Database Programming

Contents

- Why database programming?
- Options for database programming.
- Embedded SQL (in C)
- PlpgSQL
- Database programs
- Stored Procedures and Functions
- Triggers

Why?

- We would like to write programs that use the database. Examples:
  - Process salary for all employees in the end of a month
  - Compute the value of inventory (stock) at the end of a year.
  - Process each and every transaction in accounting and check if credits match debits (trial balance).
- What are the options?

Challenges

- Impedance mismatch.
  - Host language (C, java) variable may not match database data types.
  - SQL operates on (sets of rows) whereas programming language operates on variables and individual records.
  - Bind the variables to SQL.

What are the options

- Use SQL in programming languages like C, …. Java, …. PHP… python.
  - Embedded SQL
- Introduce Procedural Language Constructs to SQL.
  - PL/SQL(Oracle) or PLpgSQL(postgresql)

Part – 1
Embedded SQL (in C)

Embedded SQL in C

- We need to add SQL statements in C to
  - Declare variables for use with the SQL as well as the C programming language.
  - Connect to database.
  - Write SQL to fetch data from database into variables.

Processing SQL in C

- The SQL statements in C have to be processed by another “pre-compilation” step.
- This is the “SQL pre-processor”
  - ECPG is the standard, in the PostgreSQL database built-in, client programming interface for embedding SQL in programs written in the C programming language. It provides the option for accessing the PostgreSQL database directly from the C code in the application, using SQL commands. --Wikipedia.

Installing and testing ECPG

- Install
  - sudo apt-get install libecpg-dev
- Test
  - Write a program and save it in a file called **test.pgc**

```
#include <libpq-fe.h>
int main(int argc, char *argv[]){
   printf("Hello, World!\n");
}
```

  - Run the following commands to compile the program and run it.
    - ecpg test.pgc
    - gcc -I/usr/include/postgresql test.c
    - ./a.out
  - If everything went well, you should get the Hello World! output.

ECPG build details

- From: https://www.postgresql.org/docs/9.1/ecpg-process.html
- General 3 step procedure to compile an embedded-sql c program
  - ecpg test_1.pgc This produces a “.c” file
  - gcc -c -I/usr/include/postgresql test_1.c This produces a “*.o” file
  - gcc -o test_1 test_1.o -lecpg This produces an executable file
- Note the Special directives to the compiler to
  - -I to Include header files for epg (uppercase i)
  - -l to link the library ecpg (lowercase L)

Structure of a embedded SQL program

- Declare variables to be used in SQL and C.
- Connect to the database.
- Perform some datbase operations using SQL. Statements.
- Close the database connection.

Example – 1 Connect to a DB

```
int main(int argc, char *argv[]){
    EXEC SQL BEGIN DECLARE SECTION;
       const char *target = "tcp:postgresql://localhost:5433/postgres";
       const char *user = "postgres";
       const char *password = "postgres";
       char dbname[1024];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT TO :target AS con1 USER :user IDENTIFIED BY :password;
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("Hello, World!\n");
    printf("Connected to database=%s (should be hrb)\n", dbname);
    EXEC SQL DISCONNECT con1;
    printf("Bye!\n");
}
```

Example – 1 - Explanation

- The database connect string
  - tcp:postgresql://localhost:5433/hrb
  - Protocol:database://hostname:port/database_name
- Connecting to the database
  - EXEC SQL CONNECT TO :target AS con1 USER :user IDENTIFIED BY :password;
- Performing a simple SQL operation.
  - EXEC SQL SELECT current_database() INTO :dbname;

Example – 2 – single row fetch

```
EXEC SQL BEGIN DECLARE SECTION;
……   /* other declaration to connect to the database  */
int dnumber;
char dept_name[50];
EXEC SQL END DECLARE SECTION;
printf("Which department number do you want listed?");
scanf("%d",&dnumber);
/* Single row select from a table */
EXEC SQL SELECT dname INTO :dept_name
        FROM department WHERE dnumber = :dnumber;
/* After every exec sql, we must check for the error code and then proceed */
printf("Department name correspoding to dnumber %d is: %s\n",dnumber, dept_name);
```

Example – 3 – Multiple row processing

```
EXEC SQL BEGIN DECLARE SECTION;
……   /* other declaration to connect to the database  */
char ssn[12];
char fname[50];
char lname[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE emp_cursor CURSOR FOR
     SELECT ssn, fname, lname
     FROM employee
     WHERE dnumber = :dnumber
     ORDER BY ssn;

EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
do {
    EXEC SQL FETCH emp_cursor INTO :ssn, :fname, :lname;
    printf("%s\t%s\t%s\n",ssn, fname, lname);
}
while (1); /* infinite loop stopped by the WHENEVER NOT FOUND DO BREAK */
```

Embedded SQL and Dynamic SQL

- So far all the SQL statements we wrote were fixed and created before compile time.
- Can we create a SQL in a string and then run it?
- We can! These SQL statements which are built during run-time are called Dynamic SQL.

Dynamic SQL

- The SQL statements we wrote are used directly.
- But, the dynamic SQL statements require an additional step before they can be executed.
- It is called the “prepare” step.
- So, dynamic SQL have to “prepared” and “executed”

Example – 4 Dynamic SQL

```
    EXEC SQL BEGIN DECLARE SECTION;
    ……   /* other declaration to connect to the database  */
    char query_string[300];
    char where_clause[100];

    EXEC SQL END DECLARE SECTION;
    printf("Give where clase for this sql statement SELECT ssn, fname, lname from employee ... start with where (do not end with semi colon)\n");
    gets(where_clause);
* creating a SQL string using string functions *
    strcpy(query_string, "SELECT ssn, fname, lname from employee ");
    strcat(query_string, where_clause);
    strcat(query_string, " order by ssn");

    printf("Query string is: %s\n", query_string);

    EXEC SQL PREPARE stmt FROM :query_string;

    EXEC SQL DECLARE emp_cursor CURSOR FOR stmt;

/* Rest of the processing of data can happen  */
```

Part – 2
PLpgSQL

Introduction

- SQL is Non-procedural language.
  - That is, you specify the conditions the data must satisfy without saying how to do it.
- But, sometimes we need procedural constructs like IF ELSE and LOOPS to perform some operations.

Example – Payroll Processing

- A monthly payroll (salary computation) program will have to check leaves, the pay structure, taxes and then compute the salary.
- All these operations may not be possible to be done in one SQL statement.
- Hence we need to write programs that list the employees who need to paid salary.
  - For each employee, check the leaves taken,
  - Check the salary structure,
  - Check the income tax savings and calculate tax.
  - Check the deductions like LIC, PF, loans etc.
  - Compute the salary, deductions and amount to be deposited into the bank account.
- This requires us to run a loop and perform some comparisons using IF ELSE or SWITCH CASE constructs. (SQL does not provide these)

PLpgSQL

- Hence the need for a language that can do SQL as well as procedural language features.
- PLpgSQL is a loadable procedural language for the PostgreSQL database system.
- You can declare variables.
- Fetch data using SQL into variables.
- Fetch multiple rows into cursors.
- Perform looping and conditional operations just like in any programming language.

Part – 3
Database Programs in PLpgSQL

A Simple PlpgSQL Program

```
DO
$body$
DECLARE
    lastname varchar;
    myrec employee%ROWTYPE;
BEGIN
    lastname = 'Borg';
    SELECT * INTO STRICT myrec FROM employee WHERE lname = lastname;
    insert into outtable values (myrec.fname||' '||myrec.minit||'. '||myrec.lname);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', lastname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', lastname;
END;
$body$
LANGUAGE 'plpgsql';
```

Fetching Multiple Rows - Cursors

```
DO
$body$
DECLARE
   lastname varchar;
   myrec employee%ROWTYPE;
   empcursor CURSOR FOR select * from employee;
BEGIN
   OPEN empcursor;
   LOOP
      FETCH empcursor INTO myrec;
      EXIT WHEN NOT FOUND;

      IF myrec.salary > 1000 THEN
        insert into outtable values (myrec.fname||' '||myrec.minit||'. '||myrec.lname);
      END IF;
   END LOOP;
   CLOSE empcursor;
END;
$body$
LANGUAGE 'plpgsql';
```

Part – 4
Stored Procedures and Functions

Why

- Can we store the programs we want to run in the database?
  - Stored functions and procedures
- Functions return a value.
- Procedures are programs that do some processing.

A Simple Stored Function

```
CREATE OR REPLACE FUNCTION emp_name(lastname varchar) returns varchar AS $$
DECLARE
   myrec employee%ROWTYPE;
BEGIN
   SELECT * INTO STRICT myrec FROM employee WHERE lname = lastname;
   return myrec.fname||' '||myrec.minit||'. '||myrec.lname;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         RAISE EXCEPTION 'employee % not found', lastname;
      WHEN TOO_MANY_ROWS THEN
         RAISE EXCEPTION 'employee % not unique', lastname;
END;
$$ LANGUAGE plpgsql;
```

Try running the function by: (What does this function return? Ans: The name in firstname minit. Lastname format for a lname)

```
select emp_name('Borg');

select ssn, emp_name(lname) from employee;
```

A Simple Stored Procedure

```
CREATE OR REPLACE PROCEDURE emp_load(salary_lower_limit int) AS $$
DECLARE
   lastname varchar;
   myrec employee%ROWTYPE;
   empcursor CURSOR FOR select * from employee;
BEGIN
   OPEN empcursor;
   LOOP
      FETCH empcursor INTO myrec;
      EXIT WHEN NOT FOUND;

      IF myrec.salary > salary_lower_limit THEN
        insert into outtable values (myrec.fname||' '||myrec.minit||'. '||myrec.lname);
      END IF;
   END LOOP;
   CLOSE empcursor;
END;
$$ LANGUAGE 'plpgsql';

CALL emp_load;  * What does tis program do? Ans: It inserts into a table all the names of employees with salary greater than a limit *
```

Part – 5
Triggers

Why?

- Triggers are functions executed on the occurrence of database events.
- Simple events (often used by programmers)
  - Insert, Update, Delete
- Other events on the database
  - Create table, Create Index, ….
- Triggers can be used for validation, business rules, and audit.

Trigger in PostgreSQL

- Create a stored function.
  - See the slides in the earlier section.
- Create a trigger on a table and call the function.
  - A trigger can be fired before or after the event
    - BEFORE INSERT or AFTER DELETE etc.

Trigger For Audit

- We would like to keep a copy of the record that is being inserted, updated or deleted.
- Create an audit table with all the columns of the table to be audited
  - CREATE TABLE emp_audit as select * from employee where salary < 0; ** This creates a table with the columns of employee table with ZERO rows. Assuming that no one has salary less than ZERO*/*
- Add a few audit columns.
  - ALTER TABLE emp_audit ADD column op_date date,
  - ADD column operation varchar(1),
  - ADD column userid varchar(20);

Trigger Function

```
CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $trig_audit$
    BEGIN
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT OLD.*, now(), 'D', user;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT OLD.*, now(), 'U', user;
            RETURN OLD;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT NEW.*, now(), 'I', user;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$trig_audit$ LANGUAGE plpgsql;
```

CREATE TRIGGER

```
/* If trigger already exists, drop it */
DROP TRIGGER emp_audit on employee;
/* Create trigger */
CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON employee
    FOR EACH ROW EXECUTE PROCEDURE
process_emp_audit();
```

Triggers - Summary

- PostgreSQL does not support assertions.
- Triggers can be used to implement constraints normally implemented by assertions
- Example:
  - If we would like to ensure that an employee’s salary is never greater than his/her supervisor.
    - Create an on insert/update trigger
    - Check the salary of supervisor
    - If salary of supervisor is lesser, then raise an exception and fail the database operation.
      - The insert/Update will not occur.
    - Else do nothing.
      - The insert/Update will occur normally.

Database Programming - Summary

- This is a brief introduction to database programming
- ODBC and JDBC are other ways of connecting to a database.
- PHP is another way of accessing database.
- The embedding of SQL and use of variables and cursors.
- Creation of SQL strings in run time – Dynamic SQL and using PREPARE and EXECUTE steps is present in most languages including Microsoft platform, Java, PHP etc.
- Object-Relational Mapping using frameworks like Spring(Java) and Django (Python) are some other methods of accessing database.
- You can use these slides along with Elmasri/Navathe slides (which use Oracle as example for Database programming)

Questions