam

Name of the parameter whose value is to be returned. The parameters are listed in the `pg_settings` system view.

intval

Value of an integer parameter or the length of *strval*.

strval

Value of a string parameter.

status

Returns 0 if the parameter value is `INTEGER` or `BOOLEAN`. Returns 1 if the parameter value is a string.

Examples

The following anonymous block shows the values of two initialization parameters.

```
DECLARE
    v_intval        INTEGER;
    v_strval        VARCHAR2(80);
BEGIN
```



```
    DBMS_UTILITY.GET_PARAMETER_VALUE('max_fsm_pages', v_intval, v_strval);
    DBMS_OUTPUT.PUT_LINE('max_fsm_pages' || ': ' || v_intval);
    DBMS_UTILITY.GET_PARAMETER_VALUE('client_encoding', v_intval, v_strval);
    DBMS_OUTPUT.PUT_LINE('client_encoding' || ': ' || v_strval);
END;

max_fsm_pages: 72625
client_encoding: SQL_ASCII
```

9.15.13 GET_TIME

The `GET_TIME` function provides the capability to return the current time in hundredths of a second.

time `NUMBER GET_TIME`

Parameters

time

Number of hundredths of a second from the time in which the program is started.

Examples

The following example shows calls to the `GET_TIME` function.

```
SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
----------
  1555860

SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
----------
  1556037
```

9.15.14 NAME_TOKENIZE

The `NAME_TOKENIZE` procedure parses a name into its component parts. Names without double quotes are uppercased. The double quotes are stripped from names with double quotes.

```
NAME_TOKENIZE(name VARCHAR2, a OUT VARCHAR2, b OUT VARCHAR2,
  c OUT VARCHAR2, dblink OUT VARCHAR2,
  nextpos OUT BINARY_INTEGER)
```

Parameters

name

String containing a name in the following format:

a[.*b*[.*c*]][@*dblink*]

a

Returns the leftmost component.

b

Returns the second component, if any.

c

Returns the third component, if any.

dblink

Returns the database link name.

nextpos

Position of the last character parsed in name.

Examples

The following stored procedure is used to display the returned parameter values of the NAME_TOKENIZE procedure for various names.

```
CREATE OR REPLACE PROCEDURE name_tokenize (
    p_name          VARCHAR2
)
IS
    v_a             VARCHAR2(30);
    v_b             VARCHAR2(30);
    v_c             VARCHAR2(30);
    v_dblink        VARCHAR2(30);
    v_nextpos       BINARY_INTEGER;
BEGIN
    DBMS_UTILITY.NAME_TOKENIZE(p_name,v_a,v_b,v_c,v_dblink,v_nextpos);
    DBMS_OUTPUT.PUT_LINE('name   : ' || p_name);
    DBMS_OUTPUT.PUT_LINE('a      : ' || v_a);
    DBMS_OUTPUT.PUT_LINE('b      : ' || v_b);
    DBMS_OUTPUT.PUT_LINE('c      : ' || v_c);
    DBMS_OUTPUT.PUT_LINE('dblink : ' || v_dblink);
```



```
    DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
END;
```

Tokenize the name, `emp`:

```
BEGIN
    name_tokenize('emp');
END;

name    : emp
a       : EMP
b       :
c       :
dblink  :
nextpos: 3
```

Tokenize the name, `edb.list_emp`:

```
BEGIN
    name_tokenize('edb.list_emp');
END;

name    : edb.list_emp
a       : EDB
b       : LIST_EMP
c       :
dblink  :
nextpos: 12
```

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

```
BEGIN
    name_tokenize('"edb"."Emp_Admin".update_emp_sal');
END;

name    : "edb"."Emp_Admin".update_emp_sal
a       : edb
b       : Emp_Admin
c       : UPDATE_EMP_SAL
dblink  :
nextpos: 32
```

Tokenize the name `edb.emp@edb_dblink`:

```
BEGIN
    name_tokenize('edb.emp@edb_dblink');
END;

name    : edb.emp@edb_dblink
a       : EDB
b       : EMP
c       :
dblink  : EDB_DBLINK
nextpos: 18
```

9.15.15 TABLE_TO_COMMA

The TABLE_TO_COMMA procedure converts table of names into a comma-delimited list of names. Each table entry becomes a list entry. The names must be formatted as valid identifiers.

```
TABLE_TO_COMMA(tab { LNAME_ARRAY | UNCL_ARRAY },
  tablen OUT BINARY_INTEGER, list OUT VARCHAR2)
```

Parameters

tab

Table containing names.

LNAME_ARRAY

A DBMS_UTILITY LNAME_ARRAY (as described in Section 9.15.1).

UNCL_ARRAY

A DBMS_UTILITY UNCL_ARRAY (as described in Section 9.15.2).

tablen

Number of entries in *list*.

list

Comma-delimited list of names from *tab*.

Examples

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

```
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
```

9.16 UTL_ENCODE

The UTL_ENCODE package provides a way to encode and decode data.

Table 7.7.2 UTL_ENCODE Functions and Procedures

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

9.16.1 BASE64_DECODE

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

```
BASE64_DECODE(r IN RAW)
```


This function returns a `RAW` value.

Parameters

r

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

Examples

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

```
SET bytea_output = escape;
```

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

http://www.enterprisedb.com/docs/en/9.4/pg/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)
```

9.16.2 BASE64_ENCODE

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

```
BASE64_ENCODE(r IN RAW)
```

and

`BASE64_ENCODE(loid IN OID)`

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

Parameters

r

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

loid

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

Examples

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

```
SET bytea_output = escape;
```

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

http://www.enterprisedb.com/docs/en/9.4/pg/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)
```

9.16.3 MIMEHEADER_DECODE

Use the `MIMEHEADER_DECODE` function to decode values that are encoded by the `MIMEHEADER_ENCODE` function. The signature is:

`MIMEHEADER_DECODE(buf IN VARCHAR2)`

This function returns a `VARCHAR2` value.

Parameters

buf

buf contains the value (encoded by `MIMEHEADER_ENCODE`) that will be decoded.

Examples

The following examples use the `MIMEHEADER_ENCODE` and `MIMEHEADER_DECODE` functions to first encode, and then decode a string:

```
edb=# SELECT UTL_ENCODE.MIMEHEADER_ENCODE('What is the date?') FROM DUAL;
      mimeheader_encode
------------------------------
 =?UTF8?Q?What is the date??=
(1 row)

edb=# SELECT UTL_ENCODE.MIMEHEADER_DECODE('=?UTF8?Q?What is the date??=')
FROM DUAL;
 mimeheader_decode
-------------------
 What is the date?
(1 row)
```

9.16.4 MIMEHEADER_ENCODE

Use the `MIMEHEADER_ENCODE` function to convert a string into mime header format, and then encode the string. The signature is:

```
MIMEHEADER_ENCODE(buf IN VARCHAR2, encode_charset IN VARCHAR2
DEFAULT NULL, encoding IN INTEGER DEFAULT NULL)
```

This function returns a `VARCHAR2` value.

Parameters

buf

buf contains the string 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)
```

9.16.5 QUOTED_PRINTABLE_DECODE

Use the QUOTED_PRINTABLE_DECODE function to translate an encoded quoted-printable string into a decoded RAW string.

The signature is:

QUOTED_PRINTABLE_DECODE(*r* IN RAW)

This function returns a RAW value.

Parameters

r

r contains the encoded string that will be decoded. The string is a `RAW` value, encoded by `QUOTED_PRINTABLE_ENCODE`.

Examples

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

```
SET bytea_output = escape;
```

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

http://www.enterprisedb.com/docs/en/9.4/pg/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)
```

9.16.6 QUOTED_PRINTABLE_ENCODE

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

```
QUOTED_PRINTABLE_ENCODE(r IN RAW)
```

This function returns a `RAW` value.

Parameters

r

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

Examples

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

```
SET bytea_output = escape;
```

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

http://www.enterprisedb.com/docs/en/9.4/pg/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)
```

9.16.7 TEXT_DECODE

Use the `TEXT_DECODE` function to translate and decode an encoded string to the `VARCHAR2` value that was originally encoded by the `TEXT_ENCODE` function. The signature is:

```
TEXT_DECODE(buf IN VARCHAR2, encode_charset IN VARCHAR2 DEFAULT
NULL, encoding IN PLS_INTEGER DEFAULT NULL)
```

This function returns a `VARCHAR2` value.

Parameters

buf

> *buf* contains the encoded string that will be translated to the original value encoded by `TEXT_ENCODE`.

encode_charset

> *encode_charset* specifies the character set to which the string will be translated before encoding. The default value is `NULL`.

encoding

encoding specifies the encoding type used by TEXT_DECODE. Specify:

- UTL_ENCODE.BASE64 to specify base-64 encoding.
- UTL_ENCODE.QUOTED_PRINTABLE to specify quoted printable encoding. This is the default.

Examples

The following example uses the TEXT_ENCODE and TEXT_DECODE functions to first encode, and then decode a string:

```
edb=# SELECT UTL_ENCODE.TEXT_ENCODE('What is the date?', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
       text_encode
--------------------------
 V2hhdCBpcyB0aGUgZGF0ZT8=
(1 row)

edb=# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
    text_decode
-------------------
 What is the date?
(1 row)
```

9.16.8 TEXT_ENCODE

Use the TEXT_ENCODE function to translate a string to a user-specified character set, and then encode the string. The signature is:

```
TEXT_DECODE(buf IN VARCHAR2, encode_charset IN VARCHAR2 DEFAULT
NULL, encoding IN PLS_INTEGER DEFAULT NULL)
```

This function returns a VARCHAR2 value.

Parameters

buf

buf contains the encoded string that will be translated to the specified character set and encoded by TEXT_ENCODE.

encode_charset

encode_charset specifies the character set to which the value will be translated before encoding. The default value is NULL.

encoding

encoding specifies the encoding type used by TEXT_ENCODE. Specify:

- UTL_ENCODE.BASE64 to specify base-64 encoding.
- UTL_ENCODE.QUOTED_PRINTABLE to specify quoted printable encoding. This is the default.

Examples

The following example uses the TEXT_ENCODE and TEXT_DECODE functions to first encode, and then decode a string:

```
edb=# SELECT UTL_ENCODE.TEXT_ENCODE('What is the date?', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
       text_encode
--------------------------
 V2hhdCBpcyB0aGUgZGF0ZT8=
(1 row)

edb=# SELECT UTL_ENCODE.TEXT_DECODE('V2hhdCBpcyB0aGUgZGF0ZT8=', 'BIG5',
UTL_ENCODE.BASE64) FROM DUAL;
    text_decode
-------------------
 What is the date?
(1 row)
```

9.16.9 UUDECODE

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

UUDECODE(*r* IN RAW)

This function returns a RAW value.

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

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

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

Parameters

r

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

Examples

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

```
SET bytea_output = escape;
```

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

http://www.enterprisedb.com/docs/en/9.4/pg/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)
```

9.16.10 UUENCODE

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

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

This function returns a `RAW` value.

Parameters

r

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

type

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

Value	Constant
1	complete
2	header_piece
3	middle_piece
4	end_piece

filename

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

permission

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

Examples

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

```
SET bytea_output = escape;
```

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

http://www.enterprisedb.com/docs/en/9.4/pg/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)
```

9.17 UTL_FILE

The UTL_FILE package provides the capability to read from, and write to files on the operating system's file system. Non-superusers must be granted EXECUTE privilege on the UTL_FILE package by a superuser before using any of the functions or procedures in the package. For example the following command grants the privilege to user mary:

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

Also, the operating system username, enterprisedb, must have the appropriate read and/or write permissions on the directories and files to be accessed using the UTL_FILE functions and procedures. If the required file permissions are not in place, an exception is thrown in the UTL_FILE function or procedure.

A handle to the file to be written to, or read from is used to reference the file. The *file handle* is defined by a public variable in the UTL_FILE package named, UTL_FILE.FILE_TYPE. A variable of type FILE_TYPE must be declared to receive the file handle returned by calling the FOPEN function. The file handle is then used for all subsequent operations on the file.

References to directories on the file system are done using the directory name or alias that is assigned to the directory using the CREATE DIRECTORY command. The procedures and functions available in the UTL_FILE package are listed in the following table.

Table 7-9-18 UTL_FILE Functions/Procedures

Function/Procedure	Return Type	Description
FCLOSE(*file* IN OUT)	n/a	Closes the specified file identified by *file*.
FCLOSE_ALL	n/a	Closes all open files.
FCOPY(*location*, *filename*, *dest_dir*, *dest_file* [, *start_line* [, *end_line*]])	n/a	Copies *filename* in the directory identified by *location* to file, *dest_file*, in directory, *dest_dir*, starting from line, *start_line*, to line, *end_line*.
FFLUSH(*file*)	n/a	Forces data in the buffer to be written to disk in the file identified by *file*.
FOPEN(*location*, *filename*, *open_mode* [, *max_linesize*])	FILE_TYPE	Opens file, *filename*, in the directory identified by *location*.
FREMOVE(*location*, *filename*)	n/a	Removes the specified file from the file system.
FRENAME(*location*, *filename*, *dest_dir*, *dest_file* [, *overwrite*])	n/a	Renames the specified file.
GET_LINE(*file*, *buffer* OUT)	n/a	Reads a line of text into variable, *buffer*, from the file identified by *file*.
IS_OPEN(*file*)	BOOLEAN	Determines whether or not the given file is open.

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

UTL_FILE Exception Codes

The `UTL_FILE` package reports the following exception codes:

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

9.17.1 Setting File Permissions with utl_file.umask

When a `UTL_FILE` function or procedure creates a file, there are default file permissions as shown by the following.

```
-rw------- 1 enterprisedb enterprisedb 21 Jul 24 16:08 utlfile
```

Note that all permissions are denied on users belonging to the `enterprisedb` group as well as all other users. Only the `enterprisedb` user has read and write permissions on the created file.

If you wish to have a different set of file permissions on files created by the `UTL_FILE` functions and procedures, you can accomplish this by setting the `utl_file.umask` configuration parameter.

The `utl_file.umask` parameter sets the *file mode creation mask* or simply, the *mask*, in a manner similar to the Linux `umask` command. This is for usage only within the Advanced Server `UTL_FILE` package.

Note: The utl_file.umask parameter is not supported on Windows systems.

The value specified for utl_file.umask is a 3 or 4-character octal string that would be valid for the Linux umask command. The setting determines the permissions on files created by the UTL_FILE functions and procedures. (Refer to any information source regarding Linux or Unix systems for information on file permissions and the usage of the umask command.)

The following is an example of setting the file permissions with utl_file.umask.

First, set up the directory in the file system to be used by the UTL_FILE package. Be sure the operating system account, enterprisedb or postgres, whichever is applicable, can read and write in the directory.

```
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.

9.17.2 FCLOSE

The FCLOSE procedure closes an open file.

```
FCLOSE(file IN OUT FILE_TYPE)
```

Parameters

file

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

9.17.3 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
```

9.17.4 FCOPY

The FCOPY procedure copies text from one file to another.

```
FCOPY(location VARCHAR2, filename VARCHAR2,
  dest_dir VARCHAR2, dest_file VARCHAR2
  [, start_line PLS_INTEGER [, end_line PLS_INTEGER ] ])
```

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
```

9.17.5 FFLUSH

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

`FFLUSH(file FILE_TYPE)`

Parameters

`file`

Variable of type `FILE_TYPE` containing a file handle.

Examples

Each line is flushed after the `NEW_LINE` procedure is called.

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

9.17.6 FOPEN

The `FOPEN` function opens a file for I/O.

```
filetype FILE_TYPE FOPEN(location VARCHAR2, filename VARCHAR2,
  open_mode VARCHAR2 [, max_linesize BINARY_INTEGER ])
```

Parameters

location

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory containing the file to be opened.

filename

Name of the file to be opened.

open_mode

Mode in which the file will be opened. Modes are: `a` - append to file; `r` - read from file; `w` - write to file.

max_linesize

Maximum size of a line in characters. In read mode, an exception is thrown if an attempt is made to read a line exceeding *max_linesize*. In write and append modes, an exception is thrown if an attempt is made to write a line exceeding *max_linesize*. The end-of-line character(s) are not included in determining if the maximum line size is exceeded.

filetype

Variable of type `FILE_TYPE` containing the file handle of the opened file.

9.17.7 FREMOVE

The `FREMOVE` procedure removes a file from the system.

`FREMOVE(`*`location`*` VARCHAR2, `*`filename`*` VARCHAR2)`

An exception is thrown if the file to be removed does not exist.

Parameters

`location`

Directory name, as stored in `pg_catalog.edb_dir.dirname`, of the directory containing the file to be removed.

`filename`

Name of the file to be removed.

Examples

The following removes file `empfile.csv`.

```
DECLARE
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
BEGIN
    UTL_FILE.FREMOVE(v_directory,v_filename);
    DBMS_OUTPUT.PUT_LINE('Removed file: ' || v_filename);
    EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
            DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

Removed file: empfile.csv
```

9.17.8 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
```

9.17.9 GET_LINE

The `GET_LINE` procedure reads a line of text from a given file up to, but not including the end-of-line terminator. A `NO_DATA_FOUND` exception is thrown when there are no more lines to read.

GET_LINE(*file* FILE_TYPE, *buffer* OUT VARCHAR2)

Parameters

file

Variable of type `FILE_TYPE` containing the file handle of the opened file.

buffer

Variable to receive a line from the file.

Examples

The following anonymous block reads through and displays the records in file `empfile.csv`.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
```

```
    v_filename       VARCHAR2(20) := 'empfile.csv';
    v_emprec         VARCHAR2(120);
    v_count          INTEGER := 0;
BEGIN
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'r');
    LOOP
        UTL_FILE.GET_LINE(v_empfile,v_emprec);
        DBMS_OUTPUT.PUT_LINE(v_emprec);
        v_count := v_count + 1;
    END LOOP;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            UTL_FILE.FCLOSE(v_empfile);
            DBMS_OUTPUT.PUT_LINE('End of file ' || v_filename || ' - ' ||
                v_count || ' records retrieved');
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
            DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
END;

7369,SMITH,CLERK,7902,17-DEC-80 00:00:00,800.00,,20
7499,ALLEN,SALESMAN,7698,20-FEB-81 00:00:00,1600.00,300.00,30
7521,WARD,SALESMAN,7698,22-FEB-81 00:00:00,1250.00,500.00,30
7566,JONES,MANAGER,7839,02-APR-81 00:00:00,2975.00,,20
7654,MARTIN,SALESMAN,7698,28-SEP-81 00:00:00,1250.00,1400.00,30
7698,BLAKE,MANAGER,7839,01-MAY-81 00:00:00,2850.00,,30
7782,CLARK,MANAGER,7839,09-JUN-81 00:00:00,2450.00,,10
7788,SCOTT,ANALYST,7566,19-APR-87 00:00:00,3000.00,,20
7839,KING,PRESIDENT,,17-NOV-81 00:00:00,5000.00,,10
7844,TURNER,SALESMAN,7698,08-SEP-81 00:00:00,1500.00,0.00,30
7876,ADAMS,CLERK,7788,23-MAY-87 00:00:00,1100.00,,20
7900,JAMES,CLERK,7698,03-DEC-81 00:00:00,950.00,,30
7902,FORD,ANALYST,7566,03-DEC-81 00:00:00,3000.00,,20
7934,MILLER,CLERK,7782,23-JAN-82 00:00:00,1300.00,,10
End of file empfile.csv - 14 records retrieved
```

9.17.10 IS_OPEN

The `IS_OPEN` function determines whether or not the given file is open.

status `BOOLEAN IS_OPEN(`*file* `FILE_TYPE)`

Parameters

file

> Variable of type `FILE_TYPE` containing the file handle of the file to be tested.

status

> `TRUE` if the given file is open, `FALSE` otherwise.

9.17.11 NEW_LINE

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

NEW_LINE(*file* FILE_TYPE [, *lines* INTEGER])

Parameters

file

Variable of type FILE_TYPE containing the file handle of the file to which end-of-line character sequences are to be written.

lines

Number of end-of-line character sequences to be written. The default is one.

Examples

A file containing a double-spaced list of employee records is written.

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY empno;
BEGIN
    v_empfile := UTL_FILE.FOPEN(v_directory,v_filename,'w');
    FOR i IN emp_cur LOOP
        UTL_FILE.PUT(v_empfile,i.empno);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.ename);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.job);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.mgr);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.hiredate);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.sal);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.comm);
        UTL_FILE.PUT(v_empfile,',');
        UTL_FILE.PUT(v_empfile,i.deptno);
        UTL_FILE.NEW_LINE(v_empfile,2);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Created file: ' || v_filename);
    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
```

9.17.12 PUT

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

```
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
```

9.17.13 PUT_LINE

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

```
PUT_LINE(file FILE_TYPE, buffer { DATE | NUMBER | TIMESTAMP |
  VARCHAR2 })
```

Parameters

file

Variable of type `FILE_TYPE` containing the file handle of the file to which the given line is to be written.

buffer

Text to be written to the specified file.

Examples

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

```
DECLARE
    v_empfile       UTL_FILE.FILE_TYPE;
    v_directory     VARCHAR2(50) := 'empdir';
    v_filename      VARCHAR2(20) := 'empfile.csv';
    v_emprec        VARCHAR2(120);
    CURSOR emp_cur IS SELECT * FROM emp ORDER BY 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
```

9.17.14 PUTF

The `PUTF` procedure writes a formatted string to the given file.

```
PUTF(file FILE_TYPE, format VARCHAR2 [, arg1 VARCHAR2]
  [, ...])
```

Parameters

file

Variable of type `FILE_TYPE` containing the file handle of the file to which the formatted line is to be written.

format

String to format the text written to the file. The special character sequence, `%s`, is substituted by the value of arg. The special character sequence, `\n`, indicates a new line. Note, however, in Postgres Plus Advanced Server, a new line character must be specified with two consecutive backslashes instead of one - `\\n`.

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.

```
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
```

9.18 UTL_HTTP

The `UTL_HTTP` package provides a way to use the HTTP or HTTPS protocol to retrieve information found at an URL.

Table 7.7.2 UTL_HTTP Functions and Procedures

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

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

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

Please Note:

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

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

Advanced Server ignores all cookies it receives.

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

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

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

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

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

1XX INFORMATIONAL	
HTTP_CONTINUE	CONSTANT INTEGER := 100;
HTTP_SWITCHING_PROTOCOLS	CONSTANT INTEGER := 101;
HTTP_PROCESSING	CONSTANT INTEGER := 102;
2XX SUCCESS	
HTTP_OK	CONSTANT INTEGER := 200;
HTTP_CREATED	CONSTANT INTEGER := 201;
HTTP_ACCEPTED	CONSTANT INTEGER := 202;
HTTP_NON_AUTHORITATIVE_INFO	CONSTANT INTEGER := 203;
HTTP_NO_CONTENT	CONSTANT INTEGER := 204;
HTTP_RESET_CONTENT	CONSTANT INTEGER := 205;
HTTP_PARTIAL_CONTENT	CONSTANT INTEGER := 206;
HTTP_MULTI_STATUS	CONSTANT INTEGER := 207;
HTTP_ALREADY_REPORTED	CONSTANT INTEGER := 208;
HTTP_IM_USED	CONSTANT INTEGER := 226;
3XX REDIRECTION	
HTTP_MULTIPLE_CHOICES	CONSTANT INTEGER := 300;
HTTP_MOVED_PERMANENTLY	CONSTANT INTEGER := 301;
HTTP_FOUND	CONSTANT INTEGER := 302;
HTTP_SEE_OTHER	CONSTANT INTEGER := 303;
HTTP_NOT_MODIFIED	CONSTANT INTEGER := 304;
HTTP_USE_PROXY	CONSTANT INTEGER := 305;
HTTP_SWITCH_PROXY	CONSTANT INTEGER := 306;
HTTP_TEMPORARY_REDIRECT	CONSTANT INTEGER := 307;
HTTP_PERMANENT_REDIRECT	CONSTANT INTEGER := 308;
4XX CLIENT ERROR	
HTTP_BAD_REQUEST	CONSTANT INTEGER := 400;
HTTP_UNAUTHORIZED	CONSTANT INTEGER := 401;
HTTP_PAYMENT_REQUIRED	CONSTANT INTEGER := 402;
HTTP_FORBIDDEN	CONSTANT INTEGER := 403;
HTTP_NOT_FOUND	CONSTANT INTEGER := 404;
HTTP_METHOD_NOT_ALLOWED	CONSTANT INTEGER := 405;
HTTP_NOT_ACCEPTABLE	CONSTANT INTEGER := 406;
HTTP_PROXY_AUTH_REQUIRED	CONSTANT INTEGER := 407;
HTTP_REQUEST_TIME_OUT	CONSTANT INTEGER := 408;
HTTP_CONFLICT	CONSTANT INTEGER := 409;
HTTP_GONE	CONSTANT INTEGER := 410;
HTTP_LENGTH_REQUIRED	CONSTANT INTEGER := 411;
HTTP_PRECONDITION_FAILED	CONSTANT INTEGER := 412;
HTTP_REQUEST_ENTITY_TOO_LARGE	CONSTANT INTEGER := 413;
HTTP_REQUEST_URI_TOO_LARGE	CONSTANT INTEGER := 414;
HTTP_UNSUPPORTED_MEDIA_TYPE	CONSTANT INTEGER := 415;
HTTP_REQ_RANGE_NOT_SATISFIABLE	CONSTANT INTEGER := 416;
HTTP_EXPECTATION_FAILED	CONSTANT INTEGER := 417;
HTTP_I_AM_A_TEAPOT	CONSTANT INTEGER := 418;
HTTP_AUTHENTICATION_TIME_OUT	CONSTANT INTEGER := 419;
HTTP_ENHANCE_YOUR_CALM	CONSTANT INTEGER := 420;
HTTP_UNPROCESSABLE_ENTITY	CONSTANT INTEGER := 422;
HTTP_LOCKED	CONSTANT INTEGER := 423;
HTTP_FAILED_DEPENDENCY	CONSTANT INTEGER := 424;
HTTP_UNORDERED_COLLECTION	CONSTANT INTEGER := 425;
HTTP_UPGRADE_REQUIRED	CONSTANT INTEGER := 426;
HTTP_PRECONDITION_REQUIRED	CONSTANT INTEGER := 428;
HTTP_TOO_MANY_REQUESTS	CONSTANT INTEGER := 429;
HTTP_REQUEST_HEADER_FIELDS_TOO_LARGE	CONSTANT INTEGER := 431;
HTTP_NO_RESPONSE	CONSTANT INTEGER := 444;
HTTP_RETRY_WITH	CONSTANT INTEGER := 449;
HTTP_BLOCKED_BY_WINDOWS_PARENTAL_CONTROLS	CONSTANT INTEGER := 450;
HTTP_REDIRECT	CONSTANT INTEGER := 451;

HTTP_REQUEST_HEADER_TOO_LARGE	CONSTANT INTEGER := 494;
HTTP_CERT_ERROR	CONSTANT INTEGER := 495;
HTTP_NO_CERT	CONSTANT INTEGER := 496;
HTTP_HTTP_TO_HTTPS	CONSTANT INTEGER := 497;
HTTP_CLIENT_CLOSED_REQUEST	CONSTANT INTEGER := 499;
5XX SERVER ERROR	
HTTP_INTERNAL_SERVER_ERROR	CONSTANT INTEGER := 500;
HTTP_NOT_IMPLEMENTED	CONSTANT INTEGER := 501;
HTTP_BAD_GATEWAY	CONSTANT INTEGER := 502;
HTTP_SERVICE_UNAVAILABLE	CONSTANT INTEGER := 503;
HTTP_GATEWAY_TIME_OUT	CONSTANT INTEGER := 504;
HTTP_VERSION_NOT_SUPPORTED	CONSTANT INTEGER := 505;
HTTP_VARIANT_ALSO_NEGOTIATES	CONSTANT INTEGER := 506;
HTTP_INSUFFICIENT_STORAGE	CONSTANT INTEGER := 507;
HTTP_LOOP_DETECTED	CONSTANT INTEGER := 508;
HTTP_BANDWIDTH_LIMIT_EXCEEDED	CONSTANT INTEGER := 509;
HTTP_NOT_EXTENDED	CONSTANT INTEGER := 510;
HTTP_NETWORK_AUTHENTICATION_REQUIRED	CONSTANT INTEGER := 511;
HTTP_NETWORK_READ_TIME_OUT_ERROR	CONSTANT INTEGER := 598;
HTTP_NETWORK_CONNECT_TIME_OUT_ERROR	CONSTANT INTEGER := 599;

9.18.1 HTML_PIECES

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

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

9.18.2 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
);
```

9.18.3 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
);
```

9.18.4 BEGIN_REQUEST

The `BEGIN_REQUEST` function initiates a new HTTP request. A network connection is established to the web server with the specified URL. The signature is:

```
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.

9.18.5 END_REQUEST

The `END_REQUEST` procedure terminates an HTTP request. Use the `END_REQUEST` procedure to terminate an HTTP request without completing it and waiting for the response. The normal process is to begin the request, get the response, then close the response. The signature is:

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

Parameters

r

r is the HTTP request record.

9.18.6 END_RESPONSE

The `END_RESPONSE` procedure terminates the HTTP response. The `END_RESPONSE` procedure completes the HTTP request and response. This is the normal method to end the request and response process. The signature is:

```
END_RESPONSE(r IN OUT UTL_HTTP.RESP)
```

Parameters

r

r is the HTTP response record.

9.18.7 GET_BODY_CHARSET

The `GET_BODY_CHARSET` program is available in the form of both a procedure and a function. A call to `GET_BODY_CHARSET` returns the default character set of the body of future HTTP requests.

The procedure signature is:

```
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)
```

9.18.8 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.

9.18.9 GET_HEADER

The `GET_HEADER` procedure returns the *n*th header of the HTTP response. The signature is:

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

Parameters

r

r is the HTTP response record.

n

n is the *n*th header of the HTTP response record to retrieve.

name

name is the name of the response header.

value

value is the value of the response header.

Examples

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

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_name          VARCHAR2(30);
    v_value         VARCHAR2(200);
    v_header_cnt    INTEGER;
BEGIN
 -- Initiate request and get response
    v_req := UTL_HTTP.BEGIN_REQUEST('www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);

 -- Get header count
    v_header_cnt := UTL_HTTP.GET_HEADER_COUNT(v_resp);
    DBMS_OUTPUT.PUT_LINE('Header Count: ' || v_header_cnt);

 -- Get all headers
    FOR i IN 1 .. v_header_cnt LOOP
        UTL_HTTP.GET_HEADER(v_resp, i, v_name, v_value);
        DBMS_OUTPUT.PUT_LINE(v_name || ': ' || v_value);
    END LOOP;

 -- Terminate request
    UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The following is the output from the example.

```
Header Count: 23
Age: 570
Cache-Control: must-revalidate
Content-Type: text/html; charset=utf-8
Date: Wed, 30 Apr 2014 14:57:52 GMT
ETag: "aab02f2bd2d696eed817ca89ef411dda"
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Last-Modified: Wed, 30 Apr 2014 14:15:49 GMT
RTSS: 1-1307-3
Server: Apache/2.2.3 (Red Hat)
Set-Cookie: SESS2771d0952de2a1a84d322a262e0c173c=jn1u1j1etmdi5gg4lh8hakvs01;
expires=Fri, 23-May-2014 18:21:43 GMT; path=/; domain=.enterprisedb.com
Vary: Accept-Encoding
Via: 1.1 varnish
X-EDB-Backend: ec
X-EDB-Cache: HIT
X-EDB-Cache-Address: 10.31.162.212
X-EDB-Cache-Server: ip-10-31-162-212
X-EDB-Cache-TTL: 600.000
X-EDB-Cacheable: MAYBE: The user has a cookie of some sort. Maybe it's double
choc-chip!
X-EDB-Do-GZIP: false
X-Powered-By: PHP/5.2.17
X-Varnish: 484508634 484506789
```



```
transfer-encoding: chunked
Connection: keep-alive
```

9.18.10 GET_HEADER_BY_NAME

The `GET_HEADER_BY_NAME` procedure returns the header of the HTTP response according to the specified name. The signature is:

```
GET_HEADER_BY_NAME(r IN OUT UTL_HTTP.RESP, name VARCHAR2,
value OUT VARCHAR2, n INTEGER DEFAULT 1)
```

Parameters

r

r is the HTTP response record.

name

name is the name of the response header to retrieve.

value

value is the value of the response header.

n

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

Examples

The following example retrieves the header for Content-Type.

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

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

9.18.11 GET_HEADER_COUNT

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

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

This function returns an `INTEGER` value.

Parameters

r

r is the HTTP response record.

9.18.12 GET_RESPONSE

The `GET_RESPONSE` function sends the network request and returns any HTTP response. The signature is:

```
GET_RESPONSE(r IN OUT UTL_HTTP.REQ) RETURN UTL_HTTP.RESP
```

This function returns a `UTL_HTTP.RESP` record.

Parameters

r

r is the HTTP request record.

9.18.13 GET_RESPONSE_ERROR_CHECK

The `GET_RESPONSE_ERROR_CHECK` procedure returns whether or not response error check is set. The signature is:

```
GET_RESPONSE_ERROR_CHECK(enable OUT BOOLEAN)
```

Parameters

enable

enable returns TRUE if response error check is set, otherwise it returns FALSE.

9.18.14 GET_TRANSFER_TIMEOUT

The GET_TRANSFER_TIMEOUT procedure returns the current, default transfer timeout setting for HTTP requests. The signature is:

```
GET_TRANSFER_TIMEOUT(timeout OUT INTEGER)
```

Parameters

timeout

timeout is the transfer timeout setting in seconds.

9.18.15 READ_LINE

The READ_LINE procedure returns the data from the HTTP response body in text form until the end of line is reached. A CR character, a LF character, a CR LF sequence, or the end of the response body constitutes the end of line. The signature is:

```
READ_LINE(r IN OUT UTL_HTTP.RESP, data OUT VARCHAR2,
remove_crlf BOOLEAN DEFAULT FALSE)
```

Parameters

r

r is the HTTP response record.

data

data is the response body in text form.

remove_crlf

Set *remove_crlf* to TRUE to remove new line characters, otherwise set to FALSE. The default is FALSE.

Examples

The following example retrieves and displays the body of the specified website.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    v_resp          UTL_HTTP.RESP;
    v_value         VARCHAR2(1024);
BEGIN
    v_req := UTL_HTTP.BEGIN_REQUEST('http://www.enterprisedb.com');
    v_resp := UTL_HTTP.GET_RESPONSE(v_req);
    LOOP
        UTL_HTTP.READ_LINE(v_resp, v_value, TRUE);
        DBMS_OUTPUT.PUT_LINE(v_value);
    END LOOP;
    EXCEPTION
        WHEN OTHERS THEN
            UTL_HTTP.END_RESPONSE(v_resp);
END;
```

The following is the output.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">

  <!-- ___________________________ HEAD ___________________________ -->

  <head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />


    <title>EnterpriseDB | The Postgres Database Company</title>

    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta name="keywords" content="postgres, postgresql, postgresql installer,
mysql migration, open source database, training, replication" />
<meta name="description" content="The leader in open source database
products, services, support, training and expertise based on PostgreSQL. Free
downloads, documentation, and tutorials." />
<meta name="abstract" content="The Enterprise PostgreSQL Company" />
<link rel="EditURI" type="application/rsd+xml" title="RSD"
href="http://www.enterprisedb.com/blogapi/rsd" />
<link rel="alternate" type="application/rss+xml" title="EnterpriseDB RSS"
href="http://www.enterprisedb.com/rss.xml" />
<link rel="shortcut icon"
href="/sites/all/themes/edb_pixelcrayons/favicon.ico" type="image/x-icon" />
    <link type="text/css" rel="stylesheet" media="all"
href="/sites/default/files/css/css_db11adabae0aed6b79a2c3c52def4754.css" />
<!--[if IE 6]>
<link type="text/css" rel="stylesheet" media="all"
href="/sites/all/themes/oho_basic/css/ie6.css?g" />
<![endif]-->
<!--[if IE 7]>
<link type="text/css" rel="stylesheet" media="all"
href="/sites/all/themes/oho_basic/css/ie7.css?g" />
<![endif]-->
    <script type="text/javascript"
src="/sites/default/files/js/js_74d97b1176812e2fd6e43d62503a5204.js"></script
>
<script type="text/javascript">
<!--//--><![CDATA[//><!--
```


9.18.16 READ_RAW

The `READ_RAW` procedure returns the data from the HTTP response body in binary form. The number of bytes returned is specified by the *len* parameter. The signature is:

```
READ_RAW(r IN OUT UTL_HTTP.RESP, data OUT RAW, len INTEGER)
```

Parameters

r

r is the HTTP response record.

data

data is the response body in binary form.

len

Set *len* to the number of bytes of data to be returned.

Examples

The following example retrieves and displays the first 150 bytes in binary form.

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

The following is the output from the example.

```
\x3c21444f4354595045206868746d6c205055424c494320222d2f2f5733432f2f4454442058485
44d4c20312e30205374726963742f2f454e220d0a202022687474703a2f2f7777772e77332e6f
72672f54522f7868746d6c312f4454442f7868746d6c312d7374726963742e647464223e0d0a3
c68746d6c20786d6c6e733d22687474703a2f2f7777772e77332e6f72672f313939392f
```

9.18.17 READ_TEXT

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

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

Parameters

r

r is the HTTP response record.

data

data is the response body in text form.

len

Set *len* to the maximum number of characters to be returned.

Examples

The following example retrieves the first 150 characters.

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

The following is the output.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/
```

9.18.18 REQUEST

The `REQUEST` function returns the first 2000 bytes retrieved from a user-specified URL. The signature is:

```
REQUEST(url IN VARCHAR2) RETURN VARCHAR2
```

If the data found at the given URL is longer than 2000 bytes, the remainder will be discarded. If the data found at the given URL is shorter than 2000 bytes, the result will be shorter than 2000 bytes.

Parameters

url

> *url* is the Uniform Resource Locator from which `UTL_HTTP` will return content.

Example

The following command returns the first 2000 bytes retrieved from the EnterpriseDB website:

```
SELECT UTL_HTTP.REQUEST('http://www.enterprisedb.com/') FROM DUAL;
```

9.18.19 REQUEST_PIECES

The `REQUEST_PIECES` function returns a table of 2000-byte segments retrieved from an URL. The signature is:

```
REQUEST_PIECES(url IN VARCHAR2, max_pieces NUMBER IN DEFAULT
32767) RETURN UTL_HTTP.HTML_PIECES
```

Parameters

url

> *url* is the Uniform Resource Locator from which `UTL_HTTP` will return content.

max_pieces

> *max_pieces* specifies the maximum number of 2000-byte segments that the `REQUEST_PIECES` function will return. If *max_pieces* specifies more units than are available at the specified *url*, the final unit will contain fewer bytes.

Example

The following example returns the first four 2000 byte segments retrieved from the EnterpriseDB website:

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

9.18.20 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.

9.18.21 SET_FOLLOW_REDIRECT

The `SET_FOLLOW_REDIRECT` procedure sets the maximum number of times the HTTP redirect instruction is to be followed in the response to this request or future requests. This procedures has two signatures:

```
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.

9.18.22 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.

9.18.23 SET_RESPONSE_ERROR_CHECK

The SET_RESPONSE_ERROR_CHECK procedure determines whether or not HTTP 4xx and 5xx status codes returned by the GET_RESPONSE function should be interpreted as errors. The signature is:

```
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`.

9.18.24 SET_TRANSFER_TIMEOUT

The `SET_TRANSFER_TIMEOUT` procedure sets the default, transfer timeout setting for waiting for a response from an HTTP request. This procedure has two signatures:

```
SET_TRANSFER_TIMEOUT(timeout IN INTEGER DEFAULT 60)
```

and

```
SET_TRANSFER_TIMEOUT(r IN OUT UTL_HTTP.REQ, timeout IN
INTEGER DEFAULT 60)
```

Use the second form to change the transfer timeout setting for an individual request that a request inherits from the session default settings.

Parameters

r

r is the HTTP request record.

timeout

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

9.18.25 WRITE_LINE

The `WRITE_LINE` procedure writes data to the HTTP request body in text form; the text is terminated with a CRLF character pair. The signature is:

```
WRITE_LINE(r IN OUT UTL_HTTP.REQ, data IN VARCHAR2)
```

Parameters

r

r is the HTTP request record.

data

data is the request body in `TEXT` form.

Example

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
```

9.18.26 WRITE_RAW

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

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

Parameters

r

r is the HTTP request record.

data

data is the request body in binary form.

Example

The following example writes data in binary form to the request body to be sent using the HTTP `POST` method to a hypothetical web application that accepts and processes such data.

```
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
```

9.18.27 WRITE_TEXT

The `WRITE_TEXT` procedure writes data to the HTTP request body in text form. The signature is:

```
WRITE_TEXT(r IN OUT UTL_HTTP.REQ, data IN VARCHAR2)
```

Parameters

r

r is the HTTP request record.

data

data is the request body in text form.

Example

The following example writes data (`Account balance $500.00`) in text form to the request body to be sent using the HTTP `POST` method. The data is sent to a hypothetical web application (`post.php`) that accepts and processes data.

```
DECLARE
    v_req           UTL_HTTP.REQ;
    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
```

9.19 UTL_MAIL

The `UTL_MAIL` package provides the capability to manage e-mail.

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

Table 9-19 UTL_MAIL Functions/Procedures

Function/Procedure	Function or Procedure	Return Type	Description
`SEND(`*`sender`*`,` *`recipients`*`,` *`cc`*`,` *`bcc`*`,` *`subject`*`,` *`message`* `[,` *`mime_type`* `[,` *`priority`* `]])`	Procedure	n/a	Packages and sends an e-mail to an SMTP server.
`SEND_ATTACH_RAW(`*`sender`*`,` *`recipients`*`,` *`cc`*`,` *`bcc`*`,` *`subject`*`,` *`message`*`,` *`mime_type`*`,` *`priority`*`,` *`attachment`* `[,` *`att_inline`* `[,` *`att_mime_type`* `[,` *`att_filename`* `]]])`	Procedure	n/a	Same as the `SEND` procedure, but with `BYTEA` or large object attachments.
`SEND_ATTACH_VARCHAR2(`*`sender`*`,` *`recipients`*`,` *`cc`*`,` *`bcc`*`,` *`subject`*`,` *`message`*`,` *`mime_type`*`,` *`priority`*`,` *`attachment`* `[,` *`att_inline`* `[,` *`att_mime_type`* `[,` *`att_filename`* `]]])`	Procedure	n/a	Same as the `SEND` procedure, but with `VARCHAR2` attachments.

9.19.1 SEND

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

```
SEND(sender VARCHAR2, recipients VARCHAR2, cc VARCHAR2,
  bcc VARCHAR2, subject VARCHAR2, message VARCHAR2
  [, mime_type VARCHAR2 [, priority PLS_INTEGER ]])
```

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

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

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime_type

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

priority

Priority of the e-mail The default is 3.

Examples

The following anonymous block sends a simple e-mail message.

```
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. 15.';
    UTL_MAIL.SEND(v_sender,v_recipients,NULL,NULL,v_subj,v_msg);
END;
```

9.19.2 SEND_ATTACH_RAW

The `SEND_ATTACH_RAW` procedure provides the capability to send an e-mail to an SMTP server with an attachment containing either `BYTEA` data or a large object (identified by the large object's `OID`). The call to `SEND_ATTACH_RAW` can be written in two ways:

```
SEND_ATTACH_RAW(sender VARCHAR2, recipients VARCHAR2,
  cc VARCHAR2, bcc VARCHAR2, subject VARCHAR2, message VARCHAR2,
```

```
  mime_type VARCHAR2, priority PLS_INTEGER,
  attachment BYTEA[, att_inline BOOLEAN
  [, att_mime_type VARCHAR2[, att_filename VARCHAR2 ]]])

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

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

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

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime_type

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

priority

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

attachment

The attachment.

att_inline

If set to `TRUE`, then the attachment is viewable inline, `FALSE` otherwise. The default is `TRUE`.

att_mime_type

Mime type of the attachment. The default is `application/octet`.

att_filename

The file name containing the attachment. The default is `NULL`.

9.19.3 SEND_ATTACH_VARCHAR2

The `SEND_ATTACH_VARCHAR2` procedure provides the capability to send an e-mail to an SMTP server with a text attachment.

```
SEND_ATTACH_VARCHAR2(sender VARCHAR2, recipients VARCHAR2,
  cc VARCHAR2, bcc VARCHAR2, subject VARCHAR2, message VARCHAR2,
  mime_type VARCHAR2, priority PLS_INTEGER, attachment VARCHAR2
  [, att_inline BOOLEAN [, att_mime_type VARCHAR2
  [, att_filename VARCHAR2 ]]])
```

Parameters

sender

E-mail address of the sender.

recipients

Comma-separated e-mail addresses of the recipients.

cc

Comma-separated e-mail addresses of copy recipients.

bcc

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

subject

Subject line of the e-mail.

message

Body of the e-mail.

mime_type

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

priority

Priority of the e-mail The default is 3.

attachment

The `VARCHAR2` attachment.

att_inline

If set to `TRUE`, then the attachment is viewable inline, `FALSE` otherwise. The default is `TRUE`.

att_mime_type

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

att_filename

The file name containing the attachment. The default is `NULL`.

9.20 UTL_RAW

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

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

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

Postgres Plus 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.

9.20.1 CAST_TO_RAW

The CAST_TO_RAW function converts a VARCHAR2 string to a RAW value. The signature is:

```
CAST_TO_RAW(c VARCHAR2)
```

The function returns a RAW value if you pass a non-NULL value; if you pass a NULL value, the function will return NULL.

Parameters

c

The VARCHAR2 value that will be converted to RAW.

Example

The following example uses the `CAST_TO_RAW` function to convert a `VARCHAR2` string to a `RAW` value:

```
DECLARE
  v VARCHAR2;
  r RAW;
BEGIN
  v := 'Accounts';
  dbms_output.put_line(v);
  r := UTL_RAW.CAST_TO_RAW(v);
  dbms_output.put_line(r);
END;
```

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

```
Accounts
\x4163636f756e7473
```

9.20.2 CAST_TO_VARCHAR2

The `CAST_TO_VARCHAR2` function converts `RAW` data to `VARCHAR2` data. The signature is:

```
CAST_TO_VARCHAR2(r RAW)
```

The function returns a `VARCHAR2` value if you pass a non-`NULL` value; if you pass a `NULL` value, the function will return `NULL`.

Parameters

r

The `RAW` value that will be converted to a `VARCHAR2` value.

Example

The following example uses the `CAST_TO_VARCHAR2` function to convert a `RAW` value to a `VARCHAR2` string:

```
DECLARE
  r RAW;
  v VARCHAR2;
BEGIN
  r := '\x4163636f756e7473'
  dbms_output.put_line(v);
  v := UTL_RAW.CAST_TO_VARCHAR2(r);
  dbms_output.put_line(r);
END;
```

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

```
\x4163636f756e7473
Accounts
```

9.20.3 CONCAT

The CONCAT function concatenates multiple RAW values into a single RAW value. The signature is:

```
CONCAT(r1 RAW, r2 RAW, r3 RAW,…)
```

The function returns a RAW value. Unlike the Oracle implementation, the Advanced Server implementation is a variadic function, and does not place a restriction on the number of values that can be concatenated.

Parameters

r1, r2, r3,…

The RAW values that CONCAT will concatenate.

Example

The following example uses the CONCAT function to concatenate multiple RAW values into a single RAW value:

```
SELECT UTL_RAW.CAST_TO_VARCHAR2(UTL_RAW.CONCAT('\x61', '\x62', '\x63')) FROM
DUAL;
 concat
--------
 abc
(1 row)
```

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

9.20.4 CONVERT

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

```
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
```

9.20.5 LENGTH

The `LENGTH` function returns the length of a `RAW` value. The signature is:

```
LENGTH(r RAW)
```

The function returns a `RAW` value.

Parameters

r

The `RAW` value that `LENGTH` will evaluate.

Example

The following example uses the `LENGTH` function to return the length of a `RAW` value:

```
SELECT UTL_RAW.LENGTH(UTL_RAW.CAST_TO_RAW('Accounts')) FROM DUAL;
 length
--------
8
(1 row)
```

The following example uses the `LENGTH` function to return the length of a `RAW` value that includes multi-byte characters:

```
SELECT UTL_RAW.LENGTH(UTL_RAW.CAST_TO_RAW('独孤求败'));
 length
--------
     12
(1 row)
```


9.20.6 SUBSTR

The `SUBSTR` function returns a substring of a `RAW` value. The signature is:

```
SUBSTR (r RAW, pos INTEGER, len INTEGER)
```

This function returns a `RAW` value.

Parameters

r

The `RAW` value from which the substring will be returned.

pos

The position within the `RAW` value of the first byte of the returned substring.

- If *pos* is `0` or `1`, the substring begins at the first byte of the `RAW` value.
- If *pos* is greater than one, the substring begins at the first byte specified by *pos*. For example, if *pos* is `3`, the substring begins at the third byte of the value.
- If *pos* is negative, the substring begins at *pos* bytes from the end of the source value. For example, if *pos* is `-3`, the substring begins at the third byte from the end of the value.

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)
```


9.21 UTL_SMTP

The `UTL_SMTP` package provides the capability to send e-mails over the Simple Mail Transfer Protocol (SMTP).

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

Table 9-20 UTL_SMTP Functions/Procedures

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

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

Table 9-21 UTL_SMTP Public Variables

Public Variables	Data Type	Value	Description
connection	RECORD		Description of an SMTP connection.
reply	RECORD		SMTP reply line.

9.21.1 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
);
```

9.21.2 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;
```

9.21.3 CLOSE_DATA

The `CLOSE_DATA` procedure terminates an e-mail message by sending the following sequence:

```
<CR><LF>.<CR><LF>
```

This is a single period at the beginning of a line.

```
CLOSE_DATA(c IN OUT CONNECTION)
```

Parameters

c

The SMTP connection to be closed.

9.21.4 COMMAND

The `COMMAND` procedure provides the capability to execute an SMTP command. If you are expecting multiple reply lines, use `COMMAND_REPLIES`.


```
reply REPLY COMMAND(c IN OUT CONNECTION, cmd VARCHAR2
  [, arg VARCHAR2 ])
```

```
COMMAND(c IN OUT CONNECTION, cmd VARCHAR2 [, arg VARCHAR2 ])
```

Parameters

c

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

cmd

The SMTP command to be processed.

arg

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

reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in *reply*.

See Section 9.21.2 for a description of REPLY and REPLIES.

9.21.5 COMMAND_REPLIES

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

```
replies REPLIES COMMAND(c IN OUT CONNECTION, cmd VARCHAR2
  [, arg VARCHAR2 ])
```

Parameters

c

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

cmd

The SMTP command to be processed.

arg

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

replies

SMTP reply lines to the command. See Section 9.21.2 for a description of `REPLY` and `REPLIES`.

9.21.6 DATA

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

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

Parameters

c

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

body

Body of the e-mail message to be sent.

9.21.7 EHLO

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

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

Parameters

c

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

domain

Domain name of the sending host.

9.21.8 HELO

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

```
HELO(c IN OUT, domain VARCHAR2)
```

Parameters

c

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

domain

Domain name of the sending host.

9.21.9 HELP

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

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

Parameters

c

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

command

Command on which help is requested.

replies

SMTP reply lines to the command. See Section 9.21.2 for a description of `REPLY` and `REPLIES`.

9.21.10 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, Section 6.

9.21.11 NOOP

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

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

```
NOOP(c IN OUT CONNECTION)
```

Parameters

c

The SMTP connection on which to send the command.

reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in *reply*. See Section 9.21.2 for a description of `REPLY` and `REPLIES`.

9.21.12 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.

9.21.13 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.

9.21.14 QUIT

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

```
QUIT(c IN OUT CONNECTION)
```

Parameters

c

SMTP connection to be terminated.

9.21.15 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, Section 6.

9.21.16 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.

9.21.17 VRFY

The `VRFY` function provides the capability to validate and verify the recipient's e-mail address. If valid, the recipient's full name and fully qualified mailbox is returned.

```
reply REPLY VRFY(c IN OUT CONNECTION, recipient VARCHAR2)
```

Parameters

c

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

recipient

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

reply

SMTP reply to the command. If SMTP returns multiple replies, only the last one is returned in *reply*. See Section 9.21.2 for a description of `REPLY` and `REPLIES`.

9.21.18 WRITE_DATA

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

```
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.

9.21.19 Comprehensive Example

The following procedure constructs and sends a text e-mail message using the UTL_SMTP package.

```
CREATE OR REPLACE PROCEDURE send_mail (
    p_sender        VARCHAR2,
    p_recipient     VARCHAR2,
    p_subj          VARCHAR2,
    p_msg           VARCHAR2,
    p_mailhost      VARCHAR2
)
IS
    v_conn          UTL_SMTP.CONNECTION;
    v_crlf          CONSTANT VARCHAR2(2)  := CHR(13) || CHR(10);
    v_port          CONSTANT PLS_INTEGER := 25;
BEGIN
    v_conn := UTL_SMTP.OPEN_CONNECTION(p_mailhost,v_port);
    UTL_SMTP.HELO(v_conn,p_mailhost);
    UTL_SMTP.MAIL(v_conn,p_sender);
    UTL_SMTP.RCPT(v_conn,p_recipient);
    UTL_SMTP.DATA(v_conn, SUBSTR(
        'Date: ' || TO_CHAR(SYSDATE,
        'Dy, DD Mon YYYY HH24:MI:SS') || v_crlf
        || 'From: ' || p_sender || v_crlf
        || 'To: ' || p_recipient || v_crlf
        || 'Subject: ' || p_subj || v_crlf
        || p_msg
        , 1, 32767));
    UTL_SMTP.QUIT(v_conn);
END;

EXEC send_mail('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday
Party','Are you planning to attend?','smtp.enterprisedb.com');
```

The following example uses the OPEN_DATA, WRITE_DATA, and CLOSE_DATA procedures instead of the DATA procedure.

```
CREATE OR REPLACE PROCEDURE send_mail_2 (
    p_sender        VARCHAR2,
    p_recipient     VARCHAR2,
    p_subj          VARCHAR2,
    p_msg           VARCHAR2,
    p_mailhost      VARCHAR2
)
IS
    v_conn          UTL_SMTP.CONNECTION;
    v_crlf          CONSTANT VARCHAR2(2)  := CHR(13) || CHR(10);
    v_port          CONSTANT PLS_INTEGER := 25;
BEGIN
    v_conn := UTL_SMTP.OPEN_CONNECTION(p_mailhost,v_port);
    UTL_SMTP.HELO(v_conn,p_mailhost);
```

```
    UTL_SMTP.MAIL(v_conn,p_sender);
    UTL_SMTP.RCPT(v_conn,p_recipient);
    UTL_SMTP.OPEN_DATA(v_conn);
    UTL_SMTP.WRITE_DATA(v_conn,'From: ' || p_sender || v_crlf);
    UTL_SMTP.WRITE_DATA(v_conn,'To: ' || p_recipient || v_crlf);
    UTL_SMTP.WRITE_DATA(v_conn,'Subject: ' || p_subj || v_crlf);
    UTL_SMTP.WRITE_DATA(v_conn,v_crlf || p_msg);
    UTL_SMTP.CLOSE_DATA(v_conn);
    UTL_SMTP.QUIT(v_conn);
END;

EXEC send_mail_2('asmith@enterprisedb.com','pjones@enterprisedb.com','Holiday
Party','Are you planning to attend?','smtp.enterprisedb.com');
```

9.22 UTL_URL

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

Table 7.7.2 UTL_HTTP Functions and Procedures

Function/Procedure	Return Type	Description
`ESCAPE(`*`url`*`, `*`escape_reserved_chars`*`, `*`url_charset`*`)`	`VARCHAR2`	Use the `ESCAPE` function to escape any illegal and reserved characters in a URL.
`UNESCAPE(`*`url`*`, `*`url_charset`*`)`	`VARCHAR2`	The `UNESCAPE` function to convert an URL to its original form.

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

9.22.1 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.ht
ml
```

9.22.2 UNESCAPE

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

The signature is:

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

Parameters

`url`

`url` specifies the Uniform Resource Locator that `UTL_URL` will unescape.

`url_charset`

After unescaping a character, the character is assumed to be in *`url_charset`* encoding, and will be converted from that encoding to database encoding before being returned. If *url_charset* is NULL, the character will not be converted. The default value of *url_charset* is ISO-8859-1.

Examples

The following anonymous block uses the ESCAPE function to escape the blank spaces in the URL:

```
DECLARE
  result varchar2(400);
BEGIN
 result :=
UTL_URL.UNESCAPE('http://www.example.com/Using%20the%20UNESCAPE%2
0function.html');
  DBMS_OUTPUT.PUT_LINE(result);
END;
```

The resulting (unescaped) URL is:

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

10 Expanded Catalog Views

The Expanded Catalog Views provide comprehensive information from another perspective about database objects.

10.1 ALL_ALL_TABLES

The `ALL_ALL_TABLES` view provides information about the tables accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	The name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING (5)	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

10.2 ALL_CONS_COLUMNS

The `ALL_CONS_COLUMNS` view provides information about the columns specified in constraints placed on tables accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

10.3 ALL_CONSTRAINTS

The ALL_CONSTRAINTS view provides information about the constraints placed on tables accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	The constraint type. Possible values are: C – check constraint F – foreign key constraint P – primary key constraint U – unique key constraint R – referential integrity constraint V – constraint on a view O – with read-only, on a view
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade R – restrict N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

10.4 ALL_DB_LINKS

The ALL_DB_LINKS view provides information about the database links accessible by the current user.

Name	Type	Description
owner	TEXT	User name of the database link's owner.
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB
username	TEXT	User name of the user logging in.
host	TEXT	Name or IP address of the remote server.

10.5 ALL_IND_COLUMNS

The ALL_IND_COLUMNS view provides information about columns included in indexes on the tables accessible by the current user.

Name	Type	Description
index_owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
table_owner	TEXT	User name of the table owner.
table_name	TEXT	The name of the table to which the index belongs.
column_name	TEXT	The name of the column.
column_position	SMALLINT	The position of the column within the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

10.6 ALL_INDEXES

The ALL_INDEXES view provides information about the indexes on tables that may be accessed by the current user.

Name	Type	Description
owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
index_type	TEXT	The index type is always BTREE. Included for compatibility only.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Always set to N (not compressed). Included for compatibility only.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Always set to LOGGING. Included for compatibility only.
status	TEXT	Included for compatibility only; always set to VALID.
partitioned	CHARACTER(3)	Indicates that the index is partitioned. Currently, always set to NO.
temporary	CHARACTER(1)	Indicates that an index is on a temporary table. Always set to N; included for compatibility only.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

10.7 ALL_JOBS

The ALL_JOBS view provides information about all jobs that reside in the database.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

10.8 ALL_OBJECTS

The ALL_OBJECTS view provides information about all objects that reside in the database.

Name	Type	Description
owner	TEXT	User name of the object's owner.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	CHARACTER VARYING	Whether or not the state of the object is valid. Currently, Included for compatibility only; always set to VALID.
temporary	TEXT	Y if a temporary object; N if this is a permanent object.

10.9 ALL_PART_KEY_COLUMNS

The ALL_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

10.10ALL_PART_TABLES

The ALL_PART_TABLES view provides information about all of the partitioned tables that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the partitioned table.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The partitioning type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_next_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL

10.11 ALL_POLICIES

The ALL_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

Name	Type	Description
object_owner	TEXT	Name of the owner of the object.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object on which the policy applies.
policy_group	TEXT	Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function if the function belongs to a package.
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Included for compatibility only; always set to NO.
policy_type	TEXT	Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

10.12 ALL_SEQUENCES

The ALL_SEQUENCES view provides information about all user-defined sequences on which the user has SELECT, or UPDATE privileges.

Name	Type	Description
sequence_owner	TEXT	User name of the sequence's owner.
schema_name	TEXT	Name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	This will always return Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers stored in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

10.13 ALL_SOURCE

The ALL_SOURCE view provides a source code listing of the following program types: functions, procedures, triggers, package specifications, and package bodies.

Name	Type	Description
owner	TEXT	User name of the program's owner.
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

10.14ALL_SUBPART_KEY_COLUMNS

The ALL_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

10.15ALL_SYNONYMS

The ALL_SYNONYMS view provides information on all synonyms that may be referenced by the current user.

Name	Type	Description
owner	TEXT	User name of the synonym's owner.
schema_name	TEXT	The name of the schema in which the synonym resides.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the object's owner.
table_schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the object that the synonym refers to.
db_link	TEXT	The name of any associated database link.

10.16 ALL_TAB_COLUMNS

The ALL_TAB_COLUMNS view provides information on all columns in all user-defined tables and views.

Name	Type	Description
owner	CHARACTER VARYING	User name of the owner of the table or view in which the column resides.
schema_name	CHARACTER VARYING	Name of the schema in which the table or view resides.
table_name	CHARACTER VARYING	Name of the table or view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable. Possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the table or view.
data_default	CHARACTER VARYING	Default value assigned to the column.

10.17 ALL_TAB_PARTITIONS

The ALL_TAB_PARTITIONS view provides information about all of the partitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the partition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

10.18 ALL_TAB_SUBPARTITIONS

The ALL_TAB_SUBPARTITIONS view provides information about all of the subpartitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the subpartition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the partition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the subpartition backing table.

10.19 ALL_TABLES

The ALL_TABLES view provides information on all user-defined tables.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Whether or not the state of the table is valid. Currently, Included for compatibility only; always set to VALID.
temporary	CHARACTER(1)	Y if this is a temporary table; N if this is not a temporary table.

10.20 ALL_TRIGGERS

The ALL_TRIGGERS view provides information about the triggers on tables that may be accessed by the current user.

Name	Type	Description
owner	TEXT	User name of the trigger's owner.
schema_name	TEXT	The name of the schema in which the trigger resides.
trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW BEFORE STATEMENT AFTER ROW AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_name	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only. Value will always be SEE TRIGGER BODY FOR TEXT.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

10.21 ALL_TYPES

The ALL_TYPES view provides information about the object types available to the current user.

Name	Type	Description
owner	TEXT	The owner of the object type.
schema_name	TEXT	The name of the schema in which the type is defined.
type_name	TEXT	The name of the type.
type_oid	OID	The object identifier (OID) of the type.
typecode	TEXT	The typecode of the type. Possible values are: OBJECT COLLECTION OTHER
attributes	INTEGER	The number of attributes in the type.

10.22 ALL_USERS

The ALL_USERS view provides information on all user names.

Name	Type	Description
username	TEXT	Name of the user.
user_id	OID	Numeric user id assigned to the user.
created	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL.

10.23 ALL_VIEW_COLUMNS

The ALL_VIEW_COLUMNS view provides information on all columns in all user-defined views.

Name	Type	Description
owner	CHARACTER VARYING	User name of the view's owner.
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

10.24 ALL_VIEWS

The ALL_VIEWS view provides information about all user-defined views.

Name	Type	Description
owner	TEXT	User name of the view's owner.
schema_name	TEXT	Name of the schema in which the view belongs.
view_name	TEXT	Name of the view.
text	TEXT	The SELECT statement that defines the view.

10.25 DBA_ALL_TABLES

The DBA_ALL_TABLES view provides information about all tables in the database.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

10.26 DBA_CONS_COLUMNS

The DBA_CONS_COLUMNS view provides information about all columns that are included in constraints that are specified in on all tables in the database.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

10.27 DBA_CONSTRAINTS

The DBA_CONSTRAINTS view provides information about all constraints on tables in the database.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	The constraint type. Possible values are: C – check constraint F – foreign key constraint P – primary key constraint U – unique key constraint R – referential integrity constraint V – constraint on a view O – with read-only, on a view
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade R - restrict N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

10.28 DBA_DB_LINKS

The DBA_DB_LINKS view provides information about all database links in the database.

Name	Type	Description
owner	TEXT	User name of the database link's owner.
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB
username	TEXT	User name of the user logging in.
host	TEXT	Name or IP address of the remote server.

10.29 DBA_IND_COLUMNS

The DBA_IND_COLUMNS view provides information about all columns included in indexes, on all tables in the database.

Name	Type	Description
index_owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	Name of the index.
table_owner	TEXT	User name of the table's owner.
table_name	TEXT	Name of the table in which the index belongs.
column_name	TEXT	Name of column or attribute of object column.
column_position	SMALLINT	The position of the column in the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

10.30 DBA_INDEXES

The DBA_INDEXES view provides information about all indexes in the database.

Name	Type	Description
owner	TEXT	User name of the index's owner.
schema_name	TEXT	Name of the schema in which the index resides.
index_name	TEXT	The name of the index.
index_type	TEXT	The index type is always BTREE. Included for compatibility only.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Always set to N (not compressed). Included for compatibility only.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Included for compatibility only. Always set to LOGGING.
status	TEXT	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHARACTER(3)	Indicates that the index is partitioned. Always set to NO.
temporary	CHARACTER(1)	Indicates that an index is on a temporary table. Always set to N.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

10.31 DBA_JOBS

The DBA_JOBS view provides information about all jobs in the database.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

10.32 DBA_OBJECTS

The DBA_OBJECTS view provides information about all objects in the database.

Name	Type	Description
owner	TEXT	User name of the object's owner.
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	CHARACTER VARYING	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

10.33 DBA_PART_KEY_COLUMNS

The DBA_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

10.34 DBA_PART_TABLES

The DBA_PART_TABLES view provides information about all of the partitioned tables in the database.

Name	Type	Description
owner	TEXT	The owner of the partitioned table.
schema_name	TEXT	The schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_next_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE.
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT.
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL.

10.35 DBA_POLICIES

The DBA_POLICIES view provides information on all policies in the database. This view is accessible only to superusers.

Name	Type	Description
object_owner	TEXT	Name of the owner of the object.
schema_name	TEXT	The name of the schema in which the object resides.
object_name	TEXT	Name of the object to which the policy applies.
policy_group	TEXT	Name of the policy group. Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function (if the function belongs to a package).
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to SELECT commands. Possible values are YES or NO.
ins	TEXT	Whether or not the policy applies to INSERT commands. Possible values are YES or NO.
upd	TEXT	Whether or not the policy applies to UPDATE commands. Possible values are YES or NO.
del	TEXT	Whether or not the policy applies to DELETE commands. Possible values are YES or NO.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are YES or NO.
chk_option	TEXT	Whether or not the check option is in force for INSERT and UPDATE commands. Possible values are YES or NO.
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are YES or NO.
static_policy	TEXT	Included for compatibility only; always set to NO.
policy_type	TEXT	Included for compatibility only; always set to UNKNOWN.
long_predicate	TEXT	Included for compatibility only; always set to YES.

10.36 DBA_PROFILES

The DBA_PROFILES view provides information about existing profiles. The table includes a row for each profile/resource combination.

Name	Type	Description
profile	CHARACTER VARYING(128)	The name of the profile.
resource_name	CHARACTER VARYING(32)	The name of the resource associated with the profile.
resource_type	CHARACTER VARYING(8)	The type of resource governed by the profile; currently PASSWORD for all supported resources.
limit	CHARACTER VARYING(128)	The limit values of the resource.
common	CHARACTER VARYING(3)	YES for a user-created profile; NO for a system-defined profile.

10.37 DBA_ROLE_PRIVS

The DBA_ROLE_PRIVS view provides information on all roles that have been granted to users. A row is created for each role to which a user has been granted.

Name	Type	Description
grantee	TEXT	User name to whom the role was granted.
granted_role	TEXT	Name of the role granted to the grantee.
admin_option	TEXT	YES if the role was granted with the admin option, NO otherwise.
default_role	TEXT	YES if the role is enabled when the grantee creates a session.

10.38 DBA_ROLES

The DBA_ROLES view provides information on all roles with the NOLOGIN attribute (groups).

Name	Type	Description
role	TEXT	Name of a role having the NOLOGIN attribute – i.e., a group.
password_required	TEXT	Included for compatibility only; always N.

10.39 DBA_SEQUENCES

The DBA_SEQUENCES view provides information about all user-defined sequences.

Name	Type	Description
sequence_owner	TEXT	User name of the sequence's owner.
schema_name	TEXT	The name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	This will always return Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers stored in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

10.40 DBA_SOURCE

The DBA_SOURCE view provides the source code listing of all objects in the database.

Name	Type	Description
owner	TEXT	User name of the program's owner.
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

10.41 DBA_SUBPART_KEY_COLUMNS

The DBA_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that reside in the database.

Name	Type	Description
owner	TEXT	The owner of the table.
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the

		column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

10.42 DBA_SYNONYMS

The DBA_SYNONYM view provides information about all synonyms in the database.

Name	Type	Description
owner	TEXT	User name of the synonym's owner.
schema_name	TEXT	Name of the schema in which the synonym belongs.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the table's owner on which the synonym is defined.
table_schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	Name of the table on which the synonym is defined.
db_link	TEXT	Name of any associated database link.

10.43 DBA_TAB_COLUMNS

The DBA_TAB_COLUMNS view provides information about all columns in the database.

Name	Type	Description
owner	CHARACTER VARYING	User name of the owner of the table or view in which the column resides.
schema_name	CHARACTER VARYING	Name of the schema in which the table or view resides.
table_name	CHARACTER VARYING	Name of the table or view in which the column resides.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the table or view.
data_default	CHARACTER VARYING	Default value assigned to the column.

10.44 DBA_TAB_PARTITIONS

The DBA_TAB_PARTITIONS view provides information about all of the partitions that reside in the database.

Name	Type	Description
table_owner	TEXT	The owner of the table in which the partition resides.
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

10.45DBA_TAB_SUBPARTITIONS

The `DBA_TAB_SUBPARTITIONS` view provides information about all of the subpartitions that reside in the database.

Name	Type	Description
`table_owner`	`TEXT`	The owner of the table in which the subpartition resides.
`schema_name`	`TEXT`	The name of the schema in which the table resides.
`table_name`	`TEXT`	The name of the table.
`partition_name`	`TEXT`	The name of the subpartition.
`subpartition_name`	`TEXT`	The name of the subpartition.
`high_value`	`TEXT`	The high subpartitioning value specified in the `CREATE TABLE` statement.
`high_value_length`	`INTEGER`	The length of the subpartitioning value.
`subpartition_position`	`INTEGER`	`1` for the first subpartition; `2` for the second subpartition, etc.
`tablespace_name`	`TEXT`	The name of the tablespace in which the subpartition resides.
`pct_free`	`NUMERIC`	Included for compatibility only; always `0`
`pct_used`	`NUMERIC`	Included for compatibility only; always `0`
`ini_trans`	`NUMERIC`	Included for compatibility only; always `0`
`max_trans`	`NUMERIC`	Included for compatibility only; always `0`
`initial_extent`	`NUMERIC`	Included for compatibility only; always `NULL`
`next_extent`	`NUMERIC`	Included for compatibility only; always `NULL`
`min_extent`	`NUMERIC`	Included for compatibility only; always `0`
`max_extent`	`NUMERIC`	Included for compatibility only; always `0`
`pct_increase`	`NUMERIC`	Included for compatibility only; always `0`
`freelists`	`NUMERIC`	Included for compatibility only; always `NULL`
`freelist_groups`	`NUMERIC`	Included for compatibility only; always `NULL`
`logging`	`CHARACTER VARYING(7)`	Included for compatibility only; always `YES`
`compression`	`CHARACTER VARYING(8)`	Included for compatibility only; always `NONE`
`num_rows`	`NUMERIC`	Same as `pg_class.reltuples.`
`blocks`	`INTEGER`	Same as `pg_class.relpages.`
`empty_blocks`	`NUMERIC`	Included for compatibility only; always `NULL`
`avg_space`	`NUMERIC`	Included for compatibility only; always `NULL`
`chain_cnt`	`NUMERIC`	Included for compatibility only; always `NULL`
`avg_row_len`	`NUMERIC`	Included for compatibility only; always `NULL`
`sample_size`	`NUMERIC`	Included for compatibility only; always `NULL`
`last_analyzed`	`TIMESTAMP WITHOUT TIME ZONE`	Included for compatibility only; always `NULL`
`buffer_pool`	`CHARACTER VARYING(7)`	Included for compatibility only; always `NULL`
`global_stats`	`CHARACTER VARYING(3)`	Included for compatibility only; always `YES`
`user_stats`	`CHARACTER VARYING(3)`	Included for compatibility only; always `NO`
`backing_table`	`REGCLASS`	Name of the subpartition backing table.

10.46 DBA_TABLES

The DBA_TABLES view provides information about all tables in the database.

Name	Type	Description
owner	TEXT	User name of the table's owner.
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID.
temporary	CHARACTER(1)	Y if the table is temporary; N if the table is permanent.

10.47 DBA_TRIGGERS

The DBA_TRIGGERS view provides information about all triggers in the database.

Name	Type	Description
owner	TEXT	User name of the trigger's owner.
schema_name	TEXT	The name of the schema in which the trigger resides.
trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW BEFORE STATEMENT AFTER ROW AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_names	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only. Value will always be SEE TRIGGER BODY FOR TEXT.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

10.48 DBA_TYPES

The `DBA_TYPES` view provides information about all object types in the database.

Name	Type	Description
owner	TEXT	The owner of the object type.
schema_name	TEXT	The name of the schema in which the type is defined.
type_name	TEXT	The name of the type.
type_oid	OID	The object identifier (OID) of the type.
typecode	TEXT	The typecode of the type. Possible values are: OBJECT COLLECTION OTHER
attributes	INTEGER	The number of attributes in the type.

10.49 DBA_USERS

The DBA_USERS view provides information about all users of the database.

Name	Type	Description
username	TEXT	User name of the user.
user_id	OID	ID number of the user.
password	CHARACTER VARYING(30)	The password (encrypted) of the user.
account_status	CHARACTER VARYING(32)	The current status of the account. Possible values are: OPEN EXPIRED EXPIRED(GRACE) EXPIRED & LOCKED EXPIRED & LOCKED(TIMED) EXPIRED(GRACE) & LOCKED EXPIRED(GRACE) & LOCKED(TIMED) LOCKED LOCKED(TIMED) Use the edb_get_role_status(*role_id*) function to get the current status of the account.
lock_date	TIMESTAMP WITHOUT TIME ZONE	If the account status is LOCKED, lock_date displays the date and time the account was locked.
expiry_date	TIMESTAMP WITHOUT TIME ZONE	The expiration date of the password. Use the edb_get_password_expiry_date(*role_id*) function to get the current password expiration date.
default_tablespace	TEXT	The default tablespace associated with the account.
temporary_tablespace	CHARACTER VARYING(30)	Included for compatibility only. The value will always be '' (an empty string).
created	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only. The value is always NULL.
profile	CHARACTER VARYING(30)	The profile associated with the user.
initial_rsrc_consumer_group	CHARACTER VARYING(30)	Included for compatibility only. The value is always NULL.
external_name	CHARACTER VARYING(4000)	Included for compatibility only. The value is always NULL.

10.50 DBA_VIEW_COLUMNS

The DBA_VIEW_COLUMNS view provides information on all columns in the database.

Name	Type	Description
owner	CHARACTER VARYING	User name of the view's owner.
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

10.51 DBA_VIEWS

The DBA_VIEWS view provides information about all views in the database.

Name	Type	Description
owner	TEXT	User name of the view's owner.
schema_name	TEXT	Name of the schema in which the view belongs.
view_name	TEXT	Name of the view.
text	TEXT	The text of the SELECT statement that defines the view.

10.52 USER_ALL_TABLES

The USER_ALL_TABLES view provides information about all tables owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID..
temporary	TEXT	Y if the table is temporary; N if the table is permanent.

10.53 USER_CONS_COLUMNS

The USER_CONS_COLUMNS view provides information about all columns that are included in constraints in tables that are owned by the current user.

Name	Type	Description
owner	TEXT	User name of the constraint's owner.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
table_name	TEXT	The name of the table to which the constraint belongs.
column_name	TEXT	The name of the column referenced in the constraint.
position	SMALLINT	The position of the column within the object definition.
constraint_def	TEXT	The definition of the constraint.

10.54 USER_CONSTRAINTS

The USER_CONSTRAINTS view provides information about all constraints placed on tables that are owned by the current user.

Name	Type	Description
owner	TEXT	The name of the owner of the constraint.
schema_name	TEXT	Name of the schema in which the constraint belongs.
constraint_name	TEXT	The name of the constraint.
constraint_type	TEXT	The constraint type. Possible values are: C – check constraint F – foreign key constraint P – primary key constraint U – unique key constraint R – referential integrity constraint V – constraint on a view O – with read-only, on a view
table_name	TEXT	Name of the table to which the constraint belongs.
search_condition	TEXT	Search condition that applies to a check constraint.
r_owner	TEXT	Owner of a table referenced by a referential constraint.
r_constraint_name	TEXT	Name of the constraint definition for a referenced table.
delete_rule	TEXT	The delete rule for a referential constraint. Possible values are: C – cascade R – restrict N – no action
deferrable	BOOLEAN	Specified if the constraint is deferrable (T or F).
deferred	BOOLEAN	Specifies if the constraint has been deferred (T or F).
index_owner	TEXT	User name of the index owner.
index_name	TEXT	The name of the index.
constraint_def	TEXT	The definition of the constraint.

10.55 USER_DB_LINKS

The USER_DB_LINKS view provides information about all database links that are owned by the current user.

Name	Type	Description
db_link	TEXT	The name of the database link.
type	CHARACTER VARYING	Type of remote server. Value will be either REDWOOD or EDB
username	TEXT	User name of the user logging in.
password	TEXT	Password used to authenticate on the remote server.
host	TEXT	Name or IP address of the remote server.

10.56 USER_IND_COLUMNS

The USER_IND_COLUMNS view provides information about all columns referred to in indexes on tables that are owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
table_name	TEXT	The name of the table to which the index belongs.
column_name	TEXT	The name of the column.
column_position	SMALLINT	The position of the column within the index.
column_length	SMALLINT	The length of the column (in bytes).
char_length	NUMERIC	The length of the column (in characters).
descend	CHARACTER(1)	Always set to Y (descending); included for compatibility only.

10.57 USER_INDEXES

The USER_INDEXES view provides information about all indexes on tables that are owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the index belongs.
index_name	TEXT	The name of the index.
index_type	TEXT	Included for compatibility only. The index type is always BTREE.
table_owner	TEXT	User name of the owner of the indexed table.
table_name	TEXT	The name of the indexed table.
table_type	TEXT	Included for compatibility only. Always set to TABLE.
uniqueness	TEXT	Indicates if the index is UNIQUE or NONUNIQUE.
compression	CHARACTER(1)	Included for compatibility only. Always set to N (not compressed).
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
logging	TEXT	Included for compatibility only. Always set to LOGGING.
status	TEXT	Whether or not the state of the object is valid. (VALID or INVALID).
partitioned	CHARACTER(3)	Included for compatibility only. Always set to NO.
temporary	CHARACTER(1)	Included for compatibility only. Always set to N.
secondary	CHARACTER(1)	Included for compatibility only. Always set to N.
join_index	CHARACTER(3)	Included for compatibility only. Always set to NO.
dropped	CHARACTER(3)	Included for compatibility only. Always set to NO.

10.58 USER_JOBS

The USER_JOBS view provides information about all jobs owned by the current user.

Name	Type	Description
job	INTEGER	The identifier of the job (Job ID).
log_user	TEXT	The name of the user that submitted the job.
priv_user	TEXT	Same as log_user. Included for compatibility only.
schema_user	TEXT	The name of the schema used to parse the job.
last_date	TIMESTAMP WITH TIME ZONE	The last date that this job executed successfully.
last_sec	TEXT	Same as last_date.
this_date	TIMESTAMP WITH TIME ZONE	The date that the job began executing.
this_sec	TEXT	Same as this_date.
next_date	TIMESTAMP WITH TIME ZONE	The next date that this job will be executed.
next_sec	TEXT	Same as next_date.
total_time	INTERVAL	The execution time of this job (in seconds).
broken	TEXT	If Y, no attempt will be made to run this job. If N, this job will attempt to execute.
interval	TEXT	Determines how often the job will repeat.
failures	BIGINT	The number of times that the job has failed to complete since it's last successful execution.
what	TEXT	The job definition (PL/SQL code block) that runs when the job executes.
nls_env	CHARACTER VARYING(4000)	Always NULL. Provided for compatibility only.
misc_env	BYTEA	Always NULL. Provided for compatibility only.
instance	NUMERIC	Always 0. Provided for compatibility only.

10.59 USER_OBJECTS

The USER_OBJECTS view provides information about all objects that are owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the object belongs.
object_name	TEXT	Name of the object.
object_type	TEXT	Type of the object – possible values are: INDEX, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, SYNONYM, TABLE, TRIGGER, and VIEW.
status	CHARACTER VARYING	Included for compatibility only; always set to VALID.
temporary	TEXT	Y if the object is temporary; N if the object is not temporary.

10.60 USER_PART_KEY_COLUMNS

The USER_PART_KEY_COLUMNS view provides information about the key columns of the partitioned tables that reside in the database.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

10.61 USER_PART_TABLES

The USER_PART_TABLES view provides information about all of the partitioned tables in the database that are owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partitioning_type	TEXT	The partitioning type used to define table partitions.
subpartitioning_type	TEXT	The subpartitioning type used to define table subpartitions.
partition_count	BIGINT	The number of partitions in the table.
def_subpartition_count	INTEGER	The number of subpartitions in the table.
partitioning_key_count	INTEGER	The number of partitioning keys specified.
subpartitioning_key_count	INTEGER	The number of subpartitioning keys specified.
status	CHARACTER VARYING(8)	Provided for compatibility only. Always VALID.
def_tablespace_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
def_pct_free	NUMERIC	Provided for compatibility only. Always NULL.
def_pct_used	NUMERIC	Provided for compatibility only. Always NULL.
def_ini_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_max_trans	NUMERIC	Provided for compatibility only. Always NULL.
def_initial_extent	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_min_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_max_extents	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_pct_increase	CHARACTER VARYING(40)	Provided for compatibility only. Always NULL.
def_freelists	NUMERIC	Provided for compatibility only. Always NULL.
def_freelist_groups	NUMERIC	Provided for compatibility only. Always NULL.
def_logging	CHARACTER VARYING(7)	Provided for compatibility only. Always YES.
def_compression	CHARACTER VARYING(8)	Provided for compatibility only. Always NONE.
def_buffer_pool	CHARACTER VARYING(7)	Provided for compatibility only. Always DEFAULT.
ref_ptn_constraint_name	CHARACTER VARYING(30)	Provided for compatibility only. Always NULL.
interval	CHARACTER VARYING(1000)	Provided for compatibility only. Always NULL.

10.62 USER_POLICIES

The `USER_POLICIES` view provides information on policies where the schema containing the object on which the policy applies has the same name as the current session user. This view is accessible only to superusers.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the object resides.
object_name	TEXT	Name of the object on which the policy applies.
policy_group	TEXT	Name of the policy group. Included for compatibility only; always set to an empty string.
policy_name	TEXT	Name of the policy.
pf_owner	TEXT	Name of the schema containing the policy function, or the schema containing the package that contains the policy function.
package	TEXT	Name of the package containing the policy function if the function belongs to a package.
function	TEXT	Name of the policy function.
sel	TEXT	Whether or not the policy applies to `SELECT` commands. Possible values are `YES` or `NO`.
ins	TEXT	Whether or not the policy applies to `INSERT` commands. Possible values are `YES` or `NO`.
upd	TEXT	Whether or not the policy applies to `UPDATE` commands. Possible values are `YES` or `NO`.
del	TEXT	Whether or not the policy applies to `DELETE` commands. Possible values are `YES` or `NO`.
idx	TEXT	Whether or not the policy applies to index maintenance. Possible values are `YES` or `NO`.
chk_option	TEXT	Whether or not the check option is in force for `INSERT` and `UPDATE` commands. Possible values are `YES` or `NO`.
enable	TEXT	Whether or not the policy is enabled on the object. Possible values are `YES` or `NO`.
static_policy	TEXT	Whether or not the policy is static. Included for compatibility only; always set to `NO`.
policy_type	TEXT	Policy type. Included for compatibility only; always set to `UNKNOWN`.
long_predicate	TEXT	Included for compatibility only; always set to `YES`.

10.63 USER_ROLE_PRIVS

The USER_ROLE_PRIVS view provides information about the privileges that have been granted to the current user. A row is created for each role to which a user has been granted.

Name	Type	Description
username	TEXT	The name of the user to which the role was granted.
granted_role	TEXT	Name of the role granted to the grantee.
admin_option	TEXT	YES if the role was granted with the admin option, NO otherwise.
default_role	TEXT	YES if the role is enabled when the grantee creates a session.
os_granted	CHARACTER VARYING(3)	Included for compatibility only; always NO.

10.64 USER_SEQUENCES

The USER_SEQUENCES view provides information about all user-defined sequences that belong to the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the sequence resides.
sequence_name	TEXT	Name of the sequence.
min_value	NUMERIC	The lowest value that the server will assign to the sequence.
max_value	NUMERIC	The highest value that the server will assign to the sequence.
increment_by	NUMERIC	The value added to the current sequence number to create the next sequent number.
cycle_flag	CHARACTER VARYING	Specifies if the sequence should wrap when it reaches min_value or max_value.
order_flag	CHARACTER VARYING	Included for compatibility only; always Y.
cache_size	NUMERIC	The number of pre-allocated sequence numbers in memory.
last_number	NUMERIC	The value of the last sequence number saved to disk.

10.65 USER_SOURCE

The USER_SOURCE view provides information about all programs owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the program belongs.
name	TEXT	Name of the program.
type	TEXT	Type of program – possible values are: FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, and TRIGGER.
line	INTEGER	Source code line number relative to a given program.
text	TEXT	Line of source code text.

10.66 USER_SUBPART_KEY_COLUMNS

The USER_SUBPART_KEY_COLUMNS view provides information about the key columns of those partitioned tables which are subpartitioned that belong to the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
name	TEXT	The name of the table in which the column resides.
object_type	CHARACTER(5)	For compatibility only; always TABLE.
column_name	TEXT	The name of the column on which the key is defined.
column_position	INTEGER	1 for the first column; 2 for the second column, etc.

10.67 USER_SYNONYMS

The USER_SYNONYMS view provides information about all synonyms owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the synonym resides.
synonym_name	TEXT	Name of the synonym.
table_owner	TEXT	User name of the table's owner on which the synonym is defined.
table_schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	Name of the table on which the synonym is defined.
db_link	TEXT	Name of any associated database link.

10.68 USER_TAB_COLUMNS

The USER_TAB_COLUMNS view displays information about all columns in tables and views owned by the current user.

Name	Type	Description
schema_name	CHARACTER VARYING	Name of the schema in which the table or view resides.
table_name	CHARACTER VARYING	Name of the table or view in which the column resides.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the table.
data_default	CHARACTER VARYING	Default value assigned to the column.

10.69 USER_TAB_PARTITIONS

The USER_TAB_PARTITIONS view provides information about all of the partitions that are owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
composite	TEXT	YES if the table is subpartitioned; NO if the table is not subpartitioned.
partition_name	TEXT	The name of the partition.
subpartition_count	BIGINT	The number of subpartitions in the partition.
high_value	TEXT	The high partitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the partitioning value.
partition_position	INTEGER	1 for the first partition; 2 for the second partition, etc.
tablespace_name	TEXT	The name of the tablespace in which the partition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

10.70 USER_TAB_SUBPARTITIONS

The USER_TAB_SUBPARTITIONS view provides information about all of the subpartitions owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the table resides.
table_name	TEXT	The name of the table.
partition_name	TEXT	The name of the subpartition.
subpartition_name	TEXT	The name of the subpartition.
high_value	TEXT	The high subpartitioning value specified in the CREATE TABLE statement.
high_value_length	INTEGER	The length of the subpartitioning value.
subpartition_position	INTEGER	1 for the first subpartition; 2 for the second subpartition, etc.
tablespace_name	TEXT	The name of the tablespace in which the subpartition resides.
pct_free	NUMERIC	Included for compatibility only; always 0
pct_used	NUMERIC	Included for compatibility only; always 0
ini_trans	NUMERIC	Included for compatibility only; always 0
max_trans	NUMERIC	Included for compatibility only; always 0
initial_extent	NUMERIC	Included for compatibility only; always NULL
next_extent	NUMERIC	Included for compatibility only; always NULL
min_extent	NUMERIC	Included for compatibility only; always 0
max_extent	NUMERIC	Included for compatibility only; always 0
pct_increase	NUMERIC	Included for compatibility only; always 0
freelists	NUMERIC	Included for compatibility only; always NULL
freelist_groups	NUMERIC	Included for compatibility only; always NULL
logging	CHARACTER VARYING(7)	Included for compatibility only; always YES
compression	CHARACTER VARYING(8)	Included for compatibility only; always NONE
num_rows	NUMERIC	Same as pg_class.reltuples.
blocks	INTEGER	Same as pg_class.relpages.
empty_blocks	NUMERIC	Included for compatibility only; always NULL
avg_space	NUMERIC	Included for compatibility only; always NULL
chain_cnt	NUMERIC	Included for compatibility only; always NULL
avg_row_len	NUMERIC	Included for compatibility only; always NULL
sample_size	NUMERIC	Included for compatibility only; always NULL
last_analyzed	TIMESTAMP WITHOUT TIME ZONE	Included for compatibility only; always NULL
buffer_pool	CHARACTER VARYING(7)	Included for compatibility only; always NULL
global_stats	CHARACTER VARYING(3)	Included for compatibility only; always YES
user_stats	CHARACTER VARYING(3)	Included for compatibility only; always NO
backing_table	REGCLASS	Name of the partition backing table.

10.71 USER_TABLES

The USER_TABLES view displays information about all tables owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the table belongs.
table_name	TEXT	Name of the table.
tablespace_name	TEXT	Name of the tablespace in which the table resides if other than the default tablespace.
status	CHARACTER VARYING(5)	Included for compatibility only; always set to VALID..
temporary	CHARACTER(1)	Y if the table is temporary; N if the table is not temporary.

10.72 USER_TRIGGERS

The USER_TRIGGERS view displays information about all triggers on tables owned by the current user.

Name	Type	Description
schema_name	TEXT	The name of the schema in which the trigger resides.
trigger_name	TEXT	The name of the trigger.
trigger_type	TEXT	The type of the trigger. Possible values are: BEFORE ROW BEFORE STATEMENT AFTER ROW AFTER STATEMENT
triggering_event	TEXT	The event that fires the trigger.
table_owner	TEXT	The user name of the owner of the table on which the trigger is defined.
base_object_type	TEXT	Included for compatibility only. Value will always be TABLE.
table_name	TEXT	The name of the table on which the trigger is defined.
referencing_names	TEXT	Included for compatibility only. Value will always be REFERENCING NEW AS NEW OLD AS OLD.
status	TEXT	Status indicates if the trigger is enabled (VALID) or disabled (NOTVALID).
description	TEXT	Included for compatibility only. Value will always be SEE TRIGGER BODY FOR TEXT.
trigger_body	TEXT	The body of the trigger.
action_statement	TEXT	The SQL command that executes when the trigger fires.

10.73 USER_TYPES

The `USER_TYPES` view provides information about all object types owned by the current user.

Name	Type	Description
`schema_name`	`TEXT`	The name of the schema in which the type is defined.
`type_name`	`TEXT`	The name of the type.
`type_oid`	`OID`	The object identifier (OID) of the type.

`typecode`	`TEXT`	The typecode of the type. Possible values are:
		`OBJECT`
		`COLLECTION`
		`OTHER`

`attributes`	`INTEGER`	The number of attributes in the type.

10.74 USER_USERS

The `USER_USERS` view provides information about the current user.

Name	Type	Description
`username`	`TEXT`	User name of the user.
`user_id`	`OID`	ID number of the user.

`account_status`	`CHARACTER VARYING(32)`	The current status of the account. Possible values are:
		`EXPIRED & LOCKED`
		`OPEN`
		`LOCKED`

`lock_date`	`TIMESTAMP WITHOUT TIME ZONE`	Included for compatibility only. The value is always `NULL`.
`expiry_date`	`TIMESTAMP WITHOUT TIME ZONE`	The expiration date of the account.
`default_tablespace`	`CHARACTER VARYING(30)`	The default tablespace associated with the account.
`temporary_tablespace`	`CHARACTER VARYING(30)`	Included for compatibility only. The value will always be '' (an empty string).
`created`	`TIMESTAMP WITHOUT TIME ZONE`	Included for compatibility only. The value will always be `NULL`.
`initial_rsrc_consumer_group`	`CHARACTER VARYING(30)`	Included for compatibility only. The value will always be `NULL`.
`external_name`	`CHARACTER VARYING(4000)`	Included for compatibility only; always set to `NULL`.

10.75 USER_VIEW_COLUMNS

The USER_VIEW_COLUMNS view provides information about all columns in views owned by the current user.

Name	Type	Description
schema_name	CHARACTER VARYING	Name of the schema in which the view belongs.
view_name	CHARACTER VARYING	Name of the view.
column_name	CHARACTER VARYING	Name of the column.
data_type	CHARACTER VARYING	Data type of the column.
data_length	NUMERIC	Length of text columns.
data_precision	NUMERIC	Precision (number of digits) for NUMBER columns.
data_scale	NUMERIC	Scale of NUMBER columns.
nullable	CHARACTER(1)	Whether or not the column is nullable – possible values are: Y – column is nullable; N – column does not allow null.
column_id	NUMERIC	Relative position of the column within the view.
data_default	CHARACTER VARYING	Default value assigned to the column.

10.76 USER_VIEWS

The USER_VIEWS view provides information about all views owned by the current user.

Name	Type	Description
schema_name	TEXT	Name of the schema in which the view resides.
view_name	TEXT	Name of the view.
text	TEXT	The SELECT statement that defines the view.

10.77 V$VERSION

The V$VERSION view provides information about product compatibility.

Name	Type	Description
banner	TEXT	Displays product compatibility information.

10.78PRODUCT_COMPONENT_VERSION

The PRODUCT_COMPONENT_VERSION view provides version information about product version compatibility.

Name	Type	Description
product	CHARACTER VARYING(74)	The name of the product.
version	CHARACTER VARYING(74	The version number of the product.
status	CHARACTER VARYING(74)	Included for compatibility; always Available.

11 System Catalog Tables

The following system catalog tables contain definitions of database objects. The layout of the system tables is subject to change; if you are writing an application that depends on information stored in the system tables, it would be prudent to use an existing catalog view, or create a catalog view to isolate the application from changes to the system table.

11.1 dual

`dual` is a single-row, single-column table that is provided for Oracle compatibility only.

Column	Type	Modifiers	Description
dummy	VARCHAR2(1)		Provided for compatibility only.

11.2 edb_dir

The `edb_dir` table contains one row for each alias that points to a directory created with the `CREATE DIRECTORY` command. A directory is an alias for a pathname that allows a user limited access to the host file system.

You can use a directory to fence a user into a specific directory tree within the file system. For example, the `UTL_FILE` package offers functions that permit a user to read and write files and directories in the host file system, but only allows access to paths that the database administrator has granted access to via a `CREATE DIRECTORY` command.

Column	Type	Modifiers	Description
dirname	"name"	not null	The name of the alias.
dirowner	oid	not null	The OID of the user that owns the alias.
dirpath	text		The directory name to which access is granted.
diracl	aclitem[]		The access control list that determines which users may access the alias.

11.3 edb_all_resource_groups

The `edb_all_resource_groups` table contains one row for each resource group created with the `CREATE RESOURCE GROUP` command and displays the number of active processes in each resource group.

Column	Type	Modifiers	Description
group_name	"name"		The name of the resource group.
active_processes	integer		Number of currently active processes in the resource group.
cpu_rate_limit	float8		Maximum CPU rate limit for the resource group. 0 means no limit.
per_process_cpu_rate_limit	float8		Maximum CPU rate limit per currently active process in the resource group.
dirty_rate_limit	float8		Maximum dirty rate limit for a resource group. 0 means no limit.
per_process_dirty_rate_limit	float8		Maximum dirty rate limit per currently active process in the resource group.

11.4 edb_partdef

The `edb_partdef` table contains one row for each

Column	Type	Modifiers	Description
pdefrel	oid	not null	The OID of the partitioning root (comes from pg_class).
pdeftype	char	not null	The partitioning type: 'r' for range 'l' for list 'h' for hash.
pdefsubtype	char	not null	The subpartitioning type: 'r' for range 'l' for list 'h' for hash.
pdefcols	int2vector	not null	The partitioning key columns (a vector of pg_attribute OIDs).
pdefsubcols	int2vector	not null	The subpartitioning key columns (a vector of pg_attribute OIDs).
pdefkeyexpr	pg_node_tree		Currently unused.
pdefinsertexpr	pg_node_tree		Currently unused.

11.5 edb_partition

The `edb_partition` table contains one row for each partition or subpartition.

Column	Type	Modifiers	Description
partname	name	not_null	The partition or subpartition name.
partpos	integer	not_null	The partition or subpartition position.
partpdefid	oid	not_null	The OID of the edb_partdef tuple (points to edb_partdef).
partrelid	oid	not_null	The OID of the partition backing table (points to pg_class).
partparent	oid	not_null	The OID of the parent edb_partition tuple (for subpartitions).
partcons	oid	not_null	The OID of the CHECK constraint for the partition (points to pg_constraint).
parttablespace	oid	not_null	The OID of the TABLESPACE (points to pg_tablespace).
partistemplate	boolean	not_null	Identifies this partition as a template partition (currently unused).
partvals	pg_node_tree		A list of partition key values in pg_getexpr() form.

11.6 edb_password_history

The `edb_password_history` table contains one row for each password change. The table is shared across all databases within a cluster.

Column	Type	References	Description
passhistroleid	oid	pg_authid.oid	The ID of a role.
passhistpassword	text		Role password in md5 encrypted form.
passhistpasswordsetat	timestamptz		The time the password was set.

11.7 edb_policy

The `edb_partition` table contains one row for each policy.

Column	Type	Modifiers	Description
policyname	name	not null	The policy name.
policygroup	oid	not null	Currently unused.
policyobject	oid	not null	The OID of the table secured by this policy (the object_schema plus the object_name).
policykind	char	not null	The kind of object secured by this policy: 'r' for a table 'v' for a view = for a synonym Currently always 'r'.
policyproc	oid	not null	The OID of the policy function (function_schema plus policy_function).
policyinsert	boolean	not null	True if the policy is enforced by INSERT statements.
policyselect	boolean	not null	True if the policy is enforced by SELECT statements.
policydelete	boolean	not null	True if the policy is enforced by DELETE statements.
policyupdate	boolean	not null	True if the policy is enforced by UPDATE statements.
policyindex	boolean	not null	Currently unused.
policyenabled	boolean	not null	True if the policy is enabled.
policyupdatecheck	boolean	not null	True if rows updated by an UPDATE statement must satisfy the policy.
policystatic	boolean	not null	Currently unused.
policytype	integer	not null	Currently unused.
policyopts	integer	not null	Currently unused.
policyseccols	int2vector	not null	The column numbers for columns listed in sec_relevant_cols.

11.8 edb_profile

The `edb_profile` table stores information about the available profiles. `edb_profiles` is shared across all databases within a cluster.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected).
prfname	name		The name of the profile.
prffailedloginattempts	integer		The number of failed login attempts allowed by the profile. -1 indicates that the value from the default profile should be used. -2 indicates no limit on failed login attempts.
prfpasswordlocktime	integer		The password lock time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the account should be locked permanently.
prfpasswordlifetime	integer		The password life time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
prfpasswordgracetime	integer		The password grace time associated with the profile (in seconds). -1 indicates that the value from the default profile should be used. -2 indicates that the password never expires.
prfpasswordreusetime	integer		The number of seconds that a user must wait before reusing a password. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordreusemax	integer		The number of password changes that have to occur before a password can be reused. -1 indicates that the value from the default profile should be used. -2 indicates that the old passwords can never be reused.
prfpasswordverifyfuncdb	oid	pg_database.oid	The OID of the database in which the password verify function exists.
prfpasswordverifyfunc	oid	pg_proc.oid	The OID of the password verify function associated with the profile.

11.9 edb_resource_group

The `edb_resource_group` table contains one row for each resource group created with the `CREATE RESOURCE GROUP` command.

Column	Type	Modifiers	Description
rgrpname	"name"	not null	The name of the resource group.
rgrpcpuratelimit	float8	not null	Maximum CPU rate limit for a resource group. 0 means no limit.
rgrpdirtyratelimit	float8	not null	Maximum dirty rate limit for a resource group. 0 means no limit.

11.10 edb_variable

The `edb_variable` table contains one row for each package level variable (each variable declared within a package).

Column	Type	Modifiers	Description
varname	"name"	not null	The name of the variable.
varpackage	oid	not null	The OID of the pg_namespace row that stores the package.
vartype	oid	not null	The OID of the pg_type row that defines the type of the variable.
varaccess	"char"	not null	+ if the variable is visible outside of the package. - if the variable is only visible within the package. Note: Public variables are declared within the package header; private variables are declared within the package body.
varsrc	text		Contains the source of the variable declaration, including any default value expressions for the variable.
varseq	smallint	not null	The order in which the variable was declared in the package.

11.11pg_synonym

The pg_synonym table contains one row for each synonym created with the CREATE SYNONYM command or CREATE PUBLIC SYNONYM command.

Column	Type	Modifiers	Description
synname	"name"	not null	The name of the synonym.
synnamespace	oid	not null	Replaces synowner. Contains the OID of the pg_namespace row where the synonym is stored
synowner	oid	not null	The OID of the user that owns the synonym.
synobjschema	"name"	not null	The schema in which the referenced object is defined.
synobjname	"name"	not null	The name of the referenced object.
synlink	text		The (optional) name of the database link in which the referenced object is defined.

11.12product_component_version

The product_component_version table contains information about feature compatibility; an application can query this table at installation or run time to verify that features used by the application are available with this deployment.

Column	Type	Description
product	character varying (74)	The name of the product.
version	character varying (74)	The version number of the product.
status	character varying (74)	The status of the release.

12 Appendix

This chapter contains various miscellaneous topics.

12.1 Advanced Server Database Limits

This section lists the Advanced Server database limits.

Table 12-1 - Advanced Server Database Limits

Limit	Value
Maximum Database Size	Unlimited
Maximum Table Size	32 TB
Maximum Row Size	1.6 TB
Maximum Field Size	1 GB
Maximum Rows per Table	Unlimited
Maximum Columns per Table	250 - 1600 depending on column types
Maximum Indexes per Table	Unlimited

12.2 Advanced Server Keywords

A keyword is a word that is recognized by the Advanced Server parser as having a special meaning or association. You can use the `pg_get_keywords()` function to retrieve an up-to-date list of the Advanced Server keywords:

```
acctg=#
acctg=# SELECT * FROM pg_get_keywords();
        word          | catcode |          catdesc
---------------------+---------+----------------------------------
 abort                | U       | unreserved
 absolute             | U       | unreserved
 access               | U       | unreserved
 action               | U       | unreserved
 add                  | U       | unreserved
...
```

`pg_get_keywords` returns a table containing the keywords recognized by Advanced Server:

- The `word` column displays the keyword.
- The `catcode` column displays a category code.
- The `catdesc` column displays a brief description of the category to which the keyword belongs.

Note that any character can be used in an identifier if the name is enclosed in double quotes. You can selectively query the `pg_get_keywords()` function to retrieve an up-to-date list of the Advanced Server keywords that belong to a specific category:

```
SELECT * FROM pg_get_keywords() WHERE catcode = 'code';
```

Where *code* is:

> `R` - The word is reserved. Reserved keywords may never be used as an identifier; they are reserved for use by the server.
>
> `U` - The word is unreserved. Unreserved words are used internally in some contexts, but may be used as a name for a database object.
>
> `T` - The word is used internally, but may be used as a name for a function or type.
>
> `C` - The word is used internally, and may not be used as a name for a function or type.

For more information about Advanced Server identifiers and keywords, please see the PostgreSQL core documentation at:

http://www.enterprisedb.com/docs/en/9.4/pg/sql-syntax-lexical.html