



# **Postgres Plus® Advanced Server Guide**

**Postgres Plus Advanced Server 8.4**

**June 23, 2010**

**Postgres Plus Advanced Server Guide, Version 2.1**  
**by EnterpriseDB Corporation**  
**Copyright © 2010 EnterpriseDB Corporation**

EnterpriseDB Corporation, 235 Littleton Road, Westford, MA 01866, USA  
**T** +1 978 589 5700 **F** +1 978 589 5701 **E** [info@enterprisedb.com](mailto:info@enterprisedb.com) [www.enterprisedb.com](http://www.enterprisedb.com)

# Table of Contents

1	Introduction.....	5
1.1	Typographical Conventions Used in this Guide.....	6
1.2	About the Examples Used in this Guide.....	7
1.2.1.1	Sample Database Description.....	8
2	Database Administration.....	18
2.1	Database Management.....	18
2.1.1	Logging into DBA Management Server.....	18
2.1.2	Database Read/Write Activity Report.....	20
2.1.3	Add/Remove Cluster.....	22
2.1.4	Monitoring.....	26
2.1.4.1	User Activity.....	26
2.1.4.2	Lock Status.....	27
2.1.4.3	Buffer Cache.....	28
2.1.4.4	Configuration.....	29
2.1.4.5	View DB Logs.....	32
2.1.4.6	View Audit Logs.....	33
2.1.5	SQL.....	33
2.1.5.1	iQuery.....	33
2.1.5.2	Query Profiler.....	35
2.2	Database Auditing.....	37
2.2.1	Auditing Configuration Parameters.....	38
2.2.2	View Audit Logs.....	41
2.3	High Availability and Load Balancing.....	43
2.3.1	Shared Disk Failover.....	44
2.3.2	Warm Standby Using Point-In-Time Recovery.....	44
2.3.3	Master-Slave Replication.....	44
2.3.4	Statement-Based Replication Middleware.....	45
2.3.5	Synchronous Multi-Master Replication.....	45
2.3.6	Asynchronous Multi-Master Replication.....	45
2.3.7	Data Partitioning.....	46
2.3.8	Multi-Server Parallel Query Execution.....	46
2.4	Protecting Against SQL Injection Attacks.....	46
2.4.1	SQL/Protect Overview.....	46
2.4.1.1	Types of SQL Injection Attacks.....	46
2.4.1.2	Protected Roles.....	47
2.4.1.3	Attack Attempt Statistics.....	48
2.4.2	Installing SQL/Protect.....	48
2.4.3	Configuring SQL/Protect.....	50
2.4.3.1	Selecting Roles to Protect.....	51
2.4.3.2	Setting the Protection Level.....	51
2.4.4	Using SQL/Protect.....	52

2.4.4.1	Basic Steps.....	52
2.4.4.2	Common Maintenance Operations .....	55
2.4.5	Backing Up and Restoring a SQL/Protect Database .....	58
2.4.5.1	Object Identification Numbers in SQL/Protect Tables.....	58
2.4.5.2	Backing Up the Database.....	59
2.4.5.3	Restoring From the Backup Files .....	60
3	Application Development.....	63
3.1	ECPG – Embedded SQL in C.....	63
3.1.1	Preprocessor.....	63
3.1.2	Library .....	63
3.1.3	Compilation and Linking.....	63
3.1.4	Installation .....	64
3.1.5	Supported Platforms .....	64
3.1.6	Prerequisites.....	64
3.2	libpq C Library .....	64
3.2.1	Using libpq with EnterpriseDB SPL.....	65
3.2.2	EnterpriseDB libpq API.....	65
3.2.2.1	Preparing a Callable Statement.....	65
3.2.2.2	Executing a Callable Statement.....	65
3.2.2.3	Fetching Cursors from a Result Set.....	66
3.2.2.4	Returning the Number of Cursors Fetched from a Result Set.....	66
3.2.2.5	Retrieving the Out Parameter from the Result .....	66
3.2.2.6	Cursor Support Statements in EnterpriseDB .....	66
3.2.2.7	Array Binding.....	67
3.2.2.8	PQBulkStart.....	67
3.2.2.9	PQexecBulk .....	67
3.2.2.10	PQBulkFinish .....	68
3.2.2.11	PQexecBulkPrepared.....	68
3.2.2.12	Example Code (Using PQBulkStart, PQexecBulk, PQBulkFinish).....	69
3.2.2.13	Example Code (Using PQexecBulkPrepared).....	70

# 1 Introduction

This guide describes the features of Postgres Plus Advanced Server that have been added to enhance the capabilities of the standard Postgres Plus product. Enhancements have been made in the areas of:

- Database administration
- Application development
- Database migration

This guide explains each of these areas in more detail.

## 1.1 *Typographical Conventions Used in this Guide*

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

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

- *Italic font* introduces a new term, typically, in the sentence that defines it for the first time.
- Fixed-width (mono-spaced) font is used for terms that must be given literally such as SQL commands, specific table and column names used in the examples, programming language keywords, 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;`
- **Arial bold font** is used for names of reports, menus, menu picks, buttons, check boxes, etc.
- 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”.

## 1.2 About the Examples Used in this Guide

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

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

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

The tables and programs in the sample database can be re-created at any time by executing the script, `edb-sample.sql`, located in the `samples` subdirectory of the Postgres Plus Advanced Server home directory.

This script does the following:

- 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;
```

Altering the search path can be done using commands in PSQL.

### 1.2.1.1 Sample Database Description

The sample database represents employees in an organization.

It contains three types of records: employees, departments, and historical records of employees.

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

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

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

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

```
--
-- Script that creates the 'sample' tables, views, procedures,
-- 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      NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname       VARCHAR2(14) CONSTRAINT dept_dname_uq UNIQUE,
  loc         VARCHAR2(13)
);
--
-- Create the 'emp' table
--
CREATE TABLE emp (
  empno       NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename       VARCHAR2(10),
  job         VARCHAR2(9),
  mgr         NUMBER(4),
```

```

hiredate      DATE,
sal           NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
comm         NUMBER(7,2),
deptno       NUMBER(2) CONSTRAINT emp_ref_dept_fk
             REFERENCES dept(deptno)
);
--
-- Create the 'jobhist' table
--
CREATE TABLE jobhist (
  empno       NUMBER(4) NOT NULL,
  startdate   DATE NOT NULL,
  enddate     DATE,
  job         VARCHAR2(9),
  sal         NUMBER(7,2),
  comm        NUMBER(7,2),
  deptno      NUMBER(2),
  chgdesc     VARCHAR2(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;
--
-- Procedure that lists all employees' numbers and names
-- from the 'emp' table using a cursor.
--
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno          NUMBER(4);
    v_ename          VARCHAR2(10);
    CURSOR emp_cur IS

```

```

        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO      ENAME');
    DBMS_OUTPUT.PUT_LINE('-----      -----');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '      ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;
/
--
-- Procedure that selects an employee row given the employee
-- number and displays certain columns.
--
CREATE OR REPLACE PROCEDURE select_emp (
    p_empno          IN  NUMBER
)
IS
    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      VARCHAR2(10);
BEGIN
    SELECT ename, hiredate, sal, NVL(comm, 0), dname
        INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
        FROM emp e, dept d
        WHERE empno = p_empno
            AND e.deptno = d.deptno;
    v_disp_date := TO_CHAR(v_hiredate, 'MM/DD/YYYY');
    DBMS_OUTPUT.PUT_LINE('Number      : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name        : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Hire Date   : ' || v_disp_date);
    DBMS_OUTPUT.PUT_LINE('Salary      : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission  : ' || v_comm);
    DBMS_OUTPUT.PUT_LINE('Department  : ' || v_dname);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
/
--
-- Procedure that queries the 'emp' table based on
-- department number and employee number or name. Returns
-- employee number and name as IN OUT parameters and job,
-- hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno         IN      NUMBER,
    p_empno          IN OUT NUMBER,
    p_ename          IN OUT VARCHAR2,
    p_job            OUT   VARCHAR2,
    p_hiredate       OUT   DATE,
    p_sal            OUT   NUMBER
)

```

```

)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
            AND (empno = p_empno
                OR ename = UPPER(p_ename));
END;
/
--
-- Procedure to call 'emp_query_caller' with IN and IN OUT
-- parameters. Displays the results received from IN OUT and
-- OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
    v_deptno      NUMBER(2);
    v_empno       NUMBER(4);
    v_ename       VARCHAR2(10);
    v_job         VARCHAR2(9);
    v_hiredate    DATE;
    v_sal         NUMBER;
BEGIN
    v_deptno := 30;
    v_empno := 0;
    v_ename := 'Martin';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job         : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
--
-- Function to compute yearly compensation based on semimonthly
-- salary.
--
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal      NUMBER,
    p_comm     NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
/
--
-- 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 RETURN NUMBER
IS
    v_cnt      INTEGER := 1;
    v_new_empno NUMBER;

```

```

BEGIN
  WHILE v_cnt > 0 LOOP
    SELECT next_empno.nextval INTO v_new_empno FROM dual;
    SELECT COUNT(*) INTO v_cnt FROM emp WHERE empno = v_new_empno;
  END LOOP;
  RETURN v_new_empno;
END;
/
--
-- EDB-SPL function that adds a new clerk to table 'emp'. This function
-- uses package 'emp_admin'.
--
CREATE OR REPLACE FUNCTION hire_clerk (
  p_ename          VARCHAR2,
  p_deptno         NUMBER
) RETURN NUMBER
IS
  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);
  SELECT empno, ename, job, mgr, hiredate, sal, comm, deptno INTO
    v_empno, v_ename, v_job, v_mgr, v_hiredate, v_sal, v_comm, v_deptno
  FROM emp WHERE empno = v_empno;
  DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
  DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
  DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
  DBMS_OUTPUT.PUT_LINE('Job       : ' || v_job);
  DBMS_OUTPUT.PUT_LINE('Manager   : ' || v_mgr);
  DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_hiredate);
  DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
  DBMS_OUTPUT.PUT_LINE('Commission : ' || v_comm);
  RETURN v_empno;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
    DBMS_OUTPUT.PUT_LINE(SQLERRM);
    DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
    DBMS_OUTPUT.PUT_LINE(SQLCODE);
    RETURN -1;
END;
/
--
-- PostgreSQL PL/pgSQL 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: ';
        RAISE INFO '%', SQLERRM;
        RAISE INFO 'The following is SQLSTATE: ';
        RAISE INFO '%', 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 TRIGGER user_audit_trig

```

```

    AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE
    v_action          VARCHAR2(24);
BEGIN
    IF INSERTING THEN
        v_action := ' added employee(s) on ';
    ELSIF UPDATING THEN
        v_action := ' updated employee(s) on ';
    ELSIF DELETING THEN
        v_action := ' deleted employee(s) on ';
    END IF;
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
TO_CHAR(SYSDATE, 'YYYY-MM-DD'));
END;
/
--
-- 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 TRIGGER emp_sal_trig
    BEFORE DELETE OR INSERT OR UPDATE ON emp
    FOR EACH ROW
DECLARE
    sal_diff          NUMBER;
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
    END IF;
    IF UPDATING THEN
        sal_diff := :NEW.sal - :OLD.sal;
        DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
        DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
        DBMS_OUTPUT.PUT_LINE('..Raise      : ' || sal_diff);
    END IF;
    IF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
        DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
    END IF;
END;
/
--
-- Package specification for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE emp_admin
IS
    FUNCTION get_dept_name (
        p_deptno          NUMBER
    ) RETURN VARCHAR2;
    FUNCTION update_emp_sal (
        p_empno           NUMBER,
        p_raise           NUMBER
    ) RETURN NUMBER;
    PROCEDURE hire_emp (
        p_empno           NUMBER,
        p_ename           VARCHAR2,
        p_job             VARCHAR2,
        p_sal             NUMBER,
        p_hiredate        DATE,
        p_comm            NUMBER,
        p_mgr             NUMBER,

```

```

        p_deptno          NUMBER
    );
    PROCEDURE fire_emp (
        p_empno           NUMBER
    );
END emp_admin;
/
--
-- Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
    --
    -- Function that queries the 'dept' table based on the department
    -- number and returns the corresponding department name.
    --
    FUNCTION get_dept_name (
        p_deptno          IN NUMBER
    ) RETURN VARCHAR2
    IS
        v_dname           VARCHAR2(14);
    BEGIN
        SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
        RETURN v_dname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
            RETURN '';
    END;
    --
    -- Function that updates an employee's salary based on the
    -- employee number and salary increment/decrement passed
    -- as IN parameters. Upon successful completion the function
    -- returns the new updated salary.
    --
    FUNCTION update_emp_sal (
        p_empno           IN NUMBER,
        p_raise           IN NUMBER
    ) RETURN NUMBER
    IS
        v_sal             NUMBER := 0;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
        v_sal := v_sal + p_raise;
        UPDATE emp SET sal = v_sal WHERE empno = p_empno;
        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
            RETURN -1;
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
            DBMS_OUTPUT.PUT_LINE(SQLERRM);
            DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
            DBMS_OUTPUT.PUT_LINE(SQLCODE);
            RETURN -1;
    END;
    --
    -- Procedure that inserts a new employee record into the 'emp' table.
    --
    PROCEDURE hire_emp (
        p_empno           NUMBER,
        p_ename           VARCHAR2,

```

```
    p_job          VARCHAR2,  
    p_sal          NUMBER,  
    p_hiredate     DATE,  
    p_comm         NUMBER,  
    p_mgr          NUMBER,  
    p_deptno      NUMBER  
  )  
AS  
BEGIN  
  INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)  
    VALUES(p_empno, p_ename, p_job, p_sal,  
           p_hiredate, p_comm, p_mgr, p_deptno);  
END;  
--  
-- Procedure that deletes an employee record from the 'emp' table based  
-- on the employee number.  
--  
PROCEDURE fire_emp (  
  p_empno          NUMBER  
)  
AS  
BEGIN  
  DELETE FROM emp WHERE empno = p_empno;  
END;  
END;  
/  
COMMIT;
```

## 2 Database Administration

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

### 2.1 Database Management

The DBA Management Server allows database administrators to monitor real-time database performance, view statistics, and to identify configuration issues for an unlimited number of Postgres Plus and/or Postgres Plus Advanced Server databases. Reports can be generated in both HTML and PDF formats.

**Note: Both the DBA Management Server and the DBA Monitoring Console are assigned deprecated status in Postgres Plus Advanced Server version 8.4. The DBA Management Server and the DBA Monitoring Console will be removed from the product set in a future release. You are encouraged to use Postgres Plus HQ, pgAdmin, and the StackBuilder Plus Update Manager to replace the functionality available in the DBA Management Server and the DBA Monitoring Console.**

#### 2.1.1 Logging into DBA Management Server

When the DBA Management Server is launched, you are prompted with a login screen as shown below. Once your username is authenticated the DBA Management Server home page will appear.

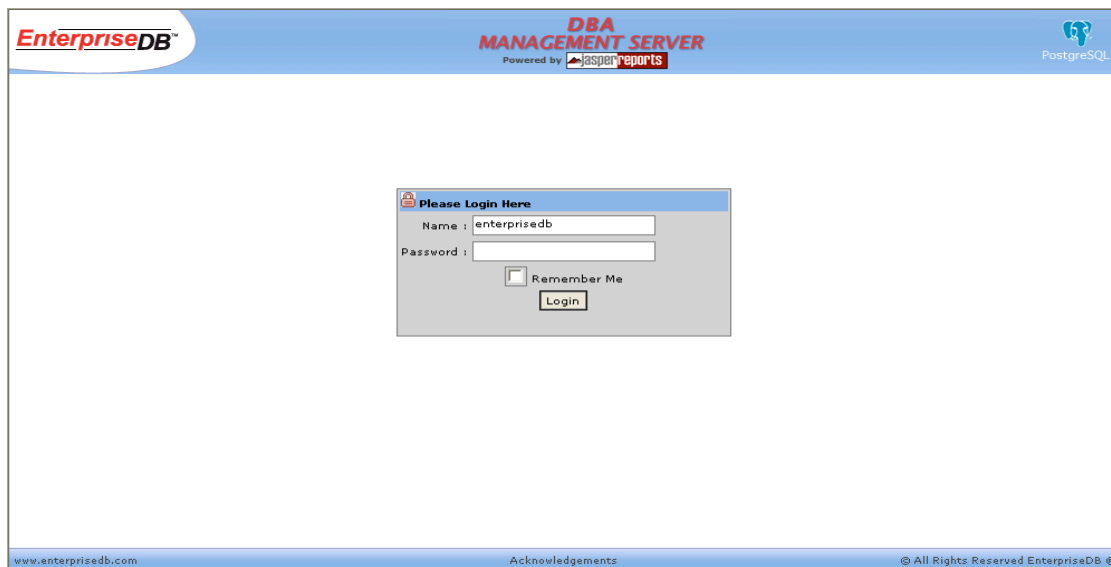


Figure 1 DBA Management Server - Login

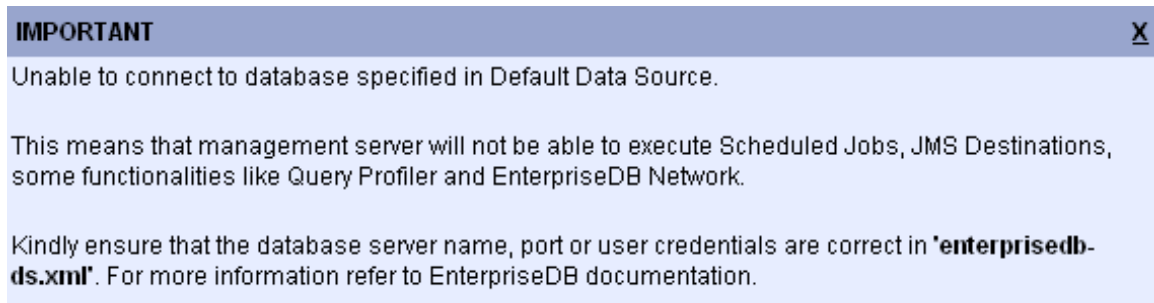
**Note:** In case of Internet Explorer, there may be a problem in redirection from `https://server:9363/edb-mgmtsvr/login.jsp` to `http://server:9000/edb-mgmtsvr/source_stats.do`

In this situation, the user may be redirected to `http://server:9363/edb-mgmtsvr/source_stats.do` instead of `http://server:9000/edb-mgmtsvr/source_stats.do`

In case this happens, update Internet Explorer using the Internet Explorer menu **Tools** and choose the **WindowsUpdate** option from the menu, and then restart Internet Explorer. The user will then be correctly redirected to `http://server:9000/edb-mgmtsvr/source_stats.do`

Only a valid superuser username and password can gain access to the DBA Management Server.

Upon login, the system checks for the availability of a connection configured against the *Default Data Source*. This information is maintained in the `enterisedb-ds.xml` file which can be found in the `mgmtsvr/server/default/deploy` subdirectory of the Postgres Plus Advanced Server home directory. In case a connection cannot be made due to some reason such as a bad username/password combination or connection timeout, the following error message is shown.



**Figure 2 Default Data Source Connection Error**

This dialog box can be closed by clicking the X symbol in the upper right corner; however, this dialog box will be displayed upon each user login until the `enterisedb-ds.xml` file is updated.

The following is an example of the `enterisedb-ds.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ===== -->
<!-- Datasource config for EnterpriseDB -->
-->
<!-- ===== -->

<datasources>
```

```
<local-tx-datasource>
  <jndi-name>DefaultDS</jndi-name>

  <!-- Connection using EnterpriseDB JDBC Driver -->
  <connection-url>jdbc:edb://127.0.0.1:5445/mgmtsvr</connection-url>
  <driver-class>com.edb.Driver</driver-class>

  <!-- Connection using PostgreSQL JDBC Driver -->
  <!--
  <connection-url>jdbc:postgresql://127.0.0.1:#port/#db</connection-url>
  <driver-class>org.postgresql.Driver</driver-class>
  -->

  <user-name>enterprisedb</user-name>
  <password>password</password>

  <!-- The minimum connections in a pool/sub-pool. Pools are lazily
  constructed on first use -->
  <min-pool-size>1</min-pool-size>

  <!-- The maximum connections in a pool/sub-pool -->
  <max-pool-size>20</max-pool-size>

  <!-- The time before an unused connection is destroyed -->
  <!-- NOTE: This is the check period. It will be destroyed somewhere
  between 1x and 2x this timeout after last use -->
  <idle-timeout-minutes>0</idle-timeout-minutes>

  <!-- corresponding type-mapping in the standardjbosscomp-jdbc.xml
  (optional) -->
  <metadata>
    <type-mapping>PostgreSQL 8.0</type-mapping>
  </metadata>

</local-tx-datasource>

</datasources>
```

### 2.1.2 Database Read/Write Activity Report

Database read/write activity is graphically represented by the bar charts on the DBA Management Server home page.

- The **Transactions** bar chart shows the number of transactions committed.
- The **Memory Reads** bar chart displays the number of cache block hits.
- The **Disk Reads** bar chart displays the number of physical disk blocks read.

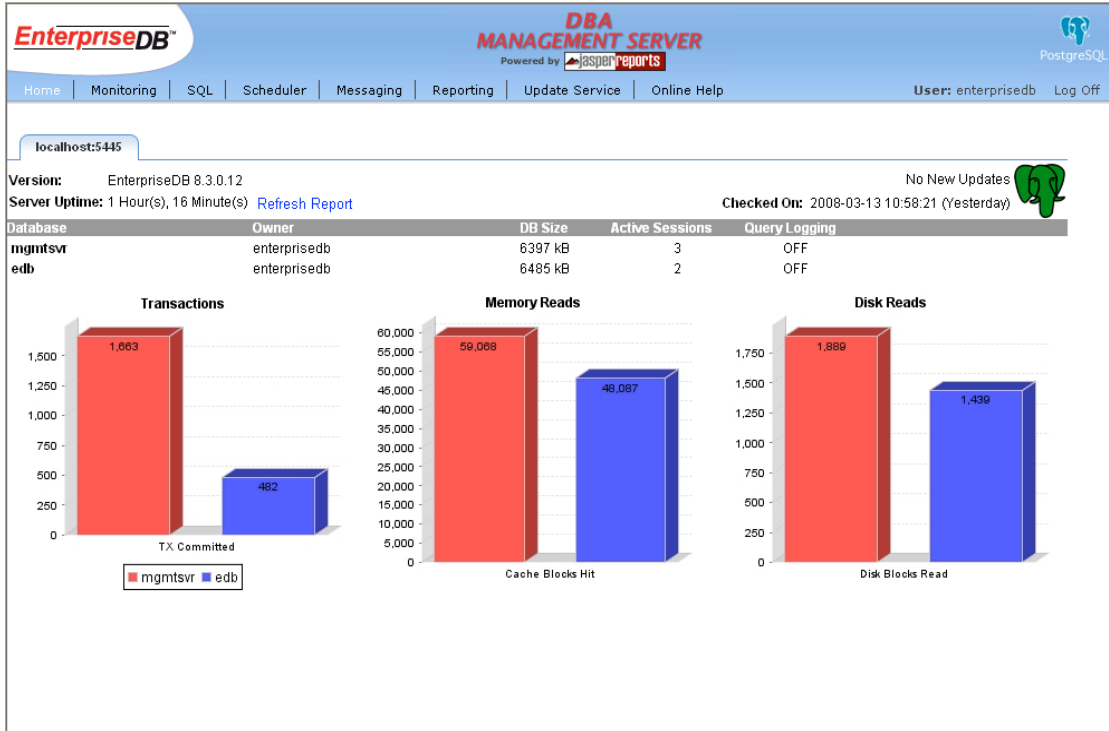


Figure 3 Database Read/Write Activity

In addition to the database read/write activity, the home page also contains a table summarizing the following information on each row for each configured database.

- **Database** - Database name in the format `hostname:port/database`
- **Version** - The database's major and minor version number
- **Owner** - The name of the database owner
- **DB Size** - Size of the database using KB, MB, GB, or TB as appropriate
- **Active Sessions** - The number of active sessions
- **Server Uptime** - The amount of time since the database server has been started
- **Query Logging** - Shows if **Query Logging** is currently on or off for a configured database. The entire row is displayed in red if **Query Logging** is on, or black if **Query Logging** is off. For more details on enabling/disabling **Query Logging** refer to the **Query Profiler**.

The **DB Size** and **Server Uptime** columns are only available with EnterpriseDB and PostgreSQL versions 8.1 and above.

If a server is not available, the tab for that server will appear greyed out with a red cross on it as shown in the following figure.

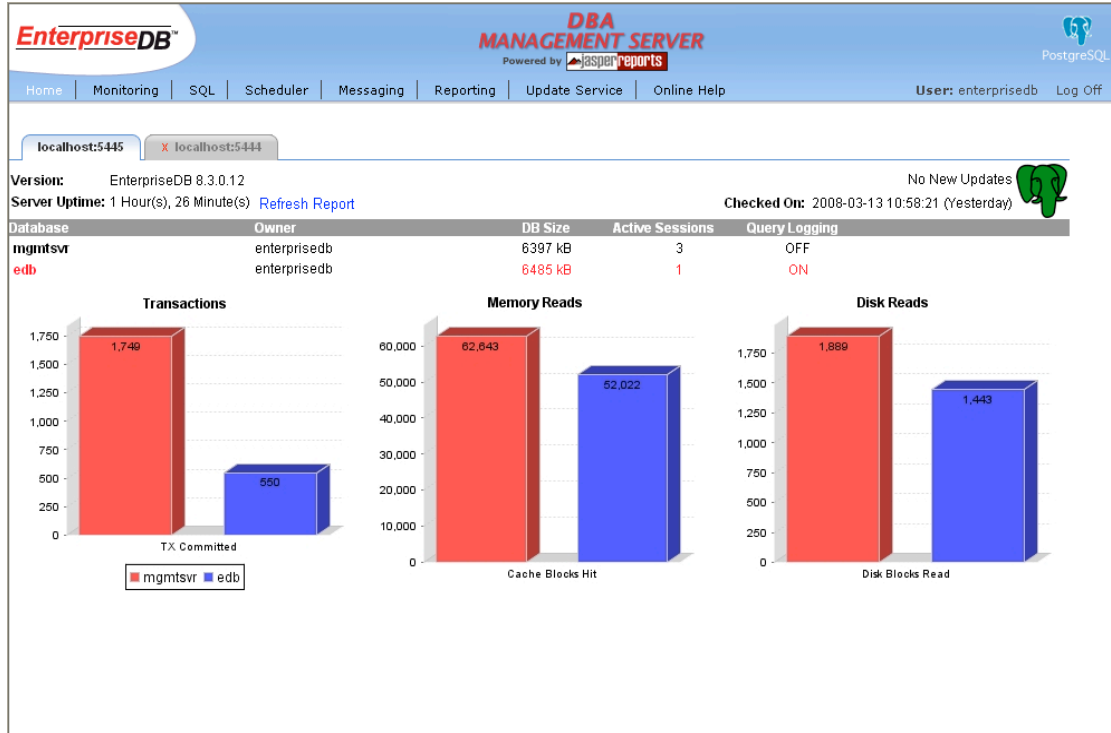


Figure 4 Unavailable Server

The report can be refreshed by clicking on the **Refresh Report** link.

When multiple data sources are configured, each is assigned a different color to differentiate each source. If a data source is not available or one of the connection parameters defined in the `configure.xml` file is incorrect, an appropriate error message is displayed.

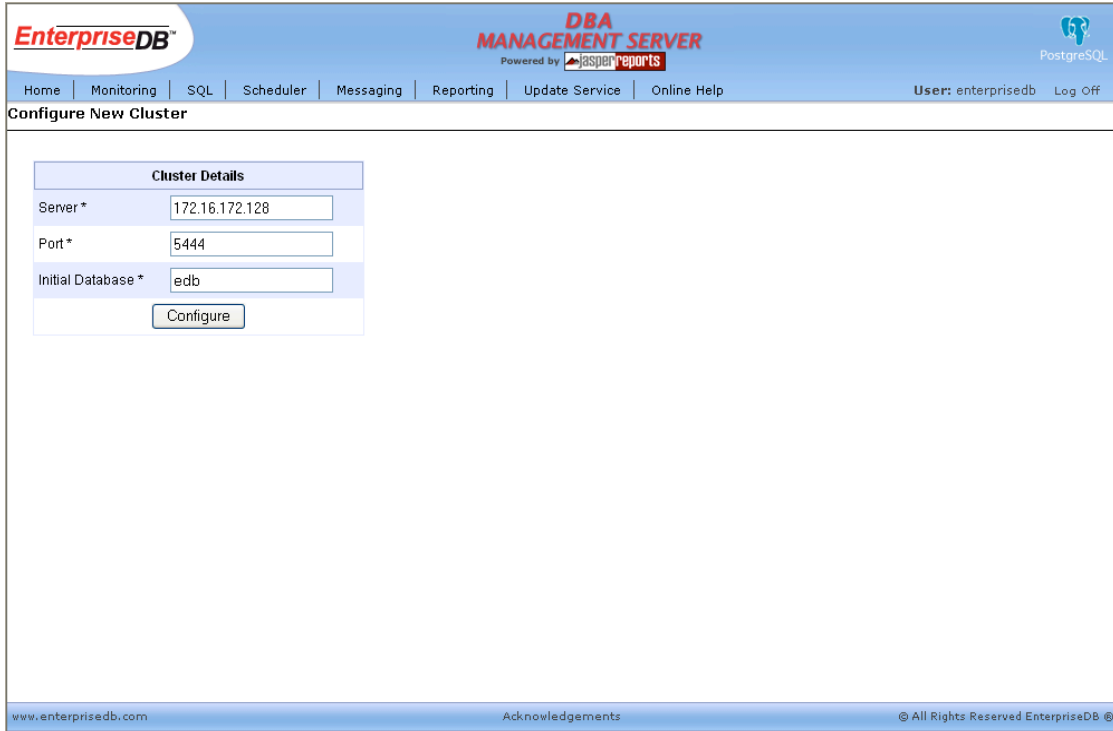
### 2.1.3 Add/Remove Cluster

DBA Management Server can monitor various databases at once. These databases can be present on the local or a remote machine. To monitor a new database cluster, select the **Add/Remove Cluster** option under the home menu. Clicking on **Add/Remove Cluster** will open the following **Configured Clusters** report:



**Figure 5 Configured Clusters**

To add a new cluster, click on the **Configure New Cluster** option. This will open the following report.



**Figure 6 Configure New Cluster**

Provide the new server IP address, port number, and database name in the **Server**, **Port** and **Initial Database** text fields. Click on the **Configure** button. The new database cluster is now added to DBA Management Server and it will be visible on the **Configured Clusters** report.



Figure 7 New Configured Cluster

The new database will be available on the home page in form of a new tab.

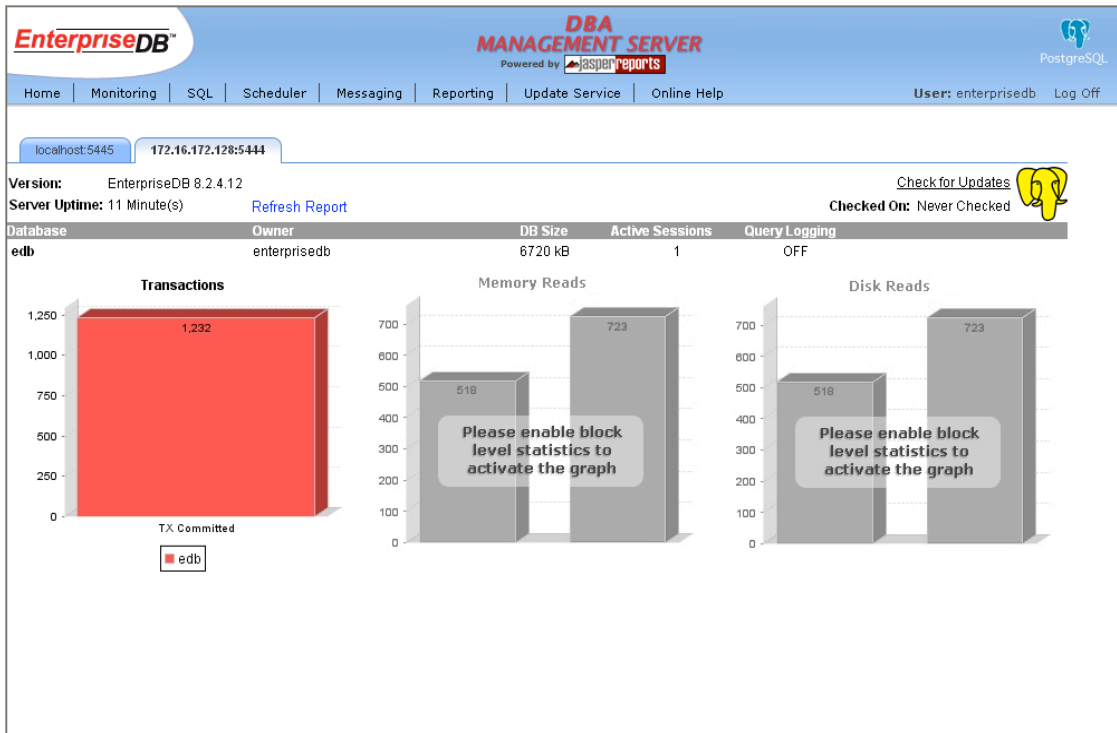


Figure 8 Read/Write Activity for New Database

## 2.1.4 Monitoring

The following reports are available for monitoring database activity:

- User Activity – Lists connected users and active SQL commands in execution
- Lock Status – Shows locks held on tables and indexes by users and what type of locks are held
- Buffer Cache – Shows count of table and index pages held in the buffer cache
- Configuration – Displays and allows for updating of database parameter settings for statistics collection
- View DB Logs – View the database server logs
- View Audit Logs – View the database server audit logs

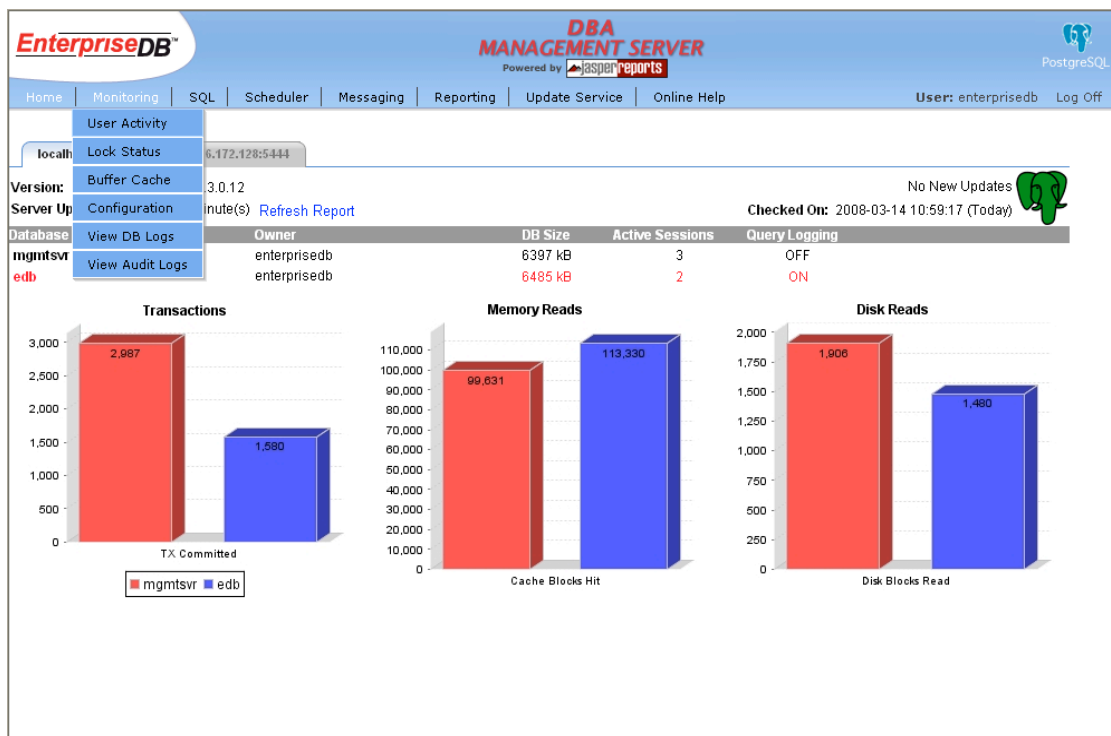


Figure 9 DBA Management Server - Monitoring

### 2.1.4.1 User Activity

This report contains information about current user activity within a specific database. This is primarily the number of active sessions associated with the database. The **User Activity** report contains the process ID, username, the time the current command was initiated, and the command being issued by that particular session.

**Note:** The current command appears in the report only if the `stats_command_string` parameter for the database is set to “true”. This parameter can be viewed and set in the Configuration report.

This report provides a real-time view of database activity. Keep in mind that there is a delay of approximately ten seconds between actual execution of the query and when it is reported in the database catalog. Hence this view lags the actual database activity by about ten seconds.

Ten seconds is the default refresh interval for this report. This default can be changed to any value between 5 - 99 seconds by typing in the desired refresh period in the **Refresh Interval** textbox.



Figure 10 User Activity

### 2.1.4.2 Lock Status

This report provides access to information about the locks held by open transactions within the database server. Each row shows an active lockable object, lock mode, the username, the transaction number, and the SQL command holding the lock. The same lockable object may appear multiple times if multiple transactions are holding or waiting against that object.

This report can be particularly useful for detecting deadlocks; however, there is a delay of approximately ten seconds between actual execution of the SQL command and when it is reported in the database catalog. Hence this view lags the actual database activity by about ten seconds.

Ten seconds is the default refresh interval for this report, this default can be changed to any value between 5-99 seconds by typing in the desired refresh period in the **Refresh Interval** textbox.

The screenshot shows the EnterpriseDB DBA Management Server interface. At the top, there is a navigation menu with options: Home, Monitoring, SQL, Scheduler, Messaging, Reporting, Update Service, and Online Help. The user is identified as 'enterprisedb' and can log off. The main section is titled 'Lock Status' and contains a form with a database dropdown set to 'localhost5445/edb', a 'Refresh Report' button, a 'Refresh Interval (in sec.)' input set to '10' (with a maximum of 99), and a checked 'Auto Refresh' checkbox. Below the form is a table titled 'LOCK STATUS' for the database 'localhost:5445/edb', with a report time of '03/14/2008 04:42 PM'. The table has columns for PID, Relation, User Name, TX, Mode, and Query.

PID	Relation	User Name	TX	Mode	Query
1940	pg_locks	enterprisedb	6062	AccessShareLock	SELECT current_timestamp as report_time, (SELECT datname FROM
2752	emp	mary	5932	RowExclusiveLock	UPDATE emp SET job = 'SALESMAN' WHERE empno = 9001;
2752	emp_pk	mary	5932	RowExclusiveLock	UPDATE emp SET job = 'SALESMAN' WHERE empno = 9001;
2752	emp	mary	5932	ExclusiveLock	UPDATE emp SET job = 'SALESMAN' WHERE empno = 9001;
3200	emp	john	5917	RowExclusiveLock	<IDLE> in transaction
3200	emp_pk	john	5917	RowExclusiveLock	<IDLE> in transaction
3656	emp_pk	enterprisedb	5961	RowExclusiveLock	UPDATE emp SET mgr = 7900 WHERE empno = 9001;
3656	emp	enterprisedb	5961	RowExclusiveLock	UPDATE emp SET mgr = 7900 WHERE empno = 9001;
3656	emp	enterprisedb	5961	ExclusiveLock	UPDATE emp SET mgr = 7900 WHERE empno = 9001;

**Figure 11 Lock Status**

**Note:** This report locks the `pg_locks` catalog table while it generates the output. Avoid refreshing this report too frequently as it will result in database performance degradation.

### 2.1.4.3 Buffer Cache

This report shows information about real-time queries on the shared buffer cache. Each row in this report shows the number of pages and amount of memory each relation or index is taking from the shared buffer cache.

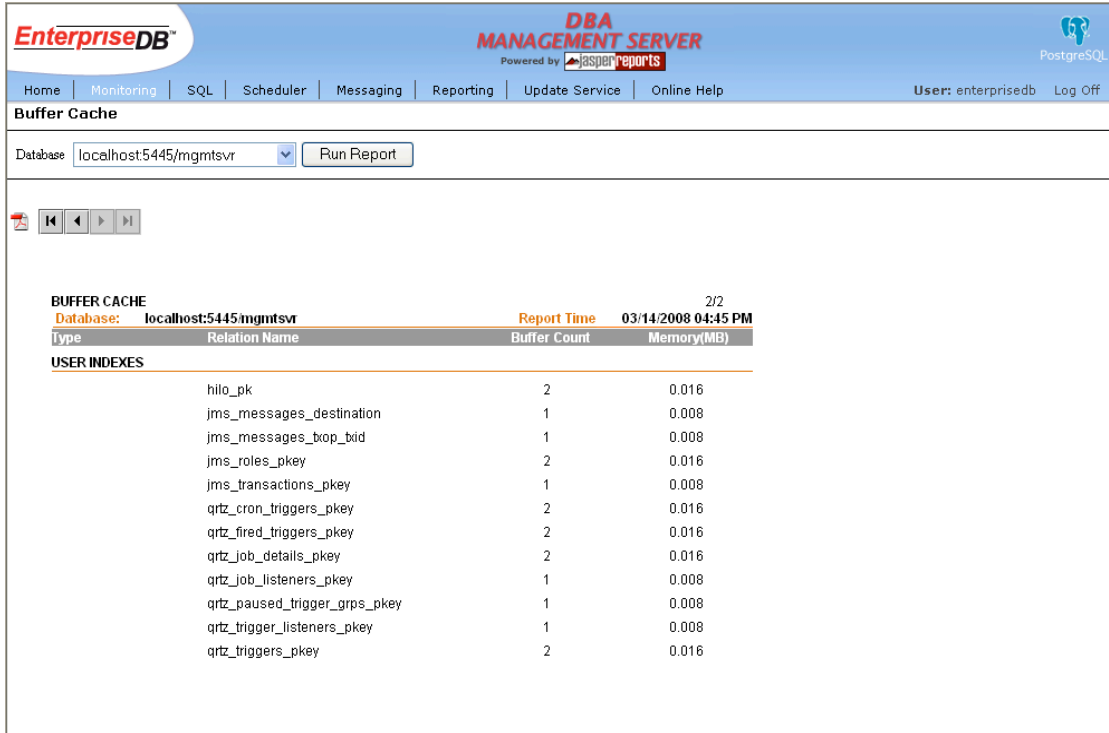


Figure 12 Buffer Cache

**Note:** This report locks the `pg_buffercache` catalog table while it generates the output. Avoid refreshing this report too frequently as it will result in database performance degradation.

### 2.1.4.4 Configuration

The parameters shown on this page are the configuration parameters that control the collection of runtime statistics for the database selected at the top of the report.

The message in red (as seen in the following figure) shows that the `readOnly` element of database `mgmtsvr` is set to “true” in the `configure.xml` file found in the `mgmtsvr/server/default/deploy/edb-mgmtsvr.war/WEB-INF` subdirectory of the Postgres Plus Advanced Server home directory.



Figure 13 Configuration – read-only mode

The following is the `configure.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<config>
  <source>
    <host>localhost</host>
    <port>5445</port>
    <database>edb</database>
    <readOnly>>false</readOnly>
  </source>
  <source>
    <host>localhost</host>
    <port>5445</port>
    <database>mgmtsvr</database>
    <readOnly>>true</readOnly>
  </source>
  <source>      <host>172.16.172.128</host>
    <port>5444</port>
    <database>edb</database>
    <readOnly>>false</readOnly>
  </source>
</config>

```

When the `readOnly` element is set to “false”, end users can freely change the status of all the parameters as shown in the following figure for database `edb`.



Figure 14 Configuration – writeable mode

The **stats\_reset\_on\_server\_start** and **stats\_start\_collector** check boxes are disabled as these parameters can be changed only at database server startup time.

To change any of the other parameter values, select the check-box and click on the **Apply** button.

The following is a description of each check box and the parameter it enables or disables for the database.

**stats\_block\_level**

This box must be checked to view the **Memory Reads** and **Disk Reads** bar charts on the home page. Furthermore, this box must be checked for accurately viewing all statistics related reports.

### **stats\_command\_string**

When this box is checked the active SQL command of any new SQL Interactive session is displayed in the User Activity report.

### **stats\_reset\_on\_server\_start**

If this box is checked and the database server is restarted, the following statistics are reset.

- **Transactions** bar chart
- **Memory Reads** bar chart
- **Disk Reads** bar chart
- **Table IO Detail** report
- **Index IO Detail** report

### **stats\_row\_level**

When this box is checked, statistics are collected and displayed on the following reports.

- **Table IO Detail** report
- **Index IO Detail** report

### **stats\_start\_collector**

When this box is checked the database server's *stats collector process* is started upon database server start. This process is responsible for collection of database statistics. This box should be checked if either or both of the **stats\_block\_level** or **stats\_row\_level** boxes are checked.

### **Reset Collected Stats**

When this checkbox is selected, the following report statistics are reset.

- **Transactions** bar chart
- **Memory Reads** bar chart
- **Disk Reads** bar chart
- **Table IO Detail** report
- **Index IO Detail** report

### 2.1.4.5 View DB Logs

This feature provides the ability to browse through the database server log files. The log files of remote as well as local database servers can be viewed.

In order to produce a database server log file, the following configuration parameters must be set in the `postgresql.conf` file prior to database server startup.

- `log_destination` must include `stderr`
- `redirect_stderr = on`
- `log_directory` must be set to a valid existing, destination directory
- `log_filename` must be set to a valid file name pattern

The **Log Name** drop down list contains all the logs for the selected database server. By default, the most recent log file is displayed in the window. In the event the log file exceeds 1 MB, the latest 1 MB of the file is shown.

To view a specific part of a log file, specify values for the following text fields.

- **Offset** – Offset in bytes from the beginning of the log to begin viewing.
- **Length** – Length of the log in bytes to view.

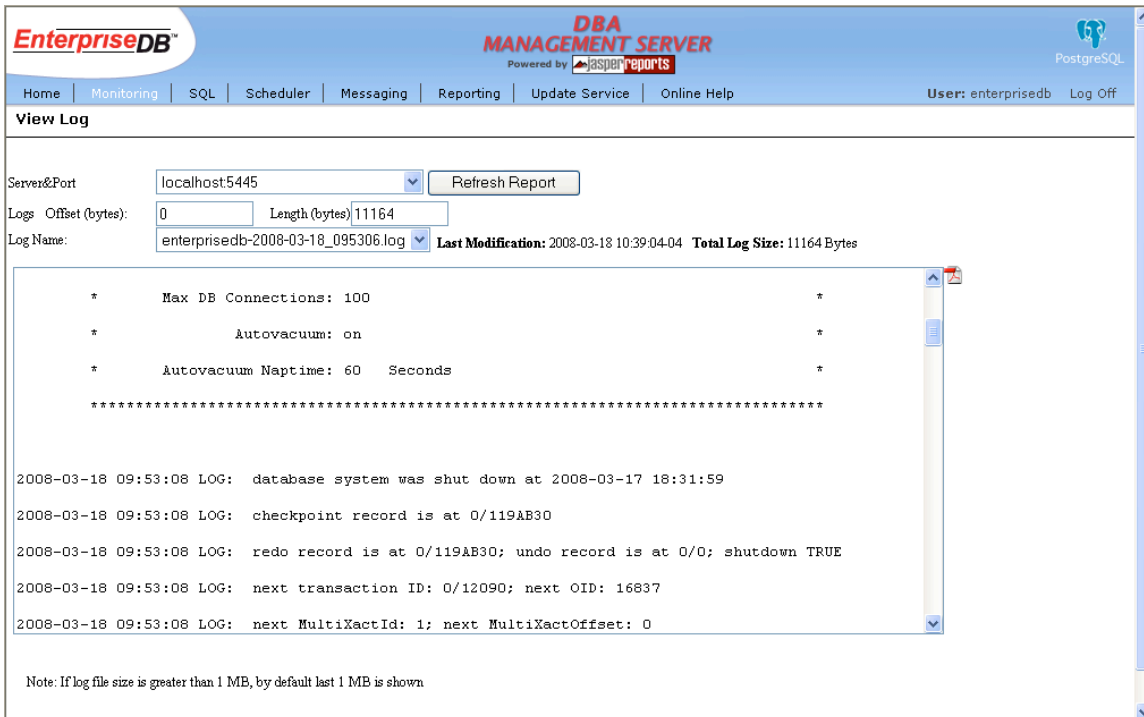


Figure 15 View DB Logs

### 2.1.4.6 View Audit Logs

See Section [2.2.2](#).

### 2.1.5 SQL

The **SQL** menu contains the following two selections:

- iQuery – SQL query executor
- Query Profiler – SQL logging and reporting

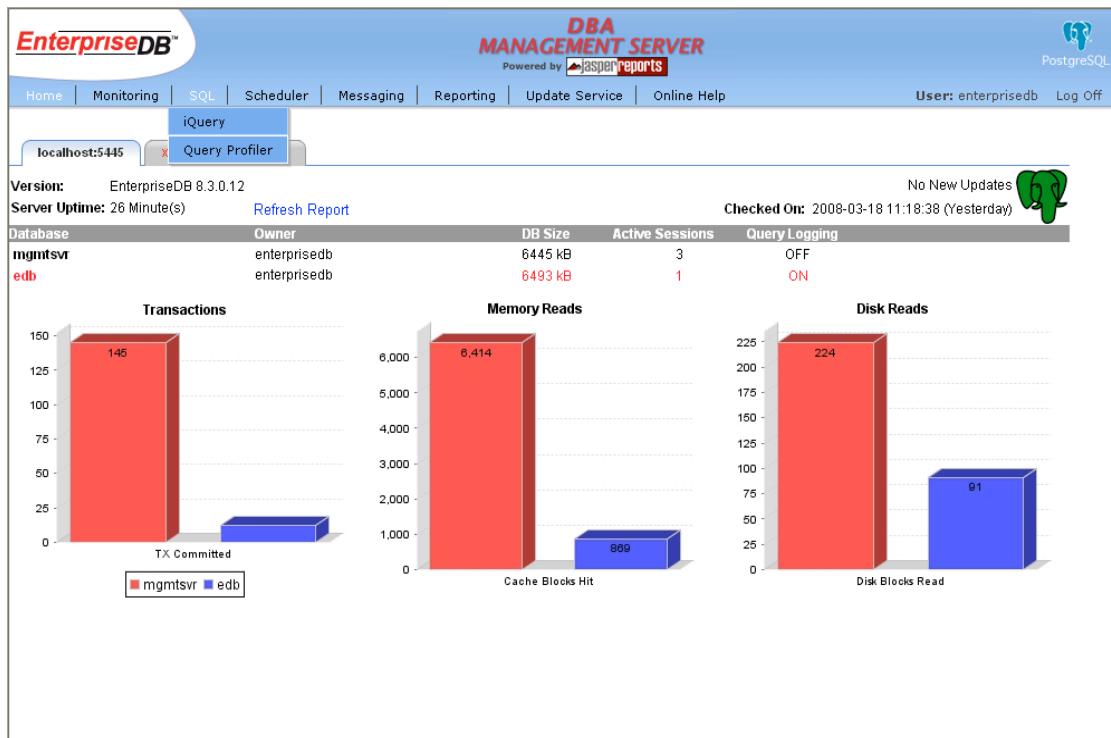


Figure 16 DBA Management Server - SQL

#### 2.1.5.1 iQuery

**iQuery** is a SQL query executor that can run one or more SQL commands, execute stored procedures, and run anonymous SPL blocks.

The database on which the commands are to be run is selected from the **Database** drop down list. The commands to be executed are entered in the **Query** text box. Click the **Submit** button to execute the commands.

The **Data Output** tab displays the results of any `SELECT` commands. The query results are limited to a maximum of 2500 records.

Runtime information such as execution time, error messages, and the output generated by SPL DBMS\_OUTPUT.PUT\_LINE statements or PL/pgSQL RAISE statements are displayed on the **Messages** tab.

The screenshot shows the EnterpriseDB iQuery interface. At the top, there is a navigation bar with links for Home, Monitoring, SQL, Scheduler, Messaging, Reporting, Update Service, and Online Help. The user is identified as 'enterprisedb'. The main area is titled 'iQuery' and contains a 'Database' dropdown set to 'localhost:5445/edb' and a 'Query' text area containing 'SELECT \* FROM emp'. A 'Submit' button is located below the query. Below the query area, there are two tabs: 'Data output' (selected) and 'Messages'. The 'Data output' tab displays a table with the following data:

empno	ename	job	mgr	hiredate	sal	comm	deptno
7369	SMITH	CLERK	7902	1980-12-17 00:00:00.0	800.00	null	20
7499	ALLEN	SALESMAN	7698	1981-02-20 00:00:00.0	1600.00	300.00	30
7521	WARD	SALESMAN	7698	1981-02-22 00:00:00.0	1250.00	500.00	30
7566	JONES	MANAGER	7839	1981-04-02 00:00:00.0	2975.00	null	20
7654	MARTIN	SALESMAN	7698	1981-09-28 00:00:00.0	1250.00	1400.00	30
7698	BLAKE	MANAGER	7839	1981-05-01 00:00:00.0	2850.00	null	30
7782	CLARK	MANAGER	7839	1981-06-09 00:00:00.0	2450.00	null	10
7788	SCOTT	ANALYST	7566	1987-04-19 00:00:00.0	3000.00	null	20
7839	KING	PRESIDENT	null	1981-11-17 00:00:00.0	5000.00	null	10
7844	TURNER	SALESMAN	7698	1981-09-08 00:00:00.0	1500.00	0.00	30
7876	ADAMS	CLERK	7788	1987-05-23 00:00:00.0	1100.00	null	20

Figure 17 iQuery Data Output

The following shows the **Messages** tab associated with the previous query.

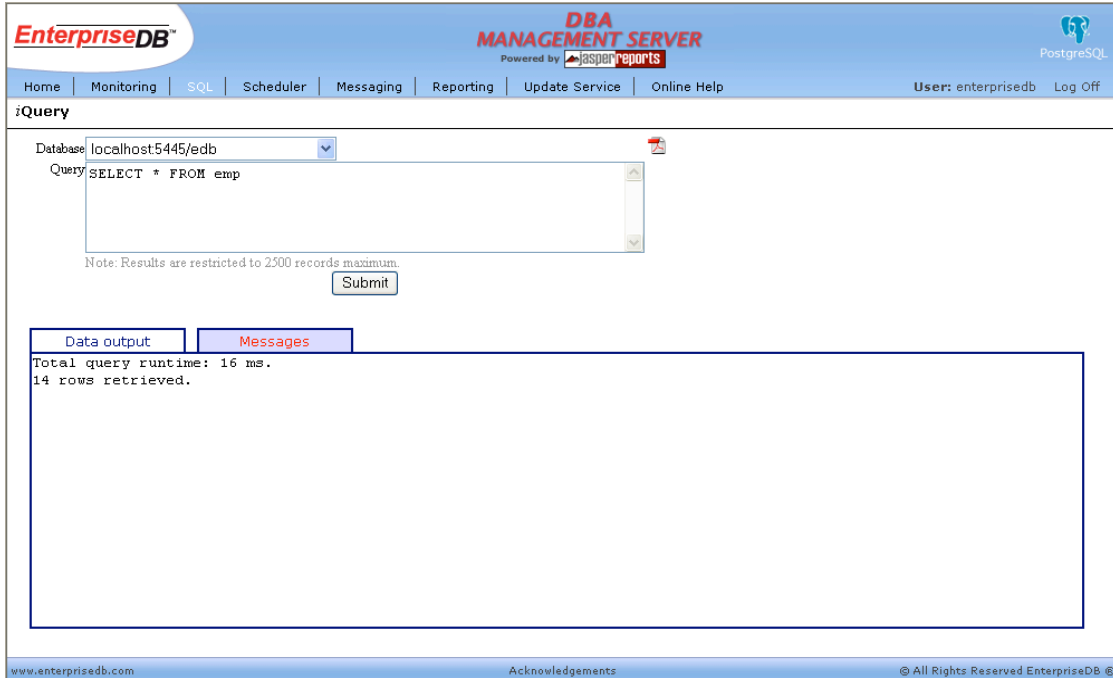


Figure 18 iQuery Messages

### 2.1.5.2 Query Profiler

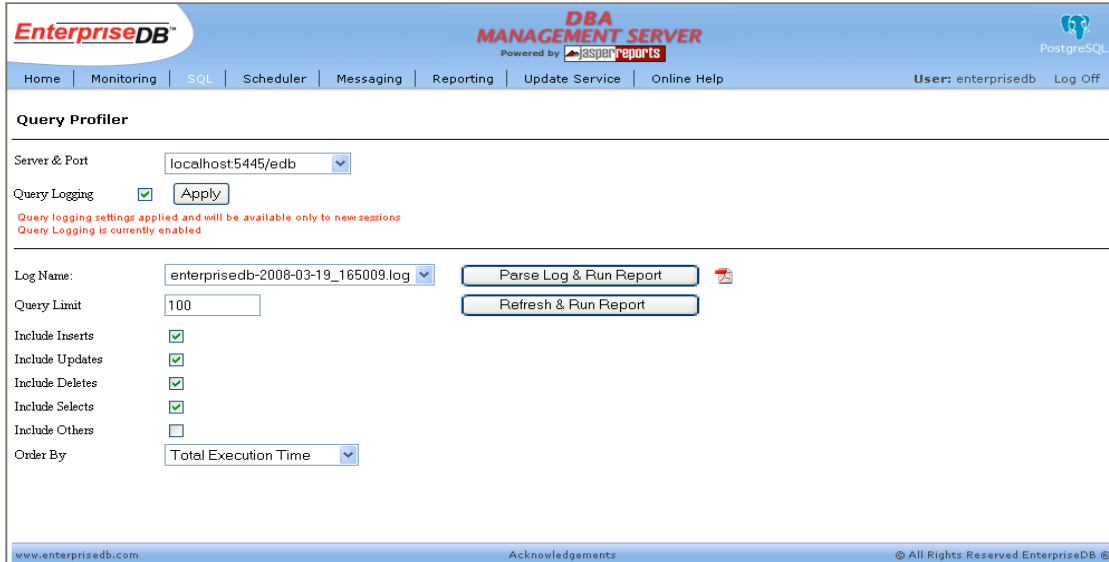
The Query Profiler parses the selected database server log file and generates a report of the SQL commands executed on that database. This can help database administrators improve performance of their systems. The information made available by Query Profiler can be utilized for two primary purposes:

- Track down the longest running SQL commands so they can be improved.
- Track down frequently used SQL commands so that these can be placed in a stored procedure for further performance optimization.

The `log_min_duration_statement` configuration parameter must be set to 0 to activate the Query Profiler for a selected database.

This can be achieved in either of the following two ways.

- At the top of the screen, check the **Query Logging** check box and click the **Apply** button as shown in the following figure (recommended method).



**Figure 19 Enabling Query Logging**

- In the `postgresql.conf` file set the `log_min_duration_statement` configuration parameter to 0 and reload the configuration file or restart the database server.

**Note:** This latter method will turn on query logging for all databases in the database cluster which is not recommended as performance on all databases will be impacted.

**Note:** Enabling Query Logging has performance implications; hence, it should only be used to sample queries for a short period of time to gather relevant information and then be turned off.

The resulting list of SQL commands can be ordered on the basis of **Total Execution Time**, **Average Execution Time**, or **Statement Count** (the number of times a given SQL command was repeated) by selecting the desired ranking from the **Order By** drop down list.

The types of SQL commands that appear can be filtered by checking the appropriate boxes:

- **Include Inserts**
- **Include Updates**
- **Include Deletes**
- **Include Selects**
- **Include Others**

Clicking the **Refresh & Run Report** button does not re-parse the log file over again, but just filters statements based upon the current filtering criteria using the data from the last time the log was parsed.

Clicking the **Parse Log & Run Report** button will re-parse the log file over again. This is recommended if there is a lot of database activity and the log file has not been re-parsed in a while.

The screenshot shows the 'Query Profiler' section of the EnterpriseDB DBA Management Server. It includes a navigation bar with links like Home, Monitoring, SQL, Scheduler, Messaging, Reporting, Update Service, and Online Help. The user is identified as 'enterprisedb'. The main area is titled 'Query Profiler' and contains several configuration options:
 

- Server & Port: localhost:5445/edb
- Query Logging:  (Apply button)
- Log Name: enterprisedb-2008-03-19\_174357.log (Parse Log & Run Report button)
- Query Limit: 100 (Refresh & Run Report button)
- Include Inserts:
- Include Updates:
- Include Deletes:
- Include Selects:
- Include Others:
- Order By: Total Execution Time

 Below the configuration is a 'Results' table with the following data:
 

Rank	Count	Total Execution Time(ms)	Average Execution Time(ms)	Query
1	1	828	828	INSERT INTO emp (empno,ename,deptno) VALUES (9001,'JONES',40);
2	11	266	24.182	select version();
3	1	62	62	UPDATE emp SET sal = 9500.00, comm = 1000.00 WHERE deptno = 40;
4	1	30.999	30.999	INSERT INTO emp (empno,ename,deptno) VALUES (9003,'ROGERS',40);
5	1	0	0	INSERT INTO emp (empno,ename,deptno) VALUES (9002,'PETERSON',40);
6	1	0	0	SELECT * FROM emp WHERE deptno = 40;

Figure 20 Query Profiler

## 2.2 Database Auditing

Postgres Plus Advanced Server provides the capability to produce audit reports. Database auditing allows database administrators, security administrators, auditors, and operators to track and analyze database activities. These activities include database access and usage along with data creation, change, or deletion. The auditing system is based on the configuration parameters defined in the `postgresql.conf` file.

Using the DBA Management Server, these audit reports can be generated and viewed; however, the audit settings in the `postgresql.conf` file must first be enabled.

## 2.2.1 Auditing Configuration Parameters

The following is a description of the configuration parameters that control database auditing. These parameters are found in the `postgresql.conf` file.

`edb_audit`

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

`edb_audit_directory`

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

`edb_audit_filename`

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

`edb_audit_rotation_day`

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

`edb_audit_rotation_size`

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

`edb_audit_rotation_seconds`

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

`edb_audit_connect`

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

`edb_audit_disconnect`

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

`edb_audit_statement`

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

Suppose we need to audit all connections, disconnections, DDL statements and statements resulting in an error. The audit file is to be rotated every Sunday.

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

Each audit line is preceded with a fixed prefix that cannot be changed. The prefix consists of user name, database name, remote host and port, process id, session id, transaction id, timestamp, and event type.

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

### CSV Audit Logfile

```

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

```

### XML Audit Logfile

```

<event process_id="2516" time="2008-03-13 13:22:42 " type="startup">
  <message>AUDIT:  database system is ready</message>
</event>
<event user="enterprisedb" database="mgmtsvr" remote_host_and_port="127.0.0.1(1281)"
process_id="352" session_id="47d96338.160" transaction="0"
time="2008-03-13 13:24:08 " type="connect">
  <message>AUDIT:  connection authorized: user=enterprisedb
database=mgmtsvr</message>
</event>
<event user="enterprisedb" database="mgmtsvr" remote_host_and_port="127.0.0.1(1281)"
process_id="352" session_id="47d96338.160" transaction="1635"
time="2008-03-13 13:24:10 " type="ddl">
  <command>AUDIT:  statement: drop table HILOSEQUENCES</command>
</event>
<event user="enterprisedb" database="mgmtsvr" remote_host_and_port="127.0.0.1(1281)"
process_id="352" session_id="47d96338.160" transaction="1637"
time="2008-03-13 13:24:10 " type="ddl">
  <command>AUDIT:  statement: create table HILOSEQUENCES (
        SEQUENCENAME varchar(50) not null,
        HIGHVALUES integer not null,
        constraint hilo_pk primary key (SEQUENCENAME)
) </command>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"

```

```
process_id="3776" session_id="47d96378.ec0" transaction="0"
time="2008-03-13 13:25:12" type="connect">
<message>AUDIT: connection authorized: user=enterprisedb database=edb</message>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
process_id="3776" session_id="47d96378.ec0" transaction="1667"
time="2008-03-13 13:25:17" type="ddl">
<command>AUDIT: statement: CREATE TABLE test (f1 INTEGER);</command>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
process_id="3776" session_id="47d96378.ec0" transaction="1669"
time="2008-03-13 13:25:17" type="sql statement">
<command>AUDIT: statement: SELECT * FROM testx;</command>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
process_id="3776" session_id="47d96378.ec0" transaction="1669"
time="2008-03-13 13:25:17" type="error">
<message>ERROR: relation &quot;testx&quot; does not exist</message>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
process_id="3776" session_id="47d96378.ec0" transaction="1670"
time="2008-03-13 13:25:18" type="ddl">
<command>AUDIT: statement: DROP TABLE test;</command>
</event>
<event user="enterprisedb" database="edb" remote_host_and_port="127.0.0.1(1283)"
process_id="3776" session_id="47d96378.ec0" transaction="0"
time="2008-03-13 13:25:22" type="disconnect">
<message>AUDIT: disconnection: session time: 0:00:10.094 user=enterprisedb
database=edb host=127.0.0.1 port=1283</message>
</event>
<event user="enterprisedb" database="mgmtsvr" remote_host_and_port="127.0.0.1(1281)"
process_id="352" session_id="47d96338.160" transaction="0"
time="2008-03-13 13:25:31" type="disconnect">
<message>AUDIT: disconnection: session time: 0:01:23.046 user=enterprisedb
database=mgmtsvr host=127.0.0.1 port=1281</message>
</event>
<event process_id="2768" time="2008-03-13 13:25:36" type="shutdown">
<message>AUDIT: database system is shut down</message>
</event>
```

### 2.2.2 View Audit Logs

In the DBA Management Server, select the **View Audit Logs** option from the **Monitoring** menu to see auditing information.

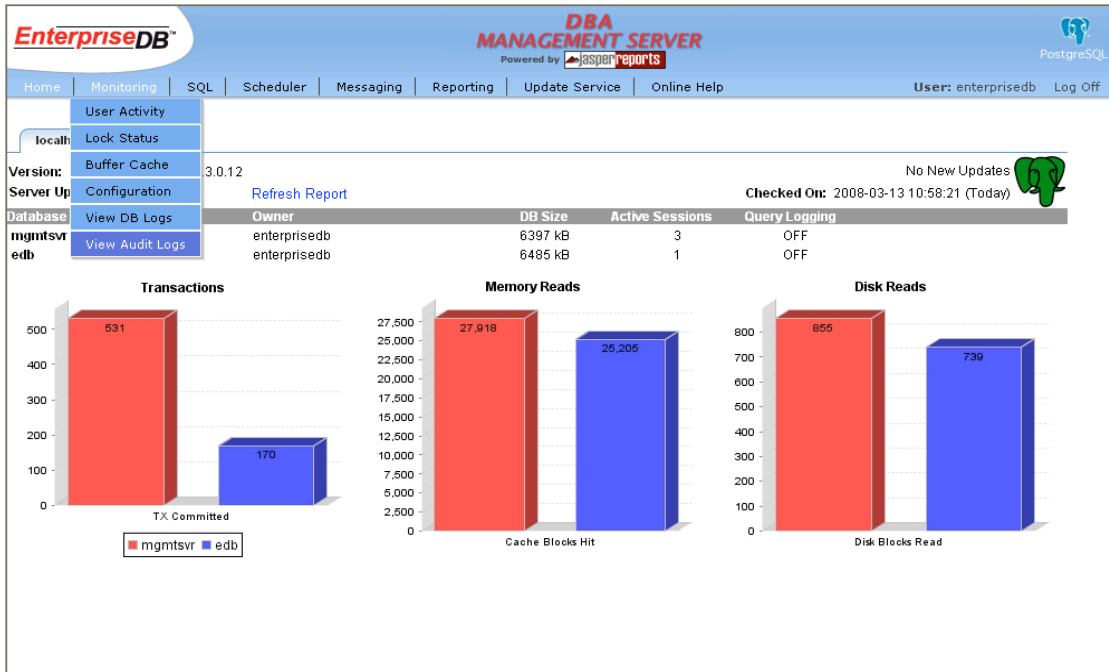


Figure 21 View Audit Logs

In the **View Audit** report, the audit options selected in the `postgresql.conf` file are shown on the left-hand side of the screen. The checkboxes in the middle of the screen provide for further screening of the types of statements that are to be shown in the report.

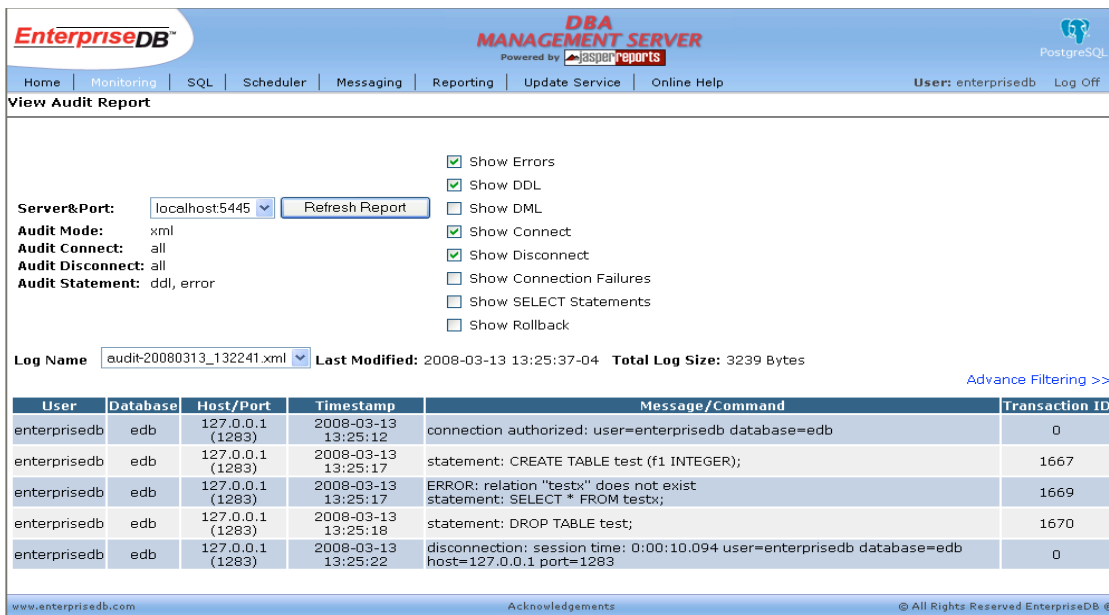


Figure 22 View Audit

The audited statements were generated by the following session in PSQL.

```

edb=# CREATE TABLE test (f1 INTEGER);
CREATE TABLE
edb=# INSERT INTO test VALUES (100);
INSERT 0 1
edb=# SELECT * FROM testx;
ERROR:  relation "testx" does not exist
edb=# DROP TABLE test;
DROP TABLE

```

Note that the successful `INSERT` command does not appear in the audit list since `edb_audit_statement` was not set to include successful DML commands.

## 2.3 High Availability and Load Balancing

Database servers can work together to allow a second server to take over quickly if the primary server fails (high availability), or to allow several computers to serve the same data (load balancing). Ideally, database servers could work together seamlessly. Web servers serving static web pages can be combined quite easily by merely load-balancing web requests to multiple machines. In fact, read-only database servers can be combined relatively easily, too. Unfortunately, most database servers have a read/write mix of requests, and read/write servers are much harder to combine. This is because though read-only data needs to be placed on each server only once, a write to any server has to be propagated to all servers so that future read requests to those servers return consistent results.

This synchronization problem is the fundamental difficulty for servers working together. Because there is no single solution that eliminates the impact of the sync problem for all use cases, there are multiple solutions. Each solution addresses this problem in a different way, and minimizes its impact for a specific workload.

Some solutions deal with synchronization by allowing only one server to modify the data. Servers that can modify data are called read/write or *master* servers. Servers that can reply to read-only queries are called *slave* servers. Servers that cannot be accessed until they are changed to master servers are called *standby* servers.

Some failover and load balancing solutions are synchronous, meaning that a data-modifying transaction is not considered committed until all servers have committed the transaction. This guarantees that a failover will not lose any data and that all load-balanced servers will return consistent results no matter which server is queried. In contrast, asynchronous solutions allow some delay between the time of a commit and its propagation to the other servers, opening the possibility that some transactions might be lost in the switch to a backup server, and that load balanced servers might return slightly stale results. Asynchronous communication is used when synchronous would be too slow.

Solutions can also be categorized by their granularity. Some solutions can deal only with an entire database server, while others allow control at the per-table or per-database level.

Performance must be considered in any failover or load balancing choice. There is usually a tradeoff between functionality and performance. For example, a full synchronous solution over a slow network might cut performance by more than half, while an asynchronous one might have a minimal performance impact.

The rest of this chapter outlines various failover, replication, and load balancing solutions.

### 2.3.1 Shared Disk Failover

*Shared disk failover* avoids synchronization overhead by having only one copy of the database. It uses a single disk array that is shared by multiple servers. If the main database server fails, the standby server is able to mount and start the database as though it was recovering from a database crash. This allows rapid failover with no data loss.

Shared hardware functionality is common in network storage devices. Using a network file system is also possible, though care must be taken that the file system has full POSIX behavior. One significant limitation of this method is that if the shared disk array fails or becomes corrupt, the primary and standby servers are both nonfunctional. Another issue is that the standby server should never access the shared storage while the primary server is running.

A modified version of shared hardware functionality is file system replication, where all changes to a file system are mirrored to a file system residing on another computer. The only restriction is that the mirroring must be done in a way that ensures the standby server has a consistent copy of the file system – specifically, writes to the standby must be done in the same order as those on the master. DRBD is a popular file system replication solution for Linux.

### 2.3.2 Warm Standby Using Point-In-Time Recovery

A *warm standby server* (see the section called “Warm Standby Servers for High Availability” in the chapter called “Backup and Restore” in the Advanced Server Documentation) can be kept current by reading a stream of write-ahead log (WAL) records. If the main server fails, the warm standby contains almost all of the data of the main server, and can be quickly made the new master database server. This is asynchronous and can only be done for the entire database server.

### 2.3.3 Master-Slave Replication

A *master-slave replication* setup sends all data modification queries to the master server. The master server asynchronously sends data changes to the slave server. The slave can answer read-only queries while the master server is running. The slave server is ideal for data warehouse queries.

Slony-I is an example of this type of replication, with per-table granularity, and support for multiple slaves. Because it updates the slave server asynchronously (in batches), there is possible data loss during fail over.

### 2.3.4 Statement-Based Replication Middleware

With *statement-based replication middleware*, a program intercepts every SQL query and sends it to one or all servers. Each server operates independently. Read-write queries are sent to all servers, while read-only queries can be sent to just one server, allowing the read workload to be distributed.

If queries are simply broadcast unmodified, functions like `random()`, `CURRENT_TIMESTAMP`, and sequences would have different values on different servers. This is because each server operates independently, and because SQL queries are broadcast (and not actual modified rows).

If this is unacceptable, either the middleware or the application must query such values from a single server and then use those values in write queries. Also, care must be taken that all transactions either commit or abort on all servers, perhaps using two-phase commit (`PREPARE TRANSACTION` and `COMMIT PREPARED`). Pgpool and Sequoia are an example of this type of replication.

### 2.3.5 Synchronous Multi-Master Replication

In *synchronous multi-master replication*, each server can accept write requests, and modified data is transmitted from the original server to every other server before each transaction commits. Heavy write activity can cause excessive locking, leading to poor performance. In fact, write performance is often worse than that of a single server. Read requests can be sent to any server. Some implementations use shared disk to reduce the communication overhead. Synchronous multi-master replication is best for mostly read workloads, though its big advantage is that any server can accept write requests – there is no need to partition workloads between master and slave servers, and because the data changes are sent from one server to another, there is no problem with non-deterministic functions like `random()`.

Postgres Plus Advanced Server does not offer this type of replication, though Postgres Plus Advanced Server two-phase commit (`PREPARE TRANSACTION` and `COMMIT PREPARED`) can be used to implement this in application code or middleware.

### 2.3.6 Asynchronous Multi-Master Replication

For servers that are not regularly connected, like laptops or remote servers, keeping data consistent among servers is a challenge. Using *asynchronous multi-master replication*, each server works independently, and periodically communicates with the other servers to identify conflicting transactions. The conflicts can be resolved by users or conflict resolution rules.

### 2.3.7 Data Partitioning

*Data partitioning* splits tables into data sets. Each set can be modified by only one server. For example, data can be partitioned by offices, e.g., EnterpriseDB U.S. and Pakistan offices, with a server in each office. If queries combining U.S. and Pakistan data are necessary, an application can query both servers, or master/slave replication can be used to keep a read-only copy of the other office's data on each server.

### 2.3.8 Multi-Server Parallel Query Execution

Many of the above solutions allow multiple servers to handle multiple queries, but none allow a single query to use multiple servers to complete faster. This solution allows multiple servers to work concurrently on a single query. This is usually accomplished by splitting the data among servers and having each server execute its part of the query and return results to a central server where they are combined and returned to the user. Pgpool-II has this capability.

## 2.4 Protecting Against SQL Injection Attacks

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

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

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

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

### 2.4.1 SQL/Protect Overview

The following is a description of how *SQL/Protect* guards against SQL injection attacks.

#### 2.4.1.1 Types of SQL Injection Attacks

There are a number of different techniques used to perpetrate SQL injection attacks. Each technique is characterized by a certain *signature*. *SQL/Protect* examines queries for the following signatures:

### Unauthorized Relations

While Postgres Plus 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.

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.

#### 2.4.1.2 Protected Roles

Monitoring for SQL injection attacks involves analyzing SQL statements originating in database sessions where the current user of the session is a protected role. A *protected role* is a Postgres Plus user or group that the database administrator has chosen to monitor

using SQL/Protect. (In Postgres Plus, users and groups are collectively referred to as *roles*.)

Each protected role can be customized for the types of SQL injection attacks (discussed in Section [2.4.1.1](#)) for which it is to be monitored, thus providing different levels of protection by role.

**Note:** A role with the superuser privilege cannot be made a protected role. If however, 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
- 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.

### 2.4.1.3 Attack Attempt Statistics

Each usage of a command by a protected role that is considered an attack by SQL/Protect is recorded in a statistics view that can be easily monitored to identify the start of a potential attack.

Statistics are collected by type of SQL injection attack as discussed in Section [2.4.1.1](#).

This gives database administrators the opportunity to react proactively in preventing theft of valuable data or other malicious actions.

### 2.4.2 Installing SQL/Protect

The library file (`sqlprotect.so` on Linux, `sqlprotect.dll` on Windows) necessary to run SQL/Protect should already be installed in the `lib` subdirectory of your Postgres Plus home directory.

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

The following are the steps to initially start up SQL/Protect:

**Step 1:** Edit the following configuration parameters in the `postgresql.conf` file located in the `data` subdirectory of your Postgres Plus home.

- **shared\_preload\_libraries.** Add `$libdir/sqlprotect` to the list of libraries.

- **custom\_variable\_classes.** Add `edb_sql_protect` to the list of classes.
- **edb\_sql\_protect.enabled.** Controls whether or not SQL/Protect is in effect. Set this parameter to `on` so that SQL/Protect will be running. 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 [2.4.3.2](#) 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`.
- **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`.

The following example shows the settings of these parameters in the `postgresql.conf` file:

```
shared_preload_libraries =
'$libdir/plugins/plugin_debugger,$libdir/plugins/plugin_sql_debugger,$libdir/sqlprotect'
                                # (change requires restart)
                                .
                                .
                                .
custom_variable_classes = 'edb_sql_protect'      # list of custom variable class names
edb_sql_protect.enabled = on
edb_sql_protect.level = learn
edb_sql_protect.max_protected_roles = 64
edb_sql_protect.max_protected_relations = 1024
```

**Note:** All `postgresql.conf` parameters are specified on single lines. The `shared_preload_libraries` parameter is shown on two lines due to space limitations in this document.

**Step 2:** For each database that you want to protect from SQL injection attacks, connect to the database as a superuser (either `enterprisedb` or `postgres`, depending upon your installation options) and run the script `sqlprotect.sql` located in the `share/contrib` subdirectory of your Postgres Plus home directory.

The script creates the SQL/Protect database objects in a schema named `sqlprotect`.

The following example shows this process on Linux:

```
$ /opt/PostgresPlus/8.4AS/bin/psql -d edb -U enterprisedb
Password for user enterprisedb:
psql (8.4.4.8)
Type "help" for help.

edb=# \i /opt/PostgresPlus/8.4AS/share/contrib/sqlprotect.sql
CREATE SCHEMA
GRANT
SET
CREATE TABLE
```

```
GRANT
CREATE TABLE
GRANT
CREATE FUNCTION
CREATE FUNCTION
CREATE FUNCTION
CREATE VIEW
GRANT
CREATE VIEW
GRANT
CREATE VIEW
GRANT
SET
```

**Step 3:** Restart the Postgres Plus database server.

**For Linux only:** Run the `/etc/init.d/ppas-8.4` script with the `restart` option as shown by the following:

```
$ su root
Password:
$ /etc/init.d/ppas-8.4 restart
Stopping Postgres Plus Advanced Server 8.4:
waiting for server to shut down.... done
server stopped
Starting Postgres Plus Advanced Server 8.4:
waiting for server to start.... done
server started
Postgres Plus Advanced Server 8.4 started successfully
```

**For Windows only:** Open Control Panel, Administrative Tools, and then Services. Restart the service named `ppas-8.4 - Postgres Plus Advanced Server 8.4`.

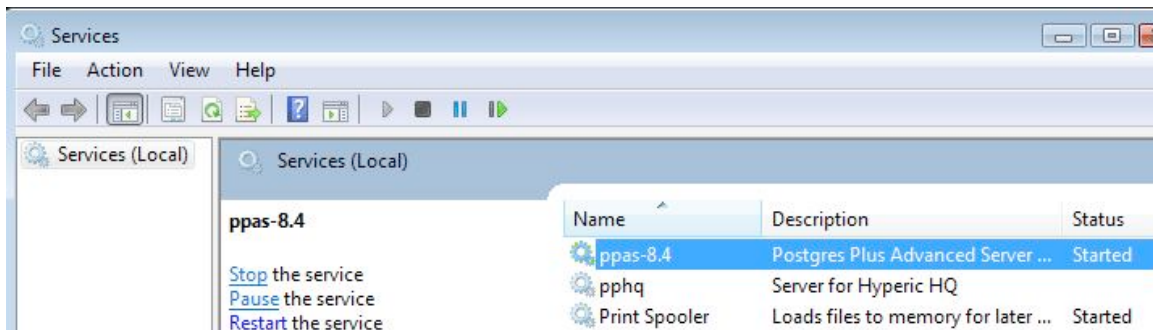


Figure 23 - ppas-8.4 service

### 2.4.3 Configuring SQL/Protect

After SQL/Protect has been installed in a database, you select the roles for which SQL queries are to be monitored for protection, and the current level of protection.

### 2.4.3.1 Selecting Roles to Protect

Since the SQL/Protect tables, functions, and views are built under the `sqlprotect` schema, use of the `SET search_path` command to include the `sqlprotect` schema eliminates the need to schema-qualify the operations discussed in this section.

```
edb=# SET search_path TO sqlprotect;
SET
```

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')`.

For example, connect to Postgres Studio or `psql` as a superuser and issue the following command to protect 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
-----+-----+-----+-----+-----+-----
 14413 | 16665 | 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
```

### 2.4.3.2 Setting the Protection Level

Configuration parameter `edb_sql_protect.level` defines the behavior of SQL/Protect when an event is found. This behavior applies to all roles in the protected roles lists of all databases configured with SQL/Protect in the database server.

The `edb_sql_protect.level` configuration parameter can be set to one of the following values:

- **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.

## 2.4.4 Using SQL/Protect

Once you have configured SQL/Protect in a database the steps to utilize it are described in the following sections.

### 2.4.4.1 Basic Steps

The SQL/Protect tables and views referenced in this section reside in the `sqlprotect` schema. Be sure your search path includes the `sqlprotect` schema, which can be set as follows:

```
edb=# SET search_path TO sqlprotect;
SET
```

**Step 1:** SQL/Protect must first learn which relations are used by a given, protected role.

As applications run by a protected role access relations, SQL/Protect makes entries into the `edb_sql_protect_rel` table. You can verify that this information is being collected by using the following query:

```
edb=# SELECT * FROM edb_sql_protect_rel;
 dbid | roleid | relid
-----+-----+-----
 14413 |  16665 | 16384
 14413 |  16665 | 16391
(2 rows)
```

The view `list_protected_rels` is provided that gives more comprehensive information using the object names instead of the Object Identification numbers (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
(2 rows)
```

SQL/Protect issues messages of NOTICE severity level to the client application as it learns each new relation.

The following example uses the `psql` application connected with role `appuser`. The entries in the `edb_sql_protect_rel` table shown in the previous example were generated by the following two SQL statements issued against the `dept` and `emp` tables:

```
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)
```

In SQL/Protect learn mode, SQL statements that are cause for suspicion are not prevented from executing, but a NOTICE severity level message is issued to alert the user to potentially dangerous statements as shown in 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: 16666
NOTICE:  SQLPROTECT: Illegal Query: empty DML
DELETE 0
```

**Step 2:** 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.

Change the `edb_sql_protect.level` parameter to either `passive` or `active`, depending upon the type of monitoring behavior desired.

The following entry in the `postgresql.conf` file shows that it has been changed to passive.

```
edb_sql_protect.level = passive
```

**Step 3:** Load the new value of `edb_sql_protect.level` by executing the function `pg_reload_conf()` while connected as a superuser.

```
edb=# SELECT pg_reload_conf();
 pg_reload_conf
-----
 t
(1 row)
```

**Step 4:** Monitor the SQL/Protect statistics.

Now SQL/Protect is in passive monitoring mode. For relations that have been learned such as the `dept` and `emp` tables, SQL statements are permitted with no special notification to the client by SQL/Protect as shown by the following:

```
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)

```

**Note:** If `edb_sql_protect.level` is set to `active`, the SQL statement is prevented from executing. Also, the message issued is severity level `ERROR` instead of `WARNING`. The following illustrates SQL statements similar to the previous example, but executed 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

```

By querying the view `edb_sql_protect_stats`, you can see the number of times SQL statements were executed that referenced unlearned relations or contained SQL injection attack signatures.

The columns in `edb_sql_protect_stats` monitor the following:

- **roleid.** OID of the monitored role.
- **superusers.** Number of SQL statements issued when the monitored role is a superuser. In effect, any SQL statement issued by a monitored superuser increases this statistic.
- **relations.** Number of SQL statements issued against relations that were not learned by the role.
- **commands.** Number of DDL statements issued by the role.
- **tautology.** Number of SQL statements issued by the role that contained a tautological condition.
- **dml.** Number of `UPDATE` and `DELETE` statements issued by the role that did not contain a `WHERE` clause.

The following is a query on `edb_sql_protect_stats`:

```

edb=# SELECT * FROM edb_sql_protect_stats;
 roleid | superusers | relations | commands | tautology | dml
-----+-----+-----+-----+-----+-----
 16665 |          0 |          3 |          1 |          1 |          0
(1 row)

```

**Note:** Whenever the database server is restarted, the previous statistics in `edb_sql_protect_stats` are no longer available.

## 2.4.4.2 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.

### Adding a Role to the Protected Roles List

To add a role to the protected roles list run `protect_role('rolename')`.

```
edb=# SELECT protect_role('newuser');
       protect_role
-----
(1 row)
```

### Removing a Role From the Protected Roles List

To remove a role from the protected roles list run `unprotect_role('rolename')`.

```
edb=# SELECT unprotect_role('newuser');
       unprotect_role
-----
(1 row)
```

### 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 user 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 utility commands issued by a given role, update the `allow_utility_cmds` column as follows:

```
UPDATE edb_sql_protect SET allow_utility_cmds = TRUE WHERE dbid = 14413 AND
roleid = 16665;
```

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
-----+-----+-----
```

```
14413 | 16665 | t
(1 row)
```

The updated rules take effect on new sessions started by the role since the change was made.

### Removing a Relation from the List of Protected Relations

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 list of protected relations.

**Step 1:** Delete its entry from the `edb_sql_protect_rel` table. Be sure you include a `WHERE` clause qualifying the following columns:

- **dbid.** OID of the database for which you are making the change
- **roleid.** OID of the role for which you are deleting an entry
- **relid.** OID of the relation you are deleting

The following example illustrates the removal of the relation with OID 16391 from the protected relations of the role with OID 16665 in database with OID 14413, which is the removal of relation `emp` from `appuser`'s list of learned relations.

```
DELETE FROM edb_sql_protect_rel WHERE dbid = 14413 AND roleid = 16665 AND
relid = 16391;
```

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)
```

**Step 2:** Restart the database server.

SQL/Protect will now issue a warning or completely block access (depending upon the setting of `edb_sql_protect.level1`) whenever the role attempts to utilize that relation.

### Stopping and Starting SQL/Protect

If you wish to turn off SQL/Protect once you have installed and configured 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:** Run `pg_reload_conf()` as shown by the following:

```
edb=# SELECT pg_reload_conf();
 pg_reload_conf
-----
 t
(1 row)
```

To re-enable SQL/Protect 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:** Run `pg_reload_conf()` as shown by the following:

```
edb=# SELECT pg_reload_conf();
 pg_reload_conf
-----
 t
(1 row)
```

## 2.4.5 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 in the SQL/Protect tables as explained in this section.

**Note:** This section is applicable if your backup and restore procedures result in the re-creation of database objects in the new database with new OIDs such as is the case when using the `pg_dump` backup program.

If you are backing up your Postgres Plus database server by simply using the operating system's copy utility to create a binary image of the Postgres Plus data files (file system backup method), then this section does not apply.

### 2.4.5.1 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' *Object Identification*

*numbers* (OIDs), and not the objects' text names. The OID is a numeric data type used by Postgres Plus to uniquely identify each database object.

When a database object is created, Postgres Plus 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.

### 2.4.5.2 Backing Up the Database

The following are the steps to back up a database that has been configured with SQL/Protect.

**Step 1:** Create a backup file using `pg_dump`.

The following example shows a plain-text backup file named `/tmp/edb.dmp` created from database `edb` using the `pg_dump` utility program:

```
$ cd /opt/PostgresPlus/8.4AS/bin
$ ./pg_dump -U enterprisedb -Fp -f /tmp/edb.dmp edb
Password:
$
```

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

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

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

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

### 2.4.5.3 Restoring From the Backup Files

**Step 1:** Restore the backup file to the new database.

The following example uses the `psql` utility program to restore the plain-text backup file `/tmp/edb.dmp` to a newly created database named `newdb`:

```
$ /opt/PostgresPlus/8.4AS/bin/psql -d newdb -U enterprisedb -f /tmp/edb.dmp
Password for user enterprisedb:
SET
SET
SET
SET
SET
SET
SET
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.

```
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:** 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 5:** 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 [2.4.5.2](#).

```
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 SQL/Protect tables are now properly restored for this database. This is verified by the following queries on the Postgres Plus system catalogs:

```
newdb=# SELECT datname, oid FROM pg_database;
 datname | oid
-----+-----
 template1 | 1
 template0 | 14412
 edb      | 14413
 newdb    | 16672
(4 rows)

newdb=# SELECT rolname, oid FROM pg_roles;
 rolname | oid
-----+-----
 enterprisedb | 10
 appuser      | 16665
(2 rows)

newdb=# SELECT relname, oid FROM pg_class WHERE relname IN ('dept','emp');
 relname | oid
-----+-----
 dept    | 16796
 emp     | 16799
(2 rows)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect;
 dbid | roleid | protect_relations | allow_utility_cmds | allow_tautology | allow_empty_dml
-----+-----+-----+-----+-----+-----
 16672 | 16665 | t                  | f                   | f               | f
(1 row)

newdb=# SELECT * FROM sqlprotect.edb_sql_protect_rel;
 dbid | roleid | relid
-----+-----+-----
 16672 | 16665 | 16796
 16672 | 16665 | 16799
(2 rows)
```

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 `emp`.

**Step 6:** Verify that the SQL/Protect configuration parameters are set as desired in the `postgresql.conf` file for the database server running the new database as described in Step 1 of Section [2.4.2](#). Restart the database server or run `pg_reload_conf` as appropriate.

You can now use SQL/Protect on the database.

## 3 Application Development

This chapter describes various application programming interfaces and other development tools.

### 3.1 ECPG – Embedded SQL in C

This section contains information that is needed when developing embedded SQL in C (ECPG) programs with Postgres Plus Advanced Server. For complete details on using ECPG, see the Advanced Server Documentation.

#### 3.1.1 Preprocessor

The preprocessor is called `ecpg`. After installation it resides in the Postgres Plus Advanced Server `dbserver/bin` directory.

#### 3.1.2 Library

The ECPG library is called `libecpg.a` or `libecpg.so`. Additionally, the ECPG library is used for communication to the database server so the source program is linked using `-lecpg -lpq`.

The following are the methods present in the ECPG library. These methods are “hidden” by default:

- `ECPGdebug(int on, FILE *stream)` turns on debug logging if called with the first argument non-zero. Debug logging is done on `stream`. Most SQL statements log the arguments and the corresponding result.

The most important being `ECPGdo` that is called on almost all SQL statements, logs both its expanded string, i.e. the string with all the input variables inserted, and the result from the database server. This method is quite useful when searching for errors in SQL statements.

- `ECPGstatus ()` returns `TRUE` if the user is connected to the underlying database and `FALSE` otherwise.

#### 3.1.3 Compilation and Linking

When compiling the preprocessed C code files, the compiler needs to be able to find the ECPG header files in the Postgres Plus Advanced Server include directory. Therefore, it may be necessary to use the `-I` option when invoking the compiler:

(e.g., `-I $EDB_HOME/dbserver/include/`).

.

Programs using C code with embedded SQL have to be linked against the `-lpgtypes` library, for example using the linker options (on Linux):

(e.g. `-L $EDB_HOME/dbserver/lib -lpgtypes`)

The above paths are valid on Linux. They may vary on other platforms.

### 3.1.4 Installation

The ECPG interface library is part of the Postgres Plus Advanced Server installation.

The preprocessor program, `ecpg`, is included in a normal Postgres Plus Advanced Server installation in the `dbserver/bin` directory under the Postgres Plus Advanced Server home directory.

### 3.1.5 Supported Platforms

The ECPG interface library is supported on Unix, Linux, Windows, and Sun Solaris platforms.

### 3.1.6 Prerequisites

- Install Postgres Plus Advanced Server.
- Locate file `libpq.so` in the `dbserver/lib` subdirectory of the Postgres Plus Advanced Server home directory.
- Check the existence of `libpq.so.5` in `/usr/lib`. If not found then execute the following command:

```
ln -s source_filename new_file_name
```

#### Example

```
ln -s $EDB_HOME/dbserver/lib/libpq.so /usr/lib/libpq.so.5
```

## 3.2 *libpq* C Library

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

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

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

### 3.2.1 Using libpq with EnterpriseDB SPL

The EnterpriseDB SPL language can be used with the libpq interface library:

- Creation of procedures, functions, packages
- Calling procedures, functions, packages using prepared statements
- REF CURSOR support
- Static cursor support
- Support for structs and typedefs
- Array support
- DML and DDL operations support
- IN/OUT/IN OUT parameter support

### 3.2.2 EnterpriseDB libpq API

In order to provide support for SPL, the EnterpriseDB libpq interface library has provided the following functions in the libpq application-programming interface:

#### 3.2.2.1 Preparing a Callable Statement

`PQprepareOut()` is used for preparing a callable statement whereas for preparing a prepared statement, the method `PQprepare()` is used.

##### API Definition

```
extern PGresult *
PQprepareOut(PGconn      *conn,
             const char *stmtName,
             const char *query,
             int         nParams,
             const Oid  *paramTypes,
             const int  *paramDirection);
```

#### 3.2.2.2 Executing a Callable Statement

The following method is used for executing a callable statement.

##### API Definition

```
extern int
PQsendQueryPreparedOut (PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char *const * paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

### 3.2.2.3 Fetching Cursors from a Result Set

This method is used for fetching cursors from `PQgetResult(resultset)`.

#### API Definition

```
extern int PQCursorResult (PGconn *conn, PQresult *res);
```

### 3.2.2.4 Returning the Number of Cursors Fetched from a Result Set

This method is used for returning the number of cursors that were fetched from `PQgetResult(resultset)`.

#### API Definition

```
extern int PQnCursor (const PQresult *res);
```

### 3.2.2.5 Retrieving the Out Parameter from the Result

`PQgetOutResult()` is used for retrieving `PQresult` which contains the values of IN/OUT/IN OUT parameters. The values of these IN/OUT/IN OUT parameters can then be retrieved by passing `PQresult` to the `PQgetvalue()` method.

#### API Definition

```
extern PQresult *PQgetOutResult (PGconn *conn);
```

### 3.2.2.6 Cursor Support Statements in EnterpriseDB

`PQgetCursorResult()` is used for fetching cursor record (`PQresult`) residing on a specific index where `tupe_num` is a row in a recordset and `field_num` is a column in the recordset table.

#### API Definition

```
extern PQresult *PQgetCursorResult (const PQresult *res, int
tupe_num, int field_num);
```

### 3.2.2.7 Array Binding

The feature of array binding is the ability to bind an array of data over the wire level protocol using prepared statements. This essentially means all the data that needs to be bound is sent over the wire in one shot. Once the back end receives the bulk data, it will make use of that data to perform insert or update operations.

You can perform bulk operations with prepared statement. So for preparing the statement you can call following function:

```
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 are the functions that can be used to perform bulk operations.

- PQBulkStart
- PQexecBulk
- PQBulkFinish
- PQexecBulkPrepared

### 3.2.2.8 PQBulkStart

This function is used to initialize bulk operations on the server. This function call is mandatory before sending bulk data to the server.

`PQBulkStart` initializes the bulk operation of the previously prepared statement specified by `stmtName`. This initializes the bulk operation to receive data in a format specified by `paramFormats`.

#### API Definition

```
PGresult * PQBulkStart(PGconn *conn,
                      const char * Stmt_Name,
                      unsigned int nCol,
                      const int *paramFmts);
```

### 3.2.2.9 PQexecBulk

This function is used to send data (`paramValues`) to the server against the statement that was previously initialized for bulk operation using `PQBulkStart()`.

This function can be used more than once after `PQBulkStart` in order to send bulk data multiple times. See the example for more details.

### API Definition

```
PGresult *PQexecBulk(PGconn *conn,
                    unsigned int nRows,
                    const char *const * paramValues,
                    const int *paramLengths);
```

### 3.2.2.10 PQBulkFinish

This function stops the previously started bulk operation. After this the previously prepared statement will not be destroyed. You can use this statement again without preparing it again.

### API Definition

```
PGresult *PQBulkFinish(PGconn *conn);
```

### 3.2.2.11 PQexecBulkPrepared

`PQexecBulkPrepared` sends a request to execute a prepared statement with given parameters, and waits for the result/status. This function is a combined function of `PQbulkStart` (), `PQexecBulk` (), and `PQBulkFinish` (). So if you are using this function you don't need to start or stop the bulk operation. This function starts the bulk operation, passes the data to the server, and closes the bulk operation.

The command to be executed is specified by naming a previously prepared statement instead of giving a query string. `stmtName` specifies the name of prepared statement. This feature allows commands that will be used repeatedly to be parsed and planned just once, rather than each time they are executed. The statement must have been prepared previously in the current session.

### 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);
```

### 3.2.2.12 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" );
}

```

### 3.2.2.13 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 );
}

```