EDB Postgres™ Advanced Server Guide

EDB Postgres™ Advanced Server 12

February 25, 2021

EDB Postgres™ Advanced Server Guide
by EnterpriseDB® Corporation

EnterpriseDB Corporation, 34 Crosby Drive, Suite 201, Bedford, MA 01730, USA
T +1 781 357 3390 **F** +1 978 467 1307 **E** info@enterprisedb.com www.enterprisedb.com

Table of Contents

1 Introduction

This guide describes the features of *EDB Postgres Advanced Server (Advanced Server).*

Advanced Server adds extended functionality to the open-source PostgreSQL database. The extended functionality supports database administration, enhanced SQL capabilities, database and application security, performance monitoring and analysis, and application development utilities. This guide documents those features that are exclusive to Advanced Server:

- **Enhanced Compatibility Features.** Chapter 2 provides an overview of compatibility features supported by Advanced Server.

- **Database Administration.** Chapter 3 contains information about features and tools that are helpful to the database administrator.

  Index Advisor described in Section 3.2 helps to determine the additional indexes needed on tables to improve application performance.

  SQL Profiler described in Section 3.3 locates and diagnoses poorly running SQL queries in applications.

  pgsnmpd described in Section 3.4 is an SNMP agent that returns hierarchical monitoring information regarding the current state of Advanced Server.

- **Security.** Chapter 4 contains information about security features supported by Advanced Server.

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

  Virtual Private Database described in Section 4.2 provides fine-grained, row level access.

  sslutils described in Section 4.3 provides SSL certificate generation functions.

  Data redaction described in Section 4.4 provides protection against sensitive data exposure.

- **EDB Resource Manager.** Chapter 5 contains information about 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.

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.

- **The libpq C Library.** The *libpq C library* described in Chapter 6 is the C application programming interface (API) language for Advanced Server.

- **The PL Debugger.** *The PL Debugger* described in Chapter 7 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.

  EDB wait states described in Section 8.2 provides a way to capture wait events and other data for performance diagnosis.

- **EDB Clone Schema.** Chapter 9 contains information about the EDB Clone Schema feature, which provides the capability to copy a schema and its database objects within a single database or from one database to another database.

- **Enhanced SQL and Other Miscellaneous Features.** Chapter 10 contains information on enhanced SQL functionality and other features that provide additional flexibility and convenience.

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

- **Advanced Server Keywords.** Chapter 12 contains information about the words that Advanced Server recognizes as keywords.

For information about the features that are shared by Advanced Server and PostgreSQL, see the PostgreSQL core documentation, available at:

https://www.postgresql.org/docs/12/static/index.html

1.1 What's New

The following features have been changed in EDB Postgres Advanced Server 11 to create Advanced Server 12:

- Advanced Server introduces `COMPOUND TRIGGERS`, which are stored as a PL block that executes in response to a specified triggering event. For information, see the *Database Compatibility for Oracle Developer's Guide*.

- Advanced Server now supports new `DATA DICTIONARY VIEWS` that provide information that is compatible with the Oracle data dictionary views. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.

- Advanced Server has added the `LISTAGG` function to support string aggregation that concatenates data from multiple rows into a single row in an ordered manner. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.

- Advanced Server now supports `CAST(MULTISET)` function, allowing subquery output to be `CAST` to a nested table type. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.

- Advanced Server has added the `MEDIAN` function to calculate a median value from the set of provided values. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.

- Advanced Server has added the `SYS_GUID` function to generate and return a globally unique identifier in the form of `16-bytes` of `RAW` data. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.

- Advanced Server now supports an Oracle-compatible `SELECT UNIQUE` clause in addition to an existing `SELECT DISTINCT` clause. For information, see the *Database Compatibility for Oracle Developer's Reference Guide*.

- Advanced Server has re-implemented `default_with_rowids`, which is used to create a table that includes a `ROWID` column in the newly created table. For more information, see Section 3.1.3.11.7.

- Advanced Server now supports logical decoding on the standby server, which allows creating a logical replication slot on a standby, independently of a primary server. For more information, see Section 10.3.

- Advanced Server introduces `INTERVAL PARTITIONING`, which allows a database to automatically create partitions of a specified interval as new data is inserted into a table. For information, see the *Database Compatibility for Oracle Developer's Guide*.

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

The following is a list of other conventions used throughout this document.

- This guide applies to both Linux and Windows systems. Directory paths are presented in the Linux format with forward slashes. When working on Windows systems, start the directory path with the drive letter followed by a colon and substitute back slashes for forward slashes.
- Some of the information in this document may apply interchangeably to the PostgreSQL and EDB Postgres Advanced Server database systems. The term *Advanced Server* is used to refer to EDB Postgres Advanced Server. The term *Postgres* is used to generically refer to both PostgreSQL and Advanced Server. When a distinction needs to be made between these two database systems, the specific names, PostgreSQL or Advanced Server are used.

- The installation directory path of the PostgreSQL or Advanced Server products is referred to as `POSTGRES_INSTALL_HOME`. For PostgreSQL Linux installations, this defaults to `/opt/PostgreSQL/x.x` for version 10 and earlier. For later versions, use the PostgreSQL community packages. For Advanced Server Linux installations accomplished using the interactive installer for version 10 and earlier, this defaults to `/opt/edb/asx.x`. For Advanced Server Linux installations accomplished using an RPM package, this defaults to `/usr/edb/asxx`. For Advanced Server Windows installations, this defaults to `C:\Program Files\edb\asxx`. The product version number is represented by `x.x` or by `xx` for version 10 and later.

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 Advanced Server is installed.

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

```
/usr/edb/asxx/share/pg-sample.sql
```

where *`xx`* is the Advanced Server version number.

In addition there is a script in the same directory containing the database objects created using syntax compatible with Oracle databases. This script file is `edb-sample.sql`.

The script:

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

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

```
SHOW SEARCH_PATH;
```

You can use PSQL commands to modify the search path.

1.4.1 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 Enhanced Compatibility Features

Advanced Server includes extended functionality that provides compatibility for syntax supported by Oracle applications. Detailed information about all of the compatibility features supported by Advanced Server is provided in the Database Compatibility for Oracle Developers Guides; the information is broken into four guides:

- The *Database Compatibility for Oracle Developers Guide* provides an overview of the compatible procedural language, profile management, partitioning syntax, and sample applications supported by Advanced Server.

- The *Database Compatibility for Oracle Developers Tools and Utilities Guide* provides information about the compatible tools supported by Advanced Server: EDB*Plus, EDB*Loader, EDB*Wrap, and DRITA.

- The *Database Compatibility for Oracle Developers Built-in Packages Guide* provides information about using the compatible syntax available in the built-in packages.

- The *Database Compatibility for Oracle Developers Reference Guide* provides reference information about using Advanced Server compatibility features, including SQL syntax, compatible views and system tables, and data types.

Version-specific guides are available at:

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

The following sections highlight some of the compatibility features supported by Advanced Server.

2.1 Enabling Compatibility Features

There are several ways to install Advanced Server that will allow you to take advantage of compatibility features:

- Use the `INITDBOPTS` variable (in the Advanced Server service configuration file) to specify --`redwood-like` before initializing your cluster.

- When invoking `initdb` to initialize your cluster, include the --`redwood-like` option.

For more information about the installation options supported by the Advanced Server installers, please see the EDB Postgres Advanced Server Installation Guide, available from the EDB website at:

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

2.2 Stored Procedural Language

Advanced Server supports a highly productive procedural language that allows you to write custom procedures, functions, triggers and packages. The procedural language:

- complements the SQL language and built-in packages.

- provides a seamless development and testing environment.

- allows you to create reusable code.

For information about using the Stored Procedural Language, see the *Database Compatibility for Oracle Developers Guide*, available at:

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

2.3 Optimizer Hints

When you invoke a `DELETE`, `INSERT`, `SELECT`, or `UPDATE` command, the server generates a set of execution plans; after analyzing those execution plans, the server selects a plan that will (generally) return the result set in the least amount of time. The server's choice of plan is dependent upon several factors:

- The estimated execution cost of data handling operations.
- Parameter values assigned to parameters in the Query Tuning section of the `postgresql.conf` file.
- Column statistics that have been gathered by the `ANALYZE` command.

As a rule, the query planner will select the least expensive plan. You can use an optimizer hint to influence the server as it selects a query plan.

An optimizer hint is a directive (or multiple directives) embedded in a comment-like syntax that immediately follows a `DELETE`, `INSERT`, `SELECT` or `UPDATE` command. Keywords in the comment instruct the server to employ or avoid a specific plan when producing the result set. For information about using optimizer hints, see the *Database Compatibility for Oracle Developers Guide*, available at:

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

2.4 Data Dictionary Views

Advanced Server includes a set of views that provide information about database objects in a manner compatible with the Oracle data dictionary views. For detailed information about the views available with Advanced Server, please see the *Database Compatibility for Oracle Developers Reference Guide*, available at:

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

2.5 dblink_ora

`dblink_ora` provides an OCI-based database link that allows you to `SELECT`, `INSERT`, `UPDATE` or `DELETE` data stored on an Oracle system from within Advanced Server. For detailed information about using `dblink_ora`, and the supported functions and procedures, see the *Database Compatibility for Oracle Developers Guide*, available at:

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

2.6 Profile Management

Advanced Server supports compatible SQL syntax for profile management. Profile management commands allow a database superuser to create and manage named *profiles*. Each profile defines rules for password management that augment `password` and `md5` authentication. The rules in a profile can:

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

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

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

For information about using profile management commands, see the *Database Compatibility for Oracle Developers Guide*, available at:

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

2.7 Built-In Packages

Advanced Server supports a number of built-in packages that provide compatibility with Oracle procedures and functions.

Package Name	Description
DBMS_ALERT	The DBMS_ALERT package provides the capability to register for, send, and receive alerts.
DBMS_AQ	The DBMS_AQ package provides message queueing and processing for Advanced Server.
DBMS_AQADM	The DBMS_AQADM package provides supporting procedures for Advanced Queueing functionality.
DBMS_CRYPTO	The DBMS_CRYPTO package provides functions and procedures that allow you to encrypt or decrypt RAW, BLOB or CLOB data. You can also use DBMS_CRYPTO functions to generate cryptographically strong random values.
DBMS_JOB	The DBMS_JOB package provides for the creation, scheduling, and managing of jobs.
DBMS_LOB	The DBMS_LOB package provides the capability to operate on large objects.
DBMS_LOCK	Advanced Server provides support for the DBMS_LOCK.SLEEP procedure.
DBMS_MVIEW	Use procedures in the DBMS_MVIEW package to manage and refresh materialized views and their dependencies.
DBMS_OUTPUT	The DBMS_OUTPUT package provides the capability to send messages to a message buffer, or get messages from the message buffer.
DBMS_PIPE	The DBMS_PIPE package provides the capability to send messages through a pipe within or between sessions connected to the same database cluster.
DBMS_PROFILER	The DBMS_PROFILER package collects and stores performance information about the PL/pgSQL and SPL statements that are executed during a performance profiling session.
DBMS_RANDOM	The DBMS_RANDOM package provides a number of methods to generate random values. The procedures and functions available in the DBMS_RANDOM package are listed in the following table.
DBMS_REDACT	The DBMS_REDACT package enables the redacting or masking of data that is returned by a query.
DBMS_RLS	The DBMS_RLS package enables the implementation of Virtual Private Database on certain Advanced Server database objects.
DBMS_SCHEDULER	The DBMS_SCHEDULER package provides a way to create and manage jobs, programs and job schedules.
DBMS_SESSION	Advanced Server provides support for the DBMS_SESSION.SET_ROLE procedure.
DBMS_SQL	The DBMS_SQL package provides an application interface to the EnterpriseDB dynamic SQL functionality.
DBMS_UTILITY	The DBMS_UTILITY package provides various utility programs.
UTL_ENCODE	The UTL_ENCODE package provides a way to encode and decode data.
UTL_FILE	The UTL_FILE package provides the capability to read from, and write to files on the operating system's file system.
UTL_HTTP	The UTL_HTTP package provides a way to use the HTTP or HTTPS protocol to retrieve information found at an URL.
UTL_MAIL	The UTL_MAIL package provides the capability to manage e-mail.

UTL_RAW	The UTL_RAW package allows you to manipulate or retrieve the length of raw data types.
UTL_SMTP	The UTL_SMTP package provides the capability to send e-mails over the Simple Mail Transfer Protocol (SMTP).
UTL_URL	The UTL_URL package provides a way to escape illegal and reserved characters within an URL.

For detailed information about the procedures and functions available within each package, please see the *Database Compatibility for Oracle Developers Built-In Package Guide*, available at:

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

2.8 Open Client Library

The Open Client Library provides application interoperability with the Oracle Call Interface – an application that was formerly "locked in" can now work with either an Advanced Server or an Oracle database with minimal to no changes to the application code. The EnterpriseDB implementation of the Open Client Library is written in C.

The following diagram compares the Open Client Library and Oracle Call Interface application stacks.

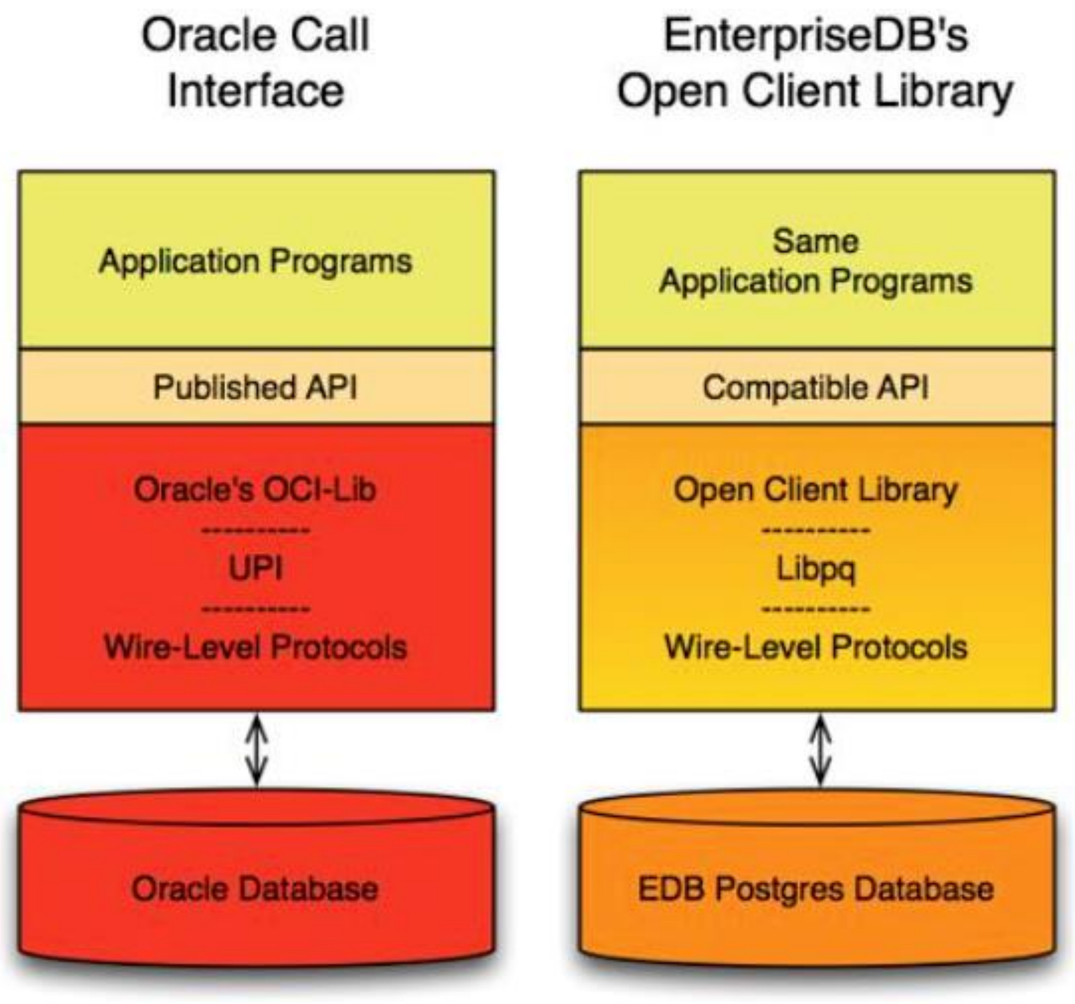


Figure 2.1 – The Open Client Library.

For detailed information about the functions supported by the Open Client Library, see the EDB Postgres Advanced Server OCL Connector Guide, available at:

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

2.9 Utilities

For detailed information about the compatible syntax supported by the utilities listed below, see the *Database Compatibility for Oracle Developers Tools and Utilities Guide*, available at:

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

*EDB*Plus*

EDB*Plus is a utility program that provides a command line user interface to the Advanced Server that will be familiar to Oracle developers and users. EDB*Plus accepts SQL commands, SPL anonymous blocks, and EDB*Plus commands.

EDB*Plus allows you to:

- Query certain database objects
- Execute stored procedures
- Format output from SQL commands
- Execute batch scripts
- Execute OS commands
- Record output

For detailed information about EDB*Plus, please see the *EDB*Plus User's Guide* available at:

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

*EDB*Loader*

EDB*Loader is a high-performance bulk data loader that provides an interface compatible with Oracle databases for Advanced Server. The EDB*Loader command line utility loads data from an input source, typically a file, into one or more tables using a subset of the parameters offered by Oracle SQL*Loader.

EDB*Loader features include:

- Support for the Oracle SQL*Loader data loading methods - conventional path load, direct path load, and parallel direct path load
- Oracle SQL*Loader compatible syntax for control file directives
- Input data with delimiter-separated or fixed-width fields
- Bad file for collecting rejected records
- Loading of multiple target tables

- Discard file for collecting records that do not meet the selection criteria of any target table
- Log file for recording the EDB*Loader session and any error messages
- Data loading from standard input and remote loading

*EDB*Wrap*

The EDB*Wrap utility protects proprietary source code and programs (functions, stored procedures, triggers, and packages) from unauthorized scrutiny. The EDB*Wrap program translates a file that contains SPL or PL/pgSQL source code (the plaintext) into a file that contains the same code in a form that is nearly impossible to read. Once you have the obfuscated form of the code, you can send that code to Advanced Server and it will store those programs in obfuscated form. While EDB*Wrap does obscure code, table definitions are still exposed.

Everything you wrap is stored in obfuscated form. If you wrap an entire package, the package body source, as well as the prototypes contained in the package header and the functions and procedures contained in the package body are stored in obfuscated form.

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.

2.10 ECPGPlus

EnterpriseDB has enhanced ECPG (the PostgreSQL pre-compiler) to create ECPGPlus. ECPGPlus allows you to include embedded SQL commands in C applications; when you use ECPGPlus to compile an application that contains embedded SQL commands, the SQL code is syntax-checked and translated into C.

ECPGPlus supports Pro*C syntax in C programs when connected to an Advanced Server database. ECPGPlus supports:

- Oracle Dynamic SQL – Method 4 (ODS-M4)
- Pro*C compatible anonymous blocks
- A `CALL` statement compatible with Oracle databases

For information about using ECPGPlus, please see the *EDB Postgres Advanced Server ECPG Connector Guide*, available from the EnterpriseDB website at:

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

2.11 Table Partitioning

In a partitioned table, one logically large table is broken into smaller physical pieces. Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning allows you to omit the partition column from the front of an index, reducing index size and making it more likely that the heavily used parts of the index fits in memory.

- When a query or update accesses a large percentage of a single partition, performance may improve because the server will perform a sequential scan of the partition instead of using an index and random access reads scattered across the whole table.

- A bulk load (or unload) can be implemented by adding or removing partitions, if you plan that requirement into the partitioning design. `ALTER TABLE` is far faster than a bulk operation. It also entirely avoids the `VACUUM` overhead caused by a bulk `DELETE`.

- Seldom-used data can be migrated to less-expensive (or slower) storage media.

Table partitioning is worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application; a good rule of thumb is that the size of the table should exceed the physical memory of the database server.

For information about database compatibility features supported by Advanced Server see the *Database Compatibility for Oracle Developer's Guide*, available at:

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

3 Database Administration

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

3.1 Configuration Parameters

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

Configuration parameters that apply only to Advanced Server are noted in Section 3.1.2.

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

https://www.postgresql.org/docs/12/static/runtime-config.html

3.1.1 Setting Configuration Parameters

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

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

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

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

Parameter values are specified as one of five types:

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

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

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

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

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

- The initial settings for almost all configurable parameters across the entire database cluster are listed in the configuration file, `postgresql.conf`. These settings are put into effect upon database server start or restart. Some of these initial parameter settings can be overridden as discussed in the following bullet points. All configuration parameters have built-in default settings that are in effect if not explicitly overridden.
- Configuration parameters in the `postgresql.conf` file are overridden when the same parameters are included in the `postgresql.auto.conf` file. The `ALTER SYSTEM` command is used to manage the configuration parameters in the `postgresql.auto.conf` file.
- Parameter settings can be modified in the configuration file while the database server is running. If the configuration file is then reloaded (meaning a SIGHUP signal is issued), for certain parameter types, the changed parameters settings immediately take effect. For some of these parameter types, the new settings are available in a currently running session immediately after the reload. For other of these parameter types, a new session must be started to use the new settings. And yet for other parameter types, modified settings do not take effect until the database server is stopped and restarted. See 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.

3.1.2 Summary of Configuration Parameters

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

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

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

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

Table 3-1 - Summary of Configuration Parameters

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

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
			account		
autovacuum_vacuum_cost_limit	Cluster	Reload	EPAS service account	Vacuum cost amount available before napping, for autovacuum.	
autovacuum_vacuum_scale_factor	Cluster	Reload	EPAS service account	Number of tuple updates or deletes prior to vacuum as a fraction of `reltuples`.	
autovacuum_vacuum_threshold	Cluster	Reload	EPAS service account	Minimum number of tuple updates or deletes prior to vacuum.	
autovacuum_work_mem	Cluster	Reload	EPAS service account	Sets the maximum memory to be used by each autovacuum worker process.	
backslash_quote	Session	Immediate	User	Sets whether "`\'`" is allowed in string literals.	
bgwriter_delay	Cluster	Reload	EPAS service account	Background writer sleep time between rounds.	
bgwriter_lru_maxpages	Cluster	Reload	EPAS service account	Background writer maximum number of LRU pages to flush per round.	
bgwriter_lru_multiplier	Cluster	Reload	EPAS service account	Multiple of the average buffer usage to free per round.	
block_size	Cluster	Preset	n/a	Shows the size of a disk block.	
bonjour	Cluster	Restart	EPAS service account	Enables advertising the server via Bonjour.	
bonjour_name	Cluster	Restart	EPAS service account	Sets the Bonjour service name.	
bytea_output	Session	Immediate	User	Sets the output format for `bytea`.	
check_function_bodies	Session	Immediate	User	Check function bodies during `CREATE FUNCTION`.	
checkpoint_completion_target	Cluster	Reload	EPAS service account	Time spent flushing dirty buffers during checkpoint, as fraction of checkpoint interval.	
checkpoint_timeout	Cluster	Reload	EPAS service account	Sets the maximum time between automatic WAL checkpoints.	
checkpoint_warning	Cluster	Reload	EPAS service account	Enables warnings if checkpoint segments are filled more frequently than this.	
client_encoding	Session	Immediate	User	Sets the client's character set encoding.	
client_min_messages	Session	Immediate	User	Sets the message levels that	

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

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

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
edb_audit_directory	Cluster	Reload	EPAS service account	Sets the destination directory for audit files.	X
edb_audit_disconnect	Cluster	Reload	EPAS service account	Audits end of a session.	X
edb_audit_filename	Cluster	Reload	EPAS service account	Sets the file name pattern for audit files.	X
edb_audit_rotation_day	Cluster	Reload	EPAS service account	Automatic rotation of log files based on day of week.	X
edb_audit_rotation_seconds	Cluster	Reload	EPAS service account	Automatic log file rotation will occur after N seconds.	X
edb_audit_rotation_size	Cluster	Reload	EPAS service account	Automatic log file rotation will occur after N Megabytes.	X
edb_audit_statement	Cluster	Reload	EPAS service account	Sets the type of statements to audit.	X
edb_audit_tag	Session	Immediate	User	Specify a tag to be included in the audit log.	X
edb_connectby_order	Session	Immediate	User	Sort results of `CONNECT BY` queries with no `ORDER BY` to depth-first order. Note: For internal use only.	X
edb_custom_plan_tries	Session	Immediate	User	Specifies the number of custom execution plans considered by the planner before the planner selects a generic execution plan.	X
edb_data_redaction	Session	Immediate	User	Enable data redaction.	X
edb_dynatune	Cluster	Restart	EPAS service account	Sets the edb utilization percentage.	X
edb_dynatune_profile	Cluster	Restart	EPAS service account	Sets the workload profile for dynatune.	X
edb_enable_pruning	Session	Immediate	User	Enables the planner to early-prune partitioned tables.	X
edb_log_every_bulk_value	Session	Immediate	Superuser	Sets the statements logged for bulk processing.	X
edb_max_resource_groups	Cluster	Restart	EPAS service account	Specifies the maximum number of resource groups for simultaneous use.	X
edb_max_spins_per_delay	Cluster	Restart	EPAS service	Specifies the number of times a session will spin while	X

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
			account	waiting for a lock.	
edb_redwood_date	Session	Immediate	User	Determines whether DATE should behave like a TIMESTAMP or not.	X
edb_redwood_greatest_least	Session	Immediate	User	Determines how GREATEST and LEAST functions should handle NULL parameters.	X
edb_redwood_raw_names	Session	Immediate	User	Return the unmodified name stored in the PostgreSQL system catalogs from Redwood interfaces.	X
edb_redwood_strings	Session	Immediate	User	Treat NULL as an empty string when concatenated with a text value.	X
edb_resource_group	Session	Immediate	User	Specifies the resource group to be used by the current process.	X
edb_sql_protect.enabled	Cluster	Reload	EPAS service account	Defines whether SQL/Protect should track queries or not.	X
edb_sql_protect.level	Cluster	Reload	EPAS service account	Defines the behavior of SQL/Protect when an event is found.	X
edb_sql_protect.max_protected_relations	Cluster	Restart	EPAS service account	Sets the maximum number of relations protected by SQL/Protect per role.	X
edb_sql_protect.max_protected_roles	Cluster	Restart	EPAS service account	Sets the maximum number of roles protected by SQL/Protect.	X
edb_sql_protect.max_queries_to_save	Cluster	Restart	EPAS service account	Sets the maximum number of offending queries to save by SQL/Protect.	X
edb_stmt_level_tx	Session	Immediate	User	Allows continuing on errors instead of requiring a transaction abort.	X
edb_wait_states.directory	Cluster	Restart	EPAS service account	The EDB wait states logs are stored in this directory.	X
edb_wait_states.retention_period	Cluster	Reload	EPAS service account	EDB wait state log files will be automatically deleted after retention period.	X
edb_wait_states.sampling_interval	Cluster	Reload	EPAS service account	The interval between two EDB wait state sampling cycles.	X
edbldr.empty_csv_field	Session	Immediate	Superuser	Specifies how EDB*Loader handles empty strings.	X
effective_cache_size	Session	Immediate	User	Sets the planner's assumption about the size of the disk cache.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
`effective_io_concurrency`	Session	Immediate	User	Number of simultaneous requests that can be handled efficiently by the disk subsystem.	
`enable_bitmapscan`	Session	Immediate	User	Enables the planner's use of bitmap-scan plans.	
`enable_hashagg`	Session	Immediate	User	Enables the planner's use of hashed aggregation plans.	
`enable_hashjoin`	Session	Immediate	User	Enables the planner's use of hash join plans.	
`enable_hints`	Session	Immediate	User	Enable optimizer hints in SQL statements.	X
`enable_indexonlyscan`	Session	Immediate	User	Enables the planner’s use of index-only-scan plans.	
`enable_indexscan`	Session	Immediate	User	Enables the planner's use of index-scan plans.	
`enable_material`	Session	Immediate	User	Enables the planner's use of materialization.	
`enable_mergejoin`	Session	Immediate	User	Enables the planner's use of merge join plans.	
`enable_nestloop`	Session	Immediate	User	Enables the planner's use of nested-loop join plans.	
`enable_seqscan`	Session	Immediate	User	Enables the planner's use of sequential-scan plans.	
`enable_sort`	Session	Immediate	User	Enables the planner's use of explicit sort steps.	
`enable_tidscan`	Session	Immediate	User	Enables the planner's use of TID scan plans.	
`escape_string_warning`	Session	Immediate	User	Warn about backslash escapes in ordinary string literals.	
`event_source`	Cluster	Restart	EPAS service account	Sets the application name used to identify PostgreSQL messages in the event log.	
`exit_on_error`	Session	Immediate	User	Terminate session on any error.	
`external_pid_file`	Cluster	Restart	EPAS service account	Writes the postmaster PID to the specified file.	
`extra_float_digits`	Session	Immediate	User	Sets the number of digits displayed for floating-point values.	
`from_collapse_limit`	Session	Immediate	User	Sets the FROM-list size beyond which subqueries are not collapsed.	
`fsync`	Cluster	Reload	EPAS service account	Forces synchronization of updates to disk.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
full_page_writes	Cluster	Reload	EPAS 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.	
geqo_seed	Session	Immediate	User	GEQO: seed for random path selection.	
geqo_selection_bias	Session	Immediate	User	GEQO: selective pressure within the population.	
geqo_threshold	Session	Immediate	User	Sets the threshold of FROM items beyond which GEQO is used.	
gin_fuzzy_search_limit	Session	Immediate	User	Sets the maximum allowed result for exact search by GIN.	
hba_file	Cluster	Restart	EPAS service account	Sets the server's "hba" configuration file.	
hot_standby	Cluster	Restart	EPAS service account	Allows connections and queries during recovery.	
hot_standby_feedback	Cluster	Reload	EPAS service account	Allows feedback from a hot standby to the primary that will avoid query conflicts.	
huge_pages	Cluster	Restart	EPAS service account	Use of huge pages on Linux.	
icu_short_form	Database	Preset	n/a	Shows the ICU collation order configuration.	X
ident_file	Cluster	Restart	EPAS service account	Sets the server's "ident" configuration file.	
ignore_checksum_failure	Session	Immediate	Superuser	Continues processing after a checksum failure.	
ignore_system_indexes	Cluster/ Session	Reload/ Immediate	EPAS service account/ User	Disables reading from system indexes. (Can also be set with PGOPTIONS at session start.)	
index_advisor.enabled	Session	Immediate	User	Enable Index Advisor plugin.	X
integer_datetimes	Cluster	Preset	n/a	Datetimes are integer based.	
intervalStyle	Session	Immediate	User	Sets the display format for	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
				interval values.	
`join_collapse_limit`	Session	Immediate	User	Sets the `FROM`-list size beyond which `JOIN` constructs are not flattened.	
`krb_caseins_users`	Cluster	Reload	EPAS service account	Sets whether Kerberos and GSSAPI user names should be treated as case-insensitive.	
`krb_server_keyfile`	Cluster	Reload	EPAS service account	Sets the location of the Kerberos server key file.	
`lc_collate`	Database	Preset	n/a	Shows the collation order locale.	
`lc_ctype`	Database	Preset	n/a	Shows the character classification and case conversion locale.	
`lc_messages`	Session	Immediate	Superuser	Sets the language in which messages are displayed.	
`lc_monetary`	Session	Immediate	User	Sets the locale for formatting monetary amounts.	
`lc_numeric`	Session	Immediate	User	Sets the locale for formatting numbers.	
`lc_time`	Session	Immediate	User	Sets the locale for formatting date and time values.	
`listen_addresses`	Cluster	Restart	EPAS service account	Sets the host name or IP address(es) to listen to.	
`local_preload_libraries`	Cluster/ Session	Reload/ Immediate	EPAS service account/ User	Lists shared libraries to preload into each backend. (Can also be set with `PGOPTIONS` at session start.)	
`lock_timeout`	Session	Immediate	User	Sets the maximum time allowed that a statement may wait for a lock.	
`lo_compat_privileges`	Session	Immediate	Superuser	Enables backward compatibility mode for privilege checks on large objects.	
`log_autovacuum_min_duration`	Cluster	Reload	EPAS service account	Sets the minimum execution time above which autovacuum actions will be logged.	
`log_checkpoints`	Cluster	Reload	EPAS service account	Logs each checkpoint.	
`log_connections`	Cluster/ Session	Reload/ Immediate	EPAS service account/ User	Logs each successful connection. (Can also be set with `PGOPTIONS` at session start.)	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
`log_destination`	Cluster	Reload	EPAS service account	Sets the destination for server log output.	
`log_directory`	Cluster	Reload	EPAS service account	Sets the destination directory for log files.	
`log_disconnections`	Cluster/ Session	Reload/ Immediate	EPAS service account/ User	Logs end of a session, including duration. (Can also be set with `PGOPTIONS` at session start.)	
`log_duration`	Session	Immediate	Superuser	Logs the duration of each completed SQL statement.	
`log_error_verbosity`	Session	Immediate	Superuser	Sets the verbosity of logged messages.	
`log_executor_stats`	Session	Immediate	Superuser	Writes executor performance statistics to the server log.	
`log_file_mode`	Cluster	Reload	EPAS service account	Sets the file permissions for log files.	
`log_filename`	Cluster	Reload	EPAS service account	Sets the file name pattern for log files.	
`log_hostname`	Cluster	Reload	EPAS service account	Logs the host name in the connection logs.	
`log_line_prefix`	Cluster	Reload	EPAS service account	Controls information prefixed to each log line.	
`log_lock_waits`	Session	Immediate	Superuser	Logs long lock waits.	
`log_min_duration_stateme nt`	Session	Immediate	Superuser	Sets the minimum execution time above which statements will be logged.	
`log_min_error_statement`	Session	Immediate	Superuser	Causes all statements generating error at or above this level to be logged.	
`log_min_messages`	Session	Immediate	Superuser	Sets the message levels that are logged.	
`log_parser_stats`	Session	Immediate	Superuser	Writes parser performance statistics to the server log.	
`log_planner_stats`	Session	Immediate	Superuser	Writes planner performance statistics to the server log.	
`log_rotation_age`	Cluster	Reload	EPAS service account	Automatic log file rotation will occur after N minutes.	
`log_rotation_size`	Cluster	Reload	EPAS service account	Automatic log file rotation will occur after N kilobytes.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
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	EPAS service account	Sets the time zone to use in log messages.	
log_truncate_on_rotation	Cluster	Reload	EPAS service account	Truncate existing log files of same name during log rotation.	
logging_collector	Cluster	Restart	EPAS service account	Start a subprocess to capture stderr output and/or csvlogs into log files.	
maintenance_work_mem	Session	Immediate	User	Sets the maximum memory to be used for maintenance operations.	
max_connections	Cluster	Restart	EPAS service account	Sets the maximum number of concurrent connections.	
max_files_per_process	Cluster	Restart	EPAS service account	Sets the maximum number of simultaneously open files for each server process.	
max_function_args	Cluster	Preset	n/a	Shows the maximum number of function arguments.	
max_identifier_length	Cluster	Preset	n/a	Shows the maximum identifier length.	
max_index_keys	Cluster	Preset	n/a	Shows the maximum number of index keys.	
max_locks_per_transaction	Cluster	Restart	EPAS service account	Sets the maximum number of locks per transaction.	
max_pred_locks_per_transaction	Cluster	Restart	EPAS service account	Sets the maximum number of predicate locks per transaction.	
max_prepared_transactions	Cluster	Restart	EPAS service account	Sets the maximum number of simultaneously prepared transactions.	
max_replication_slots	Cluster	Restart	EPAS service account	Sets the maximum number of simultaneously defined replication slots.	
max_stack_depth	Session	Immediate	Superuser	Sets the maximum stack depth, in kilobytes.	
max_standby_archive_delay	Cluster	Reload	EPAS	Sets the maximum delay	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
			service account	before canceling queries when a hot standby server is processing archived WAL data.	
max_standby_streaming_delay	Cluster	Reload	EPAS service account	Sets the maximum delay before canceling queries when a hot standby server is processing streamed WAL data.	
max_wal_senders	Cluster	Restart	EPAS service account	Sets the maximum number of simultaneously running WAL sender processes.	
max_wal_size	Cluster	Reload	EPAS service account	Sets the maximum size to which the WAL will grow between automatic WAL checkpoints. The default is 1GB.	
max_worker_processes	Cluster	Restart	EPAS service account	Maximum number of concurrent worker processes.	
min_wal_size	Cluster	Reload	EPAS service account	Sets the threshold at which WAL logs will be recycled rather than removed. The default is 80 MB.	
nls_length_semantics	Session	Immediate	Superuser	Sets the semantics to use for char, varchar, varchar2 and long columns.	X
odbc_lib_path	Cluster	Restart	EPAS service account	Sets the path for ODBC library.	X
optimizer_mode	Session	Immediate	User	Default optimizer mode.	X
oracle_home	Cluster	Restart	EPAS service account	Sets the path for the Oracle home directory.	X
password_encryption	Session	Immediate	User	Encrypt passwords.	
pg_prewarm.autoprewarm	Cluster	Restart	EPAS service account	Enables the autoprewarm background worker.	X
pg_prewarm.autoprewarm_interval	Cluster	Reload	EPAS service account	Sets the minimum number of seconds after which autoprewarm dumps shared buffers.	X
port	Cluster	Restart	EPAS service account	Sets the TCP port on which the server listens.	
post_auth_delay	Cluster/ Session	Reload/ Immediate	EPAS service account/	Waits N seconds on connection startup after authentication. (Can also be	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
			User	set with PGOPTIONS at session start.)	
pre_auth_delay	Cluster	Reload	EPAS service account	Waits N seconds on connection startup before authentication.	
qreplace_function	Session	Immediate	Superuser	The function to be used by Query Replace feature. Note: For internal use only.	X
query_rewrite_enabled	Session	Immediate	User	Child table scans will be skipped if their constraints guarantee that no rows match the query.	X
query_rewrite_integrity	Session	Immediate	Superuser	Sets the degree to which query rewriting must be enforced.	X
quote_all_identifiers	Session	Immediate	User	When generating SQL fragments, quote all identifiers.	
random_page_cost	Session	Immediate	User	Sets the planner's estimate of the cost of a nonsequentially fetched disk page.	
restart_after_crash	Cluster	Reload	EPAS service account	Reinitialize server after backend crash.	
search_path	Session	Immediate	User	Sets the schema search order for names that are not schema-qualified.	
segment_size	Cluster	Preset	n/a	Shows the number of pages per disk file.	
seq_page_cost	Session	Immediate	User	Sets the planner's estimate of the cost of a sequentially fetched disk page.	
server_encoding	Database	Preset	n/a	Sets the server (database) character set encoding.	
server_version	Cluster	Preset	n/a	Shows the server version.	
server_version_num	Cluster	Preset	n/a	Shows the server version as an integer.	
session_preload_libraries	Session	Immediate but only at connection start	Superuser	Lists shared libraries to preload into each backend.	
session_replication_role	Session	Immediate	Superuser	Sets the session's behavior for triggers and rewrite rules.	
shared_buffers	Cluster	Restart	EPAS service account	Sets the number of shared memory buffers used by the server.	
shared_preload_libraries	Cluster	Restart	EPAS service	Lists shared libraries to preload into server.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
			account		
sql_inheritance	Session	Immediate	User	Causes subtables to be included by default in various commands.	
ssl	Cluster	Restart	EPAS service account	Enables SSL connections.	
ssl_ca_file	Cluster	Restart	EPAS service account	Location of the SSL certificate authority file.	
ssl_cert_file	Cluster	Restart	EPAS service account	Location of the SSL server certificate file.	
ssl_ciphers	Cluster	Restart	EPAS service account	Sets the list of allowed SSL ciphers.	
ssl_crl_file	Cluster	Restart	EPAS service account	Location of the SSL certificate revocation list file.	
ssl_ecdh_curve	Cluster	Restart	EPAS service account	Sets the curve to use for ECDH.	
ssl_key_file	Cluster	Restart	EPAS service account	Location of the SSL server private key file.	
ssl_prefer_server_ciphers	Cluster	Restart	EPAS service account	Give priority to server ciphersuite order.	
ssl_renegotiation_limit	Session	Immediate	User	Set the amount of traffic to send and receive before renegotiating the encryption keys.	
standard_conforming_strings	Session	Immediate	User	Causes '...' strings to treat backslashes literally.	
statement_timeout	Session	Immediate	User	Sets the maximum allowed duration of any statement.	
stats_temp_directory	Cluster	Reload	EPAS service account	Writes temporary statistics files to the specified directory.	
superuser_reserved_connections	Cluster	Restart	EPAS service account	Sets the number of connection slots reserved for superusers.	
synchronize_seqscans	Session	Immediate	User	Enable synchronized sequential scans.	
synchronous_commit	Session	Immediate	User	Sets immediate fsync at commit.	
synchronous_standby_names	Cluster	Reload	EPAS	List of names of potential	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
			service account	synchronous standbys.	
syslog_facility	Cluster	Reload	EPAS service account	Sets the syslog "facility" to be used when syslog enabled.	
syslog_ident	Cluster	Reload	EPAS service account	Sets the program name used to identify PostgreSQL messages in syslog.	
tcp_keepalives_count	Session	Immediate	User	Maximum number of TCP keepalive retransmits.	
tcp_keepalives_idle	Session	Immediate	User	Time between issuing TCP keepalives.	
tcp_keepalives_interval	Session	Immediate	User	Time between TCP keepalive retransmits.	
temp_buffers	Session	Immediate	User	Sets the maximum number of temporary buffers used by each session.	
temp_file_limit	Session	Immediate	Superuser	Limits the total size of all temporary files used by each session.	
temp_tablespaces	Session	Immediate	User	Sets the tablespace(s) to use for temporary tables and sort files.	
timed_statistics	Session	Immediate	User	Enables the recording of timed statistics.	X
timezone	Session	Immediate	User	Sets the time zone for displaying and interpreting time stamps.	
timezone_abbreviations	Session	Immediate	User	Selects a file of time zone abbreviations.	
trace_hints	Session	Immediate	User	Emit debug info about hints being honored.	X
trace_notify	Session	Immediate	User	Generates debugging output for `LISTEN` and `NOTIFY`.	
trace_recovery_messages	Cluster	Reload	EPAS service account	Enables logging of recovery-related debugging information.	
trace_sort	Session	Immediate	User	Emit information about resource usage in sorting.	
track_activities	Session	Immediate	Superuser	Collects information about executing commands.	
track_activity_query_size	Cluster	Restart	EPAS service account	Sets the size reserved for `pg_stat_activity.current_query`, in bytes.	
track_counts	Session	Immediate	Superuser	Collects statistics on database activity.	
track_functions	Session	Immediate	Superuser	Collects function-level	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
				statistics on database activity.	
`track_io_timing`	Session	Immediate	Superuser	Collects timing statistics for database I/O activity.	
`transaction_deferrable`	Session	Immediate	User	Whether to defer a read-only serializable transaction until it can be executed with no possible serialization failures.	
`transaction_isolation`	Session	Immediate	User	Sets the current transaction's isolation level.	
`transaction_read_only`	Session	Immediate	User	Sets the current transaction's read-only status.	
`transform_null_equals`	Session	Immediate	User	Treats "`expr=NULL`" as "`expr IS NULL`".	
`unix_socket_directories`	Cluster	Restart	EPAS service account	Sets the directory where the Unix-domain socket will be created.	
`unix_socket_group`	Cluster	Restart	EPAS service account	Sets the owning group of the Unix-domain socket.	
`unix_socket_permissions`	Cluster	Restart	EPAS service account	Sets the access permissions of the Unix-domain socket.	
`update_process_title`	Session	Immediate	Superuser	Updates the process title to show the active SQL command.	
`utl_encode.uudecode_redwood`	Session	Immediate	User	Allows decoding of Oracle-created uuencoded data.	X
`utl_file.umask`	Session	Immediate	User	Umask used for files created through the `UTL_FILE` package.	X
`vacuum_cost_delay`	Session	Immediate	User	Vacuum cost delay in milliseconds.	
`vacuum_cost_limit`	Session	Immediate	User	Vacuum cost amount available before napping.	
`vacuum_cost_page_dirty`	Session	Immediate	User	Vacuum cost for a page dirtied by vacuum.	
`vacuum_cost_page_hit`	Session	Immediate	User	Vacuum cost for a page found in the buffer cache.	
`vacuum_cost_page_miss`	Session	Immediate	User	Vacuum cost for a page not found in the buffer cache.	
`vacuum_defer_cleanup_age`	Cluster	Reload	EPAS service account	Number of transactions by which VACUUM and HOT cleanup should be deferred, if any.	
`vacuum_freeze_min_age`	Session	Immediate	User	Minimum age at which VACUUM should freeze a table row.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
vacuum_freeze_table_age	Session	Immediate	User	Age at which VACUUM should scan whole table to freeze tuples.	
vacuum_multixact_freeze_min_age	Session	Immediate	User	Minimum age at which VACUUM should freeze a MultiXactId in a table row.	
vacuum_multixact_freeze_table_age	Session	Immediate	User	Multixact age at which VACUUM should scan whole table to freeze tuples.	
wal_block_size	Cluster	Preset	n/a	Shows the block size in the write ahead log.	
wal_buffers	Cluster	Restart	EPAS service account	Sets the number of disk-page buffers in shared memory for WAL.	
wal_keep_segments	Cluster	Reload	EPAS service account	Sets the number of WAL files held for standby servers.	
wal_level	Cluster	Restart	EPAS service account	Set the level of information written to the WAL.	
wal_log_hints	Cluster	Restart	EPAS service account	Writes full pages to WAL when first modified after a checkpoint, even for non-critical modifications.	
wal_receiver_status_interval	Cluster	Reload	EPAS service account	Sets the maximum interval between WAL receiver status reports to the primary.	
wal_receiver_timeout	Cluster	Reload	EPAS service account	Sets the maximum wait time to receive data from the primary.	
wal_segment_size	Cluster	Preset	n/a	Shows the number of pages per write ahead log segment.	
wal_sender_timeout	Cluster	Reload	EPAS service account	Sets the maximum time to wait for WAL replication.	
wal_sync_method	Cluster	Reload	EPAS service account	Selects the method used for forcing WAL updates to disk.	
wal_writer_delay	Cluster	Reload	EPAS service account	WAL writer sleep time between WAL flushes.	
work_mem	Session	Immediate	User	Sets the maximum memory to be used for query workspaces.	
xloginsert_locks	Cluster	Restart	EPAS service account	Sets the number of locks used for concurrent xlog insertions.	
xmlbinary	Session	Immediate	User	Sets how binary values are to be encoded in XML.	

Parameter	Scope of Effect	When Takes Effect	Authorized User	Description	EPAS Only
`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.	

3.1.3 Configuration Parameters by Functionality

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

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

- **Parameter Type.** Type of values the parameter can accept. See Section 3.1.1 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 3.1.2.)
- **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 3.1.2.)
- **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 EPAS service account (`enterprisedb` or `postgres` depending upon the Advanced Server compatibility installation mode). This attribute has the same meaning as the "Authorized User" column in the table of Section 3.1.2.

3.1.3.1 Top Performance Related Parameters

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

3.1.3.1.1 shared_buffers

Parameter Type: Integer

Default Value: 32MB

Range: 128kB to system dependent

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: EPAS service account

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

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

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

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

3.1.3.1.2 work_mem

Parameter Type: Integer

Default Value: 1MB

Range: 64kB to 2097151kB

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

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

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

3.1.3.1.3 maintenance_work_mem

Parameter Type: Integer

Default Value: 16MB

Range: 1024kB to 2097151kB

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

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

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.

3.1.3.1.4 wal_buffers

Parameter Type: Integer

Default Value: 64kB

Range: 32kB to system dependent

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: EPAS service account

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

Increasing this parameter might cause Advanced Server to request more System V shared memory than your operating system's default configuration allows. See 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 Advanced Server’s initial shared memory allocation, so if increasing it causes an Advanced Server start failure, you will need to increase the operating system limit.

3.1.3.1.5 checkpoint_segments

Now deprecated; this parameter is not supported by Advanced Server.

3.1.3.1.6 checkpoint_completion_target

Parameter Type: Floating point

Default Value: 0.5

Range: 0 to 1

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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.

3.1.3.1.7 checkpoint_timeout

Parameter Type: Integer

Default Value: 5min

Range: 30s to 3600s

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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

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

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

3.1.3.1.8 max_wal_size

Parameter Type: Integer

Default Value: 1 GB

Range: 2 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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

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

3.1.3.1.9 min_wal_size

Parameter Type: Integer

Default Value: 80 MB

Range: 2 to 2147483647

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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

3.1.3.1.10 bgwriter_delay

Parameter Type: Integer

Default Value: 200ms

Range: 10ms to 10000ms

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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

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

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

3.1.3.1.11 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.

3.1.3.1.12 random_page_cost

Parameter Type: Floating point

Default Value: 4

Range: 0 to 1.79769e+308

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

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

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

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

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

3.1.3.1.13 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 Advanced Server's shared buffers and the portion of the kernel's disk cache that will be used for Advanced Server data files. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by Advanced Server, nor does it reserve kernel disk cache; it is used only for estimation purposes. The default is 128 megabytes (`128MB`).

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

3.1.3.1.14 synchronous_commit

Parameter Type: Boolean

Default Value: `true`

Range: {`true | false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

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

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

So, turning `synchronous_commit` off can be a useful alternative when performance is more important than exact certainty about the durability of a transaction. See 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.

3.1.3.1.15 edb_max_spins_per_delay

Parameter Type: Integer

Default Value: 1000

Range: 10 to 1000

Minimum Scope of Effect: Per cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: EPAS service account

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

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

3.1.3.1.16 pg_prewarm.autoprewarm

Parameter Type: Boolean

Default Value: `true`

Range: {`true` | `false`}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: EPAS service account

This parameter controls whether or not the database server should run *autoprewarm*, which is a background worker process that automatically dumps shared buffers to disk before a shutdown. It then *prewarms* the shared buffers the next time the server is started, meaning it loads blocks from the disk back into the buffer pool.

The advantage is that it shortens the warm up times after the server has been restarted by loading the data that has been dumped to disk before shutdown.

If `pg_prewarm.autoprewarm` is set to on, the `autoprewarm` worker is enabled. If the parameter is set to off, `autoprewarm` is disabled. The parameter is on by default.

Before `autoprewarm` can be used, you must add `$libdir/pg_prewarm` to the libraries listed in the `shared_preload_libraries` configuration parameter of the `postgresql.conf` file as shown by the following example:

```
shared_preload_libraries =
'$libdir/dbms_pipe,$libdir/edb_gen,$libdir/dbms_aq,$libdir/pg_prewarm'
```

After modifying the `shared_preload_libraries` parameter, restart the database server after which the `autoprewarm` background worker is launched immediately after the server has reached a consistent state.

The `autoprewarm` process will start loading blocks that were previously recorded in `$PGDATA/autoprewarm.blocks` until there is no free buffer space left in the buffer pool. In this manner, any new blocks that were loaded either by the recovery process or by the querying clients, are not replaced.

Once the `autoprewarm` process has finished loading buffers from disk, it will periodically dump shared buffers to disk at the interval specified by the `pg_prewarm.autoprewarm_interval` parameter (see Section 3.1.3.1.17). Upon the next server restart, the `autoprewarm` process will prewarm shared buffers with the blocks that were last dumped to disk.

3.1.3.1.17 pg_prewarm.autoprewarm_interval

Parameter Type: Integer

Default Value: 300s

Range: 0s to 2147483s

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

This is the minimum number of seconds after which the `autoprewarm` background worker dumps shared buffers to disk. The default is 300 seconds. If set to 0, shared buffers are not dumped at regular intervals, but only when the server is shut down.

See Section 3.1.3.1.16 for information on the `autoprewarm` background worker.

3.1.3.2 Resource Usage / Memory

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

3.1.3.2.1 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: EPAS service account

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

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

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

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

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

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

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

3.1.3.2.2 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: EPAS service account

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

The following are the possible values:

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

3.1.3.3 Resource Usage / EDB Resource Manager

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

3.1.3.3.1 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: EPAS service account

This parameter controls the maximum number of resource groups that can be used simultaneously by EDB Resource Manager. More resource groups can 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.

3.1.3.3.2 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.

3.1.3.4 Query Tuning

This section describes the configuration parameters used for optimizer hints.

3.1.3.4.1 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.

3.1.3.5 Query Tuning / Planner Method Configuration

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

3.1.3.5.1 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`).

3.1.3.6 Reporting and Logging / What to Log

The configuration parameters in this section control reporting and logging.

3.1.3.6.1 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)
```

3.1.3.6.2 edb_log_every_bulk_value

Parameter Type: Boolean

Default Value: `false`

Range: {`true | false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Superuser

Bulk processing logs the resulting statements into both the Advanced Server log file and the EDB Audit log file. However, logging each and every statement in bulk processing is costly. This can be controlled by the `edb_log_every_bulk_value` configuration parameter. When set to `true`, each and every statement in bulk processing is logged. When set to `false`, a log message is recorded once per bulk processing. In addition, the duration is emitted once per bulk processing. Default is set to `false`.

3.1.3.7 Auditing Settings

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

3.1.3.7.1 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: EPAS service account

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

3.1.3.7.2 edb_audit_directory

Parameter Type: String

Default Value: `edb_audit`

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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

3.1.3.7.3 edb_audit_filename

Parameter Type: String

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

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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

3.1.3.7.4 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: EPAS service account

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

3.1.3.7.5 edb_audit_rotation_size

Parameter Type: Integer

Default Value: 0MB

Range: 0MB to 5000MB

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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

3.1.3.7.6 edb_audit_rotation_seconds

Parameter Type: Integer

Default Value: 0s

Range: 0s to 2147483647s

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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

3.1.3.7.7 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: EPAS service account

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

3.1.3.7.8 edb_audit_disconnect

Parameter Type: Enum

Default Value: `none`

Range: {`none` | `all`}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

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

3.1.3.7.9 edb_audit_statement

Parameter Type: String

Default Value: `ddl, error`

Range: {`none` | `ddl` | `dml` | `insert` | `update` | `delete` | `truncate` | `select` | `error` | `create` | `drop` | `alter` | `grant` | `revoke` | `rollback` | `all`} ...

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

This configuration parameter is used to specify auditing of different categories of SQL statements as well as those statements related to specific SQL commands. To log errors, set the parameter value to `error`. To audit all DDL statements such as `CREATE TABLE`, `ALTER TABLE`, etc., set the parameter value to `ddl`. To audit specific types of DDL statements, the parameter values can include those specific SQL commands (`create`, `drop`, or `alter`). In addition, the object type may be specified following the command such as `create table`, `create view`, `drop role`, etc. All modification statements such as `INSERT`, `UPDATE`, `DELETE` or `TRUNCATE` can be audited by setting `edb_audit_statement` to `dml`. To audit specific types of DML statements, the parameter values can include the specific SQL commands, `insert`, `update`, `delete`, or `truncate`. Include parameter values `select`, `grant`, `revoke`, or `rollback` to audit statements regarding those SQL commands. Setting the value to `all` will audit every statement while `none` disables this feature.

3.1.3.7.10 edb_audit_tag

Parameter Type: String

Default Value: none

Minimum Scope of Effect: Session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: User

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

3.1.3.7.11 edb_audit_destination

Parameter Type: Enum

Default Value: `file`

Range: {`file` | `syslog`}

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

Specifies whether the audit log information is to be recorded in the directory as given by the `edb_audit_directory` parameter or to the directory and file managed by the *syslog* process. Set to `file` to use the directory specified by `edb_audit_directory` (the default setting).

Set to `syslog` to use the syslog process and its location as configured in the `/etc/syslog.conf` file. The `syslog` setting is valid only for Advanced Server running on a Linux host, and is not supported on Windows systems. **Note:** In recent Linux versions, syslog has been replaced by *rsyslog* and the configuration file is in `/etc/rsyslog.conf`.

3.1.3.7.12 edb_log_every_bulk_value

For information on `edb_log_every_bulk_value`, see Section 3.1.3.6.2.

3.1.3.8 Client Connection Defaults / Locale and Formatting

This section describes configuration parameters affecting locale and formatting.

3.1.3.8.1 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 3.6 for general information about the ICU short form and the Unicode Collation Algorithm.

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

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

3.1.3.9 Client Connection Defaults / Statement Behavior

This section describes configuration parameters affecting statement behavior.

3.1.3.9.1 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.

3.1.3.9.2 edb_data_redaction

Parameter Type: Boolean

Default Value: `true`

Range: {`true | false`}

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

Data redaction is the support of policies to limit the exposure of certain sensitive data to certain users by altering the displayed information.

The default setting is `TRUE` so the data redaction is applied to all users except for superusers and the table owner:

- Superusers and table owner bypass data redaction.
- All other users get the redaction policy applied and see the reformatted data.

If the parameter is disabled by setting it to `FALSE`, then the following occurs:

- Superusers and table owner still bypass data redaction.
- All other users will get an error.

For information on data redaction, see Section 4.4.

3.1.3.10 Client Connection Defaults / Other Defaults

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

3.1.3.10.1 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: EPAS 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.

3.1.3.10.2 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: EPAS service account

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

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

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

3.1.3.11 Compatibility Options

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

3.1.3.11.1 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` at the time the table definition is stored in the database if the configuration parameter `edb_redwood_date` is set to `TRUE`. Thus, a time component will also be stored in the column along with the date.

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

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

3.1.3.11.2 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)
```

3.1.3.11.3 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_`). 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)
```

3.1.3.11.4 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)
```

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

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

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

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

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

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

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

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

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

```
\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.

3.1.3.11.6 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 an extended catalog view. This is the `sys` catalog for the expanded catalog view. The `db_dialect` parameter controls the order in which these catalogs are searched for name resolution.

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

3.1.3.11.7 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. In earlier versions of Advanced Server `ROWIDs` were mapped to `OIDs`, but from Advanced Server version 12 onwards the `ROWID` is an auto-incrementing value based on a sequence that starts with `1` and assigned to each row of a table created with `ROWIDs` option. By default, a unique index is created on a `ROWID` column.

The `ALTER` and `DROP` operations are restricted on a `ROWID` column.

To restore a database with `ROWIDs` from Advanced Server 11 or an earlier version, you must perform the following:

- pg_dump: If a table includes `OIDs` then specify `--convert-oids-to-rowids` to dump a database. Otherwise, ignore the `OIDs` to continue table creation on Advanced Server version 12 onwards.

- pg_upgrade: Errors out. But if a table includes `OIDs` or `ROWIDs`, then you must perform the following steps:

    1. Take a dump of the tables by specifying `--convert-oids-to-rowids` option.
    2. Drop the tables and then perform the upgrade.
    3. Restore the dump after the upgrade is successful into a new cluster that contains the dumped tables into a target database.

3.1.3.11.8 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 3-2 - 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.

3.1.3.12 Customized Options

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

3.1.3.12.1 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.

3.1.3.12.2 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: EPAS service account

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

3.1.3.12.3 dbms_pipe.total_message_buffer

Parameter Type: Integer

Default Value: 30 Kb

Range: 30 Kb to 256 Kb

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: EPAS service account

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

3.1.3.12.4 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 (12.0.0)
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)
```

3.1.3.12.5 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: EPAS service account

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

3.1.3.12.6 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: EPAS service account

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

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

- **learn.** Tracks the activities of protected roles and records the relations used by the roles. This is used when initially configuring SQL/Protect so the expected behaviors of the protected applications are learned.

- **passive.** Issues warnings if protected roles are breaking the defined rules, but does not stop any SQL statements from executing. This is the next step after SQL/Protect has learned the expected behavior of the protected roles. This essentially behaves in intrusion detection mode and can be run in production when properly monitored.
- **active.** Stops all invalid statements for a protected role. This behaves as a SQL firewall preventing dangerous queries from running. This is particularly effective against early penetration testing when the attacker is trying to determine the vulnerability point and the type of database behind the application. Not only does SQL/Protect close those vulnerability points, but it tracks the blocked queries allowing administrators to be alerted before the attacker finds an alternate method of penetrating the system.

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

3.1.3.12.7 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: EPAS service account

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

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

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

3.1.3.12.8 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: EPAS service account

Sets the maximum number of roles that can be protected.

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

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

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

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

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

As long as the space required can be reserved in shared memory, the value will be acceptable. If the value is such that the space in shared memory cannot 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.

3.1.3.12.9 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: EPAS service account

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

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

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

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

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

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

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

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

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

3.1.3.12.10 edb_wait_states.directory

Parameter Type: String

Default Value: `edb_wait_states`

Range: n/a

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Restart

Required Authorization to Activate: EPAS service account

Sets the directory path where the EDB wait states log files are stored. The specified path should be a full, absolute path and not a relative path. However, the default setting is `edb_wait_states`, which makes `$PGDATA/edb_wait_states` the default directory location. See Section 8.2 for information on EDB wait states.

3.1.3.12.11 edb_wait_states.retention_period

Parameter Type: Integer

Default Value: 604800s

Range: 86400s to 2147483647s

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

Sets the period of time after which the log files for EDB wait states will be deleted. The default is 604800 seconds, which is 7 days. See Section 8.2 for information on EDB wait states.

3.1.3.12.12 edb_wait_states.sampling_interval

Parameter Type: Integer

Default Value: 1s

Range: 1s to 2147483647s

Minimum Scope of Effect: Cluster

When Value Changes Take Effect: Reload

Required Authorization to Activate: EPAS service account

Sets the timing interval between two sampling cycles for EDB wait states. The default setting is 1 second. See Section 8.2 for information on EDB wait states.

3.1.3.12.13 edbldr.empty_csv_field

Parameter Type: Enum

Default Value: `NULL`

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

Minimum Scope of Effect: Per session

When Value Changes Take Effect: Immediate

Required Authorization to Activate: Session user

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

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

For more information about the `edbldr.empty_csv_field` parameter in EDB*Loader, see the *Database Compatibility for Oracle Developers Tools and Utilities Guide* at the EnterpriseDB website:

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

3.1.3.12.14 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.

3.1.3.12.15 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
```

3.1.3.13 Ungrouped

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

3.1.3.13.1 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 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`.

3.1.3.13.2 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 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`.

3.1.3.13.3 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 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`.

3.1.3.13.4 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.

3.2 *Index Advisor*

The Index Advisor utility helps determine which columns you should index to improve performance in a given workload. Index Advisor considers B-tree (single-column or composite) index types, and does not identify other index types (GIN, GiST, Hash) that may improve performance. Index Advisor is installed with EDB Postgres Advanced Server.

Index Advisor works with Advanced Server's query planner by creating *hypothetical indexes* that the query planner uses to calculate execution costs as if such indexes were available. Index Advisor identifies the indexes by analyzing SQL queries supplied in the workload.

There are three ways to use Index Advisor to analyze SQL queries:

- 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 the *PEM Getting Started Guide* available from the EnterpriseDB website at:

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

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.

3.2.1 Index Advisor Components

The Index Advisor shared library interacts with the query planner to make indexing recommendations. On Windows, the Advanced Server installer creates the following shared library in the `libdir` subdirectory of your Advanced Server home directory. For Linux, install the `edb-as`*`xx`*`-server-indexadvisor` RPM package where *`xx`* is the Advanced Server version number.

On Linux:

```
    index_advisor.so
```

On Windows:

```
    index_advisor.dll
```

Please note that libraries in the `libdir` directory can only be loaded by a superuser. A database administrator can allow a non-superuser to use Index Advisor by manually copying the Index Advisor file from the `libdir` directory into the `libdir/plugins` directory (under your Advanced Server home directory). Only a trusted non-superuser should be allowed access to the plugin; this is an unsafe practice in a production environment.

The installer also creates the Index Advisor utility program and setup script:

`pg_advise_index`

> `pg_advise_index` is a utility program that reads a user-supplied input file containing SQL queries and produces a text file containing `CREATE INDEX` statements that can be used to create the indexes recommended by the Index Advisor. The `pg_advise_index` program is located in the `bin` subdirectory of the Advanced Server home directory.

`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 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.

3.2.2 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
psql.bin (12.0.0, server 12.0.0)
Type "help" for help.

edb=# CREATE SCHEMA ia;
CREATE SCHEMA
edb=# SET search_path TO ia;
SET
edb=# \i /usr/edb/as12/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.

3.2.3 Using Index Advisor

When you invoke Index Advisor, you must supply a workload; the workload is either a query (specified at the command line), or a file that contains a set of queries (executed by the `pg_advise_index()` function). After analyzing the workload, Index Advisor will either store the result set in a temporary table, or in a permanent table. You can review the indexing recommendations generated by Index Advisor and use the `CREATE INDEX` statements generated by Index Advisor to create the recommended indexes.

Note: You should not run Index Advisor in read-only transactions.

The following examples assume that superuser `enterprisedb` is the Index Advisor user, and the Index Advisor database objects have been created in a schema in the `search_path` of superuser `enterprisedb`.

The examples in the following sections use the table created with the statement shown below:

```
CREATE TABLE t( a INT, b INT );
INSERT INTO t SELECT s, 99999 - s FROM generate_series(0,99999) AS s;
ANALYZE t;
```

The resulting table contains the following rows:

```
   a   |   b
-------+-------
     0 | 99999
     1 | 99998
     2 | 99997
     3 | 99996
       .
       .
       .
 99997 |     2
 99998 |     1
 99999 |     0
```

3.2.3.1 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
```

3.2.3.2 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)
```

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
```

3.2.4 Reviewing the Index Advisor Recommendations

There are several ways to review the index recommendations generated by Index Advisor. You can:

- Query the `index_advisor_log` table.
- Run the `show_index_recommendations` function.
- Query the `index_recommendations` view.

3.2.4.1 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;
```

3.2.4.2 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.

3.2.4.3 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.

3.2.5 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;
```

3.3 SQL Profiler

Inefficient SQL code is one of, if not the leading cause of database performance problems. The challenge for database administrators and developers is locating and then optimizing this code in large, complex systems.

SQL Profiler helps you locate and optimize poorly running SQL code.

Specific features and benefits of SQL Profiler include the following:

- **On-Demand Traces.** You can capture SQL traces at any time by manually setting up your parameters and starting the trace.
- **Scheduled Traces.** For inconvenient times, you can also specify your trace parameters and schedule them to run at some later time.
- **Save Traces.** Execute your traces and save them for later review.
- **Trace Filters.** Selectively filter SQL captures by database and by user, or capture every SQL statement sent by all users against all databases.
- **Trace Output Analyzer.** A graphical table lets you quickly sort and filter queries by duration or statement, and a graphical or text based `EXPLAIN` plan lays out your query paths and joins.
- **Index Advisor Integration.** Once you have found your slow queries and optimized them, you can also let the Index Advisor recommend the creation of underlying table indices to further improve performance.

The following describes the installation process.

Step 1: Install SQL Profiler

SQL Profiler is installed by the Advanced Server installer on Windows or from the `edb-as`*`xx`*`-server-sqlprofiler` RPM package on Linux where *`xx`* is the Advanced Server version number.

Step 2: Add the SQL Profiler library

Modify the `postgresql.conf` parameter file for the instance to include the SQL Profiler library in the `shared_preload_libraries` configuration parameter.

For Linux installations, the parameter value should include:

```
$libdir/sql-profiler
```

On Windows, the parameter value should include:

```
$libdir\sql-profiler.dll
```

Step 3: Create the functions used by SQL Profiler

The SQL Profiler installation program places a SQL script (named `sql-profiler.sql`) in:

On Linux:

```
/usr/edb/as12/share/contrib/
```

On Windows:

```
C:\Program Files\edb\as12\share\contrib\
```

Use the `psql` command line interface to run the `sql-profiler.sql` script in the database specified as the Maintenance Database on the server you wish to profile. If you are using Advanced Server, the default maintenance database is named `edb`. If you are using a PostgreSQL instance, the default maintenance database is named `postgres`.

The following command uses the `psql` command line to invoke the `sql-profiler.sql` script on a Linux system:

```
$ /usr/edb/as12/bin/psql -U user_name database_name <
/usr/edb/as12/share/contrib/sql-profiler.sql
```

Step 4: Stop and restart the server for the changes to take effect.

After configuring SQL Profiler, it is ready to use with all databases that reside on the server. You can take advantage of SQL Profiler functionality with EDB Postgres Enterprise Manager; for more information about Postgres Enterprise Manager, visit the EnterpriseDB website at:

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

Troubleshooting

If (after performing an upgrade to a newer version of SQL Profiler) you encounter an error that contains the following text:

```
An error has occurred:
ERROR: function return row and query-specified return row do not match.
DETAIL: Returned row contains 11 attributes, but the query expects 10.
```

To correct this error, you must replace the existing query set with a new query set. First, uninstall SQL Profiler by invoking the `uninstall-sql-profiler.sql` script, and then reinstall SQL Profiler by invoking the `sql-profiler.sql` script.

3.4 pgsnmpd

`pgsnmpd` is an SNMP agent that can return hierarchical information about the current state of Advanced Server on a Linux system. `pgsnmpd` is distributed and installed using the `edb-as`*`xx`*`-pgsnmpd` RPM package where *`xx`* is the Advanced Server version number. The `pgsnmpd` agent can operate as a stand-alone SNMP agent, as a pass-through sub-agent, or as an AgentX sub-agent.

After installing Advanced Server, you will need to update the `LD_LIBRARY_PATH` variable. Use the command:

```
$ export LD_LIBRARY_PATH=/usr/edb/as12/lib:$LD_LIBRARY_PATH
```

This command does not persistently alter the value of `LD_LIBRARY_PATH`. Consult the documentation for your distribution of Linux for information about persistently setting the value of `LD_LIBRARY_PATH`.

The examples that follow demonstrate the simplest usage of `pgsnmpd`, implementing read only access. `pgsnmpd` is based on the net-snmp library; for more information about net-snmp, visit:

http://net-snmp.sourceforge.net/

3.4.1 Configuring pgsnmpd

The `pgsnmpd` configuration file is named `snmpd.conf`. For information about the directives that you can specify in the configuration file, please review the `snmpd.conf` man page (`man snmpd.conf`).

You can create the configuration file by hand, or you can use the `snmpconf` perl script to create the configuration file. The perl script is distributed with net-snmp package.

net-snmp is an open-source package available from:

http://www.net-snmp.org/

To use the snmpconf configuration file wizard, download and install net-snmp. When the installation completes, open a command line and enter:

```
snmpconf
```

When the configuration file wizard opens, it may prompt you to read in an existing configuration file. Enter `none` to generate a new configuration file (not based on a previously existing configuration file).

`snmpconf` is a menu-driven wizard. Select menu item `1: snmpd.conf` to start the configuration wizard. As you select each top-level menu option displayed by `snmpconf`, the wizard walks through a series of questions, prompting you for information required to build the configuration file. When you have provided information in each of the category relevant to your system, enter `Finished` to generate a configuration file named `snmpd.conf`. Copy the file to:

```
/usr/edb/as12/share/
```

3.4.2 Setting the Listener Address

By default, `pgsnmpd` listens on port `161`. If the listener port is already being used by another service, you may receive the following error:

```
Error opening specified endpoint "udp:161".
```

You can specify an alternate listener port by adding the following line to your `snmpd.conf` file:

```
agentaddress $host_address:2000
```

The example instructs `pgsnmpd` to listen on UDP port `2000`, where *`$host_address`* is the IP address of the server (e.g., `127.0.0.1`).

3.4.3 Invoking pgsnmpd

Ensure that an instance of Advanced Server is up and running (`pgsnmpd` will connect to this server). Open a command line and assume superuser privileges, before invoking pgsnmpd with a command that takes the following form:

```
POSTGRES_INSTALL_HOME/bin/pgsnmpd -b
  -c POSTGRES_INSTALL_HOME/share/snmpd.conf
  -C "user=enterprisedb dbname=edb password=safe_password
      port=5444"
```

Where *`POSTGRES_INSTALL_HOME`* specifies the Advanced Server installation directory.

Include the `-b` option to specify that `pgsnmpd` should run in the background.

Include the `-c` option, specifying the path and name of the `pgsnmpd` configuration file.

Include connection information for your installation of Advanced Server (in the form of a `libpq` connection string) after the `-C` option.

3.4.4 Viewing pgsnmpd Help

Include the `--help` option when invoking the `pgsnmpd` utility to view other `pgsnmpd` command line options:

```
pgsnmpd --help
  Version PGSQL-SNMP-Ver1.0
  usage: pgsnmpd [-s] [-b] [-c FILE ] [-x address ] [-g] [-C "Connect String"]
     -s : run as AgentX sub-agent of an existing snmpd process
     -b : run in the background
     -c : configuration file name
     -g : use syslog
     -C : libpq connection string
     -x : address:port of a network interface
     -V : display version strings
```

3.4.5 Requesting Information from pgsnmpd

You can use `net-snmp` commands to query the `pgsnmpd` service. For example:

```
snmpgetnext -v 2c -c public localhost
.1.3.6.1.4.1.5432.1.4.2.1.1.0
```

In the above example:

`-v 2c` option instructs the `snmpgetnext` client to send the request in SNMP version 2c format.

`-c public` specifies the community name.

`localhost` indicates the host machine running the `pgsnmpd` server.

`.1.3.6.1.4.1.5432.1.4.2.1.1.0` is the identity of the requested object. To see a list of all databases, increment the last digit by one (e.g. .1.1, .1.2, .1.3 etc.).

The encodings required to query any given object are defined in the MIB (Management Information Base). An SNMP client can monitor a variety of servers; the server type determines the information exposed by a given server. Each SNMP server describes the exposed data in the form of a MIB (Management information base). By default, pgsnmpd searches for MIB's in the following locations:

```
/usr/share/snmp/mibs

$HOME/.snmp/mibs
```

3.5 EDB Audit Logging

Advanced Server allows database and security administrators, auditors, and operators to track and analyze database activities using the *EDB Audit Logging* functionality. EDB Audit Logging generates audit log files, which contains all of the relevant information. The audit logs can be configured to record information such as:

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

The parameters specified in the configuration files, `postgresql.conf` or `postgresql.auto.conf`, control the information included in the audit logs.

3.5.1 Audit Logging Configuration Parameters

Use the following configuration parameters to control database auditing. See Section 3.1.2 to determine if a change to the configuration parameter takes effect immediately, or if the configuration needs to be reloaded, or if the database server needs to be restarted.

`edb_audit`

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

`edb_audit_directory`

> Specifies the directory where the log files will be created. The path of the directory can be relative or absolute to the data folder. The default is the `PGDATA/edb_audit` directory.

`edb_audit_filename`

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

`edb_audit_rotation_day`

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

`edb_audit_rotation_size`

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

`edb_audit_rotation_seconds`

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

`edb_audit_connect`

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

`edb_audit_disconnect`

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

`edb_audit_statement`

> This configuration parameter is used to specify auditing of different categories of SQL statements. Various combinations of the following values may be specified: `none`, `dml`, `insert`, `update`, `delete`, `truncate`, `select`, `error`, `rollback`, `ddl`, `create`, `drop`, `alter`, `grant`, `revoke`, and `all`. The default is `ddl` and `error`. See Section 3.5.2 for information regarding the setting of this parameter.

`edb_audit_tag`

> Use this configuration parameter to specify a string value that will be included in the audit log file for each entry as a tracking tag.

`edb_log_every_bulk_value`

Bulk processing logs the resulting statements into both the Advanced Server log file and the EDB Audit log file. However, logging each and every statement in bulk processing is costly. This can be controlled by the `edb_log_every_bulk_value` configuration parameter. When set to `true`, each and every statement in bulk processing is logged. When set to `false`, a log message is recorded once per bulk processing. In addition, the duration is emitted once per bulk processing. Default is `false`.

`edb_audit_destination`

Specifies whether the audit log information is to be recorded in the directory as given by the `edb_audit_directory` parameter or to the directory and file managed by the *syslog* process. Set to `file` to use the directory specified by `edb_audit_directory`, which is the default setting.

Set to `syslog` to use the syslog process and its location as configured in the `/etc/syslog.conf` file. The `syslog` setting is valid for Advanced Server running on a Linux host and is not supported on Windows systems. **Note:** In recent Linux versions, syslog has been replaced by *rsyslog* and the configuration file is in `/etc/rsyslog.conf`.

The following section describes selection of specific SQL statements for auditing using the `edb_audit_statement` parameter.

3.5.2 Selecting SQL Statements to Audit

The `edb_audit_statement` permits inclusion of one or more, comma-separated values to control which SQL statements are to be audited. The following is the general format:

```
edb_audit_statement = 'value_1[, value_2]...'
```

The comma-separated values may include or omit space characters following the comma. The values can be specified in any combination of lowercase or uppercase letters.

The basic parameter values are the following:

- `all` – Results in the auditing and logging of every statement including any error messages on statements.
- `none` – Disables all auditing and logging. A value of `none` overrides any other value included in the list.

- `ddl` – Results in the auditing of all data definition language (DDL) statements (`CREATE`, `ALTER`, and `DROP`) as well as `GRANT` and `REVOKE` data control language (DCL) statements.
- `dml` – Results in the auditing of all data manipulation language (DML) statements (`INSERT`, `UPDATE`, `DELETE`, and `TRUNCATE`).
- `select` – Results in the auditing of `SELECT` statements.
- `rollback` – Results in the auditing of `ROLLBACK` statements.
- `error` – Results in the logging of all error messages that occur. Unless the `error` value is included, no error messages are logged regarding any errors that occur on SQL statements related to any of the other preceding parameter values except when `all` is used.

Section 3.5.2.1 describes additional parameter values for selecting particular DDL or DCL statements for auditing.

Section 3.5.2.2 describes additional parameter values for selecting particular DML statements for auditing.

If an unsupported value is included in the `edb_audit_statement` parameter, then an error occurs when applying the setting to the database server. See the database server log file for the error such as in the following example where `create materialized vw` results in the error. (The correct value is `create materialized view`.)

```
2017-07-16 11:20:39 EDT LOG:  invalid value for parameter "edb_audit_statement": "create
materialized vw, create sequence, grant"
2017-07-16 11:20:39 EDT FATAL:  configuration file "/var/lib/edb/as12/data/postgresql.conf"
contains errors
```

The following sections describe the values for the SQL language types DDL, DCL, and DML.

3.5.2.1 Data Definition Language and Data Control Language Statements

This section describes values that can be included in the `edb_audit_statement` parameter to audit DDL and DCL statements.

The following general rules apply:

- If the `edb_audit_statement` parameter includes either `ddl` or `all`, then all DDL statements are audited. In addition, the DCL statements `GRANT` and `REVOKE` are audited as well.
- If the `edb_audit_statement` is set to `none`, then no DDL nor DCL statements are audited.

- Specific types of DDL and DCL statements can be chosen for auditing by including a combination of values within the `edb_audit_statement` parameter.

Use the following syntax to specify an `edb_audit_statement` parameter value for DDL statements:

```
{ create | alter | drop } [ object_type ]
```

`object_type` is any of the following:

```
ACCESS METHOD
AGGREGATE
CAST
COLLATION
CONVERSION
DATABASE
EVENT TRIGGER
EXTENSION
FOREIGN TABLE
FUNCTION
INDEX
LANGUAGE
LARGE OBJECT
MATERIALIZED VIEW
OPERATOR
OPERATOR CLASS
OPERATOR FAMILY
POLICY
PUBLICATION
ROLE
RULE
SCHEMA
SEQUENCE
SERVER
SUBSCRIPTION
TABLE
TABLESPACE
TEXT SEARCH CONFIGURATION
TEXT SEARCH DICTIONARY
TEXT SEARCH PARSER
TEXT SEARCH TEMPLATE
TRANSFORM
TRIGGER
TYPE
USER MAPPING
VIEW
```

Descriptions of object types as used in SQL commands can be found in the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/12/static/sql-commands.html

If *object_type* is omitted from the parameter value, then all of the specified command statements (either `create`, `alter`, or `drop`) are audited.

Use the following syntax to specify an `edb_audit_statement` parameter value for DCL statements:

```
{ grant | revoke }
```

The following are some DDL and DCL examples.

Example 1

The following is an example where `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
edb_audit_connect = 'all'
edb_audit_statement = 'create, alter, error'
```

Thus, only SQL statements invoked by the `CREATE` and `ALTER` commands are audited. Error messages are also included in the audit log.

The database session that occurs is the following:

```
$ psql edb enterprisedb
Password for user enterprisedb:
psql.bin (12.0.0)
Type "help" for help.

edb=# SHOW edb_audit_connect;
 edb_audit_connect
-------------------
 all
(1 row)

edb=# SHOW edb_audit_statement;
 edb_audit_statement
----------------------
 create, alter, error
(1 row)

edb=# CREATE ROLE adminuser;
CREATE ROLE
edb=# ALTER ROLE adminuser WITH LOGIN, SUPERUSER, PASSWORD 'password';
ERROR:  syntax error at or near ","
LINE 1: ALTER ROLE adminuser WITH LOGIN, SUPERUSER, PASSWORD 'passwo...
                                       ^
edb=# ALTER ROLE adminuser WITH LOGIN SUPERUSER PASSWORD 'password';
ALTER ROLE
edb=# CREATE DATABASE auditdb;
CREATE DATABASE
```

```
edb=# ALTER DATABASE auditdb OWNER TO adminuser;
ALTER DATABASE
edb=# \c auditdb adminuser
Password for user adminuser:
You are now connected to database "auditdb" as user "adminuser".
auditdb=# CREATE SCHEMA edb;
CREATE SCHEMA
auditdb=# SET search_path TO edb;
SET
auditdb=# CREATE TABLE department (
auditdb(#     deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
auditdb(#     dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
auditdb(#     loc             VARCHAR2(13)
auditdb(# );
CREATE TABLE
auditdb=# DROP TABLE department;
DROP TABLE
auditdb=# CREATE TABLE dept (
auditdb(#     deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
auditdb(#     dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
auditdb(#     loc             VARCHAR2(13)
auditdb(# );
CREATE TABLE
```

The resulting audit log file contains the following.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
2017-07-16 12:59:42.125 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,1,"authentication",2017-07-16 12:59:42 EDT,6/2,0,AUDIT,00000,
"connection authorized: user=enterprisedb database=edb",,,,,,,,,"","",""

2017-07-16 12:59:42.125 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,2,"idle",2017-07-16 12:59:42 EDT,6/6,0,AUDIT,00000,
"statement: CREATE ROLE adminuser;",,,,,,,,,"psql.bin","CREATE ROLE",""

2017-07-16 13:00:28.469 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,3,"idle",2017-07-16 12:59:42 EDT,6/7,0,ERROR,42601,
"syntax error at or near "","""",,,,,,
"ALTER ROLE adminuser WITH LOGIN, SUPERUSER, PASSWORD 'password';",32,,"psql.bin","",""

2017-07-16 13:00:28.469 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,4,"idle",2017-07-16 12:59:42 EDT,6/8,0,AUDIT,00000,
"statement: ALTER ROLE adminuser WITH LOGIN SUPERUSER PASSWORD 'password';",,,,,,,,,
"psql.bin","ALTER ROLE",""

2017-07-16 13:00:28.469 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,5,"idle",2017-07-16 12:59:42 EDT,6/9,0,AUDIT,00000,
"statement: CREATE DATABASE auditdb;",,,,,,,,,"psql.bin","CREATE DATABASE",""

2017-07-16 13:00:28.469 EDT,"enterprisedb","edb",3356,"[local]",
596b9b7e.d1c,6,"idle",2017-07-16 12:59:42 EDT,6/10,0,AUDIT,00000,
"statement: ALTER DATABASE auditdb OWNER TO adminuser;",,,,,,,,,"psql.bin","ALTER DATABASE",""

2017-07-16 13:01:13.735 EDT,"adminuser","auditdb",3377,"[local]",
596b9bd9.d31,1,"authentication",2017-07-16 13:01:13 EDT,4/15,0,AUDIT,00000,
"connection authorized: user=adminuser database=auditdb",,,,,,,,,"","",""

2017-07-16 13:01:13.735 EDT,"adminuser","auditdb",3377,"[local]",
596b9bd9.d31,2,"idle",2017-07-16 13:01:13 EDT,4/17,0,AUDIT,00000,
"statement: CREATE SCHEMA edb;",,,,,,,,,"psql.bin","CREATE SCHEMA",""

2017-07-16 13:01:13.735 EDT,"adminuser","auditdb",3377,"[local]",
596b9bd9.d31,3,"idle",2017-07-16 13:01:13 EDT,4/19,0,AUDIT,00000,
"statement: CREATE TABLE department (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);",,,,,,,,,"psql.bin","CREATE TABLE",""

2017-07-16 13:01:13.735 EDT,"adminuser","auditdb",3377,"[local]",
596b9bd9.d31,4,"idle",2017-07-16 13:01:13 EDT,4/21,0,AUDIT,00000,
"statement: CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);",,,,,,,,,"psql.bin","CREATE TABLE",""
```

The `CREATE` and `ALTER` statements for the `adminuser` role and `auditdb` database are audited. The error for the `ALTER ROLE adminuser` statement is also logged since error is included in the `edb_audit_statement` parameter.

Similarly, the `CREATE` statements for schema `edb` and tables `department` and `dept` are audited.

Note that the `DROP TABLE department` statement is not in the audit log since there is no `edb_audit_statement` setting that would result in the auditing of successfully processed `DROP` statements such as `ddl`, `all`, or `drop`.

Example 2

The following is an example where `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
edb_audit_connect = 'all'
edb_audit_statement = create view,create materialized view,create sequence,grant'
```

Thus, only SQL statements invoked by the `CREATE VIEW` , `CREATE MATERIALIZED VIEW`, `CREATE SEQUENCE` and `GRANT` commands are audited.

The database session that occurs is the following:

```
$ psql auditdb adminuser
Password for user adminuser:
psql.bin (12.0.0)
Type "help" for help.

auditdb=# SHOW edb_audit_connect;
 edb_audit_connect
-------------------
 all
(1 row)

auditdb=# SHOW edb_audit_statement;
                    edb_audit_statement
------------------------------------------------------------
 create view,create materialized view,create sequence,grant
(1 row)

auditdb=# SET search_path TO edb;
```

```
SET
auditdb=# CREATE TABLE emp (
auditdb(#     empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
auditdb(#     ename           VARCHAR2(10),
auditdb(#     job             VARCHAR2(9),
auditdb(#     mgr             NUMBER(4),
auditdb(#     hiredate        DATE,
auditdb(#     sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
auditdb(#     comm            NUMBER(7,2),
auditdb(#     deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
auditdb(#                         REFERENCES dept(deptno)
auditdb(# );
CREATE TABLE
auditdb=# CREATE VIEW salesemp AS
auditdb-#     SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
CREATE VIEW
auditdb=# CREATE MATERIALIZED VIEW managers AS
auditdb-#     SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'MANAGER';
SELECT 0
auditdb=# CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
CREATE SEQUENCE
auditdb=# GRANT ALL ON dept TO PUBLIC;
GRANT
auditdb=# GRANT ALL ON emp TO PUBLIC;
GRANT
```

The resulting audit log file contains the following.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,1,"authentication",2017-07-16 13:20:09 EDT,4/10,0,AUDIT,00000,
"connection authorized: user=adminuser database=auditdb",,,,,,,,,"","",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,2,"idle",2017-07-16 13:20:09 EDT,4/16,0,AUDIT,00000,
"statement: CREATE VIEW salesemp AS
    SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job =
'SALESMAN';",,,,,,,,,"psql.bin","CREATE VIEW",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,3,"idle",2017-07-16 13:20:09 EDT,4/17,0,AUDIT,00000,
"statement: CREATE MATERIALIZED VIEW managers AS
    SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job =
'MANAGER';",,,,,,,,,"psql.bin","CREATE MATERIALIZED VIEW",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,4,"idle",2017-07-16 13:20:09 EDT,4/18,0,AUDIT,00000,
"statement: CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY
1;",,,,,,,,,"psql.bin","CREATE SEQUENCE",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,5,"idle",2017-07-16 13:20:09 EDT,4/19,0,AUDIT,00000,
"statement: GRANT ALL ON dept TO PUBLIC;",,,,,,,,,"psql.bin","GRANT",""

2017-07-16 13:20:09.836 EDT,"adminuser","auditdb",4143,"[local]",
596ba049.102f,6,"idle",2017-07-16 13:20:09 EDT,4/20,0,AUDIT,00000,
"statement: GRANT ALL ON emp TO PUBLIC;",,,,,,,,,"psql.bin","GRANT",""
```

The `CREATE VIEW` and `CREATE MATERIALIZED VIEW` statements are audited. Note that the prior `CREATE TABLE emp` statement is not audited since none of the values `create`, `create table`, `ddl`, nor `all` are included in the `edb_audit_statement` parameter.

The `CREATE SEQUENCE` and `GRANT` statements are audited since those values are included in the `edb_audit_statement` parameter.

3.5.2.2 Data Manipulation Language Statements

This section describes the values that can be included in the `edb_audit_statement` parameter to audit DML statements.

The following general rules apply:

- If the `edb_audit_statement` parameter includes either `dml` or `all`, then all DML statements are audited.
- If the `edb_audit_statement` is set to `none`, then no DML statements are audited.
- Specific types of DML statements can be chosen for auditing by including a combination of values within the `edb_audit_statement` parameter.

Use the following syntax to specify an `edb_audit_statement` parameter value for SQL `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE` statements:

```
{ insert | update | delete | truncate }
```

Example

The following is an example where `edb_audit_connect` and `edb_audit_statement` are set with the following non-default values:

```
edb_audit_connect = 'all'
edb_audit_statement = 'UPDATE, DELETE, error'
```

Thus, only SQL statements invoked by the `UPDATE` and `DELETE` commands are audited. All errors are also included in the audit log (even errors not related to the `UPDATE` and `DELETE` commands).

The database session that occurs is the following:

```
$ psql auditdb adminuser
Password for user adminuser:
psql.bin (12.0.0)
Type "help" for help.

auditdb=# SHOW edb_audit_connect;
 edb_audit_connect
-------------------
 all
```

```
(1 row)

auditdb=# SHOW edb_audit_statement;
  edb_audit_statement
-----------------------
 UPDATE, DELETE, error
(1 row)

auditdb=# SET search_path TO edb;
SET
auditdb=# INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT 0 1
auditdb=# INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT 0 1
auditdb=# INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT 0 1
auditdb=# INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
INSERT 0 1
auditdb=# INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT 0 1
auditdb=# INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
INSERT 0 1
auditdb=# INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT 0 1
                    .
                    .
                    .
auditdb=# INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
INSERT 0 1
auditdb=# UPDATE dept SET loc = 'BEDFORD' WHERE deptno = 40;
UPDATE 1
auditdb=# SELECT * FROM dept;
 deptno |   dname    |   loc
--------+------------+----------
     10 | ACCOUNTING | NEW YORK
     20 | RESEARCH   | DALLAS
     30 | SALES      | CHICAGO
     40 | OPERATIONS | BEDFORD
(4 rows)

auditdb=# DELETE FROM emp WHERE deptno = 10;
DELETE 3
auditdb=# TRUNCATE employee;
ERROR:  relation "employee" does not exist
auditdb=# TRUNCATE emp;
TRUNCATE TABLE
auditdb=# \q
```

The resulting audit log file contains the following.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
2017-07-16 13:43:26.638 EDT,"adminuser","auditdb",4574,"[local]",
596ba5be.11de,1,"authentication",2017-07-16 13:43:26 EDT,4/11,0,AUDIT,00000,
"connection authorized: user=adminuser database=auditdb",,,,,,,,,"","",""

2017-07-16 13:43:26.638 EDT,"adminuser","auditdb",4574,"[local]",
596ba5be.11de,2,"idle",2017-07-16 13:43:26 EDT,4/34,0,AUDIT,00000,
"statement: UPDATE dept SET loc = 'BEDFORD' WHERE deptno = 40;",,,,,,,,,"psql.bin","UPDATE",""

2017-07-16 13:43:26.638 EDT,"adminuser","auditdb",4574,"[local]",
596ba5be.11de,3,"idle",2017-07-16 13:43:26 EDT,4/36,0,AUDIT,00000,
"statement: DELETE FROM emp WHERE deptno = 10;",,,,,,,,,"psql.bin","DELETE",""
```

```
2017-07-16 13:45:46.999 EDT,"adminuser","auditdb",4574,"[local]",
596ba5be.11de,4,"TRUNCATE TABLE",2017-07-16 13:43:26 EDT,4/37,0,ERROR,42P01,
"relation ""employee"" does not exist",,,,,,"TRUNCATE employee;",,,"psql.bin","",""

2017-07-16 13:46:26.362 EDT,,,4491,,596ba59c.118b,1,,2017-07-16 13:42:52 EDT,,0,LOG,00000,
"database system is shut down",,,,,,,,,"","",""
```

The `UPDATE dept` and `DELETE FROM emp` statements are audited. Note that all of the prior `INSERT` statements are not audited since none of the values `insert`, `dml`, nor `all` are included in the `edb_audit_statement` parameter.

The `SELECT * FROM dept` statement is not audited as well since neither `select` nor `all` is included in the `edb_audit_statement` parameter.

Since `error` is specified in the `edb_audit_statement` parameter, but not the `truncate` value, the error on the `TRUNCATE employee` statement is logged in the audit file, but not the successful `TRUNCATE emp` statement.

3.5.3 Enabling Audit Logging

The following steps describe how to configure Advanced Server to log all connections, disconnections, DDL statements, DCL statements, DML statements, and any statements resulting in an error.

1. Enable auditing by the setting the `edb_audit` parameter to `xml` or `csv`.
2. Set the file rotation day when the new file will be created by setting the parameter `edb_audit_rotation_day` to the desired value.
3. To audit all connections, set the parameter, `edb_audit_connect`, to `all`.
4. To audit all disconnections, set the parameter, `edb_audit_disconnect`, to `all`.
5. To audit DDL, DCL, DML and other statements, set the parameter, `edb_audit_statement` according to the instructions in Section 3.5.2.

The setting of the `edb_audit_statement` parameter in the configuration file affects the entire database cluster.

The type of statements that are audited as controlled by the `edb_audit_statement` parameter can be further refined according to the database in use as well as the database role running the session:

- The `edb_audit_statement` parameter can be set as an attribute of a specified database with the `ALTER DATABASE` *`dbname`* `SET edb_audit_statement` command. This setting overrides the `edb_audit_statement` parameter in the configuration file for statements executed when connected to database *`dbname`*.
- The `edb_audit_statement` parameter can be set as an attribute of a specified role with the `ALTER ROLE` *`rolename`* `SET edb_audit_statement`

command. This setting overrides the `edb_audit_statement` parameter in the configuration file as well as any setting assigned to the database by the previously described `ALTER DATABASE` command when the specified role is running the current session.

- The `edb_audit_statement` parameter can be set as an attribute of a specified role when using a specified database with the `ALTER ROLE` *`rolename`* `IN DATABASE` *`dbname`* `SET edb_audit_statement` command. This setting overrides the `edb_audit_statement` parameter in the configuration file as well as any setting assigned to the database by the previously described `ALTER DATABASE` command as well as any setting assigned to the role with the `ALTER ROLE` command without the `IN DATABASE` clause as previously described.

The following are examples of this technique.

The database cluster is established with `edb_audit_statement` set to `all` as shown in its `postgresql.conf` file:

```
edb_audit_statement = 'all'             # Statement type to be audited:
                                        # none, dml, insert, update, delete, truncate,
                                        # select, error, rollback, ddl, create, drop,
                                        # alter, grant, revoke, all
```

A database and role are established with the following settings for the `edb_audit_statement` parameter:

- Database `auditdb` with `ddl`, `insert`, `update`, and `delete`
- Role `admin` with `select` and `truncate`
- Role `admin` in database `auditdb` with `create table`, `insert`, and `update`

Creation and alteration of the database and role are shown by the following:

```
$ psql edb enterprisedb
Password for user enterprisedb:
psql.bin (12.0.0)
Type "help" for help.

edb=# SHOW edb_audit_statement;
 edb_audit_statement
---------------------
 all
(1 row)

edb=# CREATE DATABASE auditdb;
CREATE DATABASE
edb=# ALTER DATABASE auditdb SET edb_audit_statement TO 'ddl, insert, update, delete';
ALTER DATABASE
edb=# CREATE ROLE admin WITH LOGIN SUPERUSER PASSWORD 'password';
CREATE ROLE
edb=# ALTER ROLE admin SET edb_audit_statement TO 'select, truncate';
ALTER ROLE
edb=# ALTER ROLE admin IN DATABASE auditdb SET edb_audit_statement TO 'create table, insert,
update';
ALTER ROLE
```

The following demonstrates the changes made and the resulting audit log file for three cases.

Case 1: Changes made in database `auditdb` by role `enterprisedb`. Only `ddl`, `insert`, `update`, and `delete` statements are audited:

```
$ psql auditdb enterprisedb
Password for user enterprisedb:
psql.bin (12.0.0)
Type "help" for help.

auditdb=# SHOW edb_audit_statement;
     edb_audit_statement
-----------------------------
 ddl, insert, update, delete
(1 row)

auditdb=# CREATE TABLE audit_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);
CREATE TABLE
auditdb=# INSERT INTO audit_tbl VALUES (1, 'Row 1');
INSERT 0 1
auditdb=# UPDATE audit_tbl SET f2 = 'Row A' WHERE f1 = 1;
UPDATE 1
auditdb=# SELECT * FROM audit_tbl;                            <== Should not be audited
 f1 |  f2
----+-------
  1 | Row A
(1 row)

auditdb=# TRUNCATE audit_tbl;                                 <== Should not be audited
TRUNCATE TABLE
```

The following audit log file shows entries only for the `CREATE TABLE`, `INSERT INTO audit_tbl`, and `UPDATE audit_tbl` statements. The `SELECT * FROM audit_tbl` and `TRUNCATE audit_tbl` statements were not audited.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,1,"idle",2017-07-13 15:25:59 EDT,7/4,0,AUDIT,00000,
"statement: CREATE TABLE audit_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);",,,,,,,,,
"psql.bin","CREATE TABLE",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,2,"idle",2017-07-13 15:25:59 EDT,7/5,0,AUDIT,00000,
"statement: INSERT INTO audit_tbl VALUES (1, 'Row 1');",,,,,,,,,"psql.bin","INSERT",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,3,"idle",2017-07-13 15:25:59 EDT,7/6,0,AUDIT,00000,
"statement: UPDATE audit_tbl SET f2 = 'Row A' WHERE f1 = 1;",,,,,,,,,"psql.bin","UPDATE",""
```

Case 2: Changes made in database `edb` by role `admin`. Only `select` and `truncate` statements are audited:

```
$ psql edb admin
Password for user admin:
psql.bin (12.0.0)
```

```
Type "help" for help.

edb=# SHOW edb_audit_statement;
 edb_audit_statement
---------------------
 select, truncate
(1 row)

edb=# CREATE TABLE edb_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT) <== Should not be audited
CREATE TABLE
edb=# INSERT INTO edb_tbl VALUES (1, 'Row 1');               <== Should not be audited
INSERT 0 1
edb=# SELECT * FROM edb_tbl;
 f1 |  f2
----+-------
  1 | Row 1
(1 row)

edb=# TRUNCATE edb_tbl;
TRUNCATE TABLE
```

Continuation of the audit log file now appears as follows. The last two entries representing the second case show only the `SELECT * FROM edb_tbl` and `TRUNCATE edb_tbl` statements. The `CREATE TABLE edb_tbl` and `INSERT INTO edb_tbl` statements were not audited.

```
2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,1,"idle",2017-07-13 15:25:59 EDT,7/4,0,AUDIT,00000,
"statement: CREATE TABLE audit_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);",,,,,,,,,
"psql.bin","CREATE TABLE",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,2,"idle",2017-07-13 15:25:59 EDT,7/5,0,AUDIT,00000,
"statement: INSERT INTO audit_tbl VALUES (1, 'Row 1');",,,,,,,,,"psql.bin","INSERT",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,3,"idle",2017-07-13 15:25:59 EDT,7/6,0,AUDIT,00000,
"statement: UPDATE audit_tbl SET f2 = 'Row A' WHERE f1 = 1;",,,,,,,,,"psql.bin","UPDATE",""

2017-07-13 15:29:45.616 EDT,"admin","edb",4047,"[local]",
5967ca05.fcf,1,"idle",2017-07-13 15:29:09 EDT,4/33,0,AUDIT,00000,
"statement: SELECT * FROM edb_tbl;",,,,,,,,,"psql.bin","SELECT",""

2017-07-13 15:29:45.616 EDT,"admin","edb",4047,"[local]",
5967ca05.fcf,2,"idle",2017-07-13 15:29:09 EDT,4/34,0,AUDIT,00000,
"statement: TRUNCATE edb_tbl;",,,,,,,,,"psql.bin","TRUNCATE TABLE",""
```

Case 3: Changes made in database `auditdb` by role `admin`. Only `create table`, `insert`, and `update` statements are audited:

```
$ psql auditdb admin
Password for user admin:
psql.bin (12.0.0)
Type "help" for help.

auditdb=# SHOW edb_audit_statement;
     edb_audit_statement
------------------------------
 create table, insert, update
(1 row)

auditdb=# CREATE TABLE audit_tbl_2 (f1 INTEGER PRIMARY KEY, f2 TEXT);
CREATE TABLE
auditdb=# INSERT INTO audit_tbl_2 VALUES (1, 'Row 1');
```

```
INSERT 0 1
auditdb=# SELECT * FROM audit_tbl_2;                          <== Should not be audited
 f1 |  f2
----+-------
  1 | Row 1
(1 row)

auditdb=# TRUNCATE audit_tbl_2;                               <== Should not be audited
TRUNCATE TABLE
```

Continuation of the audit log file now appears as follows. The next to last two entries representing the third case show only `CREATE TABLE audit_tbl_2` and `INSERT INTO audit_tbl_2` statements. The `SELECT * FROM audit_tbl_2` and `TRUNCATE audit_tbl_2` statements were not audited.

```
2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,1,"idle",2017-07-13 15:25:59 EDT,7/4,0,AUDIT,00000,
"statement: CREATE TABLE audit_tbl (f1 INTEGER PRIMARY KEY, f2 TEXT);",,,,,,,,,
"psql.bin","CREATE TABLE",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,2,"idle",2017-07-13 15:25:59 EDT,7/5,0,AUDIT,00000,
"statement: INSERT INTO audit_tbl VALUES (1, 'Row 1');",,,,,,,,,"psql.bin","INSERT",""

2017-07-13 15:26:17.426 EDT,"enterprisedb","auditdb",4024,"[local]",
5967c947.fb8,3,"idle",2017-07-13 15:25:59 EDT,7/6,0,AUDIT,00000,
"statement: UPDATE audit_tbl SET f2 = 'Row A' WHERE f1 = 1;",,,,,,,,,"psql.bin","UPDATE",""

2017-07-13 15:29:45.616 EDT,"admin","edb",4047,"[local]",
5967ca05.fcf,1,"idle",2017-07-13 15:29:09 EDT,4/33,0,AUDIT,00000,
"statement: SELECT * FROM edb_tbl;",,,,,,,,,"psql.bin","SELECT",""

2017-07-13 15:29:45.616 EDT,"admin","edb",4047,"[local]",
5967ca05.fcf,2,"idle",2017-07-13 15:29:09 EDT,4/34,0,AUDIT,00000,
"statement: TRUNCATE edb_tbl;",,,,,,,,,"psql.bin","TRUNCATE TABLE",""

2017-07-13 15:35:45.309 EDT,"admin","auditdb",4085,"[local]",
5967cb81.ff5,1,"idle",2017-07-13 15:35:29 EDT,4/72,0,AUDIT,00000,
"statement: CREATE TABLE audit_tbl_2 (f1 INTEGER PRIMARY KEY, f2 TEXT);",,,,,,,,,
"psql.bin","CREATE TABLE",""

2017-07-13 15:35:45.309 EDT,"admin","auditdb",4085,"[local]",
5967cb81.ff5,2,"idle",2017-07-13 15:35:29 EDT,4/73,0,AUDIT,00000,
"statement: INSERT INTO audit_tbl_2 VALUES (1, 'Row 1');",,,,,,,,,"psql.bin","INSERT",""

2017-07-13 15:38:42.028 EDT,,,3942,,5967c934.f66,1,,2017-07-13 15:25:40
EDT,,0,LOG,00000,"database system is shut down",,,,,,,,,"","",""
```

3.5.4 Audit Log File

The audit log file can be generated in either CSV or XML format depending upon the setting of the `edb_audit` configuration parameter. The XML format contains less information than the CSV format.

The information in the audit log is based on the logging performed by PostgreSQL as described in Section 19.8.4 “Using CSV-Format Log Output” within Section 19.8 “Error Reporting and Logging” in the PostgreSQL core documentation, available at:

https://www.postgresql.org/docs/12/static/runtime-config-logging.html

The following table lists the fields in the order they appear in the CSV audit log format. The table contains the following information:

- **Field.** Name of the field as shown in the sample table definition in the PostgreSQL documentation as previously referenced.
- **XML Element/Attribute.** For the XML format, name of the XML element and its attribute (if used), referencing the value. **Note:** `n/a` indicates that there is no XML representation for this field.
- **Data Type.** Data type of the field as given by the PostgreSQL sample table definition.
- **Description.** Description of the field. For certain fields, no output is generated in the audit log as those fields are not supported by auditing. Those fields are indicated by "Not supported".

The fields with the `Description` of "Not supported" appear as consecutive commas (`,,`) in the CSV format.

Table 3-3 - Audit Log Fields

Field	XML Element/Attribute	Data Type	Description
`log_time`	`event/time`	`timestamp with time zone`	Log date/time of the statement.
`user_name`	`event/user`	`text`	Database user who executed the statement.
`database_name`	`event/database`	`text`	Database in which the statement was executed.
`process_id`	`event/process_id`	`integer`	Operating system process ID in which the statement was executed.
`connection_from`	`event/remote_host`	`text`	Host and port location from where the statement was executed.
`session_id`	`event/session_id`	`text`	Session ID in which the statement was executed.
`session_line_num`	`n/a`	`bigint`	Order of the statement within the session.
`process_status`	`n/a`	`text`	Processing status.
`session_start_time`	`n/a`	`timestamp with time zone`	Date/time when the session was started.
`virtual_transaction_id`	`n/a`	`text`	Virtual transaction ID of the statement.
`transaction_id`	`event/transaction_id`	`bigint`	Regular transaction ID of the statement.
`error_severity`	`message`	`text`	Statement severity. Values are `AUDIT` for audited statements and `ERROR` for any resulting error messages.

Field	XML Element/Attribute	Data Type	Description
`sql_state_code`	`n/a`	`text`	SQL state code returned for the statement.
`message`	`message`	`text`	The SQL statement that was attempted for execution.
`detail`	`n/a`	`text`	Error message detail. (Not supported)
`hint`	`n/a`	`text`	Hint (Not supported)
`internal_query`	`n/a`	`text`	Internal query that led to the error, if any. (Not supported)
`internal_query_pos`	`n/a`	`integer`	Character count of the error position, therein. (Not supported)
`context`	`n/a`	`text`	Error context. (Not supported)
`query`	`n/a`	`text`	User query that led to the error. (For errors only)
`query_pos`	`n/a`	`integer`	Character count of the error position, therein. (For errors only)
`location`	`n/a`	`text`	Location of the error in the PostgreSQL source code. (Not supported)
`application_name`	`n/a`	`text`	Name of the application from which the statement was executed. (for example, `psql.bin`).
`command_tag`	`event/command_tag`	`text`	SQL command of the statement.
`audit_tag`	`event/audit_tag`	`text`	Value specified by the `audit_tag` parameter in the configuration file.

The following examples are generated in the CSV and XML formats.

The non-default audit settings in the `postgresql.conf` file are as follows:

```
edb_audit = 'csv'
edb_audit_connect = 'all'
edb_audit_disconnect = 'all'
edb_audit_statement = 'ddl, dml, select, error'
edb_audit_tag = 'edbaudit'
```

The `edb_audit` parameter is changed to `xml` when generating the XML format.

The audited session is the following:

```
$ psql edb enterprisedb
Password for user enterprisedb:
psql.bin (12.0.0)
Type "help" for help.

edb=# CREATE SCHEMA edb;
CREATE SCHEMA
```

```
edb=# SET search_path TO edb;
SET
edb=# CREATE TABLE dept (
edb(#     deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
edb(#     dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
edb(#     loc             VARCHAR2(13)
edb(# );
CREATE TABLE
edb=# INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT 0 1
edb=# UPDATE department SET loc = 'BOSTON' WHERE deptno = 10;
ERROR:  relation "department" does not exist
LINE 1: UPDATE department SET loc = 'BOSTON' WHERE deptno = 10;
               ^
edb=# UPDATE dept SET loc = 'BOSTON' WHERE deptno = 10;
UPDATE 1
edb=# SELECT * FROM dept;
 deptno |   dname    |  loc
--------+------------+--------
     10 | ACCOUNTING | BOSTON
(1 row)

edb=# \q
```

CSV Audit Log File

The following is the CSV format of the audit log file.

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,1,"authentication",2017-07-17 13:28:44 EDT,6/2,0,AUDIT,00000,
"connection authorized: user=enterprisedb database=edb",,,,,,,,,"","","edbaudit"

2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,2,"idle",2017-07-17 13:28:44 EDT,6/4,0,AUDIT,00000,
"statement: CREATE SCHEMA edb;",,,,,,,,,"psql.bin","CREATE SCHEMA","edbaudit"

2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,3,"idle",2017-07-17 13:28:44 EDT,6/6,0,AUDIT,00000,
"statement: CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);",,,,,,,,,"psql.bin","CREATE TABLE","edbaudit"

2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,4,"idle",2017-07-17 13:28:44 EDT,6/7,0,AUDIT,00000,
"statement: INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');",,,,,,,,,
"psql.bin","INSERT","edbaudit"

2017-07-17 13:28:44.235 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,5,"idle",2017-07-17 13:28:44 EDT,6/8,0,AUDIT,00000,
"statement: UPDATE department SET loc = 'BOSTON' WHERE deptno = 10;",,,,,,,,,
"psql.bin","UPDATE","edbaudit"

2017-07-17 13:29:59.833 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,6,"UPDATE",2017-07-17 13:28:44 EDT,6/8,0,ERROR,42P01,
"relation ""department"" does not exist",,,,,,
"UPDATE department SET loc = 'BOSTON' WHERE deptno = 10;",8,,"psql.bin","","edbaudit"

2017-07-17 13:29:59.833 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,7,"idle",2017-07-17 13:28:44 EDT,6/9,0,AUDIT,00000,
"statement: UPDATE dept SET loc = 'BOSTON' WHERE deptno = 10;",,,,,,,,,
"psql.bin","UPDATE","edbaudit"

2017-07-17 13:29:59.833 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,8,"idle",2017-07-17 13:28:44 EDT,6/10,0,AUDIT,00000,
"statement: SELECT * FROM dept;",,,,,,,,,"psql.bin","SELECT","edbaudit"

2017-07-17 13:29:59.833 EDT,"enterprisedb","edb",4068,"[local]",
596cf3cc.fe4,9,"idle",2017-07-17 13:28:44 EDT,,0,AUDIT,00000,
"disconnection: session time: 0:02:01.511 user=enterprisedb database=edb
host=[local]",,,,,,,,,"psql.bin","SELECT","edbaudit"

2017-07-17 13:30:53.617 EDT,,,3987,,596cf3b3.f93,1,,2017-07-17 13:28:19 EDT,,0,LOG,00000,
"database system is shut down",,,,,,,,,"","","edbaudit"
```

XML Audit Log File

The following is the XML format of the audit log file. The output has been formatted for more clarity in the appearance in the example.

```
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:36:55 EDT"
       transaction_id="0" type="connect" audit_tag="edbaudit">
  <message>AUDIT:  connection authorized: user=enterprisedb database=edb</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:02 EDT"
       transaction_id="0" type="create" command_tag="CREATE SCHEMA" audit_tag="edbaudit">
    <message>AUDIT:  statement: CREATE SCHEMA edb;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:19 EDT"
       transaction_id="0" type="create" command_tag="CREATE TABLE" audit_tag="edbaudit">
  <message>AUDIT:  statement: CREATE TABLE dept (
              deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
              dname           VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
              loc             VARCHAR2(13));
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:29 EDT"
       transaction_id="0" type="insert" command_tag="INSERT" audit_tag="edbaudit">
  <message>AUDIT:  statement: INSERT INTO dept VALUES
                              (10,&apos;ACCOUNTING&apos;,&apos;NEW YORK&apos;);
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:40 EDT"
       transaction_id="0" type="update" command_tag="UPDATE" audit_tag="edbaudit">
  <message>AUDIT:  statement: UPDATE department SET
                              loc = &apos;BOSTON&apos; WHERE deptno = 10;
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:40 EDT"
       transaction_id="0" type="error" audit_tag="edbaudit">
  <message>ERROR:  relation &quot;department&quot; does not exist at character 8
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:51 EDT"
       transaction_id="0" type="update" command_tag="UPDATE" audit_tag="edbaudit">
  <message>AUDIT:  statement: UPDATE dept SET loc = &apos;BOSTON&apos; WHERE deptno = 10;
  </message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
```

```
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:37:59 EDT"
       transaction_id="0" type="select" command_tag="SELECT" audit_tag="edbaudit">
  <message>AUDIT:  statement: SELECT * FROM dept;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="596cf5b7.12a8" process_id="4776" time="2017-07-17 13:38:01 EDT"
       transaction_id="0" type="disconnect" command_tag="SELECT" audit_tag="edbaudit">
  <message>AUDIT:  disconnection: session time: 0:01:05.814
                   user=enterprisedb database=edb host=[local]
  </message>
</event>
<event process_id="4696" time="2017-07-17 13:38:08 EDT"
       transaction_id="0" type="shutdown" audit_tag="edbaudit">
  <message>LOG:  database system is shut down</message>
</event>
```

3.5.5 Using Error Codes to Filter Audit Logs

Advanced Server includes an extension that you can use to exclude log file entries that include a user-specified error code from the Advanced Server log files. To filter audit log entries, you must first enable the extension by modifying the `postgresql.conf` file, adding the following value to the values specified in the `shared_preload_libraries` parameter:

```
$libdir/edb_filter_log
```

Then, use the `edb_filter_log.errcodes` parameter to specify any error codes you wish to omit from the log files:

```
edb_filter_log.errcode = 'error_code'
```

Where *`error_code`* specifies one or more error codes that you wish to omit from the log file. Provide multiple error codes in a comma-delimited list.

For example, if `edb_filter_log` is enabled, and `edb_filter_log.errcode` is set to `'23505,23502,22012'`, any log entries that return one of the following SQLSTATE errors:

`23505` (for violating a unique constraint)

`23502` (for violating a not-null constraint)

`22012` (for dividing by zero)

will be omitted from the log file.

For a complete list of the error codes supported by Advanced Server audit log filtering, please see the core documentation at:

https://www.postgresql.org/docs/12/static/errcodes-appendix.html

3.5.6 Using Command Tags to Filter Audit Logs

Each entry in the log file except for those displaying an error message contains a *command tag*, which is the SQL command executed for that particular log entry.

The command tag makes it possible to use subsequent tools to scan the log file to find entries related to a particular SQL command.

The following is an example in XML form. The output has been formatted for easier appearance in the example.

The command tag is displayed as the `command_tag` attribute of the `event` element with values `CREATE ROLE`, `ALTER ROLE`, and `DROP ROLE` in the example.

```
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="595e8537.10f1" process_id="4337" time="2017-07-06 14:45:18 EDT"
       transaction_id="0" type="create"
       command_tag="CREATE ROLE">
  <message>AUDIT:  statement: CREATE ROLE newuser WITH LOGIN;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="595e8537.10f1" process_id="4337" time="2017-07-06 14:45:31 EDT"
       transaction_id="0" type="error">
  <message>ERROR:  unrecognized role option &quot;super&quot; at character 25
           STATEMENT:  ALTER ROLE newuser WITH SUPER USER;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="595e8537.10f1" process_id="4337" time="2017-07-06 14:45:38 EDT"
       transaction_id="0" type="alter" command_tag="ALTER ROLE">
  <message>AUDIT:  statement: ALTER ROLE newuser WITH SUPERUSER;</message>
</event>
<event user="enterprisedb" database="edb" remote_host="[local]"
       session_id="595e8537.10f1" process_id="4337" time="2017-07-06 14:45:46 EDT"
       transaction_id="0" type="drop" command_tag="DROP ROLE">
  <message>AUDIT:  statement: DROP ROLE newuser;</message>
</event>
```

The following is the same example in CSV form. The command tag is the next to last column of each entry. (The last column appears empty as `""`, which would be the value provided by the `edb_audit_tag` parameter.)

Each audit log entry has been split and displayed across multiple lines, and a blank line has been inserted between the audit log entries for more clarity in the appearance of the results.

```
2017-07-06 14:47:22.294 EDT,"enterprisedb","edb",4720,"[local]",
595e85b2.1270,1,"idle",2017-07-06 14:47:14 EDT,6/4,0,AUDIT,00000,
"statement: CREATE ROLE newuser WITH LOGIN;",,,,,,,,,"psql.bin","CREATE ROLE",""

2017-07-06 14:47:29.694 EDT,"enterprisedb","edb",4720,"[local]",
595e85b2.1270,2,"idle",2017-07-06 14:47:14 EDT,6/5,0,ERROR,42601,
"unrecognized role option ""super""",,,,,,"ALTER ROLE newuser WITH SUPER USER;",25,,
"psql.bin","",""

2017-07-06 14:47:29.694 EDT,"enterprisedb","edb",4720,"[local]",
595e85b2.1270,3,"idle",2017-07-06 14:47:14 EDT,6/6,0,AUDIT,00000,
"statement: ALTER ROLE newuser WITH SUPERUSER;",,,,,,,,,"psql.bin","ALTER ROLE",""

2017-07-06 14:47:29.694 EDT,"enterprisedb","edb",4720,"[local]",
595e85b2.1270,4,"idle",2017-07-06 14:47:14 EDT,6/7,0,AUDIT,00000,
"statement: DROP ROLE newuser;",,,,,,,,,"psql.bin","DROP ROLE",""
```

3.5.7 Redacting Passwords from Audit Logs

You can use the `edb_filter_log.redact_password_commands` extension to instruct the server to redact stored passwords from the log file. Note that the module only recognizes the following syntax:

```
{CREATE|ALTER} {USER|ROLE|GROUP} identifier { [WITH] [ENCRYPTED]
PASSWORD 'nonempty_string_literal' | IDENTIFIED BY {
'nonempty_string_literal' | bareword } } [ REPLACE {
'nonempty_string_literal' | bareword } ]
```

When such a statement is logged by `log_statement`, the server will redact the old and new passwords to 'x'. For example, the command:

```
ALTER USER carol PASSWORD '1safepwd' REPLACE 'old_pwd';
```

Will be added to log files as:

```
statement: ALTER USER carol PASSWORD 'x' REPLACE 'x';
```

When a statement that includes a redacted password is logged, the server redacts the statement text. When the statement is logged as context for some other message, the server omits the statement from the context.

To enable password redaction, you must first enable the extension by modifying the `postgresql.conf` file, adding the following value to the values specified in the `shared_preload_libraries` parameter:

```
$libdir/edb_filter_log
```

Then, set `edb_filter_log.redact_password_commands` to `true`:

```
edb_filter_log.redact_password_commands = true
```

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

3.6 Unicode Collation Algorithm

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

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

- Unicode supports a vast number of languages. Letters that may be common to several languages may be expected to collate in different orders depending upon the language.
- Characters that appear with letters in certain languages such as accents or umlauts have an impact on the expected collation depending upon the language.
- In some languages, combinations of several consecutive characters should be treated as a single character with regards to its collation sequence.
- There may be certain preferences as to the collation of letters according to case. For example, should the lowercase form of a letter collate before 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.

3.6.1 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.

3.6.2 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*.

3.6.2.1 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 3.6.2.2 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.

3.6.2.2 Collation Attributes

When creating an ICU collation, the desired characteristics of the collation must be specified. As discussed in Section 3.6.2.1, 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 14 "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.

3.6.3 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)
```

4 Security

The chapter describes various features providing for added security.

4.1 Protecting Against SQL Injection Attacks

Advanced Server provides protection against SQL injection attacks. A *SQL injection attack* is an attempt to compromise a database by running SQL statements whose results provide clues to the attacker as to the content, structure, or security of that database.

Preventing a SQL injection attack is normally the responsibility of the application developer. The database administrator typically has little or no control over the potential threat. The difficulty for database administrators is that the application must have access to the data to function properly.

SQL/Protect is a module that allows a database administrator to protect a database from SQL injection attacks. SQL/Protect provides a layer of security in addition to the normal database security policies by examining incoming queries for common SQL injection profiles.

SQL/Protect gives the control back to the database administrator by alerting the administrator to potentially dangerous queries and by blocking these queries.

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 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 an Advanced Server user or group that the database administrator has chosen to monitor using SQL/Protect. (In Advanced Server, users and groups are collectively referred to as *roles*.)

Each protected role can be customized for the types of SQL injection attacks for which it is to be monitored, thus providing different levels of protection by role and significantly reducing the user maintenance load for DBAs.

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 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 be installed in the `lib` subdirectory of your Advanced Server home directory. For Windows, this should be done by the Advanced Server installer. For Linux, install the `edb-as`*`xx`*`-server-sqlprotect` RPM package where *`xx`* is the Advanced Server version number.

You will also need the SQL script file `sqlprotect.sql` located in the `share/contrib` subdirectory of your Advanced Server home directory.

You must configure the database server to use SQL/Protect, and you must configure each database that you want SQL/Protect to monitor:

- The database server configuration file, `postgresql.conf`, must be modified by adding and enabling configuration parameters used by SQL/Protect.
- Database objects used by SQL/Protect must be installed in each database that you want SQL/Protect to monitor.

Step 1: Edit the following configuration parameters in the `postgresql.conf` file located in the `data` subdirectory of your Advanced Server home directory.

- **shared_preload_libraries.** Add `$libdir/sqlprotect` to the list of libraries.
- **edb_sql_protect.enabled.** Controls whether or not SQL/Protect is actively monitoring protected roles by analyzing SQL statements issued by those roles and reacting according to the setting of `edb_sql_protect.level`. When you are ready to begin monitoring with SQL/Protect set this parameter to `on`. If this parameter is omitted, the default is `off`.
- **edb_sql_protect.level.** Sets the action taken by SQL/Protect when a SQL statement is issued by a protected role. If this parameter is omitted, the default behavior is `passive`. Initially, set this parameter to `learn`. See 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 3.1.3.12.8 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 3.1.3.12.7 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 3.1.3.12.9 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.

On Linux: Invoke the Advanced Server service script with the `restart` option:

On a Redhat or CentOS 7.x installation, use the command:

```
systemctl restart edb-as-12
```

On Windows: Use the Windows `Services` applet to restart the service named `edb-as-12`.

Step 3: For each database that you want to protect from SQL injection attacks, connect to the database as a superuser (either `enterprisedb` or `postgres`, depending upon your installation options) and run the script `sqlprotect.sql` located in the `share/contrib` subdirectory of your Advanced Server home directory. The script creates the SQL/Protect database objects in a schema named `sqlprotect`.

The following example shows this process to set up protection for a database named `edb`:

```
$ /usr/edb/as12/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (12.0.0, server 12.0.0)
Type "help" for help.
```

```
edb=# \i /usr/edb/as12/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.

```
$ /usr/edb/as12/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql.bin (12.0.0, server 12.0.0)
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 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 Advanced Server database server by simply using the operating system's copy utility to create a binary image of the Advanced Server data files (file system backup method), then this section does not apply.

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 Advanced Server to uniquely identify each database object.

When a database object is created, Advanced Server assigns an OID to the object, which is then used whenever a reference is needed to the object in the database catalogs. If you create the same database object in two databases, such as a table with the same `CREATE TABLE` statement, each table is assigned a different OID in each database.

In a backup and restore operation that results in the re-creation of the backed up database objects, the restored objects end up with different OIDs in the new database than what they were assigned in the original database. As a result, the OIDs referencing databases, roles, and relations stored in the `edb_sql_protect` and `edb_sql_protect_rel` tables are no longer valid when these tables are simply dumped to a backup file and then restored to a new database.

The following sections describe two functions, `export_sqlprotect` and `import_sqlprotect`, that are used specifically for backing up and restoring SQL/Protect tables in order to ensure the OIDs in the SQL/Protect tables reference the correct database objects after the 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 /usr/edb/as12/bin
$ ./pg_dump -U enterprisedb -Fp -f /tmp/edb.dmp edb
Password:
$
```

Step 2: Connect to the database as a superuser and export the SQL/Protect data using the `export_sqlprotect('`*`sqlprotect_file`*`')` function where *`sqlprotect_file`* is the fully qualified path to a file where the SQL/Protect data is to be saved.

The `enterprisedb` operating system account (`postgres` if you installed Advanced Server in PostgreSQL compatibility mode) must have read and write access to the directory specified in *`sqlprotect_file`*.

```
edb=# SELECT sqlprotect.export_sqlprotect('/tmp/sqlprotect.dmp');
 export_sqlprotect
-------------------

(1 row)
```

The files `/tmp/edb.dmp` and `/tmp/sqlprotect.dmp` comprise your total database backup.

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`:

```
$ /usr/edb/as12/bin/psql -d newdb -U enterprisedb -f /tmp/edb.dmp
Password for user enterprisedb:
SET
SET
SET
SET
SET
COMMENT
CREATE SCHEMA
    .
    .
    .
```

Step 2: Connect to the new database as a superuser and delete all rows from the `edb_sql_protect_rel` table.

This step removes any existing rows in the `edb_sql_protect_rel` table that were backed up from the original database. These rows do not contain the correct OIDs relative to the database where the backup file has been restored.

```
$ /usr/edb/as12/bin/psql -d newdb -U enterprisedb
Password for user enterprisedb:
psql.bin (12.0.0, server 12.0.0)
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 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. Restart the database server or reload the configuration file as appropriate.

You can now monitor the database using SQL/Protect.

4.2 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.

4.3 sslutils

`sslutils` is a Postgres extension that provides SSL certificate generation functions to Advanced Server for use by the EDB Postgres Enterprise Manager server. `sslutils` is installed by using the `edb-as`*`xx`*`-server-sslutils` RPM package where *`xx`* is the Advanced Server version number.

The `sslutils` package provides the functions shown in the following sections.

In these sections, each parameter in the function's parameter list is described by *`parameter n`* under the **Parameters** subsection where *`n`* refers to the *`n`*th ordinal position (for example, first, second, third, etc.) within the function's parameter list.

4.3.1 openssl_rsa_generate_key

The `openssl_rsa_generate_key` function generates an RSA private key. The function signature is:

```
    openssl_rsa_generate_key(integer) RETURNS text
```

When invoking the function, pass the number of bits as an integer value; the function returns the generated key.

4.3.2 openssl_rsa_key_to_csr

The `openssl_rsa_key_to_csr` function generates a certificate signing request (CSR). The signature is:

```
    openssl_rsa_key_to_csr(text, text, text, text, text, text,
    text) RETURNS text
```

The function generates and returns the certificate signing request.

Parameters

`parameter 1`

The name of the RSA key file.

`parameter 2`

The common name (e.g., `agentN`) of the agent that will use the signing request.

`parameter 3`

The name of the country in which the server resides.

parameter 4

The name of the state in which the server resides.

parameter 5

The location (city) within the state in which the server resides.

parameter 6

The name of the organization unit requesting the certificate.

parameter 7

The email address of the user requesting the certificate.

4.3.3 openssl_csr_to_crt

The `openssl_csr_to_crt` function generates a self-signed certificate or a certificate authority certificate. The signature is:

```
    openssl_csr_to_crt(text, text, text) RETURNS text
```

The function returns the self-signed certificate or certificate authority certificate.

Parameters

parameter 1

The name of the certificate signing the request.

parameter 2

The path to the certificate authority certificate, or `NULL` if generating a certificate authority certificate.

parameter 3

The path to the certificate authority's private key or (if argument 2 is `NULL`) the path to a private key.

4.3.4 openssl_rsa_generate_crl

The `openssl_rsa_generate_crl` function generates a default certificate revocation list. The signature is:

```
    openssl_rsa_generate_crl(text, text) RETURNS text
```

The function returns the certificate revocation list.

Parameters

`parameter 1`

The path to the certificate authority certificate.

`parameter 2`

The path to the certificate authority private key.

4.4 Data Redaction

Data redaction is a technique that limits sensitive data exposure by dynamically changing data as it is displayed for certain users.

For example, a social security number (SSN) is stored as `021-23-9567`. Privileged users can see the full SSN, while other users only see the last four digits `xxx-xx-9567`.

Data redaction is implemented by defining a function for each field to which redaction is to be applied. The function returns the value that should be displayed to the users subject to the data redaction.

So for example, for the SSN field, the redaction function would return `xxx-xx-9567` for an input SSN of `021-23-9567`.

For a salary field, a redaction function would always return `$0.00` regardless of the input salary value.

These functions are then incorporated into a redaction policy by using the `CREATE REDACTION POLICY` command. This command specifies the table on which the policy applies, the table columns to be affected by the specified redaction functions, expressions to determine which session users are to be affected, and other options.

The `edb_data_redaction` parameter in the `postgresql.conf` file then determines whether or not data redaction is to be applied.

By default, the parameter is enabled so the redaction policy is in effect and the following occurs:

- Superusers and the table owner bypass data redaction and see the original data.
- All other users get the redaction policy applied and see the reformatted data.

If the parameter is disabled by having it set to `FALSE` during the session, then the following occurs:

- Superusers and the table owner bypass data redaction and see the original data.
- All other users get will get an error.

A redaction policy can be changed by using the `ALTER REDACTION POLICY` command, or it can be eliminated using the `DROP REDACTION POLICY` command.

The redaction policy commands are described in more detail in the subsequent sections.

4.4.1 CREATE REDACTION POLICY

`CREATE REDACTION POLICY` defines a new data redaction policy for a table.

Synopsis

```
CREATE REDACTION POLICY name ON table_name
  [ FOR ( expression ) ]
  [ ADD [ COLUMN ] column_name USING funcname_clause
    [ WITH OPTIONS ( [ redaction_option ]
      [, redaction_option ] )
    ]
  ] [, ...]
```

where *redaction_option* is:

```
{ SCOPE scope_value |
  EXCEPTION exception_value }
```

Description

The `CREATE REDACTION POLICY` command defines a new column-level security policy for a table by redacting column data using redaction function. A newly created data redaction policy will be enabled by default. The policy can be disabled using `ALTER REDACTION POLICY ... DISABLE`.

`FOR ( expression )`

This form adds a redaction policy expression.

`ADD [ COLUMN ]`

This optional form adds a column of the table to the data redaction policy. The `USING` specifies a redaction function expression. Multiple `ADD [ COLUMN ]` form can be used, if you want to add multiple columns of the table to the data redaction policy being created. The optional `WITH OPTIONS ( ... )` clause specifies a scope and/or an exception to the data redaction policy to be applied. If the scope and/or exception are not specified, the default values for scope and exception will be `query` and `none` respectively.

Parameters

name

The name of the data redaction policy to be created. This must be distinct from the name of any other existing data redaction policy for the table.

`table_name`

The name (optionally schema-qualified) of the table the data redaction policy applies to.

`expression`

The data redaction policy expression. No redaction will be applied if this expression evaluates to false.

`column_name`

Name of the existing column of the table on which the data redaction policy being created.

`funcname_clause`

The data redaction function which decides how to compute the redacted column value. Return type of the redaction function should be same as the column type on which data redaction policy being added.

`scope_value`

The scope identified the query part where redaction to be applied for the column. Scope value could be `query`, `top_tlist` or `top_tlist_or_error`. If the scope is `query` then, the redaction applied on the column irrespective of where it appears in the query. If the scope is `top_tlist` then, the redaction applied on the column only when it appears in the query's top target list. If the scope is `top_tlist_or_error` the behavior will be same as the `top_tlist`, but throws an errors when the column appears anywhere else in the query.

`exception_value`

The exception identified the query part where redaction to be exempted. Exception value could be `none`, `equal` or `leakproof`. If exception is `none` then there is no exemption. If exception is `equal`, then the column is not redacted when used in an equality test. If exception is `leakproof`, the column will is not redacted when a leakproof function is applied to it.

Notes:

You must be the owner of a table to create or change data redaction policies for it.

The superuser and the table owner are exempt from the data redaction policy.

Examples

Below is an example of how this feature can be used in production environments. Create the components for a data redaction policy on `employees` table:

```
CREATE TABLE employees (
 id          integer GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
 name        varchar(40) NOT NULL,
 ssn         varchar(11) NOT NULL,
 phone       varchar(10),
 birthday    date,
 salary      money,
 email       varchar(100)
);

-- Insert some data
INSERT INTO employees (name, ssn, phone, birthday, salary, email)
VALUES
( 'Sally Sample', '020-78-9345', '5081234567', '1961-02-02', 51234.34,
'sally.sample@enterprisedb.com'),
( 'Jane Doe', '123-33-9345', '6171234567', '1963-02-14', 62500.00, 'jane.doe@gmail.com'),
( 'Bill Foo', '123-89-9345', '9781234567','1963-02-14', 45350, 'william.foe@hotmail.com');

-- Create a user hr who can see all the data in employees
CREATE USER hr;
-- Create a normal user
CREATE USER alice;
GRANT ALL ON employees TO hr, alice;

-- Create redaction function in which actual redaction logic resides
CREATE OR REPLACE FUNCTION redact_ssn (ssn varchar(11)) RETURN varchar(11) IS
BEGIN
   /* replaces 020-12-9876 with xxx-xx-9876 */
   return overlay (ssn placing 'xxx-xx' from 1) ;
END;

CREATE OR REPLACE FUNCTION redact_salary () RETURN money IS BEGIN return 0::money; END;
```

Now create a data redaction policy on `employees` to redact column `ssn` which should be accessible in equality condition and `salary` with default scope and exception. The redaction policy will be exempt for the `hr` user.

```
CREATE REDACTION POLICY redact_policy_personal_info ON employees FOR (session_user != 'hr')
ADD COLUMN ssn USING redact_ssn(ssn) WITH OPTIONS (SCOPE query, EXCEPTION equal),
ADD COLUMN salary USING redact_salary();
```

The visible data for the `hr` user will be:

```
-- hr can view all columns data
edb=# \c edb hr
edb=> SELECT * FROM employees;
 id | name         | ssn         | phone      | birthday           | salary     | email
----+--------------+-------------+------------+--------------------+------------+-------------
-------------------
  1 | Sally Sample | 020-78-9345 | 5081234567 | 02-FEB-61 00:00:00 | $51,234.34 |
sally.sample@enterprisedb.com
  2 | Jane Doe     | 123-33-9345 | 6171234567 | 14-FEB-63 00:00:00 | $62,500.00 |
jane.doe@gmail.com
```

```
  3 | Bill Foo     | 123-89-9345 | 9781234567 | 14-FEB-63 00:00:00 | $45,350.00 |
william.foe@hotmail.com
(3 rows)
```

The visible data for the normal user `alice` will be:

```
-- Normal user cannot see salary and ssn number.
edb=> \c edb alice
edb=> SELECT * FROM employees;
id  | name         | ssn         | phone      | birthday             | salary | email
----+--------------+-------------+------------+----------------------+--------+-----------------
--------------
  1 | Sally Sample | xxx-xx-9345 | 5081234567 | 02-FEB-61 00:00:00 |  $0.00 |
sally.sample@enterprisedb.com
  2 | Jane Doe     | xxx-xx-9345 | 6171234567 | 14-FEB-63 00:00:00 |  $0.00 |
jane.doe@gmail.com
  3 | Bill Foo     | xxx-xx-9345 | 9781234567 | 14-FEB-63 00:00:00 |  $0.00 |
william.foe@hotmail.com
(3 rows)
```

But `ssn` data is accessible when it used for equality check due to `exception_value` setting.

```
-- Get ssn number starting from 123
edb=> SELECT * FROM employees WHERE substring(ssn from 0 for 4) = '123';
 id | name     | ssn         | phone      | birthday           | salary | email
----+----------+-------------+------------+--------------------+--------+---------------------
----
  2 | Jane Doe | xxx-xx-9345 | 6171234567 | 14-FEB-63 00:00:00 |  $0.00 | jane.doe@gmail.com
  3 | Bill Foo | xxx-xx-9345 | 9781234567 | 14-FEB-63 00:00:00 |  $0.00 |
william.foe@hotmail.com
(2 rows)
```

Caveats

1. The data redaction policy created on inheritance hierarchies will not be cascaded. For example, if the data redaction policy is created for a parent, it will not be applied to the child table, which inherits it and vice versa. Someone who has access to these child tables can see the non-redacted data. For information about inheritance hierarchies, see Section 5.9 in the PostgreSQL Core Documentation available at:

   https://www.postgresql.org/docs/12/static/ddl-inherit.html

2. If the superuser or the table owner has created any materialized view on the table and has provided the access rights `GRANT SELECT` on the table and the materialized view to any non-superuser, then the non-superuser will be able to access the non-redacted data through the materialized view.

3. The objects accessed in the redaction function body should be schema qualified otherwise `pg_dump` might fail.

Compatibility

`CREATE REDACTION POLICY` is an EnterpriseDB extension.

See Also

ALTER REDACTION POLICY, DROP REDACTION POLICY

4.4.2 ALTER REDACTION POLICY

`ALTER REDACTION POLICY` changes the definition of data redaction policy for a table.

Synopsis

```
ALTER REDACTION POLICY name ON table_name RENAME TO new_name

ALTER REDACTION POLICY name ON table_name FOR ( expression )

ALTER REDACTION POLICY name ON table_name { ENABLE | DISABLE}

ALTER REDACTION POLICY name ON table_name
  ADD [ COLUMN ] column_name USING funcname_clause
    [ WITH OPTIONS ( [ redaction_option ]
      [, redaction_option ] )
    ]

ALTER REDACTION POLICY name ON table_name
  MODIFY [ COLUMN ] column_name
  {
    [ USING funcname_clause ]
  |
    [ WITH OPTIONS ( [ redaction_option ]
      [, redaction_option ] )
    ]
  }

ALTER REDACTION POLICY name ON table_name
  DROP [ COLUMN ] column_name
```

where *redaction_option* is:

```
{ SCOPE scope_value |
  EXCEPTION exception_value }
```

Description

`ALTER REDACTION POLICY` changes the definition of an existing data redaction policy.

To use `ALTER REDACTION POLICY`, you must own the table that the data redaction policy applies to.

`FOR ( expression )`

This form adds or replaces the data redaction policy expression.

`ENABLE`

Enables the previously disabled data redaction policy for a table.

`DISABLE`

Disables the data redaction policy for a table.

`ADD [ COLUMN ]`

This form adds a column of the table to the existing redaction policy. See CREATE REDACTION POLICY for the details.

`MODIFY [ COLUMN ]`

This form modifies the data redaction policy on the column of the table. You can update the redaction function clause and/or the redaction options for the column. The `USING` clause specifies the redaction function expression to be updated and the `WITH OPTIONS ( ... )` clause specifies the scope and/or the exception. For more details on the redaction function clause, the redaction scope and the redaction exception, see CREATE REDACTION POLICY.

`DROP [ COLUMN ]`

This form removes the column of the table from the data redaction policy.

Parameters

`name`

The name of an existing data redaction policy to alter.

`table_name`

The name (optionally schema-qualified) of the table that the data redaction policy is on.

`new_name`

The new name for the data redaction policy. This must be distinct from the name of any other existing data redaction policy for the table.

`expression`

The data redaction policy expression.

column_name

Name of existing column of the table on which the data redaction policy being altered or dropped.

funcname_clause

The data redaction function expression for the column. See CREATE REDACTION POLICY for details.

scope_value

The scope identified the query part where redaction to be applied for the column. See CREATE REDACTION POLICY for the details.

exception_value

The exception identified the query part where redaction to be exempted. See CREATE REDACTION POLICY for the details.

Examples

Update data redaction policy called `redact_policy_personal_info` on the table named `employees`:

```
ALTER REDACTION POLICY redact_policy_personal_info ON employees
FOR (session_user != 'hr' AND session_user != 'manager');
```

And to update data redaction function for the column `ssn` in the same policy:

```
ALTER REDACTION POLICY redact_policy_personal_info ON employees
MODIFY COLUMN ssn USING redact_ssn_new(ssn);
```

Compatibility

`ALTER REDACTION POLICY` is an EnterpriseDB extension.

See Also

CREATE REDACTION POLICY, DROP REDACTION POLICY

4.4.3 DROP REDACTION POLICY

`DROP REDACTION POLICY` removes a data redaction policy from a table.

Synopsis

```
DROP REDACTION POLICY [ IF EXISTS ] name ON table_name
  [ CASCADE | RESTRICT ]
```

Description

`DROP REDACTION POLICY` removes the specified data redaction policy from the table.

To use `DROP REDACTION POLICY`, you must own the table that the redaction policy applies to.

Parameters

`IF EXISTS`

> Do not throw an error if the data redaction policy does not exist. A notice is issued in this case.

`name`

> The name of the data redaction policy to drop.

`table_name`

> The name (optionally schema-qualified) of the table that the data redaction policy is on.

`CASCADE`
`RESTRICT`

> These keywords do not have any effect, since there are no dependencies on the data redaction policies.

Examples

To drop the data redaction policy called `redact_policy_personal_info` on the table named `employees`:

```
DROP REDACTION POLICY redact_policy_personal_info ON employees;
```

Compatibilities

`DROP REDACTION POLICY` is an EnterpriseDB extension.

See Also

CREATE REDACTION POLICY, ALTER REDACTION POLICY

4.4.4 System Catalogs

This section describes the system catalogs that store the redaction policy information.

4.4.4.1 edb_redaction_column

The catalog `edb_redaction_column` stores information of data redaction policy attached to the columns of the table.

Column	Type	References	Description
`oid`	`oid`		Row identifier (hidden attribute; must be explicitly selected)
`rdpolicyid`	`oid`	`edb_redaction_policy.oid`	The data redaction policy applies to the described column
`rdrelid`	`oid`	`pg_class.oid`	The table that the described column belongs to
`rdattnum`	`int2`	`pg_attribute.attnum`	The number of the described column
`rdscope`	`int2`		The redaction scope: `1` = query, `2` = top_tlist, `4` = top_tlist_or_error
`rdexception`	`int2`		The redaction exception: `8` = none, `16` = equal, `32` = leakproof
`rdfuncexpr`	`pg_node_tree`		Data redaction function expression

Note: The described column will be redacted if the redaction policy `edb_redaction_column.rdpolicyid` on the table is enabled and the redaction policy expression `edb_redaction_policy.rdexpr` evaluates to true.

4.4.4.2 edb_redaction_policy

The catalog `edb_redaction_policy` stores information of the redaction policies for tables.

Column	Type	References	Description
`oid`	`oid`		Row identifier (hidden attribute; must be explicitly selected)
`rdname`	`name`		The name of the data redaction policy
`rdrelid`	`oid`	`pg_class.oid`	The table to which the data redaction policy applies
`rdenable`	`boolean`		Is the data redaction policy enabled?
`rdexpr`	`pg_node_tree`		The data redaction policy expression

Note: The data redaction policy applies for the table if it is enabled and the expression ever evaluated true.

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, please refer to the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/12/static/sql-alterrole.html

A default resource group can be assigned to a database by the `ALTER DATABASE ... SET` command. For more information about the `ALTER DATABASE` command, please refer to the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/12/static/sql-alterdatabase.html

The entire database server instance can be assigned a default resource group by setting the `edb_resource_group` configuration parameter in the `postgresql.conf` file as shown by the following.

```
# - EDB Resource Manager -
#edb_max_resource_groups = 16          # 0-65536 (change requires restart)
edb_resource_group = 'resgrp_b'
```

A change to `edb_resource_group` in the `postgresql.conf` file requires a configuration file reload before it takes effect on the database server instance.

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 libpq C Library

libpq is the C application programmer's interface to Advanced Server. libpq is a set of library functions that allow client programs to pass queries to the Advanced Server and to receive the results of these queries.

libpq is also the underlying engine for several other EnterpriseDB application interfaces including those written for C++, Perl, Python, Tcl and ECPG. So some aspects of libpq's behavior will be important to the user if one of those packages is used.

Client programs that use libpq must include the header file `libpq-fe.h` and must link with the `libpq` library.

6.1 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

6.2 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 `REFCURSORs`:

- The first `REFCURSOR` contains the name of a cursor (`employees`) that contains all employees who work in a department within the range specified by the caller.

- The second REFCURSOR contains the name of a cursor (departments) that contains all of the departments in the range specified by the caller.

In this example, instead of returning a single REFCURSOR, the function returns a SETOF REFCURSOR (which means 0 or more REFCURSORS). One other important difference is that the libpq program should not expect a single REFCURSOR in the result set, but should expect two rows, each of which will contain a single value (the first row contains the name of the employees cursor, and the second row contains the name of the departments cursor).

```
CREATE OR REPLACE FUNCTION getEmpsAndDepts(p_min NUMERIC,
                                           p_max NUMERIC)
RETURN SETOF REFCURSOR AS
  employees   REFCURSOR;
  departments REFCURSOR;
BEGIN
  OPEN employees FOR
    SELECT * FROM emp WHERE deptno BETWEEN p_min AND p_max;
  RETURN NEXT employees;

  OPEN departments FOR
    SELECT * FROM dept WHERE deptno BETWEEN p_min AND p_max;
  RETURN NEXT departments;
END;
```

As in the previous example, you can use PQexec() and PQgetvalue() to call the SPL function:

```
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void fetchAllRows(PGconn *conn,
                         const char *cursorName,
                         const char *description);
static void fail(PGconn *conn, const char *msg);

int
main(int argc, char *argv[])
{
  PGconn   *conn = PQconnectdb(argv[1]);
  PGresult *result;

  if (PQstatus(conn) != CONNECTION_OK)
    fail(conn, PQerrorMessage(conn));

  result = PQexec(conn, "BEGIN TRANSACTION");
  if (PQresultStatus(result) != PGRES_COMMAND_OK)
    fail(conn, PQerrorMessage(conn));

  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 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` (see Section 6.3.1)
- `PQexecBulk` (see Section 6.3.2)
- `PQBulkFinish` (see Section 6.3.3)
- `PQexecBulkPrepared` (see Section 6.3.4)

6.3.1 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);
```

6.3.2 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 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);
```

6.3.4 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);
```

6.3.5 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" );
}
```

6.3.6 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 );
}
```

7 Debugger

The Debugger gives developers and DBAs the ability to test and debug server-side programs using a graphical, dynamic environment. The types of programs that can be debugged are SPL stored procedures, functions, triggers, and packages as well as PL/pgSQL functions and triggers.

The Debugger is integrated with and invoked from *pgAdmin 4*. On Linux, the `edb-as`*`xx`*`-server-pldebugger` RPM package where *`xx`* is the Advanced Server version number, must be installed as well. Information about pgAdmin 4 is available at:

https://www.pgadmin.org/

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 7.5.3.

7.1 Configuring the Debugger

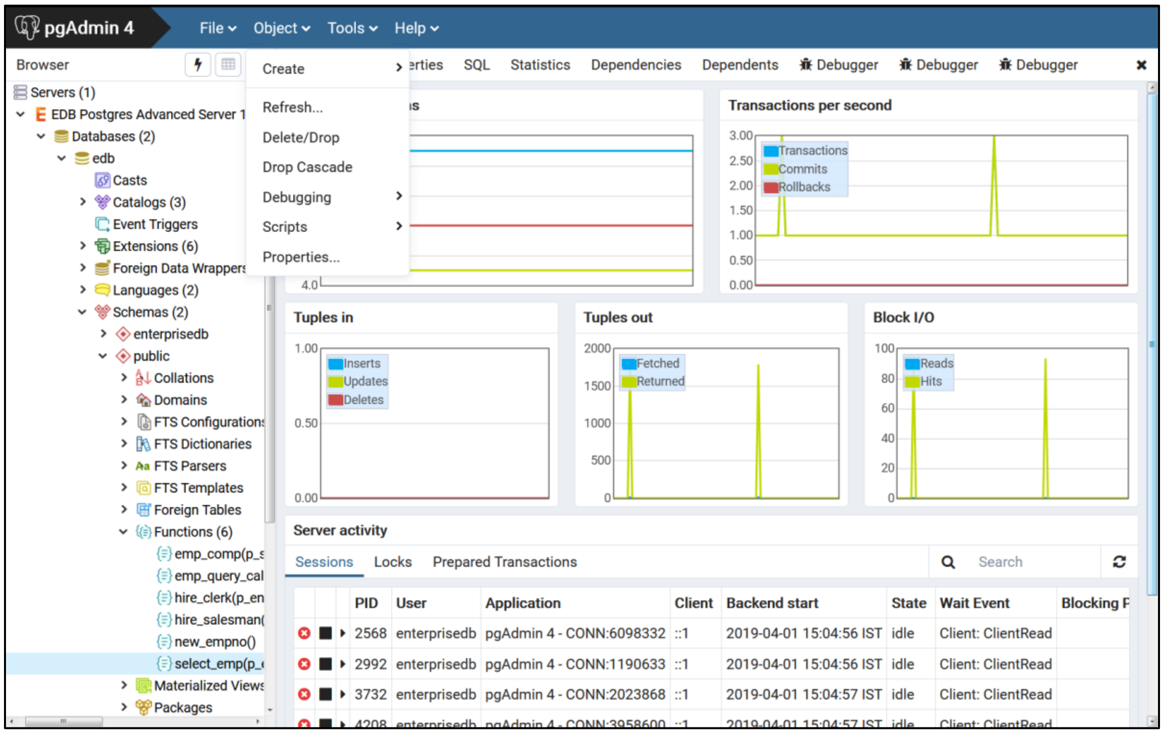
Before using the Debugger, edit the `postgresql.conf` file (located in the `data` subdirectory of your Advanced Server home directory), adding `$libdir/plugin_debugger` to the libraries listed in the `shared_preload_libraries` configuration parameter:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/edb_gen,$libdir/plugin_debugger'
```

After modifying the `shared_preload_libraries` parameter, you must restart the database server.

7.2 Starting the Debugger

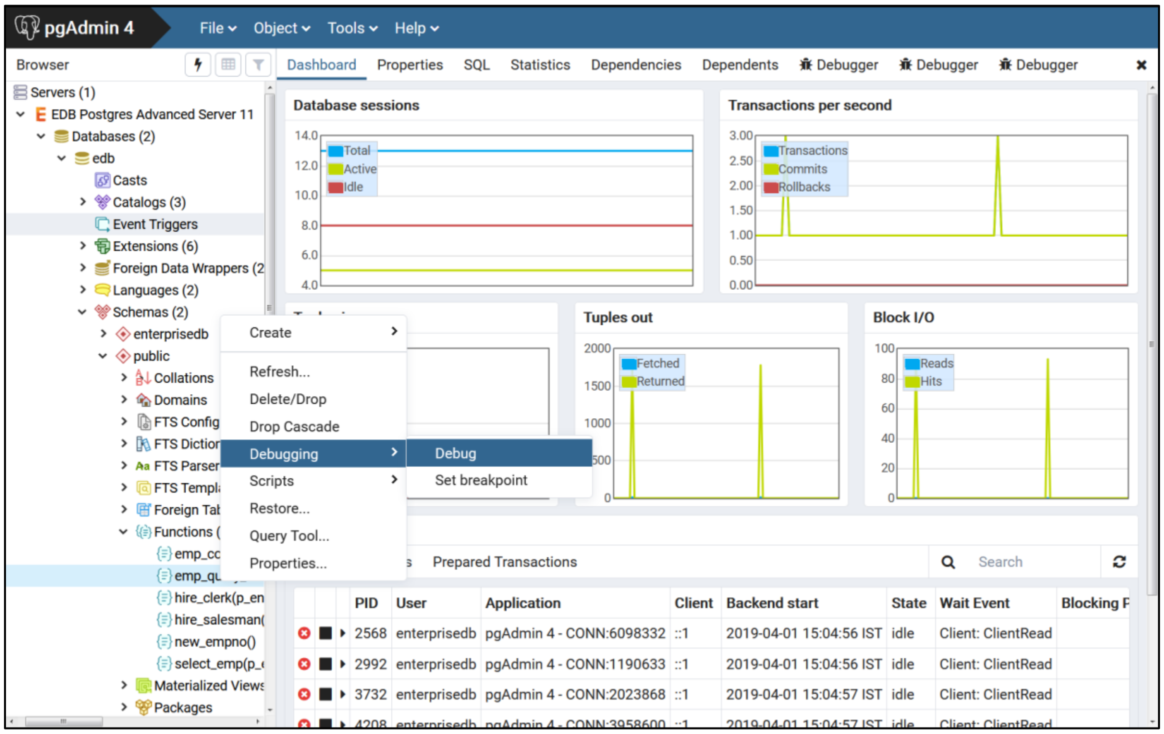
Use pgAdmin 4 to access the Debugger for standalone debugging. To open the Debugger, highlight the name of the stored procedure or function you wish to debug in the pgAdmin 4 `Browser` panel. Then, navigate through the `Object` menu to the `Debugging` menu and select `Debug` from the submenu.



PID	User	Application	Client	Backend start	State	Wait Event	Blocking P
2568	enterprisedb	pgAdmin 4 - CONN:6098332	::1	2019-04-01 15:04:56 IST	idle	Client: ClientRead	
2992	enterprisedb	pgAdmin 4 - CONN:1190633	::1	2019-04-01 15:04:56 IST	idle	Client: ClientRead	
3732	enterprisedb	pgAdmin 4 - CONN:2023868	::1	2019-04-01 15:04:57 IST	idle	Client: ClientRead	
4208	enterprisedb	pgAdmin 4 - CONN:3958600	::1	2019-04-01 15:04:57 IST	idle	Client: ClientRead	

Figure 7.1 - Starting the Debugger from the Object menu

You can also right-click on the name of the stored procedure or function in the pgAdmin 4 `Browser`, and select `Debugging`, and the `Debug` from the context menu.



PID	User	Application	Client	Backend start	State	Wait Event	Blocking P
2568	enterprisedb	pgAdmin 4 - CONN:6098332	::1	2019-04-01 15:04:56 IST	idle	Client: ClientRead	
2992	enterprisedb	pgAdmin 4 - CONN:1190633	::1	2019-04-01 15:04:56 IST	idle	Client: ClientRead	
3732	enterprisedb	pgAdmin 4 - CONN:2023868	::1	2019-04-01 15:04:57 IST	idle	Client: ClientRead	
4208	enterprisedb	pgAdmin 4 - CONN:3958600	::1	2019-04-01 15:04:57 IST	idle	Client: ClientRead	

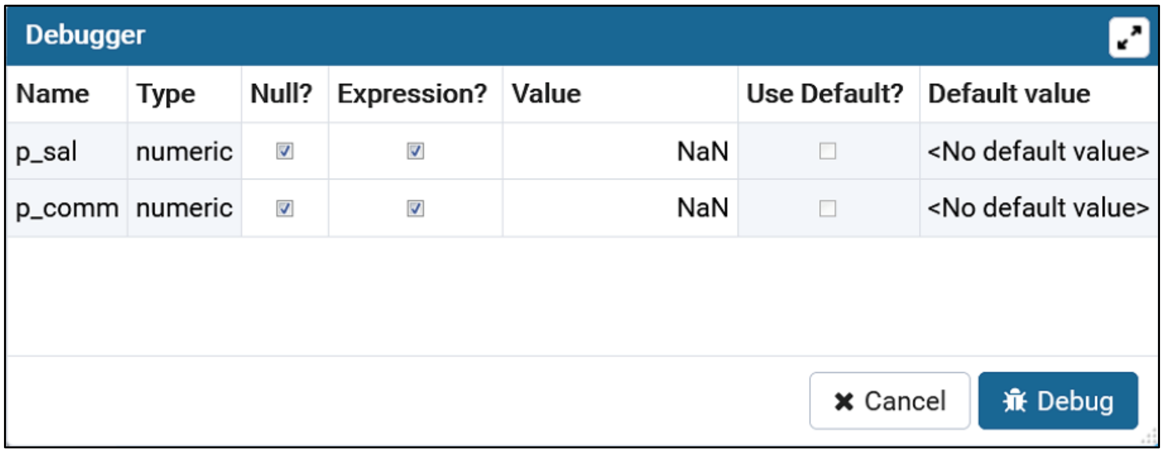
Figure 7.2 - Starting the Debugger from the object's context menu

Note that triggers cannot be debugged using standalone debugging. Triggers must be debugged using in-context debugging. See Section 7.5.3 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.

7.3 The Debugger Window

You can use the `Debugger` window to pass parameter values when you are standalone-debugging a program that expects parameters. When you start the debugger, the `Debugger` window opens automatically to display any `IN` or `IN OUT` parameters expected by the program. If the program declares no `IN` or `IN OUT` parameters, the `Debugger` window does not open.



Name	Type	Null?	Expression?	Value	Use Default?	Default value
p_sal	numeric	✓	✓	NaN	☐	<No default value>
p_comm	numeric	✓	✓	NaN	☐	<No default value>

Figure 7.3 - The Debugger window

Use the fields on the `Debugger` window to provide a value for each parameter:

- The `Name` field contains the formal parameter name.
- The `Type` field contains the parameter data type.
- Check the `Null?` checkbox to indicate that the parameter is a `NULL` value.
- Check the `Expression?` checkbox if the `Value` field contains an expression.
- The `Value` field contains the parameter value that will be passed to the program.
- Check the `Use Default?` checkbox to indicate that the program should use the value in the `Default Value` field.
- The `Default Value` field contains the default value of the parameter.

Press the `Tab` key to select the next parameter in the list for data entry, or click on a `Value` field to select the parameter for data entry.

If you are debugging a procedure or function that is a member of a package that has an initialization section, check the `Debug Package Initializer` check box to instruct the Debugger to step into the package initialization section, allowing you to debug the initialization section code before debugging the procedure or function. If you do not select the check box, the Debugger executes the package initialization section without allowing you to see or step through the individual lines of code as they are executed.

After entering the desired parameter values, click the `Debug` button to start the debugging process. Click the `Cancel` button to terminate the Debugger.

Note: The `Debugger` window does not open during in-context debugging. Instead, the application calling the program to be debugged must supply any required input parameter values.

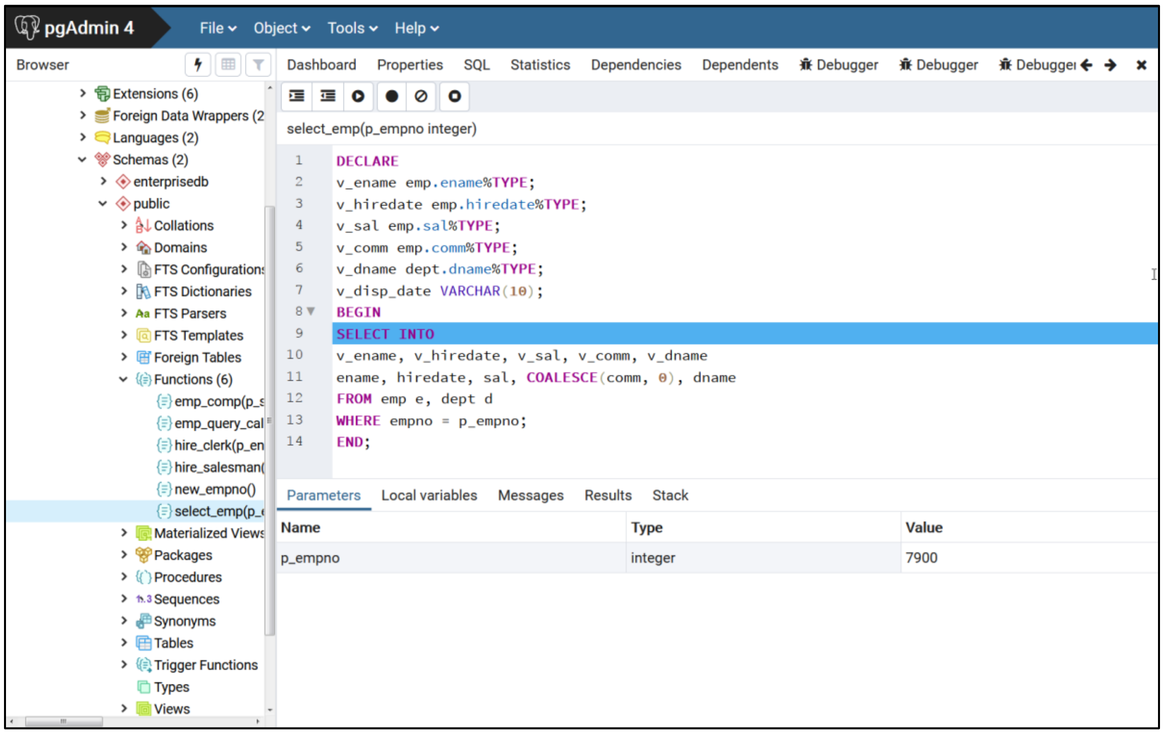
When you have completed a full debugging cycle by stepping through the program code, the `Debugger` window re-opens, allowing you to enter new parameter values and repeat the debugging cycle, or end the debugging session.

7.4 Main Debugger Window

The Main Debugger window contains two panels:

- The top `Program Body` panel displays the program source code.
- The bottom `Tabs` panel provides a set of tabs for different information.

Use the `Tool Bar` icons located at the top panel to access debugging functions.



select_emp(p_empno integer)

```
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;
END;
```

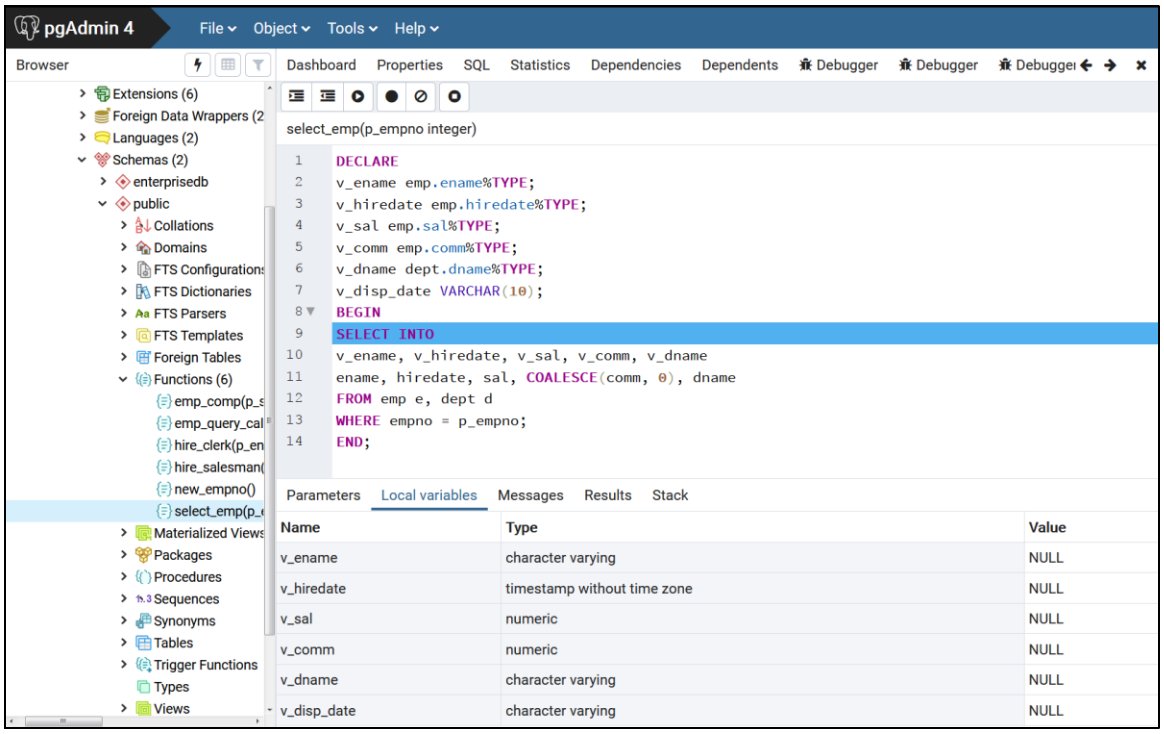
Name	Type	Value
p_empno	integer	7900

Figure 7.4 - The Main Debugger window

The two panels are described in the following sections.

7.4.1 The Program Body Panel

The `Program Body` panel displays the source code of the program that is being debugged.



```
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;
END;
```

Name	Type	Value
v_ename	character varying	NULL
v_hiredate	timestamp without time zone	NULL
v_sal	numeric	NULL
v_comm	numeric	NULL
v_dname	character varying	NULL
v_disp_date	character varying	NULL

Figure 7.5 - The Program Body

Figure 7.5 shows that the Debugger is about to execute the `SELECT` statement. The blue indicator in the program body highlights the next statement to execute.

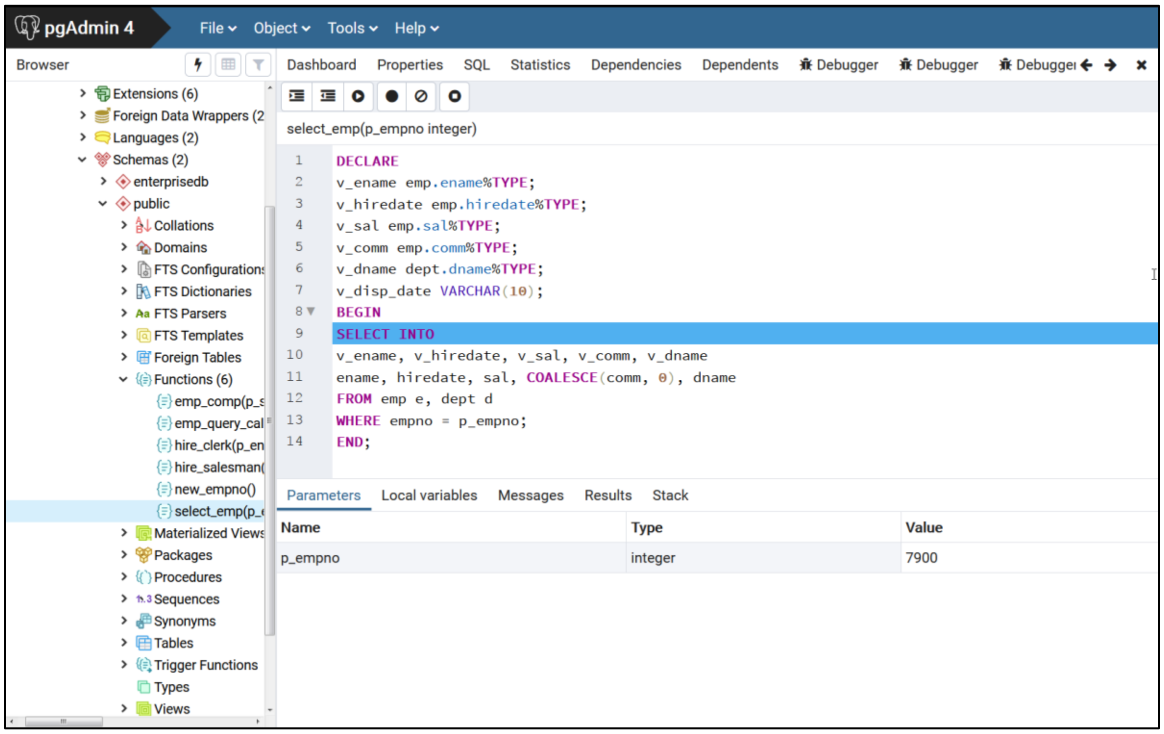
7.4.2 The Tabs Panel

You can use the bottom `Tabs` panel to view or modify parameter values or local variables, or to view messages generated by `RAISE INFO` and function results.

The following is the information displayed by the tabs in the panel:

- The `Parameters` tab displays the current parameter values.
- The `Local variables` tab displays the value of any variables declared within the program.
- The `Messages` tab displays any results returned by the program as it executes.
- The `Results` tab displays program results (if applicable) such as the value from the `RETURN` statement of a function.
- The `Stack` tab has functionality described in Section 7.4.3.

The following figures show the results from the various tabs.

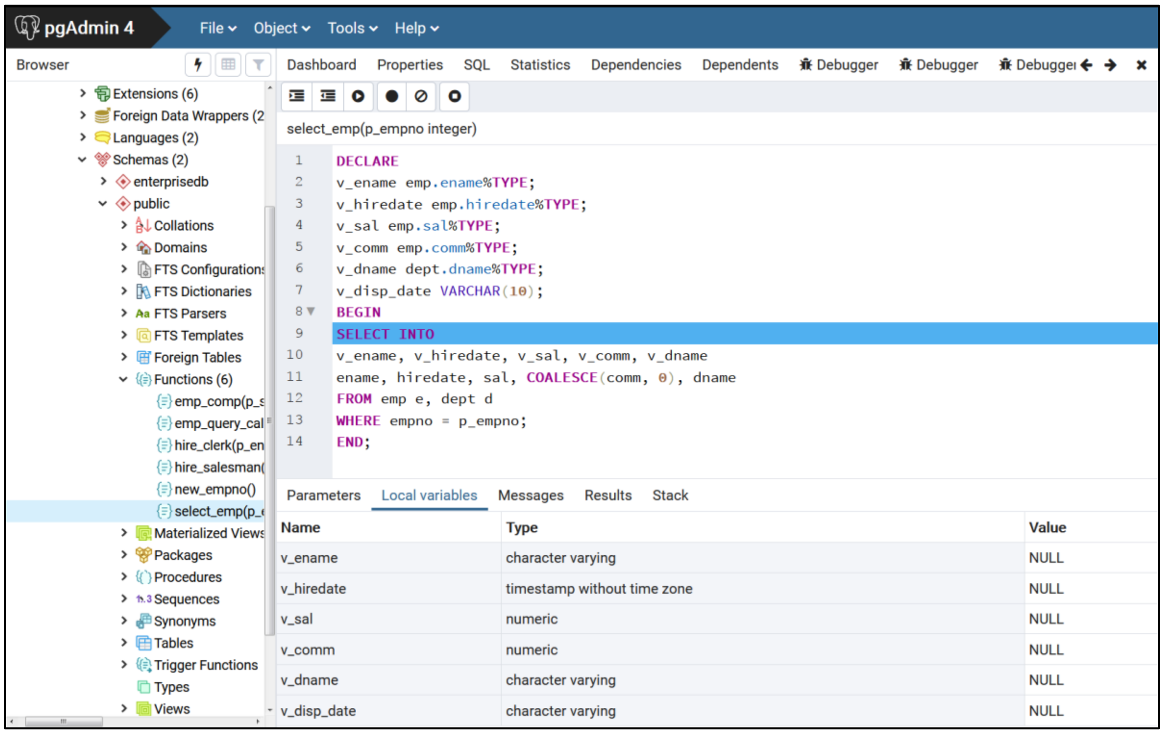


select_emp(p_empno integer)

```
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;
END;
```

Name	Type	Value
p_empno	integer	7900

Figure 7.6 – The Parameters tab

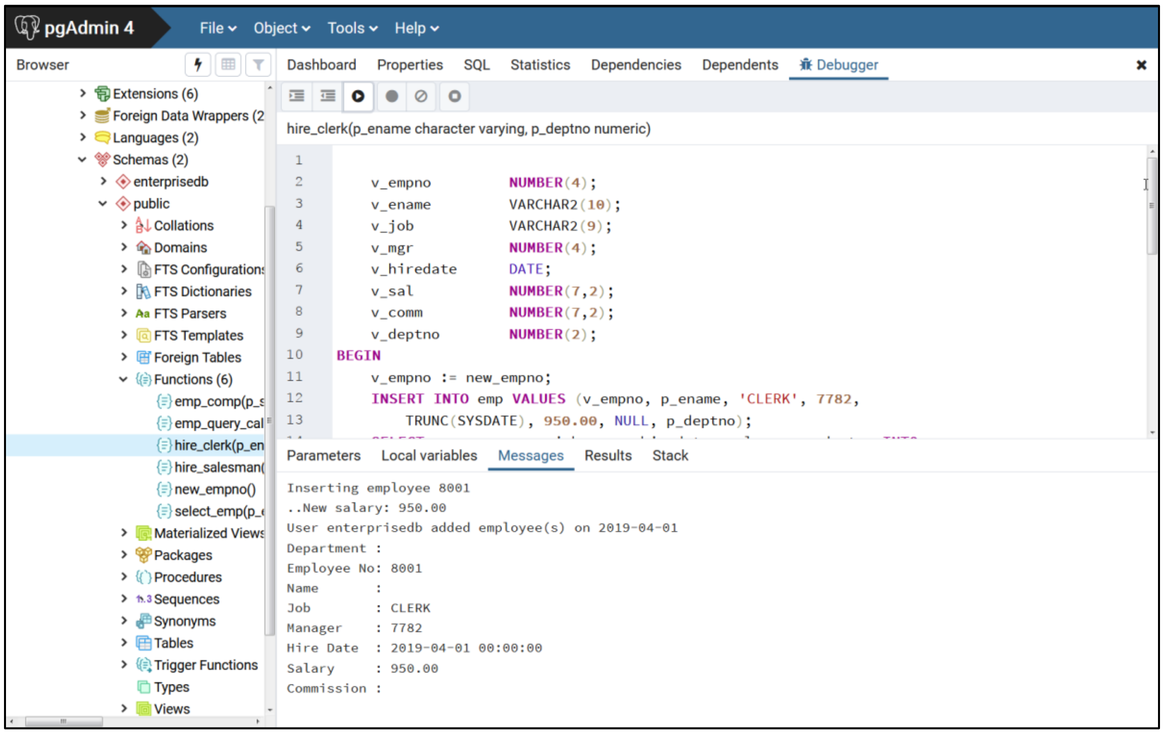


select_emp(p_empno integer)

```
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;
END;
```

Name	Type	Value
v_ename	character varying	NULL
v_hiredate	timestamp without time zone	NULL
v_sal	numeric	NULL
v_comm	numeric	NULL
v_dname	character varying	NULL
v_disp_date	character varying	NULL

Figure 7.7 – The Local variables tab

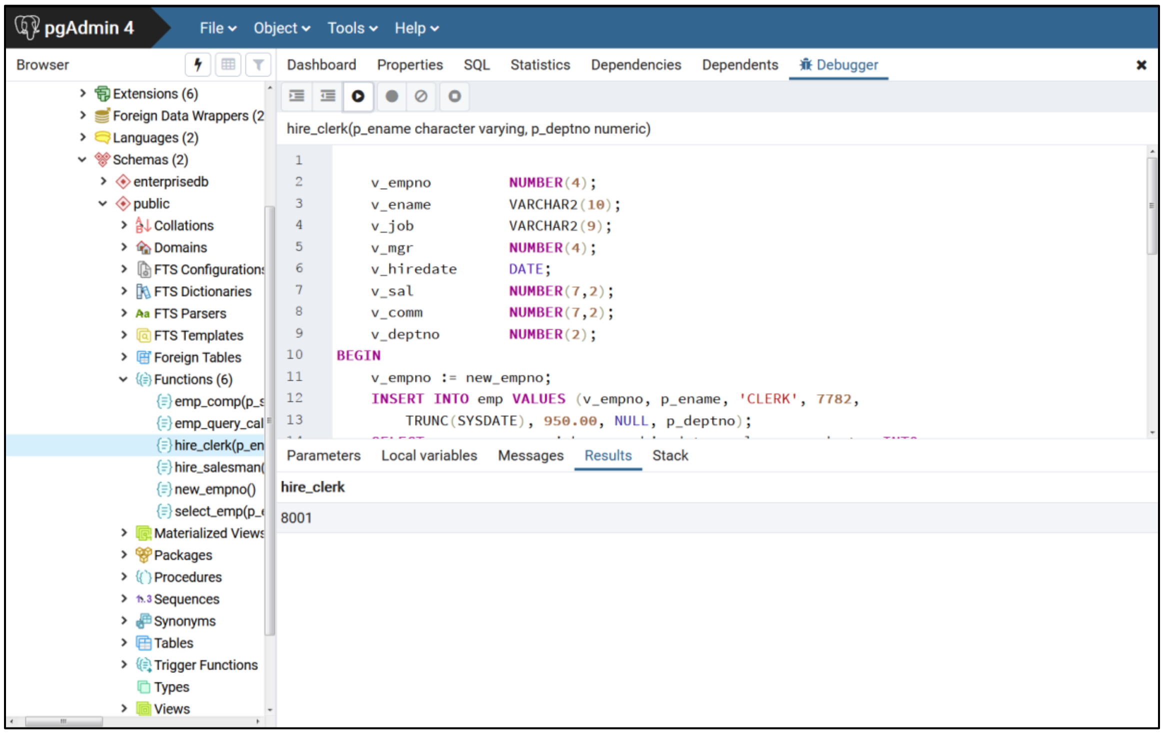


hire_clerk(p_ename character varying, p_deptno numeric)

```

    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_mgr           NUMBER(4);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_deptno        NUMBER(2);
BEGIN
    v_empno := new_empno;
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        TRUNC(SYSDATE), 950.00, NULL, p_deptno);
```

```
Inserting employee 8001
..New salary: 950.00
User enterprisedb added employee(s) on 2019-04-01
Department :
Employee No: 8001
Name       :
Job        : CLERK
Manager    : 7782
Hire Date  : 2019-04-01 00:00:00
Salary     : 950.00
Commission :
```

Figure 7.8 – The Messages tab



hire_clerk(p_ename character varying, p_deptno numeric)

```

    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_mgr           NUMBER(4);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_deptno        NUMBER(2);
BEGIN
    v_empno := new_empno;
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        TRUNC(SYSDATE), 950.00, NULL, p_deptno);
```

hire_clerk
8001

Figure 7.9 – The Results tab

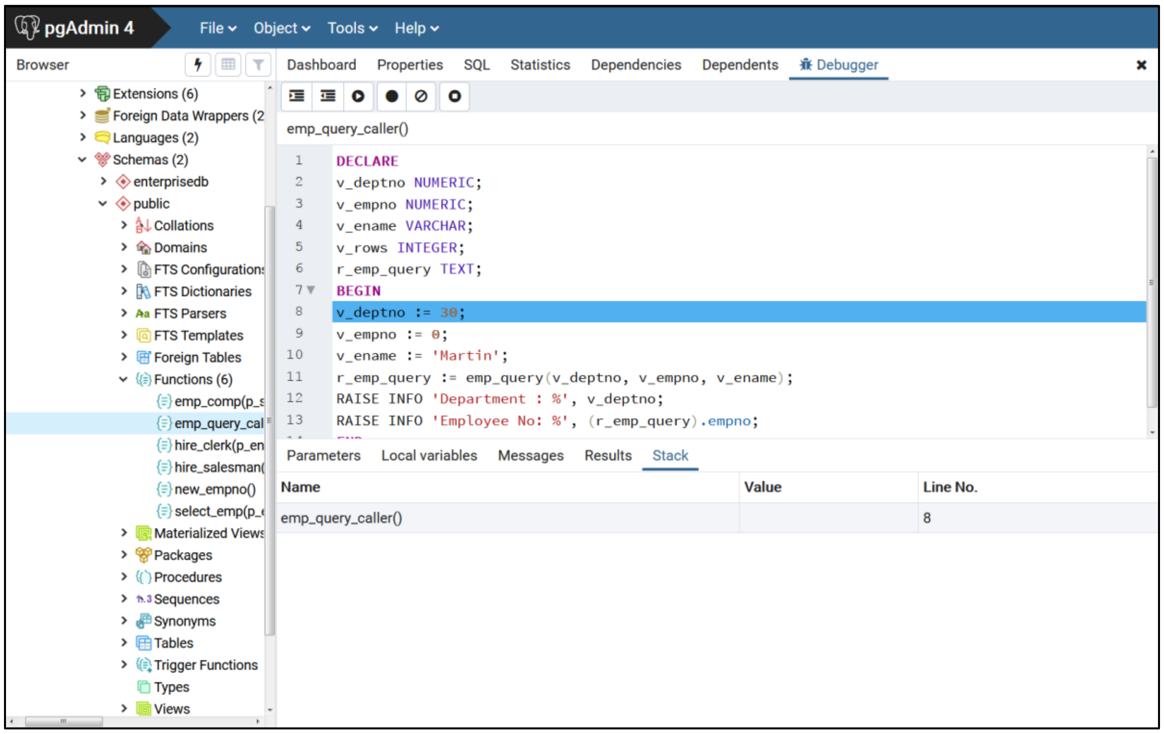
7.4.3 The Stack Tab

The `Stack` tab displays a list of programs that are currently on the call stack (programs that have been invoked, but which have not yet completed). When a program is called, the name of the program is added to the top of the list displayed in the `Stack` tab. `When` the program ends, its name is removed from the list.

The `Stack` tab also displays information about program calls. The information includes:

- The location of the call within the program
- The call arguments
- The name of the program being called

Reviewing the call stack can help you trace the course of execution through a series of nested programs.



emp_query_caller()

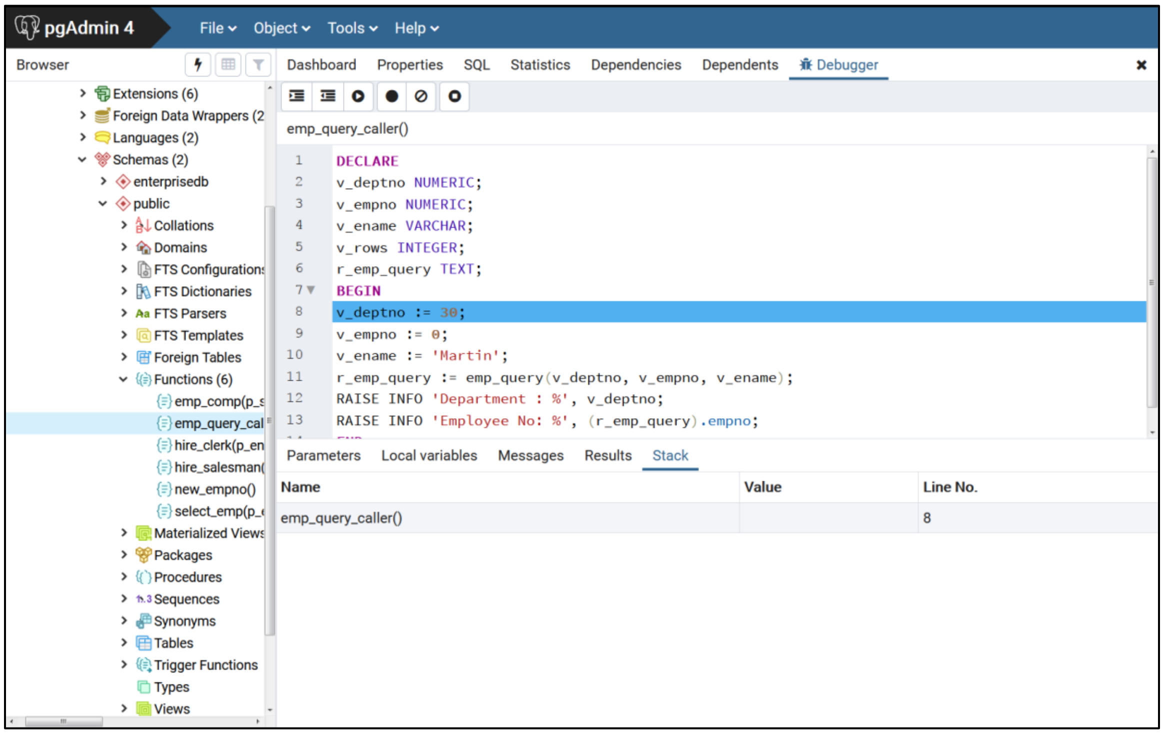
```
DECLARE
v_deptno NUMERIC;
v_empno NUMERIC;
v_ename VARCHAR;
v_rows INTEGER;
r_emp_query TEXT;
BEGIN
v_deptno := 30;
v_empno := 0;
v_ename := 'Martin';
r_emp_query := emp_query(v_deptno, v_empno, v_ename);
RAISE INFO 'Department : %', v_deptno;
RAISE INFO 'Employee No: %', (r_emp_query).empno;
```

Name	Value	Line No.
emp_query_caller()		8

Figure 7.10 – A debugged program calling a subprogram

Figure 7.10 shows that `emp_query_caller` is about to call a subprogram named `emp_query`. `emp_query_caller` is currently at the top of the call stack.

After the call to `emp_query` executes, `emp_query` is displayed at the top of the `Stack` tab, and its code is displayed in the `Program Body` panel as shown in Figure 7.11.



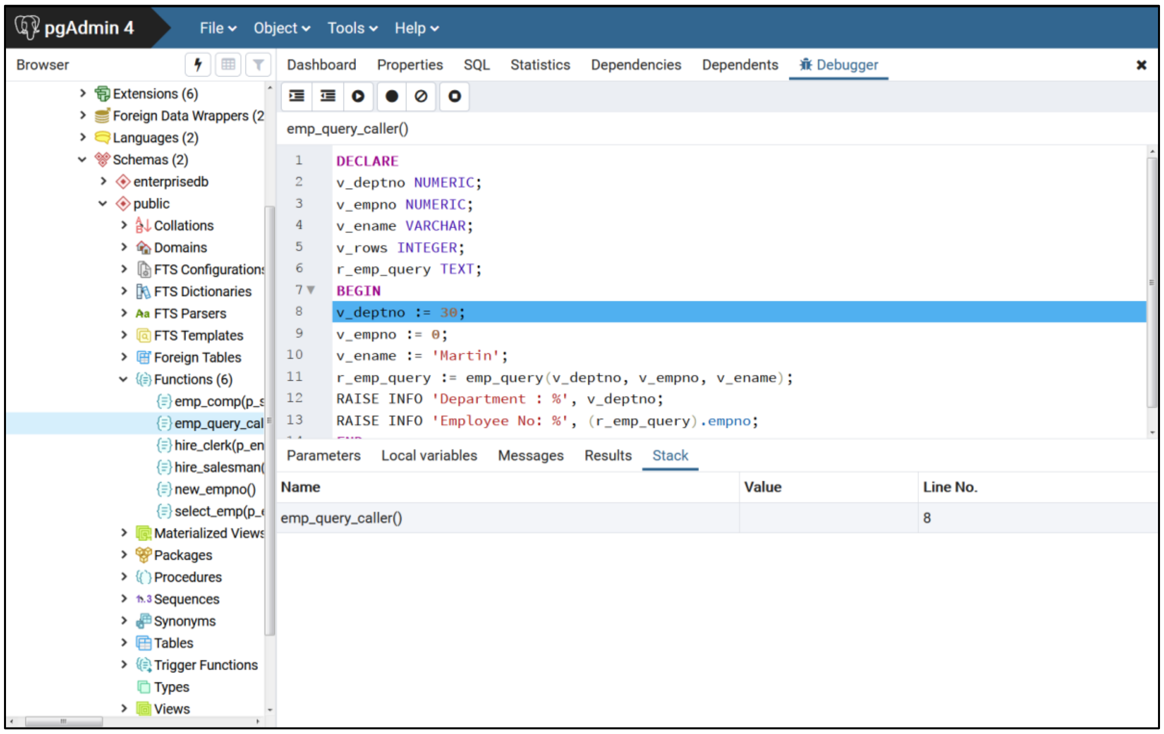
emp_query_caller()

```
DECLARE
v_deptno NUMERIC;
v_empno NUMERIC;
v_ename VARCHAR;
v_rows INTEGER;
r_emp_query TEXT;
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;
```

Name	Value	Line No.
emp_query_caller()		8

Figure 7.11 - Debugging the called subprogram

Upon completion of execution of the subprogram, control returns to the calling program (`emp_query_caller`), now displayed at the top of the `Stack` tab as shown in Figure 7.12.



emp_query_caller()

```
DECLARE
v_deptno NUMERIC;
v_empno NUMERIC;
v_ename VARCHAR;
v_rows INTEGER;
r_emp_query TEXT;
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;
```

Name	Value	Line No.
emp_query_caller()		8

Figure 7.12 – Control returns from debugged subprogram

7.5 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

7.5.1 Stepping Through the Code

Use the tool bar icons to step through a program with the Debugger:

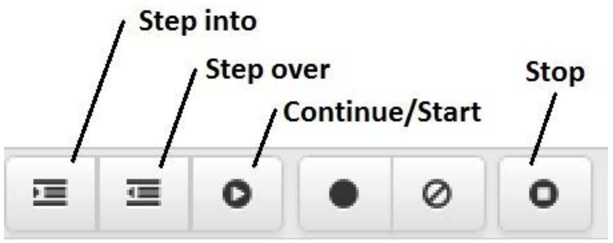


Figure 7.13 - The Tool bar icons

The icons serve the following purposes:

- **Step into.** Click the `Step into` icon to execute the currently highlighted line of code.
- **Step over.** Click the `Step over` icon to execute a line of code, stepping over any sub-functions invoked by the code. The sub-function executes, but is not debugged unless it contains a breakpoint.
- **Continue/Start.** Click the `Continue/Start` icon to execute the highlighted code, and continue until the program encounters a breakpoint or completes.
- **Stop.** Click the `Stop` icon to halt the execution of a program.

7.5.2 Using Breakpoints

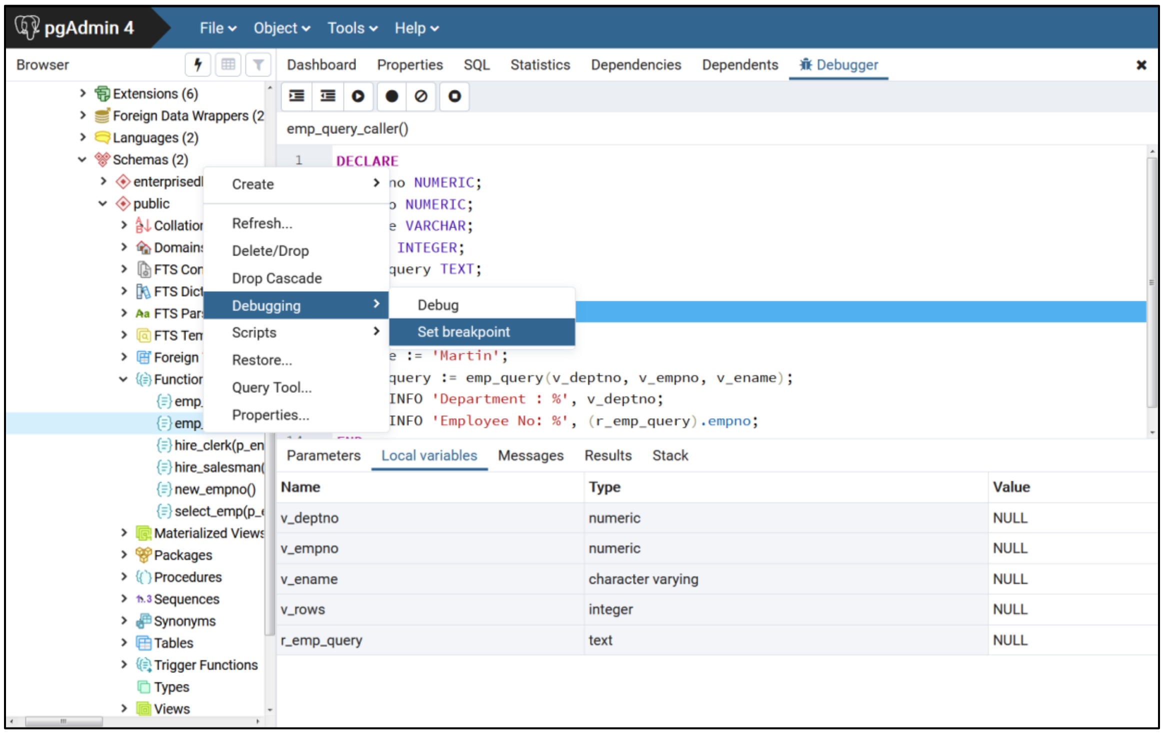
As the Debugger executes a program, it pauses whenever it reaches a breakpoint. When the Debugger pauses, you can observe or change local variables, or navigate to an entry in the call stack to observe variables or set other breakpoints. The next step into, step over, or continue operation forces the 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 within the grey shaded margin to the left of the line of code where you want the local breakpoint set. Where you click in the grey shaded margin should be close to the right side of the margin as in the spot where the breakpoint dot is shown on source code line 12 in Figure 7.14.

When created, the Debugger displays a dark dot in the margin, indicating a breakpoint has been set at the selected line of code.



```
emp_query_caller()
DECLARE
no NUMERIC;
o NUMERIC;
e VARCHAR;
 INTEGER;
query TEXT;

e := 'Martin';
query := emp_query(v_deptno, v_empno, v_ename);
INFO 'Department : %', v_deptno;
INFO 'Employee No: %', (r_emp_query).empno;
```

Name	Type	Value
v_deptno	numeric	NULL
v_empno	numeric	NULL
v_ename	character varying	NULL
v_rows	integer	NULL
r_emp_query	text	NULL

Figure 7.14 - Set a breakpoint by clicking in left-hand margin

You can set as many local breakpoints as desired. Local breakpoints remain in effect for the duration of a debugging session until they are removed.

Removing a Local Breakpoint

To remove a local breakpoint, left-click the mouse on the breakpoint dot in the grey shaded margin of the `Program Body` panel. The dot disappears, indicating that the breakpoint has been removed.

You can remove all of the breakpoints from the program that currently appears in the `Program Body` frame by clicking the `Clear all breakpoints` icon.

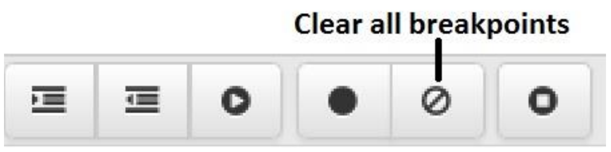
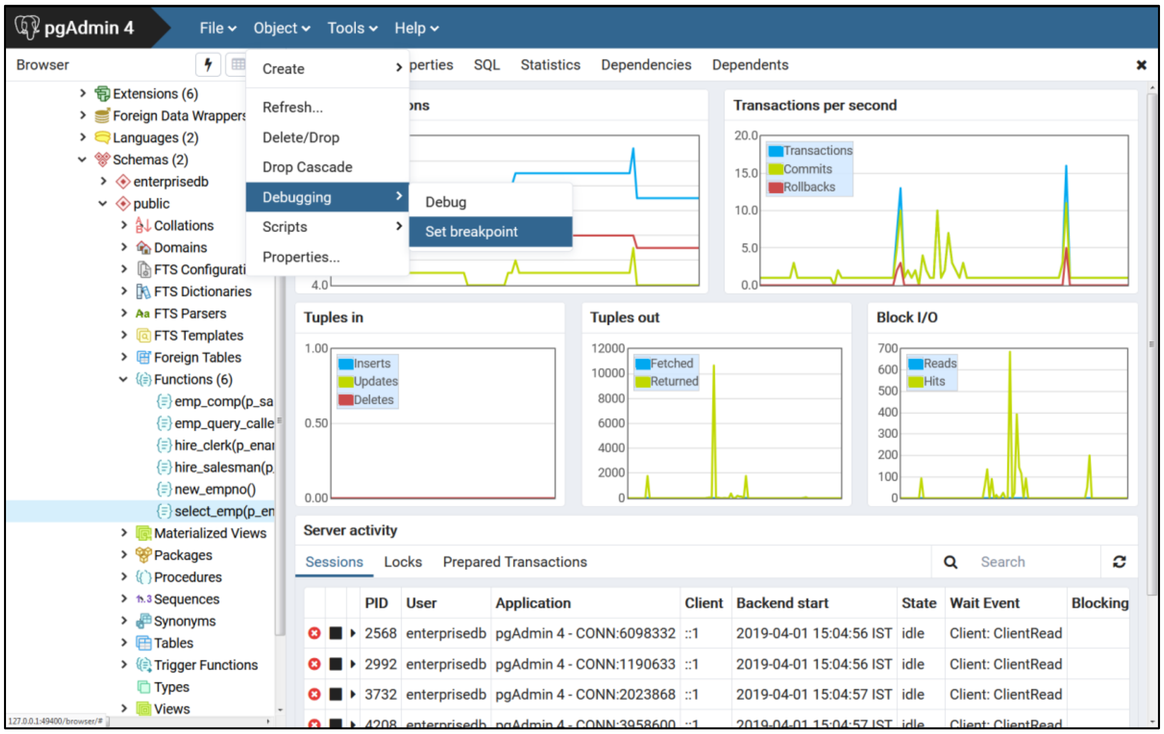


Figure 7.15 – Clear all breakpoints icon

Note: When you perform any of the preceding actions, only the breakpoints in the program that currently appears in the `Program Body` panel are removed. Breakpoints in called subprograms or breakpoints in programs that call the program currently appearing in the `Program Body` panel are not removed.

7.5.3 Setting a Global Breakpoint for In-Context Debugging

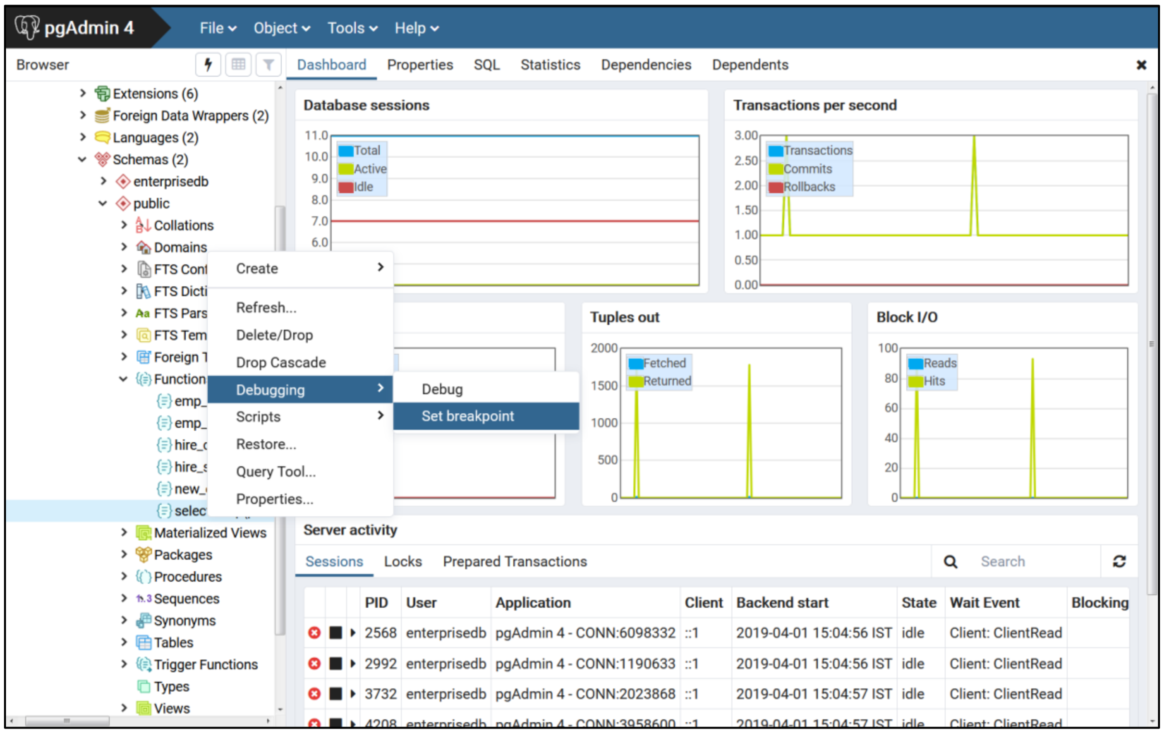
To set a global breakpoint for in-context debugging, highlight the stored procedure, function, or trigger on which you wish to set the breakpoint in the `Browser` panel. Navigate through the `Object` menu to select `Debugging`, and then `Set Breakpoint`.



PID	User	Application	Client	Backend start	State	Wait Event	Blocking
2568	enterprisedb	pgAdmin 4 - CONN:6098332	::1	2019-04-01 15:04:56 IST	idle	Client: ClientRead	
2992	enterprisedb	pgAdmin 4 - CONN:1190633	::1	2019-04-01 15:04:56 IST	idle	Client: ClientRead	
3732	enterprisedb	pgAdmin 4 - CONN:2023868	::1	2019-04-01 15:04:57 IST	idle	Client: ClientRead	
4208	enterprisedb	pgAdmin 4 - CONN:3958600	::1	2019-04-01 15:04:57 IST	idle	Client: ClientRead	

Figure 7.16 - Setting a global breakpoint from the Object menu

Alternatively, you can right-click on the name of the stored procedure, function, or trigger on which you wish to set a global breakpoint and select `Debugging`, then `Set Breakpoint` from the context menu as shown by the following.



PID	User	Application	Client	Backend start	State	Wait Event	Blocking
2568	enterprisedb	pgAdmin 4 - CONN:6098332	::1	2019-04-01 15:04:56 IST	idle	Client: ClientRead	
2992	enterprisedb	pgAdmin 4 - CONN:1190633	::1	2019-04-01 15:04:56 IST	idle	Client: ClientRead	
3732	enterprisedb	pgAdmin 4 - CONN:2023868	::1	2019-04-01 15:04:57 IST	idle	Client: ClientRead	
4208	enterprisedb	pgAdmin 4 - CONN:3958600	::1	2019-04-01 15:04:57 IST	idle	Client: ClientRead	

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.

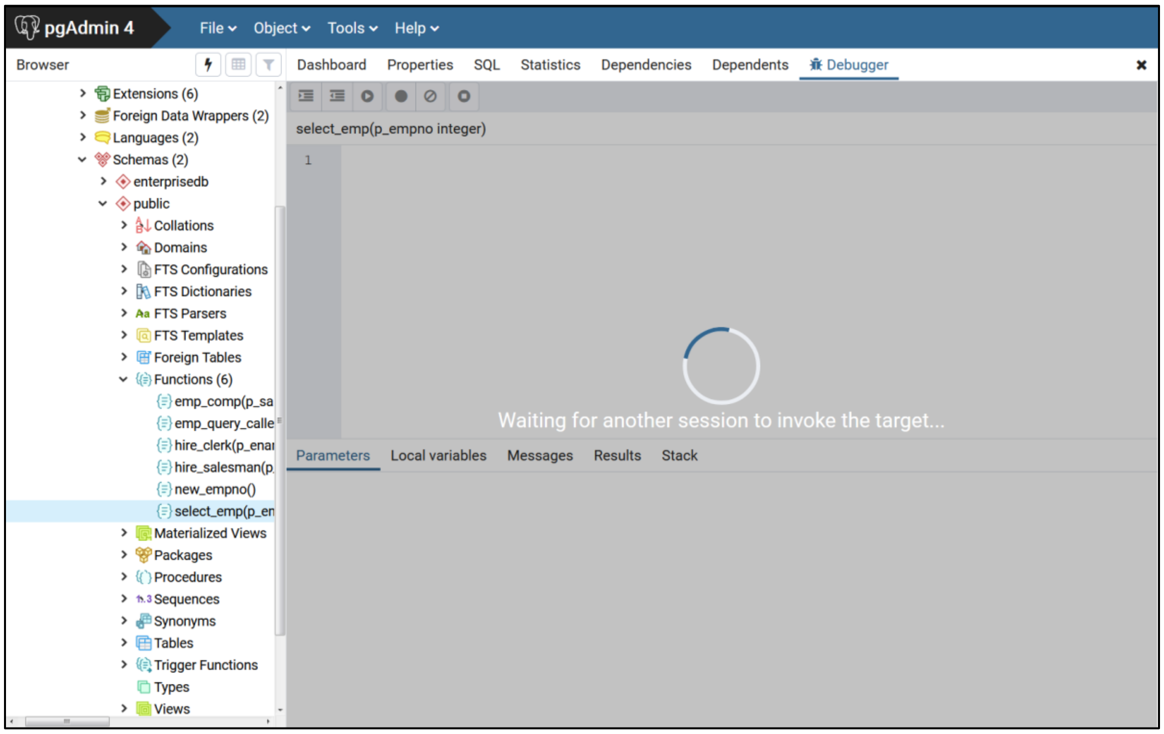
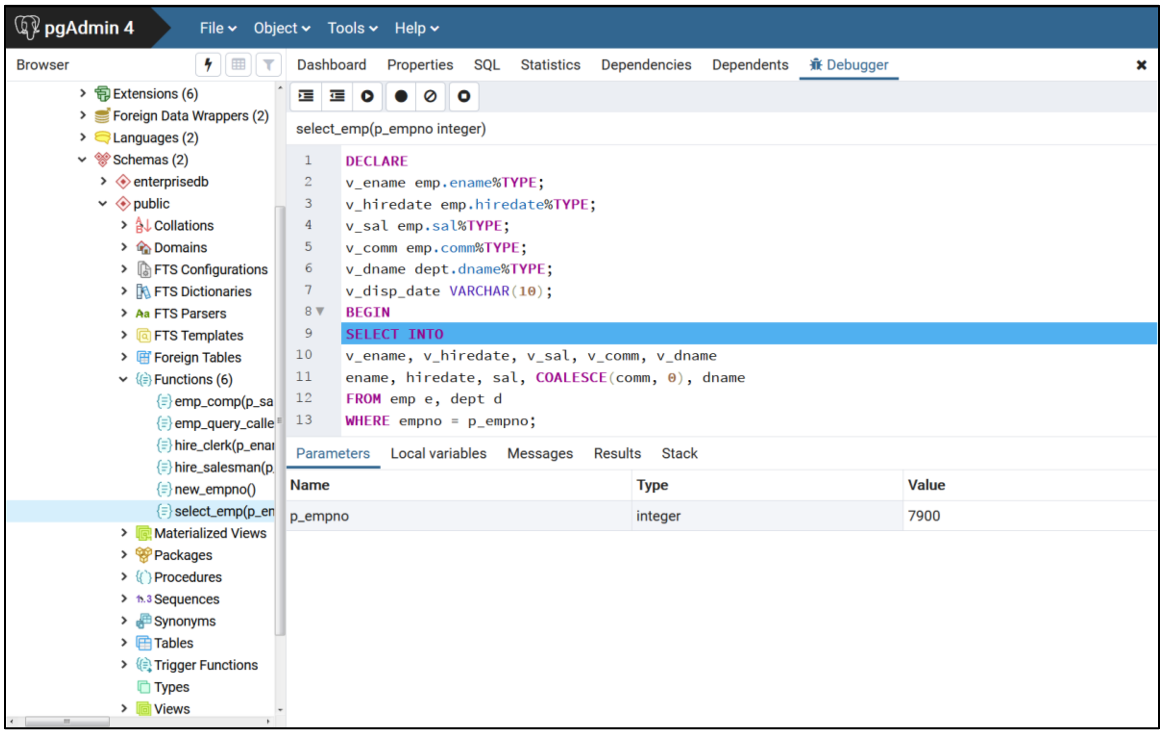


Figure 7.18 - Waiting for invocation of program to be debugged

The PSQL client invokes the `select_emp` function (on which a global breakpoint has been set).

```
$ psql edb enterprisedb
psql.bin (12.0.0, server 12.0.0)
Type "help" for help.

edb=# SELECT select_emp(7900);
```

The `select_emp` function does not complete until you step through the program in the Debugger.



select_emp(p_empno integer)

```
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;
```

Name	Type	Value
p_empno	integer	7900

Figure 7.19 - Program on which a global breakpoint has been set

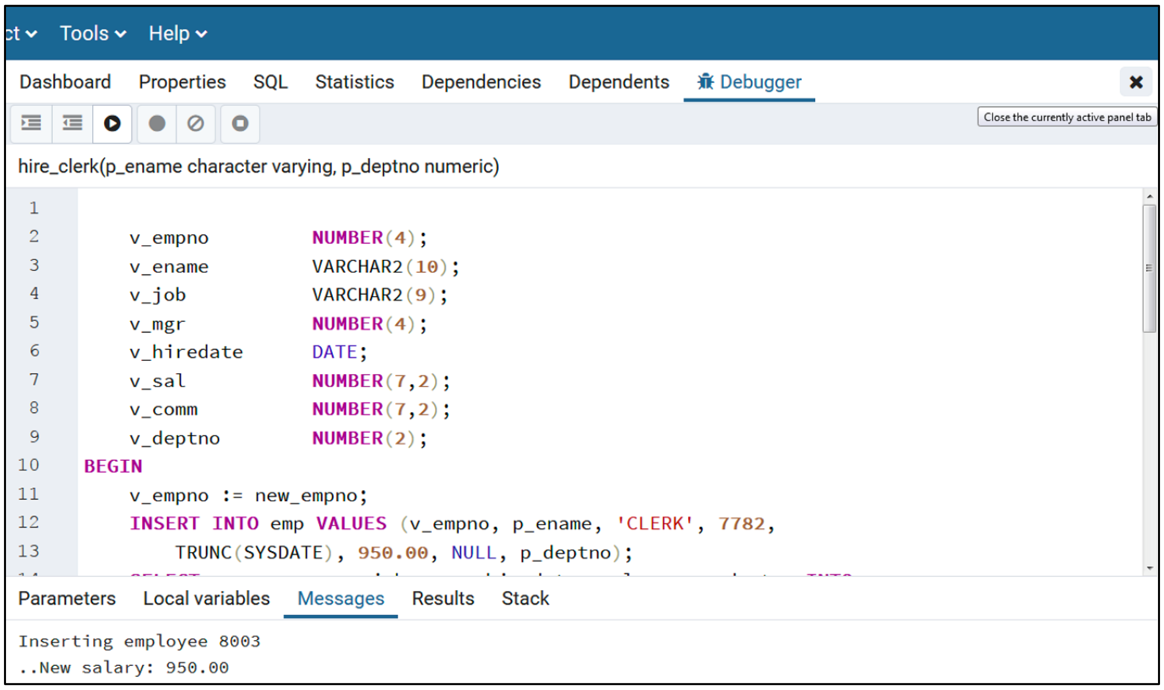
You can now debug the program using any of the previously discussed operations such as step into, step over, and continue, or set local breakpoints. When you have stepped through execution of the program, the calling application (PSQL) regains control and the `select_emp` function completes execution and its output is displayed.

```
$ psql edb enterprisedb
psql.bin (12.0.0, server 12.0.0)
Type "help" for help.

edb=# SELECT select_emp(7900);
INFO:  Number     : 7900
INFO:  Name       : JAMES
INFO:  Hire Date  : 12/03/1981
INFO:  Salary     : 950.00
INFO:  Commission: 0.00
INFO:  Department: SALES
 select_emp
------------

(1 row)
```

At this point, you can end the Debugger session as shown in Section 7.5.4. 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.

7.5.4 Exiting the Debugger

To end a Debugger session and exit the Debugger, click on the close icon (x) located in the upper-right corner to close the tab.



hire_clerk(p_ename character varying, p_deptno numeric)

```

    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_mgr           NUMBER(4);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_comm          NUMBER(7,2);
    v_deptno        NUMBER(2);
BEGIN
    v_empno := new_empno;
    INSERT INTO emp VALUES (v_empno, p_ename, 'CLERK', 7782,
        TRUNC(SYSDATE), 950.00, NULL, p_deptno);
```

```
Inserting employee 8003
..New salary: 950.00
```

Figure 7.20 - Exiting from the Debugger

8 Performance Analysis and Tuning

Advanced Server provides various tools for performance analysis and tuning. These features are described in this section.

8.1 Dynatune

Advanced Server supports dynamic tuning of the database server to make the optimal usage of the system resources available on the host machine on which it is installed. The two parameters that control this functionality are located in the `postgresql.conf` file. These parameters are:

- `edb_dynatune`
- `edb_dynatune_profile`

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

You can change the value of the `edb_dynatune` parameter after the initial installation of Advanced Server by editing the `postgresql.conf` file. The postmaster must be restarted in order for the new configuration to take effect.

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

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

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

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

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

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 EDB Wait States

The *EDB wait states* contrib module contains two main components.

EDB Wait States Background Worker (EWSBW)

When the wait states background worker is registered as one of the shared preload libraries, EWSBW probes each of the running sessions at regular intervals.

For every session it collects information such as the database to which it is connected, the logged in user of the session, the query running in that session, and the wait events on which it is waiting.

This information is saved in a set of files in a user-configurable path and directory folder given by the `edb_wait_states.directory` parameter to be added to the `postgresql.conf` file. The specified path must be a full, absolute path and not a relative path.

The following describes the installation process on a Linux system.

Step 1: EDB wait states is installed with the `edb-as`*`xx`*`-server-edb-modules` RPM package where *`xx`* is the Advanced Server version number.

Step 2: To launch the worker, it must be registered in the `postgresql.conf` file using the `shared_preload_libraries` parameter, for example:

```
shared_preload_libraries = '$libdir/edb_wait_states'
```

Step 3: Restart the database server. After a successful restart, the background worker begins collecting data.

Step 4: To review the data, create the following extension:

```
CREATE EXTENSION edb_wait_states;
```

Step 5: To terminate the EDB wait states worker, remove `$libdir/edb_wait_states` from the `shared_preload_libraries` parameter and restart the database server.

The following describes the installation process on a Windows system.

Step 1: EDB wait states module is installed with the `EDB Modules` installer by invoking StackBuilder Plus utility. Follow the onscreen instructions to complete the installation of the `EDB Modules`.

Step 2: To register the worker, modify the `postgresql.conf` file to include the wait states library in the `shared_preload_libraries` configuration parameter. The parameter value must include:

```
shared_preload_libraries = '$libdir/edb_wait_states.dll'
```

The EDB wait states installation places the `edb_wait_states.dll` library file in the following path:

```
C:\Program Files\edb\as12\lib\
```

Step 3: Restart the database server for the changes to take effect. After a successful restart, the background worker gets started and starts collecting the data.

Step 4: To view the data, create the following extension:

```
CREATE EXTENSION edb_wait_states;
```

The installer places the `edb_wait_states.control` file in the following path:

```
C:\Program Files\edb\as12\share\extension
```

Terminating the Wait States Worker

To terminate the EDB wait states worker, use the `DROP EXTENSION` command to drop the `edb_wait_states` extension; then modify the `postgresql.conf` file, removing `$libdir/edb_wait_states.dll` from the `shared_preload_libraries` parameter. Restart the database server after modifying the postgresql.conf file to apply your changes.

The Wait States Interface

The interface includes the functions listed in the following sections. Each of these functions has common input and output parameters. Those parameters are as follows:

- **start_ts and end_ts (IN).** Together these specify the time interval and the data within which is to be read. If only `start_ts` is specified, the data starting from `start_ts` is output. If only `end_ts` is provided, data up to `end_ts` is output. If none of those are provided, all the data is output. Every function outputs different data. The output of each function will be explained below.

- **query_id (OUT).** Identifies a normalized query. It is internal hash code computed from the query.
- **session_id (OUT).** Identifies a session.
- **ref_start_ts and ref_end_ts (OUT).** Provide the timestamps of a file containing a particular data point. A data point may be a wait event sample record or a query record or a session record.

The syntax of each function is given in the following sections.

Note: The examples shown in the following sections are based on the following three queries executed on four different sessions connected to different databases using different users, simultaneously:

```
SELECT schemaname FROM pg_tables, pg_sleep(15) WHERE schemaname <> 'pg_catalog'; /* ran on 2
sessions */
SELECT tablename FROM pg_tables, pg_sleep(10) WHERE schemaname <> 'pg_catalog';
SELECT tablename, schemaname FROM pg_tables, pg_sleep(10) WHERE schemaname <> 'pg_catalog';
```

8.2.1 edb_wait_states_data

This function is used to read the data collected by EWSBW.

```
edb_wait_states_data(
  IN start_ts timestamptz default '-infinity'::timestamptz,
  IN end_ts timestamptz default 'infinity'::timestamptz,
  OUT session_id int4,
  OUT dbname text,
  OUT username text,
  OUT query text,
  OUT query_start_time timestamptz,
  OUT sample_time timestamptz,
  OUT wait_event_type text,
  OUT wait_event text
)
```

This function can be used to find out the following:

The queries running in the given duration (defined by *start_ts* and *end_ts*) in all the sessions, and the wait events, if any, they were waiting on. For example:

```
SELECT query, session_id, wait_event_type, wait_event
  FROM edb_wait_states_data(start_ts, end_ts);
```

The progress of a session within a given duration (that is, the queries run in a session (session_id = 100000) and the wait events the queries waited on). For example:

```
SELECT query, wait_event_type, wait_event
  FROM edb_wait_states_data(start_ts, end_ts)
  WHERE session_id = 100000;
```

The duration for which the samples are available. For example:

```
SELECT min(sample_time), max(sample_time)
  FROM edb_wait_states_data();
```

Parameters

In addition to the common parameters described previously, each row of the output gives the following:

dbname

The session's database

username

The session's logged in user

query

The query running in the session

query_start_time

The time when .the query started

sample_time

The time when wait event data was collected

wait_event_type

The type of wait event the session (backend) is waiting on

wait_event

The wait event the session (backend) is waiting on

Example

The following is a sample output from the `edb_wait_states_data()` function.

```
edb=# SELECT * FROM edb_wait_states_data();
```

```
-[ RECORD 1 ]----+----------------------------------------------------------------------
session_id       | 4398
dbname           | edb
username         | enterprisedb
query            | SELECT schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
query_start_time | 17-AUG-18 11:56:05.271962 -04:00
sample_time      | 17-AUG-18 11:56:19.700236 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
-[ RECORD 2 ]----+----------------------------------------------------------------------
session_id       | 4398
dbname           | edb
username         | enterprisedb
query            | SELECT schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
query_start_time | 17-AUG-18 11:56:05.271962 -04:00
sample_time      | 17-AUG-18 11:56:18.699938 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
-[ RECORD 3 ]----+----------------------------------------------------------------------
session_id       | 4398
dbname           | edb
username         | enterprisedb
query            | SELECT schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
query_start_time | 17-AUG-18 11:56:05.271962 -04:00
sample_time      | 17-AUG-18 11:56:17.700253 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
                    .
                    .
                    .
```

8.2.2 edb_wait_states_queries

This function gives information about the queries sampled by EWSBW.

```
edb_wait_states_queries(
  IN start_ts timestamptz default '-infinity'::timestamptz,
  IN end_ts timestamptz default 'infinity'::timestamptz,
  OUT query_id int8,
  OUT query text,
  OUT ref_start_ts timestamptz
  OUT ref_end_ts timestamptz
)
```

A new queries file is created periodically and thus, there can be multiple query files generated corresponding to specific intervals.

This function returns all the queries in query files that overlap with the given time interval. A query as shown below, gives all the queries in query files that contained queries sampled between *start_ts* and *end_ts*.

In other words, the function may output queries that did not run in the given interval. To exactly know that the user should use `edb_wait_states_data()`.

```
SELECT query FROM edb_wait_states_queries(start_ts, end_ts);
```

Parameters

In addition to the common parameters described previously, each row of the output gives the following:

query

Normalized query text

Example

The following is a sample output from the `edb_wait_states_queries()` function.

```
edb=# SELECT * FROM edb_wait_states_queries();
-[ RECORD 1 ]+------------------------------------------------------------------------------
query_id     | 4292540138852956818
query        | SELECT schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 2 ]+------------------------------------------------------------------------------
query_id     | 3754591102365859187
query        | SELECT tablename FROM pg_tables, pg_sleep($1) WHERE schemaname <> $2
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 3 ]+------------------------------------------------------------------------------
query_id     | 349089096300352897
query        | SELECT tablename, schemaname FROM pg_tables, pg_sleep($1) WHERE schemaname <>
$2
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
```

8.2.3 edb_wait_states_sessions

This function gives information about the sessions sampled by EWSBW.

```
edb_wait_states_sessions(
  IN start_ts timestamptz default '-infinity'::timestamptz,
  IN end_ts timestamptz default 'infinity'::timestamptz,
  OUT session_id int4,
  OUT dbname text,
  OUT username text,
  OUT ref_start_ts timestamptz
  OUT ref_end_ts timestamptz
)
```

This function can be used to identify the databases that were connected and/or which users started those sessions. For example:

```
SELECT dbname, username, session_id
  FROM edb_wait_states_sessions();
```

Similar to `edb_wait_states_queries()`, this function outputs all the sessions logged in session files that contain sessions sampled within the given interval and not necessarily only the sessions sampled within the given interval. To identify that one should use `edb_wait_states_data()`.

Parameters

In addition to the common parameters described previously, each row of the output gives the following:

`dbname`

The database to which the session is connected

`username`

Login user of the session

Example

The following is a sample output from the `edb_wait_states_sessions()` function.

```
edb=# SELECT * FROM edb_wait_states_sessions();
-[ RECORD 1 ]+---------------------------------
session_id   | 4340
dbname       | edb
username     | enterprisedb
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 2 ]+---------------------------------
session_id   | 4398
dbname       | edb
username     | enterprisedb
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 3 ]+---------------------------------
session_id   | 4410
dbname       | db1
username     | user1
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
-[ RECORD 4 ]+---------------------------------
session_id   | 4422
dbname       | db2
username     | user2
ref_start_ts | 17-AUG-18 11:52:38.698793 -04:00
ref_end_ts   | 18-AUG-18 11:52:38.698793 -04:00
```

8.2.4 edb_wait_states_samples

This function gives information about wait events sampled by EWSBW.

```
edb_wait_states_samples(
  IN start_ts timestamptz default '-infinity'::timestamptz,
  IN end_ts timestamptz default 'infinity'::timestamptz,
  OUT query_id int8,
  OUT session_id int4,
  OUT query_start_time timestamptz,
  OUT sample_time timestamptz,
  OUT wait_event_type text,
  OUT wait_event text
)
```

Usually, a user would not be required to call this function directly.

Parameters

In addition to the common parameters described previously, each row of the output gives the following:

query_start_time

The time when the query started in this session

sample_time

The time when wait event data was collected

wait_event_type

The type of wait event on which the session is waiting

wait_event

The wait event on which the session (backend) is waiting

Example

The following is a sample output from the `edb_wait_states_samples()` function.

```
edb=# SELECT * FROM edb_wait_states_samples();
-[ RECORD 1 ]----+---------------------------------
query_id         | 4292540138852956818
session_id       | 4340
query_start_time | 17-AUG-18 11:56:00.39421 -04:00
sample_time      | 17-AUG-18 11:56:00.699934 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
-[ RECORD 2 ]----+---------------------------------
query_id         | 4292540138852956818
session_id       | 4340
query_start_time | 17-AUG-18 11:56:00.39421 -04:00
sample_time      | 17-AUG-18 11:56:01.699003 -04:00
wait_event_type  | Timeout
```

```
wait_event       | PgSleep
-[ RECORD 3 ]----+---------------------------------
query_id         | 4292540138852956818
session_id       | 4340
query_start_time | 17-AUG-18 11:56:00.39421 -04:00
sample_time      | 17-AUG-18 11:56:02.70001 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
-[ RECORD 4 ]----+---------------------------------
query_id         | 4292540138852956818
session_id       | 4340
query_start_time | 17-AUG-18 11:56:00.39421 -04:00
sample_time      | 17-AUG-18 11:56:03.700081 -04:00
wait_event_type  | Timeout
wait_event       | PgSleep
                    .
                    .
                    .
```

8.2.5 edb_wait_states_purge

The function deletes all the sampled data files (queries, sessions and wait event samples) that were created after *start_ts* and aged (rotated) before *end_ts*.

```
    edb_wait_states_purge(
      IN start_ts timestamptz default '-infinity'::timestamptz,
      IN end_ts timestamptz default 'infinity'::timestamptz
    )
```

Usually a user does not need to run this function. The backend should purge those according to the retention age, but in case, that doesn't happen for some reason, this function may be used.

In order to know the duration for which the samples have been retained, use `edb_wait_states_data()` as explained in the previous examples of that function.

Example

The `$PGDATA/edb_wait_states` directory before running `edb_wait_states_purge()`:

```
[root@localhost data]# pwd
/var/lib/edb/as12/data
[root@localhost data]# ls -l edb_wait_states
total 12
-rw------- 1 enterprisedb ...  253 Aug 17 11:56 edb_ws_queries_587836358698793_587922758698793
-rw------- 1 enterprisedb ... 1600 Aug 17 11:56 edb_ws_samples_587836358698793_587839958698793
-rw------- 1 enterprisedb ...   94 Aug 17 11:56
edb_ws_sessions_587836358698793_587922758698793
```

The `$PGDATA/edb_wait_states` directory after running `edb_wait_states_purge()`:

```
edb=# SELECT * FROM edb_wait_states_purge();
 edb_wait_states_purge
-----------------------

(1 row)

[root@localhost data]# pwd
/var/lib/edb/as12/data
[root@localhost data]# ls -l edb_wait_states
total 0
```

9 EDB Clone Schema

EDB Clone Schema is an extension module for Advanced Server that allows you to copy a schema and its database objects from a local or remote database (the source database) to a receiving database (the target database).

The source and target databases can be the same physical database, or different databases within the same database cluster, or separate databases running under different database clusters on separate database server hosts.

Use the following functions with EDB Clone Schema:

- **localcopyschema.** This function makes a copy of a schema and its database objects from a source database back into the same database (the target), but with a different schema name than the original. Use this function when the original source schema and the resulting copy are to reside within the same database. See Section 9.2.1 for information on the `localcopyschema` function.
- **localcopyschema_nb.** This function performs the same purpose as `localcopyschema`, but as a background job, thus freeing up the terminal from which the function was initiated. This is referred to as a *non-blocking* function. See Section 9.2.2 for information on the `localcopyschema_nb` function.
- **remotecopyschema.** This function makes a copy of a schema and its database objects from a source database to a different target database. Use this function when the original source schema and the resulting copy are to reside in two, separate databases. The separate databases can reside in the same, or in different Advanced Server database clusters. See Section 9.2.3 for information on the `remotecopyschema` function.
- **remotecopyschema_nb.** This function performs the same purpose as `remotecopyschema`, but as a background job, thus freeing up the terminal from which the function was initiated. This is referred to as a *non-blocking* function. See Section 9.2.4 for information on the `remotecopyschema_nb` function.
- **process_status_from_log.** This function displays the status of the cloning functions. The information is obtained from a log file that must be specified when a cloning function is invoked. See Section 9.2.5 for information on the `process_status_from_log` function.
- **remove_log_file_and_job.** This function deletes the log file created by a cloning function. This function can also be used to delete a job created by the non-blocking form of the function. See Section 9.2.6 for information on the `remove_log_file_and_job` function.

The database objects that can be cloned from one schema to another are the following:

- Data types

- Tables including partitioned tables, but not foreign tables
- Indexes.
- Constraints
- Sequences
- View definitions
- Materialized views
- Private synonyms
- Table triggers, but not event triggers
- Rules
- Functions
- Procedures
- Packages
- Comments for all supported object types
- Access control lists (ACLs) for all supported object types

The following database objects cannot cloned:

- Large objects (Postgres `LOB`s and `BFILE`s)
- Logical replication attributes for a table
- Database links
- Foreign data wrappers
- Foreign tables
- Event triggers
- Extensions (For cloning objects that rely on extensions, see the third bullet point in the following limitations list.)
- Row level security
- Policies
- Operator class

In addition, the following limitations apply:

- EDB Clone Schema is supported on Advanced Server only when a dialect of `Compatible with Oracle` is specified on the Advanced Server `Dialect` dialog during installation, or when the --`redwood`-`like` keywords are included during a text mode installation or cluster initialization.

- The source code within functions, procedures, triggers, packages, etc., are not modified after being copied to the target schema. If such programs contain coded references to objects with schema names, the programs may fail upon invocation in the target schema if such schema names are no longer consistent within the target schema.

- Cross schema object dependencies are not resolved. If an object in the target schema depends upon an object in another schema, this dependency is not resolved by the cloning functions.

- For remote cloning, if an object in the source schema is dependent upon an extension, then this extension must be created in the `public` schema of the remote database before invoking the remote cloning function.

- At most, 16 copy jobs can run in parallel to clone schemas, whereas each job can have at most 16 worker processes to copy table data in parallel.

- Queries being run by background workers cannot be cancelled.

The following section describes how to set up EDB Clone Schema on the databases.

9.1 Setup Process

Several extensions along with the PL/Perl language must be installed on any database to be used as the source or target database by an EDB Clone Schema function.

In addition, some configuration parameters in the `postgresql.conf` file of the database servers may benefit from some modification.

The following is the setup instructions for these requirements.

9.1.1 Installing Extensions and PL/Perl

The following describes the steps to install the required extensions and the PL/Perl language.

These steps must be performed on any database to be used as the source or target database by an EDB Clone Schema function.

Step 1: The following extensions must be installed on the database:

- `postgres_fdw`
- `dblink`
- `adminpack`
- `pgagent`

Ensure that pgAgent is installed before creating the `pgagent` extension. On Linux, you can use the `edb-as`*`xx`*`-pgagent` RPM package where *`xx`* is the Advanced Server version number to install pgAgent. On Windows, use StackBuilder Plus to download and install pgAgent.

The previously listed extensions can be installed by the following commands if they do not already exist:

```
CREATE EXTENSION postgres_fdw SCHEMA public;
CREATE EXTENSION dblink SCHEMA public;
CREATE EXTENSION adminpack;
CREATE EXTENSION pgagent;
```

For more information about using the `CREATE EXTENSION` command, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/12/static/sql-createextension.html

Step 2: Modify the `postgresql.conf` file.

Modify the `postgresql.conf` file by adding `$libdir/parallel_clone` to the `shared_preload_libraries` configuration parameter as shown by the following example:

```
shared_preload_libraries = '$libdir/dbms_pipe,$libdir/dbms_aq,$libdir/parallel_clone'
```

Step 3: The Perl Procedural Language (PL/Perl) must be installed on the database and the `CREATE TRUSTED LANGUAGE plperl` command must be run. For Linux, install PL/Perl using the `edb-as`*`xx`*`-server-plperl` RPM package where *`xx`* is the Advanced Server version number. For Windows, use the EDB Postgres Language Pack. For information on EDB Language Pack, see the *EDB Postgres Language Pack Guide* available at:

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

Step 4: Connect to the database as a superuser and run the following command:

```
CREATE TRUSTED LANGUAGE plperl;
```

For more information about using the `CREATE LANGUAGE` command, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/12/static/sql-createlanguage.html

9.1.2 Setting Configuration Parameters

The following sections describe certain configuration parameters that may need to be altered in the `postgresql.conf` file.

9.1.2.1 Performance Configuration Parameters

You may need to tune the system for copying a large schema as part of one transaction.

Tuning of configuration parameters is for the source database server referenced in a cloning function.

The configuration parameters in the `postgresql.conf` file that may need to be tuned include the following:

- **work_mem.** Specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files.
- **maintenance_work_mem.** Specifies the maximum amount of memory to be used by maintenance operations, such as `VACUUM`, `CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY`.
- **max_worker_processes.** Sets the maximum number of background processes that the system can support.
- **checkpoint_timeout.** Maximum time between automatic WAL checkpoints, in seconds.
- **checkpoint_completion_target.** Specifies the target of checkpoint completion, as a fraction of total time between checkpoints.
- **checkpoint_flush_after.** Whenever more than `checkpoint_flush_after` *`bytes`* have been written while performing a checkpoint, attempt to force the OS to issue these writes to the underlying storage.
- **max_wal_size.** Maximum size to let the WAL grow to between automatic WAL checkpoints.
- **max_locks_per_transaction.** This parameter controls the average number of object locks allocated for each transaction; individual transactions can lock more objects as long as the locks of all transactions fit in the lock table.

For information about the configuration parameters, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/12/static/runtime-config.html

9.1.2.2 Status Logging

Status logging by the cloning functions creates log files in the directory specified by the `log_directory` parameter in the `postgresql.conf` file for the database server to which you are connected when invoking the cloning function.

The default location is `PGDATA/log` as shown by the following:

```
#log_directory = 'log'          # directory where log files are written,
                                # can be absolute or relative to PGDATA
```

This directory must exist prior to running a cloning function.

The name of the log file is determined by what you specify in the parameter list when invoking the cloning function.

To display the status from a log file, use the `process_status_from_log` function as described in Section 9.2.5.

To delete a log file, use the `remove_log_file_and_job` function as described in Section 9.2.6, or simply navigate to the log directory and delete it manually.

9.1.3 Installing EDB Clone Schema

The following are the directions for installing EDB Clone Schema.

These steps must be performed on any database to be used as the source or target database by an EDB Clone Schema function.

Step 1: If you had previously installed an older version of the `edb_cloneschema` extension, then you must run the following command:

```
DROP EXTENSION parallel_clone CASCADE;
```

This command also drops the `edb_cloneschema` extension.

Step 2: Install the extensions using the following commands:

```
CREATE EXTENSION parallel_clone SCHEMA public;
CREATE EXTENSION edb_cloneschema;
```

Make sure you create the `parallel_clone` extension before creating the `edb_cloneschema` extension.

9.1.4 Creating the Foreign Servers and User Mappings

When using one of the local cloning functions, `localcopyschema` or `localcopyschema_nb`, one of the required parameters includes a single, foreign server for identifying the database server along with its database that is the source and the receiver of the cloned schema.

When using one of the remote cloning functions, `remotecopyschema` or `remotecopyschema_nb`, two of the required parameters include two foreign servers.

The foreign server specified as the first parameter identifies the source database server along with its database that is the provider of the cloned schema. The foreign server specified as the second parameter identifies the target database server along with its database that is the receiver of the cloned schema.

For each foreign server, a user mapping must be created. When a selected database superuser invokes a cloning function, that database superuser who invokes the function must have been mapped to a database user name and password that has access to the foreign server that is specified as a parameter in the cloning function.

For general information about foreign data, foreign servers, and user mappings, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/12/static/ddl-foreign-data.html

The following two sections describe how these foreign servers and user mappings are defined.

9.1.4.1 Foreign Server and User Mapping for Local Cloning Functions

For the `localcopyschema` and `localcopyschema_nb` functions, the source and target schemas are both within the same database of the same database server. Thus, only one foreign server must be defined and specified for these functions. This foreign server is also referred to as the *local server*.

This server is referred to as the local server because this server is the one to which you must be connected when invoking the `localcopyschema` or `localcopyschema_nb` function.

The user mapping defines the connection and authentication information for the foreign server.

This foreign server and user mapping must be created within the database of the local server in which the cloning is to occur.

The database user for whom the user mapping is defined must be a superuser and the user connected to the local server when invoking an EDB Clone Schema function.

The following example creates the foreign server for the database containing the schema to be cloned, and to receive the cloned schema as well.

```
CREATE SERVER local_server FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS(
    host 'localhost',
    port '5444',
    dbname 'edb'
);
```

For more information about using the `CREATE SERVER` command, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/12/static/sql-createserver.html

The user mapping for this server is the following:

```
    CREATE USER MAPPING FOR enterprisedb SERVER local_server
      OPTIONS (
        user 'enterprisedb',
        password 'password'
    );
```

For more information about using the `CREATE USER MAPPING` command, see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/12/static/sql-createusermapping.html

The following `psql` commands show the foreign server and user mapping:

```
edb=# \des+
List of foreign servers
-[ RECORD 1 ]--------+----------------------------------------------
Name                 | local_server
Owner                | enterprisedb
Foreign-data wrapper | postgres_fdw
Access privileges    |
Type                 |
Version              |
FDW options          | (host 'localhost', port '5444', dbname 'edb')
Description          |

edb=# \deu+
                           List of user mappings
    Server    |  User name   |                 FDW options
--------------+--------------+----------------------------------------------
 local_server | enterprisedb | ("user" 'enterprisedb', password 'password')
(1 row)
```

When database superuser `enterprisedb` invokes a cloning function, the database user `enterprisedb` with its password is used to connect to `local_server` on the `localhost` with port `5444` to database `edb`.

In this case, the mapped database user, `enterprisedb`, and the database user, `enterprisedb`, used to connect to the local `edb` database happen to be the same, identical database user, but that is not an absolute requirement.

For specific usage of these foreign server and user mapping examples, see the example given in Section 9.2.1.

9.1.4.2 Foreign Server and User Mapping for Remote Cloning Functions

For the `remotecopyschema` and `remotecopyschema_nb` functions, the source and target schemas are in different databases of either the same or different database servers. Thus, two foreign servers must be defined and specified for these functions.

The foreign server defining the originating database server and its database containing the source schema to be cloned is referred to as the *source server* or the *remote server*.

The foreign server defining the database server and its database to receive the schema to be cloned is referred to as the *target server* or the *local server*.

The target server is also referred to as the local server because this server is the one to which you must be connected when invoking the `remotecopyschema` or `remotecopyschema_nb` function.

The user mappings define the connection and authentication information for the foreign servers.

All of these foreign servers and user mappings must be created within the target database of the target/local server.

The database user for whom the user mappings are defined must be a superuser and the user connected to the local server when invoking an EDB Clone Schema function.

The following example creates the foreign server for the local, target database that is to receive the cloned schema.

```
CREATE SERVER tgt_server FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS(
    host 'localhost',
    port '5444',
    dbname 'tgtdb'
);
```

The user mapping for this server is the following:

```
CREATE USER MAPPING FOR enterprisedb SERVER tgt_server
  OPTIONS (
    user 'tgtuser',
    password 'tgtpassword'
);
```

The following example creates the foreign server for the remote, source database that is to be the source for the cloned schema.

```
CREATE SERVER src_server FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS(
    host '192.168.2.28',
    port '5444',
    dbname 'srcdb'
);
```

The user mapping for this server is the following:

```
CREATE USER MAPPING FOR enterprisedb SERVER src_server
  OPTIONS (
    user 'srcuser',
    password 'srcpassword'
);
```

The following `psql` commands show the foreign servers and user mappings:

```
tgtdb=# \des+
List of foreign servers
-[ RECORD 1 ]--------+----------------------------------------------------
Name                 | src_server
Owner                | tgtuser
Foreign-data wrapper | postgres_fdw
Access privileges    |
Type                 |
Version              |
FDW options          | (host '192.168.2.28', port '5444', dbname 'srcdb')
Description          |
-[ RECORD 2 ]--------+----------------------------------------------------
Name                 | tgt_server
Owner                | tgtuser
Foreign-data wrapper | postgres_fdw
Access privileges    |
Type                 |
Version              |
FDW options          | (host 'localhost', port '5444', dbname 'tgtdb')
Description          |

tgtdb=# \deu+
                         List of user mappings
   Server    |  User name   |                 FDW options
-------------+--------------+-------------------------------------------
 src_server | enterprisedb | ("user" 'srcuser', password 'srcpassword')
 tgt_server | enterprisedb | ("user" 'tgtuser', password 'tgtpassword')
(2 rows)
```

When database superuser `enterprisedb` invokes a cloning function, the database user `tgtuser` with password `tgtpassword` is used to connect to `tgt_server` on the `localhost` with port `5444` to database `tgtdb`.

In addition, database user `srcuser` with password `srcpassword` connects to `src_server` on host `192.168.2.28` with port `5444` to database `srcdb`.

Note: Be sure the `pg_hba.conf` file of the database server running the source database `srcdb` has an appropriate entry permitting connection from the target server location (address `192.168.2.27` in the following example) connecting with the database user `srcuser` that was included in the user mapping for the foreign server `src_server` defining the source server and database.

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     md5
# IPv4 local connections:
host    srcdb           srcuser         192.168.2.27/32         md5
```

For specific usage of these foreign server and user mapping examples, see the example given in Section 9.2.3.

9.2 EDB Clone Schema Functions

The EDB Clone Schema functions are created in the `edb_util` schema when the `parallel_clone` and `edb_cloneschema` extensions are installed.

Verify the following conditions before using an EDB Clone Schema function:

- You are connected to the target or local database as the database superuser defined in the `CREATE USER MAPPING` command for the foreign server of the target or local database. See Section 9.1.4.1 for information on the user mapping for the `localcopyschema` or `localcopyschema_nb` function. See Section 9.1.4.2 for information on the user mapping for the `remotecopyschema` or `remotecopyschema_nb` function.
- The `edb_util` schema is in the search path, or the cloning function is to be invoked with the `edb_util` prefix.
- The target schema does not exist in the target database.
- When using the remote copy functions, if the *`on_tblspace`* parameter is to be set to `true`, then the target database cluster contains all tablespaces that are referenced by objects in the source schema, otherwise creation of the DDL statements for those database objects will fail in the target schema. This causes a failure of the cloning process.
- When using the remote copy functions, if the *`copy_acls`* parameter is to be set to `true`, then all roles that have `GRANT` privileges on objects in the source schema exist in the target database cluster, otherwise granting of privileges to those roles will fail in the target schema. This causes a failure of the cloning process.
- pgAgent is running against the target database if the non-blocking form of the function is to be used.

For information about pgAgent, see the following section of the pgAdmin documentation available at:

https://www.pgadmin.org/docs/pgadmin4/dev/pgagent.html

Note that pgAgent is provided as a component with Advanced Server.

9.2.1 localcopyschema

The `localcopyschema` function copies a schema and its database objects within a local database specified within the *`source_fdw`* foreign server from the source schema to the specified target schema within the same database.

```
localcopyschema(
  source_fdw TEXT,
  source_schema TEXT,
  target_schema TEXT,
  log_filename TEXT
  [, on_tblspace BOOLEAN
  [, verbose_on BOOLEAN
  [, copy_acls BOOLEAN
  [, worker_count INTEGER ]]]]
)
```

A `BOOLEAN` value is returned by the function. If the function succeeds, then `true` is returned. If the function fails, then `false` is returned.

The *`source_fdw`*, *`source_schema`*, *`target_schema`*, and *`log_filename`* are required parameters while all other parameters are optional.

Parameters

`source_fdw`

Name of the foreign server managed by the `postgres_fdw` foreign data wrapper from which database objects are to be cloned.

`source_schema`

Name of the schema from which database objects are to be cloned.

`target_schema`

Name of the schema into which database objects are to be cloned from the source schema.

log_filename

> Name of the log file in which information from the function is recorded. The log file is created under the directory specified by the `log_directory` configuration parameter in the `postgresql.conf` file.

on_tblspace

> `BOOLEAN` value to specify whether or not database objects are to be created within their tablespaces. If `false` is specified, then the `TABLESPACE` clause is not included in the applicable `CREATE` DDL statement when added to the target schema. If `true` is specified, then the `TABLESPACE` clause is included in the `CREATE` DDL statement when added to the target schema. If the *on_tblspace* parameter is omitted, the default value is `false`.

verbose_on

> `BOOLEAN` value to specify whether or not the DDLs are to be printed in *log_filename* when creating objects in the target schema. If `false` is specified, then DDLs are not printed. If `true` is specified, then DDLs are printed. If omitted, the default value is `false`.

copy_acls

> `BOOLEAN` value to specify whether or not the access control list (ACL) is to be included while creating objects in the target schema. The access control list is the set of `GRANT` privilege statements. If `false` is specified, then the access control list is not included for the target schema. If `true` is specified, then the access control list is included for the target schema. If the *copy_acls* parameter is omitted, the default value is `false`.

worker_count

> Number of background workers to perform the clone in parallel. If omitted, the default value is `1`.

Example

The following example shows the cloning of schema `edb` containing a set of database objects to target schema `edbcopy`, both within database `edb` as defined by `local_server`.

The example is for the following environment:

- Host on which the database server is running: `localhost`

- Port of the database server: `5444`
- Database source/target of the clone: `edb`
- Foreign server (`local_server`) and user mapping (see Section 9.1.4.1) with the information of the preceding bullet points
- Source schema: `edb`
- Target schema: `edbcopy`
- Database superuser to invoke `localcopyschema`: `enterprisedb`

Before invoking the function, the connection is made by database user `enterprisedb` to database `edb`.

```
edb=# SET search_path TO "$user",public,edb_util;
SET
edb=# SHOW search_path;
        search_path
---------------------------
 "$user", public, edb_util
(1 row)

edb=# SELECT localcopyschema ('local_server','edb','edbcopy','clone_edb_edbcopy');
 localcopyschema
-----------------
 t
(1 row)
```

The following displays the logging status using the `process_status_from_log` function:

```
edb=# SELECT process_status_from_log('clone_edb_edbcopy');
                                    process_status_from_log
----------------------------------------------------------------------------------------------
--
 (FINISH,"2017-06-29 11:07:36.830783-04",3855,INFO,"STAGE: FINAL","successfully cloned
schema")
(1 row)
```

After the clone has completed, the following shows some of the database objects copied to the `edbcopy` schema:

```
edb=# SET search_path TO edbcopy;
SET
edb=# \dt+
                          List of relations
 Schema  |  Name   | Type  |    Owner     |    Size    | Description
---------+---------+-------+--------------+------------+-------------
 edbcopy | dept    | table | enterprisedb | 8192 bytes |
 edbcopy | emp     | table | enterprisedb | 8192 bytes |
 edbcopy | jobhist | table | enterprisedb | 8192 bytes |
(3 rows)

edb=# \dv
            List of relations
 Schema  |   Name   | Type |    Owner
---------+----------+------+--------------
 edbcopy | salesemp | view | enterprisedb
(1 row)
```

```
edb=# \di
                    List of relations
 Schema  |     Name      | Type  |    Owner     |  Table
---------+---------------+-------+--------------+---------
 edbcopy | dept_dname_uq | index | enterprisedb | dept
 edbcopy | dept_pk       | index | enterprisedb | dept
 edbcopy | emp_pk        | index | enterprisedb | emp
 edbcopy | jobhist_pk    | index | enterprisedb | jobhist
(4 rows)

edb=# \ds
               List of relations
 Schema  |    Name    |   Type   |    Owner
---------+------------+----------+--------------
 edbcopy | next_empno | sequence | enterprisedb
(1 row)

edb=# SELECT DISTINCT schema_name, name, type FROM user_source WHERE
schema_name = 'EDBCOPY' ORDER BY type, name;
 schema_name |              name              |     type
-------------+--------------------------------+--------------
 EDBCOPY     | EMP_COMP                       | FUNCTION
 EDBCOPY     | HIRE_CLERK                     | FUNCTION
 EDBCOPY     | HIRE_SALESMAN                  | FUNCTION
 EDBCOPY     | NEW_EMPNO                      | FUNCTION
 EDBCOPY     | EMP_ADMIN                      | PACKAGE
 EDBCOPY     | EMP_ADMIN                      | PACKAGE BODY
 EDBCOPY     | EMP_QUERY                      | PROCEDURE
 EDBCOPY     | EMP_QUERY_CALLER               | PROCEDURE
 EDBCOPY     | LIST_EMP                       | PROCEDURE
 EDBCOPY     | SELECT_EMP                     | PROCEDURE
 EDBCOPY     | EMP_SAL_TRIG                   | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_19991" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_19992" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_19999" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_20000" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_20004" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_a_20005" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_19993" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_19994" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_20001" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_20002" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_20006" | TRIGGER
 EDBCOPY     | "RI_ConstraintTrigger_c_20007" | TRIGGER
 EDBCOPY     | USER_AUDIT_TRIG                | TRIGGER
(24 rows)
```

9.2.2 localcopyschema_nb

The `localcopyschema_nb` function copies a schema and its database objects within a local database specified within the *`source_fdw`* foreign server from the source schema to the specified target schema within the same database, but in a non-blocking manner as a job submitted to pgAgent.

```
localcopyschema_nb(
  source_fdw TEXT,
  source TEXT,
  target TEXT,
  log_filename TEXT
  [, on_tblspace BOOLEAN
  [, verbose_on BOOLEAN
  [, copy_acls BOOLEAN
  [, worker_count INTEGER ]]]]
)
```

An `INTEGER` value job ID is returned by the function for the job submitted to pgAgent. If the function fails, then null is returned.

The *source_fdw*, *source*, *target*, and *log_filename* are required parameters while all other parameters are optional.

After completion of the pgAgent job, remove the job with the `remove_log_file_and_job` function (see Section 9.2.6).

Parameters

source_fdw

> Name of the foreign server managed by the `postgres_fdw` foreign data wrapper from which database objects are to be cloned.

source

> Name of the schema from which database objects are to be cloned.

target

> Name of the schema into which database objects are to be cloned from the source schema.

log_filename

> Name of the log file in which information from the function is recorded. The log file is created under the directory specified by the `log_directory` configuration parameter in the `postgresql.conf` file.

on_tblspace

> `BOOLEAN` value to specify whether or not database objects are to be created within their tablespaces. If `false` is specified, then the `TABLESPACE` clause is not included in the applicable `CREATE` DDL statement when added to the target schema. If `true` is specified, then the `TABLESPACE` clause is included in the

CREATE DDL statement when added to the target schema. If the *on_tblspace* parameter is omitted, the default value is false.

verbose_on

BOOLEAN value to specify whether or not the DDLs are to be printed in *log_filename* when creating objects in the target schema. If false is specified, then DDLs are not printed. If true is specified, then DDLs are printed. If omitted, the default value is false.

copy_acls

BOOLEAN value to specify whether or not the access control list (ACL) is to be included while creating objects in the target schema. The access control list is the set of GRANT privilege statements. If false is specified, then the access control list is not included for the target schema. If true is specified, then the access control list is included for the target schema. If the *copy_acls* parameter is omitted, the default value is false.

worker_count

Number of background workers to perform the clone in parallel. If omitted, the default value is 1.

Example

The same cloning operation is performed as the example in Section 9.2.1, but using the non-blocking function localcopyschema_nb.

The following command can be used to observe if pgAgent is running on the appropriate local database:

```
[root@localhost ~]# ps -ef | grep pgagent
root       4518      1  0 11:35 pts/1    00:00:00 pgagent -s /tmp/pgagent_edb_log
hostaddr=127.0.0.1 port=5444 dbname=edb user=enterprisedb password=password
root       4525   4399  0 11:35 pts/1    00:00:00 grep --color=auto pgagent
```

If pgAgent is not running, it can be started as shown by the following. The pgagent program file is located in the bin subdirectory of the Advanced Server installation directory.

```
[root@localhost bin]# ./pgagent -l 2 -s /tmp/pgagent_edb_log hostaddr=127.0.0.1 port=5444
dbname=edb user=enterprisedb password=password
```

Note: the pgagent -l 2 option starts pgAgent in DEBUG mode, which logs continuous debugging information into the log file specified with the -s option. Use a lower value for the -l option, or omit it entirely to record less information.

The `localcopyschema_nb` function returns the job ID shown as `4` in the example.

```
edb=# SELECT edb_util.localcopyschema_nb ('local_server','edb','edbcopy','clone_edb_edbcopy');
 localcopyschema_nb
--------------------
                  4
(1 row)
```

The following displays the job status:

```
edb=# SELECT edb_util.process_status_from_log('clone_edb_edbcopy');
                                      process_status_from_log
---------------------------------------------------------------------------------------------------
-----
 (FINISH,"29-JUN-17 11:39:11.620093 -04:00",4618,INFO,"STAGE: FINAL","successfully cloned
schema")
(1 row)
```

The following removes the pgAgent job:

```
edb=# SELECT edb_util.remove_log_file_and_job (4);
 remove_log_file_and_job
-------------------------
 t
(1 row)
```

9.2.3 remotecopyschema

The `remotecopyschema` function copies a schema and its database objects from a source schema in the remote source database specified within the *`source_fdw`* foreign server to a target schema in the local target database specified within the *`target_fdw`* foreign server.

```
remotecopyschema(
  source_fdw TEXT,
  target_fdw TEXT,
  source_schema TEXT,
  target_schema TEXT,
  log_filename TEXT
  [, on_tblspace BOOLEAN
  [, verbose_on BOOLEAN
  [, copy_acls BOOLEAN
  [, worker_count INTEGER ]]]]
)
```

A `BOOLEAN` value is returned by the function. If the function succeeds, then `true` is returned. If the function fails, then `false` is returned.

The *`source_fdw`*, *`target_fdw`*, *`source_schema`*, *`target_schema`*, and *`log_filename`* are required parameters while all other parameters are optional.

Parameters

source_fdw

Name of the foreign server managed by the `postgres_fdw` foreign data wrapper from which database objects are to be cloned.

target_fdw

Name of the foreign server managed by the `postgres_fdw` foreign data wrapper to which database objects are to be cloned.

source_schema

Name of the schema from which database objects are to be cloned.

target_schema

Name of the schema into which database objects are to be cloned from the source schema.

log_filename

Name of the log file in which information from the function is recorded. The log file is created under the directory specified by the `log_directory` configuration parameter in the `postgresql.conf` file.

on_tblspace

`BOOLEAN` value to specify whether or not database objects are to be created within their tablespaces. If `false` is specified, then the `TABLESPACE` clause is not included in the applicable `CREATE` DDL statement when added to the target schema. If `true` is specified, then the `TABLESPACE` clause is included in the `CREATE` DDL statement when added to the target schema. If the *on_tblspace* parameter is omitted, the default value is `false`.

Note: If `true` is specified and a database object has a `TABLESPACE` clause, but that tablespace does not exist in the target database cluster, then the cloning function fails.

verbose_on

`BOOLEAN` value to specify whether or not the DDLs are to be printed in *log_filename* when creating objects in the target schema. If `false` is specified, then DDLs are not printed. If `true` is specified, then DDLs are printed. If omitted, the default value is `false`.

copy_acls

> `BOOLEAN` value to specify whether or not the access control list (ACL) is to be included while creating objects in the target schema. The access control list is the set of `GRANT` privilege statements. If `false` is specified, then the access control list is not included for the target schema. If `true` is specified, then the access control list is included for the target schema. If the *copy_acls* parameter is omitted, the default value is `false`.
>
> **Note:** If `true` is specified and a role with `GRANT` privilege does not exist in the target database cluster, then the cloning function fails.

worker_count

> Number of background workers to perform the clone in parallel. If omitted, the default value is `1`.

Example

The following example shows the cloning of schema `srcschema` within database `srcdb` as defined by `src_server` to target schema `tgtschema` within database `tgtdb` as defined by `tgt_server`.

The source server environment:

- Host on which the source database server is running: `192.168.2.28`
- Port of the source database server: `5444`
- Database source of the clone: `srcdb`
- Foreign server (`src_server`) and user mapping (see Section 9.1.4.2) with the information of the preceding bullet points
- Source schema: `srcschema`

The target server environment:

- Host on which the target database server is running: `localhost`
- Port of the target database server: `5444`
- Database target of the clone: `tgtdb`
- Foreign server (`tgt_server`) and user mapping (see Section 9.1.4.2) with the information of the preceding bullet points
- Target schema: `tgtschema`
- Database superuser to invoke `remotecopyschema`: `enterprisedb`

Before invoking the function, the connection is made by database user `enterprisedb` to database `tgtdb`. A `worker_count` of `4` is specified for this function.

```
tgtdb=# SELECT edb_util.remotecopyschema
('src_server','tgt_server','srcschema','tgtschema','clone_rmt_src_tgt',worker_count => 4);
 remotecopyschema
------------------
 t
(1 row)
```

The following displays the status from the log file during various points in the cloning process:

```
tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                                          process_status_from_log

---------------------------------------------------------------------------------------------
------------------------------------------
---
 (RUNNING,"28-JUN-17 13:18:05.299953 -04:00",4021,INFO,"STAGE: DATA-COPY","[0][0] successfully
copied data in [tgtschema.pgbench_tellers]
")
(1 row)

tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                                          process_status_from_log

---------------------------------------------------------------------------------------------
------------------------------------------
---
 (RUNNING,"28-JUN-17 13:18:06.634364 -04:00",4022,INFO,"STAGE: DATA-COPY","[0][1] successfully
copied data in [tgtschema.pgbench_history]
")
(1 row)

tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                                          process_status_from_log

---------------------------------------------------------------------------------------------
------------------------------------------
--
 (RUNNING,"28-JUN-17 13:18:10.550393 -04:00",4039,INFO,"STAGE: POST-DATA","CREATE PRIMARY KEY
CONSTRAINT pgbench_tellers_pkey successful"
)
(1 row)

tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                             process_status_from_log
---------------------------------------------------------------------------------------------
-------------------
 (FINISH,"28-JUN-17 13:18:12.019627 -04:00",4039,INFO,"STAGE: FINAL","successfully clone
schema into tgtschema")
(1 row)
```

The following shows the cloned tables:

```
tgtdb=# \dt+
                               List of relations
  Schema   |       Name       | Type  |    Owner     |    Size    | Description
-----------+------------------+-------+--------------+------------+-------------
 tgtschema | pgbench_accounts | table | enterprisedb | 256 MB     |
 tgtschema | pgbench_branches | table | enterprisedb | 8192 bytes |
 tgtschema | pgbench_history  | table | enterprisedb | 25 MB      |
 tgtschema | pgbench_tellers  | table | enterprisedb | 16 kB      |
(4 rows)
```

When the `remotecopyschema` function was invoked, four background workers were specified.

The following portion of the log file `clone_rmt_src_tgt` shows the status of the parallel data copying operation using four background workers:

```
Wed Jun 28 13:18:05.232949 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0] table count [4]
Wed Jun 28 13:18:05.233321 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0][0] worker started to
copy data
Wed Jun 28 13:18:05.233640 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0][1] worker started to
copy data
Wed Jun 28 13:18:05.233919 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0][2] worker started to
copy data
Wed Jun 28 13:18:05.234231 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0][3] worker started to
copy data
Wed Jun 28 13:18:05.298174 2017 EDT: [4024] INFO: [STAGE: DATA-COPY] [0][3] successfully
copied data in [tgtschema.pgbench_branches]
Wed Jun 28 13:18:05.299913 2017 EDT: [4021] INFO: [STAGE: DATA-COPY] [0][0] successfully
copied data in [tgtschema.pgbench_tellers]
Wed Jun 28 13:18:06.634310 2017 EDT: [4022] INFO: [STAGE: DATA-COPY] [0][1] successfully
copied data in [tgtschema.pgbench_history]
Wed Jun 28 13:18:10.477333 2017 EDT: [4023] INFO: [STAGE: DATA-COPY] [0][2] successfully
copied data in [tgtschema.pgbench_accounts]
Wed Jun 28 13:18:10.477609 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0] all workers finished
[4]
Wed Jun 28 13:18:10.477654 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] [0] copy done [4] tables
Wed Jun 28 13:18:10.493938 2017 EDT: [4019] INFO: [STAGE: DATA-COPY] successfully copied data
into tgtschema
```

Note that the `DATA-COPY` log message includes two, square bracket numbers (for example, `[0][3]`).

The first number is the job index whereas the second number is the worker index. The worker index values range from `0` to `3` for the four background workers.

In case two clone schema jobs are running in parallel, the first log file will have `0` as the job index whereas the second will have `1` as the job index.

9.2.4 remotecopyschema_nb

The `remotecopyschema_nb` function copies a schema and its database objects from a source schema in the remote source database specified within the *`source_fdw`* foreign server to a target schema in the local target database specified within the *`target_fdw`* foreign server, but in a non-blocking manner as a job submitted to pgAgent.

```
remotecopyschema_nb(
  source_fdw TEXT,
  target_fdw TEXT,
  source TEXT,
  target TEXT,
  log_filename TEXT
  [, on_tblspace BOOLEAN
  [, verbose_on BOOLEAN
```

```
      [, copy_acls BOOLEAN
      [, worker_count INTEGER ]]]]
    )
```

An `INTEGER` value job ID is returned by the function for the job submitted to pgAgent. If the function fails, then null is returned.

The *`source_fdw`*, *`target_fdw`*, *`source`*, *`target`*, and *`log_filename`* are required parameters while all other parameters are optional.

After completion of the pgAgent job, remove the job with the `remove_log_file_and_job` function (see Section 9.2.6).

Parameters

`source_fdw`

> Name of the foreign server managed by the `postgres_fdw` foreign data wrapper from which database objects are to be cloned.

`target_fdw`

> Name of the foreign server managed by the `postgres_fdw` foreign data wrapper to which database objects are to be cloned.

`source`

> Name of the schema from which database objects are to be cloned.

`target`

> Name of the schema into which database objects are to be cloned from the source schema.

`log_filename`

> Name of the log file in which information from the function is recorded. The log file is created under the directory specified by the `log_directory` configuration parameter in the `postgresql.conf` file.

`on_tblspace`

> `BOOLEAN` value to specify whether or not database objects are to be created within their tablespaces. If `false` is specified, then the `TABLESPACE` clause is not included in the applicable `CREATE` DDL statement when added to the target schema. If `true` is specified, then the `TABLESPACE` clause is included in the

`CREATE` DDL statement when added to the target schema. If the *`on_tblspace`* parameter is omitted, the default value is `false`.

Note: If `true` is specified and a database object has a `TABLESPACE` clause, but that tablespace does not exist in the target database cluster, then the cloning function fails.

`verbose_on`

`BOOLEAN` value to specify whether or not the DDLs are to be printed in *`log_filename`* when creating objects in the target schema. If `false` is specified, then DDLs are not printed. If `true` is specified, then DDLs are printed. If omitted, the default value is `false`.

`copy_acls`

`BOOLEAN` value to specify whether or not the access control list (ACL) is to be included while creating objects in the target schema. The access control list is the set of `GRANT` privilege statements. If `false` is specified, then the access control list is not included for the target schema. If `true` is specified, then the access control list is included for the target schema. If the *`copy_acls`* parameter is omitted, the default value is `false`.

Note: If `true` is specified and a role with `GRANT` privilege does not exist in the target database cluster, then the cloning function fails.

`worker_count`

Number of background workers to perform the clone in parallel. If omitted, the default value is `1`.

Example

The same cloning operation is performed as the example in Section 9.2.3, but using the non-blocking function `remotecopyschema_nb`.

The following command starts pgAgent on the target database `tgtdb`. The `pgagent` program file is located in the `bin` subdirectory of the Advanced Server installation directory.

```
[root@localhost bin]# ./pgagent -l 1 -s /tmp/pgagent_tgtdb_log hostaddr=127.0.0.1 port=5444
user=enterprisedb dbname=tgtdb password=password
```

The `remotecopyschema_nb` function returns the job ID shown as `2` in the example.

```
tgtdb=# SELECT edb_util.remotecopyschema_nb
('src_server','tgt_server','srcschema','tgtschema','clone_rmt_src_tgt',worker_count => 4);
 remotecopyschema_nb
---------------------
                   2
(1 row)
```

The completed status of the job is shown by the following:

```
tgtdb=# SELECT edb_util.process_status_from_log('clone_rmt_src_tgt');
                                             process_status_from_log
---------------------------------------------------------------------------------------------
-------------------
 (FINISH,"29-JUN-17 13:16:00.100284 -04:00",3849,INFO,"STAGE: FINAL","successfully clone
schema into tgtschema")
(1 row)
```

The following removes the log file and the pgAgent job:

```
tgtdb=# SELECT edb_util.remove_log_file_and_job ('clone_rmt_src_tgt',2);
 remove_log_file_and_job
-------------------------
 t
(1 row)
```

9.2.5 process_status_from_log

The `process_status_from_log` function provides the status of a cloning function from its log file.

```
process_status_from_log (
  log_file TEXT
)
```

The function returns the following fields from the log file:

Table 9-1 - Clone Schema Log File

Field Name	Description
`status`	Displays either `STARTING`, `RUNNING`, `FINISH`, or `FAILED`.
`execution_time`	When the command was executed. Displayed in timestamp format.
`pid`	Session process ID in which clone schema is getting called.
`level`	Displays either `INFO`, `ERROR`, or `SUCCESSFUL`.
`stage`	Displays either `STARTUP`, `INITIAL`, `DDL-COLLECTION`, `PRE-DATA`, `DATA-COPY`, `POST-DATA`, or `FINAL`.
`message`	Information respective to each command or failure.

Parameters

`log_file`

Name of the log file recording the cloning of a schema as specified when the cloning function was invoked.

Example

The following shows usage of the `process_status_from_log` function:

```
edb=# SELECT edb_util.process_status_from_log('clone_edb_edbcopy');
                                    process_status_from_log
-------------------------------------------------------------------------------------------
-----
 (FINISH,"26-JUN-17 11:57:03.214458 -04:00",3691,INFO,"STAGE: FINAL","successfully cloned
schema")
(1 row)
```

9.2.6 remove_log_file_and_job

The `remove_log_file_and_job` function performs cleanup tasks by removing the log files created by the schema cloning functions and the jobs created by the non-blocking functions.

```
remove_log_file_and_job (
  { log_file TEXT |
    job_id INTEGER |
    log_file TEXT, job_id INTEGER
  }
)
```

Values for any or both of the two parameters may be specified when invoking the `remove_log_file_and_job` function:

- If only *`log_file`* is specified, then the function will only remove the log file.
- If only *`job_id`* is specified, then the function will only remove the job.
- If both are specified, then the function will remove the log file and the job.

Parameters

`log_file`

Name of the log file to be removed.

`job_id`

Job ID of the job to be removed.

Example

The following examples removes only the log file, given the log filename.

```
edb=# SELECT edb_util.remove_log_file_and_job ('clone_edb_edbcopy');
 remove_log_file_and_job
-------------------------
 t
(1 row)
```

The following example removes only the job, given the job ID.

```
edb=# SELECT edb_util.remove_log_file_and_job (3);
 remove_log_file_and_job
-------------------------
 t
(1 row)
```

The following example removes the log file and the job, given both values:

```
tgtdb=# SELECT edb_util.remove_log_file_and_job ('clone_rmt_src_tgt',2);
 remove_log_file_and_job
-------------------------
 t
(1 row)
```

10 Enhanced SQL and Other Miscellaneous Features

Advanced Server includes enhanced SQL functionality and various other features that provide additional flexibility and convenience. This chapter discusses some of these additions.

10.1 COMMENT

In addition to commenting on objects supported by the PostgreSQL `COMMENT` command, Advanced Server supports comments on additional object types. The complete supported syntax is:

```
COMMENT ON
{
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  COLUMN relation_name.column_name |
  CONSTRAINT constraint_name ON table_name |
  CONSTRAINT constraint_name ON DOMAIN domain_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  EXTENSION object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION func_name ([[argmode] [argname] argtype [, ...]])|
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  PACKAGE object_name
  POLICY policy_name ON table_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE proc_name [([[argmode] [argname] argtype [, ...]])]
  PUBLIC SYNONYM object_name
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TABLESPACE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name
} IS 'text'
```

where *aggregate_signature* is:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ]
ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

Parameters

object_name

The name of the object on which you are commenting.

AGGREGATE *aggregate_name* (*aggregate_signature*)

Include the AGGREGATE clause to create a comment about an aggregate. *aggregate_name* specifies the name of an aggregate, and *aggregate_signature* specifies the associated signature in one of the following forms:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ]
ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

Where *argmode* is the mode of a function, procedure, or aggregate argument; argmode may be IN, OUT, INOUT, or VARIADIC. If omitted, the default is IN.

argname is the name of an aggregate argument.

argtype is the data type of an aggregate argument.

CAST (*source_type* AS *target_type*)

Include the CAST clause to create a comment about a cast. When creating a comment about a cast, *source_type* specifies the source data type of the cast, and *target_type* specifies the target data type of the cast.

`COLUMN` *`relation_name.column_name`*

> Include the `COLUMN` clause to create a comment about a column. *`column_name`* specifies name of the column to which the comment applies. *`relation_name`* is the table, view, composite type, or foreign table in which a column resides.

`CONSTRAINT` *`constraint_name`* `ON` *`table_name`*
`CONSTRAINT` *`constraint_name`* `ON DOMAIN` *`domain_name`*

> Include the `CONSTRAINT` clause to add a comment about a constraint. When creating a comment about a constraint, *`constraint_name`* specifies the name of the constraint; *`table_name`* or *`domain_name`* specifies the name of the table or domain on which the constraint is defined.

`FUNCTION` *`func_name`* `([[`*`argmode`*`] [`*`argname`*`]` *`argtype`* `[, ...]])`

> Include the `FUNCTION` clause to add a comment about a function. *`func_name`* specifies the name of the function. *`argmode`* specifies the mode of the function; *`argmode`* may be `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, the default is `IN`. *`argname`* specifies the name of a function, procedure, or aggregate argument. *`argtype`* specifies the data type of a function, procedure, or aggregate argument.

`large_object_oid`

> *`large_object_oid`* is the system-assigned OID of the large object about which you are commenting.

`OPERATOR` *`operator_name`* `(left_type, right_type)`

> Include the `OPERATOR` clause to add a comment about an operator. *`operator_name`* specifies the (optionally schema-qualified) name of an operator on which you are commenting. *`left_type`* and *`right_type`* are the (optionally schema-qualified) data type(s) of the operator's arguments.

`OPERATOR CLASS` *`object_name`* `USING` *`index_method`*

> Include the `OPERATOR CLASS` clause to add a comment about an operator class. *`object_name`* specifies the (optionally schema-qualified) name of an operator on which you are commenting. *`index_method`* specifies the associated index method of the operator class.

`OPERATOR FAMILY` *`object_name`* `USING` *`index_method`*

> Include the `OPERATOR FAMILY` clause to add a comment about an operator family. *`object_name`* specifies the (optionally schema-qualified) name of an

operator family on which you are commenting. `index_method` specifies the associated index method of the operator family.

`POLICY policy_name ON table_name`

Include the `POLICY` clause to add a comment about a policy. `policy_name` specifies the name of the policy, and `table_name` specifies the table that the policy is associated with.

`PROCEDURE proc_name [([[argmode] [argname] argtype [, ...]])]`

Include the `PROCEDURE` clause to add a comment about a procedure. `proc_name` specifies the name of the procedure. `argmode` specifies the mode of the procedure; `argmode` may be `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, the default is `IN`. `argname` specifies the name of a function, procedure, or aggregate argument. `argtype` specifies the data type of a function, procedure, or aggregate argument.

`RULE rule_name ON table_name`

Include the `RULE` clause to specify a `COMMENT` on a rule. `rule_name` specifies the name of the rule, and `table_name` specifies the name of the table on which the rule is defined.

`TRANSFORM FOR type_name LANGUAGE lang_name |`

Include the `TRANSFORM FOR` clause to specify a `COMMENT` on a `TRANSFORM`. `type_name` specifies the name of the data type of the transform and `lang_name` specifies the name of the language of the transform.

`TRIGGER trigger_name ON table_name`

Include the `TRIGGER` clause to specify a `COMMENT` on a trigger. `trigger_name` specifies the name of the trigger, and `table_name` specifies the name of the table on which the trigger is defined.

`text`

The comment, written as a string literal; or `NULL` to drop the comment.

Notes:

Names of tables, aggregates, collations, conversions, domains, foreign tables, functions, indexes, operators, operator classes, operator families, packages, procedures, sequences, text search objects, types, and views can be schema-qualified.

Example:

The following example adds a comment to a table named `new_emp`:

```
COMMENT ON TABLE new_emp IS 'This table contains information about new
employees.';
```

For more information about using the `COMMENT` command, please see the PostgreSQL core documentation at:

https://www.postgresql.org/docs/12/static/sql-comment.html

10.2 Output of Function version()

The text string output of the `version()` function displays the name of the product, its version, and the host system on which it has been installed.

For Advanced Server, the `version()` output is in a format similar to the PostgreSQL community version in that the first text word is *PostgreSQL* instead of *EnterpriseDB* as in Advanced Server version 10 and earlier.

The general format of the version() output is the following:

```
PostgreSQL $PG_VERSION_EXT (EnterpriseDB Advanced Server $PG_VERSION) on $host
```

So for the current Advanced Server the version string appears as follows:

```
edb@45032=#select version();
version
-----------------------------------------------------------------------------------------------
-------------------------------------------------
PostgreSQL 12.0 (EnterpriseDB Advanced Server 12.0.0) on x86_64-pc-linux-gnu, compiled by gcc
(GCC) 4.8.5 20150623 (Red Hat 4.8.5-11), 64-bit
(1 row)
```

In contrast, for Advanced Server 10, the version string was the following:

```
edb=# select version();
                                                   version
-----------------------------------------------------------------------------------------------
---------------
 EnterpriseDB 10.4.9 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.4.7 20120313 (Red Hat
4.4.7-18), 64-bit
(1 row)
```

10.3 Logical Decoding on Standby

Logical decoding on a standby server allows you to create a logical replication slot on a standby server that can respond to API operations such as `get`, `peek`, `advance`, etc..

For more information about the `LOGICAL DECODING`, please refer to the PostgreSQL core documentation available at:

https://www.postgresql.org/docs/12/logicaldecoding-explanation.html

For a logical slot on a standby server to work, the `hot_standby_feedback` parameter must be set to `ON` on the standby. The `hot_standby_feedback` parameter prevents `VACCUM` from removing recently-dead rows that are required by an existing logical replication slot on the standby server. If a slot conflict occurs on the standby, the slots will be dropped.

For logical decoding on a standby to work, `wal_level` must be set to `logical` on both the primary and standby server. If `wal_level` is set to a value other than `logical`, then slots are not created. If you set `wal_level` to a value other than `logical` on primary and if there are existing logical slots on standby, such slots are dropped and new slots cannot be created.

When transactions are written to the primary server, the activity will trigger the creation of a logical slot on the standby server. If a primary server is idle, creating a logical slot on a standby server may take noticeable time.

For more information about functions that support replication and logical decoding example, please refer to the PostgreSQL documentation available at:

https://www.postgresql.org/docs/12/functions-admin.html#FUNCTIONS-REPLICATION

https://www.postgresql.org/docs/12/logicaldecoding-example.html

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 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.

11.2 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.

11.3 edb_policy

The `edb_policy` table contains one row for each policy.

Column	Type	Modifiers	Description
policyname	name	not null	The policy name.
policygroup	oid	not null	Currently unused.
policyobject	oid	not null	The OID of the table secured by this policy (the object_schema plus the object_name).
policykind	char	not null	The kind of object secured by this policy: 'r' for a table 'v' for a view = for a synonym Currently always 'r'.
policyproc	oid	not null	The OID of the policy function (function_schema plus policy_function).
policyinsert	boolean	not null	True if the policy is enforced by INSERT statements.
policyselect	boolean	not null	True if the policy is enforced by SELECT statements.
policydelete	boolean	not null	True if the policy is enforced by DELETE statements.
policyupdate	boolean	not null	True if the policy is enforced by UPDATE statements.
policyindex	boolean	not null	Currently unused.
policyenabled	boolean	not null	True if the policy is enabled.
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.

11.4 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

Column	Type	References	Description
			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.

11.5 edb_redaction_column

The catalog `edb_redaction_column` stores information of data redaction policy attached to the columns of the table.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
rdpolicyid	oid	edb_redaction_policy.oid	The data redaction policy applies to the described column
rdrelid	oid	pg_class.oid	The table that the described

Column	Type	References	Description
			column belongs to
rdattnum	int2	pg_attribute.attnum	The number of the described column
rdscope	int2		The redaction scope: 1 = query, 2 = top_tlist, 4 = top_tlist_or_error
rdexception	int2		The redaction exception: 8 = none, 16 = equal, 32 = leakproof
rdfuncexpr	pg_node_tree		Data redaction function expression

Note: The described column will be redacted if the redaction policy `edb_redaction_column.rdpolicyid` on the table is enabled and the redaction policy expression `edb_redaction_policy.rdexpr` evaluates to true.

11.6 edb_redaction_policy

The catalog `edb_redaction_policy` stores information of the redaction policies for tables.

Column	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
rdname	name		The name of the data redaction policy
rdrelid	oid	pg_class.oid	The table to which the data redaction policy applies
rdenable	boolean		Is the data redaction policy enabled?
rdexpr	pg_node_tree		The data redaction policy expression

Note: The data redaction policy applies for the table if it is enabled and the expression ever evaluated true.

11.7 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.

11.8 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.

11.9 pg_synonym

The `pg_synonym` table contains one row for each synonym created with the `CREATE SYNONYM` command or `CREATE PUBLIC SYNONYM` command.

Column	Type	Modifiers	Description
`synname`	`"name"`	`not null`	The name of the synonym.
`synnamespace`	`oid`	`not null`	Replaces `synowner`. Contains the OID of the `pg_namespace` row where the synonym is stored
`synowner`	`oid`	`not null`	The OID of the user that owns the synonym.
`synobjschema`	`"name"`	`not null`	The schema in which the referenced object is defined.
`synobjname`	`"name"`	`not null`	The name of the referenced object.
`synlink`	`text`		The (optional) name of the database link in which the referenced object is defined.

11.10 product_component_version

The `product_component_version` table contains information about feature compatibility; an application can query this table at installation or run time to verify that features used by the application are available with this deployment.

Column	Type	Description
`product`	`character varying (74)`	The name of the product.
`version`	`character varying (74)`	The version number of the product.
`status`	`character varying (74)`	The status of the release.

12 Advanced Server Keywords

A keyword is a word that is recognized by the Advanced Server parser as having a special meaning or association. You can use the `pg_get_keywords()` function to retrieve an up-to-date list of the Advanced Server keywords:

```
acctg=#
acctg=# SELECT * FROM pg_get_keywords();
        word         | catcode |          catdesc
---------------------+---------+--------------------------------
 abort               | U       | unreserved
 absolute            | U       | unreserved
 access              | U       | unreserved
...
```

`pg_get_keywords` returns a table containing the keywords recognized by Advanced Server:

- The `word` column displays the keyword.
- The `catcode` column displays a category code.
- The `catdesc` column displays a brief description of the category to which the keyword belongs.

Note that any character can be used in an identifier if the name is enclosed in double quotes. You can selectively query the `pg_get_keywords()` function to retrieve an up-to-date list of the Advanced Server keywords that belong to a specific category:

```
SELECT * FROM pg_get_keywords() WHERE catcode = 'code';
```

Where *`code`* is:

`R` - The word is reserved. Reserved keywords may never be used as an identifier; they are reserved for use by the server.

`U` - The word is unreserved. Unreserved words are used internally in some contexts, but may be used as a name for a database object.

`T` - The word is used internally, but may be used as a name for a function or type.

`C` - The word is used internally, and may not be used as a name for a function or type.

For more information about Advanced Server identifiers and keywords, please refer to the PostgreSQL core documentation at:

https://www.postgresql.org/docs/12/static/sql-syntax-lexical.html