09. Programming Databases

Different facets of accessing databases in a program

How do you run SQL statements on a database instance?

Do you know what is role of PgAdmin-III?
[ps: you understand PgAdmin-III and Postgresql server as two different products?]

Following are different types of programs that manipulate databases. Each type of programs has their own place and suitability.

- SQL queries
  - Declarative language – allows writing expressions manipulating relations
  - SQL is not enough to perform many computations involving database manipulations

- Stored Procedures
  - In many applications, database manipulations are to be done in iterative manner; involve complex conditional checks, etc.
  - To address the said requirement, most DBMS do provide procedural extensions to SQL. Stored procedures are written in such languages. Stored procedures are stored as elements in database instance.

- General programming language programs accessing databases through API
  - Most popular way of manipulating database.
  - Most application are written in Programming languages like Java, JSP/Servlet, Python, PHP, C++, or so.
  - Programs written in these languages access (read/write) databases through standard programming interfaces JDBC, Python DB-API, MS-ADO, ODBC, etc.

- Embedded SQL: SQL is permitted in programming
  - Oracle's Pro*C, PostgreSQL's ECPG, and SQLJ are such environments

- Program accessing databases through Native Programming Interfaces:
  - Each DBMS typically provide a lower level access to databases – typically at physical level schema. This makes most efficient access of databases.
  - Programs written using standard API may typically have to be translated to this level calls by things like JDBC driver.
  - Oracle's OCI (Oracle Call Interface), PostgreSQL's (libpq) are such examples.

In any real application we write more than one type of programs. Considering following scenario of DA-IICT inviting applications for admission, you should be able to identify what functionality is implemented using what type of program or so-

- Student Applies: showing up blank application form - will require getting various inputs from databases.

- Applicant enters his/her details. On submission, various validation happens, may be by looking into databases.
- Finally new tuples (for new applicant) are inserted into relevant relations.
- DA-IICT prepares its own merit list; a process updates few attributes of applications relation.
- DA-IICT allocates seats as per available sheet matrix (program wise, category wise, etc); updates again some fields of some relations.

Stored Procedures

Two terms here "stored" and "procedure"?

"procedure": is a generic term used for "function" in a program. It has historic importance as many programming languages had function and procedures as separate constructs. With the time procedure, as a separate construct, has been phased out, and are not available in most modern programming languages. A "cohesive" procedure basically turns out to be a function only.

"stored": procedures are stored in "database instance". They run in database instance context under the control of DBMS.

Why do we need stored Procedure? Here is an motivating example-

> Consider above scenario of DA-IICT inviting applications for B.Tech. admissions. Let us say we have a relation **Applications** as following-
> Applications (AppNo, Name, Marks_Phy, Marks_Chem, Marks_Math, DA_Rank)

Given this relation, if we want to allocate DA_Rank to each applicant, we cannot compute this merely by writing SQL queries. Below is some pseudo code typically performing required computation-

```
rank = 0
for each row in SELECT * FROM Applicants ORDER BY marks DESC
      rank = rank + 1
      UPDATE applicants set DA_RANK = rank
            WHERE application_no = row.application_no
end for
```

Exercise: To make it more complex, let us say, we need to allocate Category Rank as well. Try figuring out appropriate solution for this.

Above pseudo code can be implemented either as stored procedure or in a host programming language like java using appropriate access API.

Below is representative Java code for the same-

```
resultset = conn.execute("SELECT * FROM Applicants ORDER BY marks DESC");
rank = 0
while ( rs.next() ) {
      rank = rank + 1
      con.Execute("update applicants set da_rank = " + rank
               + " where application_no = " + resultset.application_no
   resultset.moveNext()
}
```

It is important to note here that java "host" program runs on the client, and require all tuples bringing to the client. This adds transportation cost, and lowers the program performance.

Stored Procedures are stored "program" as part of database instance, and run on the same computer where data are, avoiding data transportation and thus program performance.

Stored Procedures – applications and benefits

1. Programs runs in immediate vicinity of data, and does not require any data transportation over networks, and this speeds up data processing
2. Functionality that is "batch" in nature, that is, does not require any user intervention; need not to bring any data to the client. Examples
    a. Example above for allocating ranks to a large number of applicants is such a requirement. Once started, no intervention from user is needed.
    b. Computation of SPI and CPI at the end of semester; done once and as "batch" task.
    c. Run month-end database updates for an online Transaction Processing system
    d. Build various materialized views.
    e. Data import/export from other database sources
3. Helps in enabling various operational abstractions over data?
4. Stored procedure stores "application functionality" as part of database; this makes the functionality to be more shareable and reusable.
5. Database triggers are implemented as stored procedures
6. Complex constraints are also enforced using stored procedures.

Language features for Stored Procedures

- SQL is not enough?
- Stored procedures are written for requirements that cannot be expressed in expressive queries like SQL, and **require procedural constructs** like branching and iterations.
- Note that newer releases of SQL do support some branching expressive power in SQL queries (IF, SWITCH or so)
- ANSI added as SQL/PSM (SQL/Persistent Stored Module) as part of SQL extension with SQL-1999.
- RDBMS like Oracle had PL/SQL even before this standardization.

- Most RDBMS, today, provide their own procedural languages for creating stored procedures. For example: PL/SQL (Oracle), PL/PgSQL (postgresql), SQL/PL (DB2), TSQL(MS SQL)

Stored Procedures in PostgreSQL

- PostgreSQL provides an open architecture for creating stored procedures.
- It allows you creating stored procedures in many languages like C, PL/PgSQL, PL/perl, PL/Python, PL/Tcl, and PL/Java.
- PostgreSQL may require you to load the language before programming in that language.
- It also allows you to write simple stored procedures in SQL.
- PostgreSQL calls stored procedures as “stored functions”.

For simplicity, we may be using term “function” for “stored functions” in further discussion.

Learning Stored Procedure Language (SPL)

To repeat: Stored Procedure Language (SPL) is “procedural extension” to SQL.

Let us see what those “extensions”? On the way you should also be checking that how does a stored procedure programming language defer from language like C?

Stored Procedure Language (SPL) = SQL + What ?

- Should allow to create and use variables
- Should allow mixing variables with attribute-names in SQL.
- Allow to submit a SQL statement to the database, and collect the responses into variables so that can be manipulated further
- Support for various procedure constructs like if .. then .. else, loops, exception handling, etc..

Quick tour to PL/PgSQL

- We use “CREATE OR REPLACE FUNCTION” command for creating a new function. Like in any function definition, we specify following things as part of this command- function name, parameters with their types, return type, and finally instructions within the function body.
- Once created successfully, that is no compilation error in the function; the function is added as a component in database instance.
- Below is an example function code with relevant annotations.

Anatomy of a PL/PgSQL

```
CREATE FUNCTION somefunc(x integer) RETURNS integer AS $BODY$
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50
    RETURN quantity - x;
END;
$BODY$ LANGUAGE plpgsql;
```

Below are certain observations for a PL/PgSQL program

1. Function Header: contain name and parameters
2. Function Body: begin with a tag enclosed in $...$; here we have used tag $BODY$; body ends with same tag appearing again.
3. There is a notion of PL/PgSQL block. DECLARE …. BEGIN … END; is structure of a block; has two parts variable declaration and instructions. All variables are declared in declaration section, and instructions are between BEGIN and END separated by “;”.
4. Blocks can be nested as shown above; variable from outer block are visible to inner block;
5. In code above RAISE NOTICE is just a command that outputs on console.

Varibale Declarations

All SQL data types are available in PL/PgSQL.

General syntax of a variable declaration is:

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

Following are some of declarations and self-explanatory-

```
user_id integer;
quantity numeric(5);
url varchar(50);
emprecord employee%ROWTYPE;
mpno project.pno%TYPE;
arow RECORD;
```

Some Examples

Example #01. The function takes two parameters ssn and pno; check if department of the employee and project are same, and returns true or false accordingly.

The function also demonstrates how a result from query can be collected into a variable using SELECT … INTO command.

You can call the function as following: SELECT is_same_dept(123, 'p1');

The call will show true or false accordingly.

```
REATE OR REPLACE FUNCTION is_same_dept(ppno integer, pssn char(2)) RETURNS
bool AS $BODY$
DECLARE
        edno employee.dno%TYPE;
        pdno project.dno%TYPE;
BEGIN
        SELECT dno into edno FROM employee WHERE ssn = pssn;
        SELECT dno into pdno FROM project WHERE pno = ppno;
        if pdno = edno then
                return true;
        else
                return false;
        end if;
END $BODY$ LANGUAGE 'plpgsql';
```

Example #02. The function takes two parameters sales total and sale type. The function looks into table SALES_TYPE in database, finds appropriate tax_rate from the table, compute total tax and returns.

```
CREATE FUNCTION sales_tax(subtotal real, stype varchar(5))
RETURNS real AS $BODY$
DECLARE
        tax_rate numeric(5,2);
BEGIN
        SELECT st.tax_rate into tax_rate FROM SALES_TYPE  WHERE sale_type = stype
        RETURN subtotal * tax_rate / 100;
END;
$BODY$ LANGUAGE plpgsql;
```

Here is typical call to the function

```
SELECT sales_tax(2250, 'C2');
```
Output: **456.56**

Parameters to PL/pgSQL functions

Parameters to PL/pgSQL functions are following types -

IN (default) – remains as constant within the function implementation.
Popularly known as “by value”.
INOUT: carries value in and returns.
OUT: return, cannot carry in value

Example #03: demonstrates various types of parameter types

```
CREATE OR REPLACE FUNCTION test(IN x integer,  INOUT y integer, OUT a integer, OUT b integer)
RETURNS record as $BODY$
BEGIN
        a = x + y;
        b = x * y;
        y = y / 2;
END $BODY$ LANGUAGE 'plpgsql';
```

Procedure called: **SELECT test(5, 6);**
Outputs: **{3, 11, 30}**

Note that for functions having output parameters, either you do not specify the return type, or specify it to of type RECORD, as all out parameters are returned as a record

Looping through query results

```
CREATE or REPLACE FUNCTION test_emp(pdno integer)
RETURNS integer AS $BODY$
DECLARE
        r record;
BEGIN
        FOR r IN SELECT * FROM employee WHERE dno = pdno
        LOOP
                --do whatever you want to do
                raise notice 'Name: %', r.fname;
        END LOOP;
        RETURN 1;
END;
$BODY$ LANGUAGE plpgsql;
```

Functions returning ROW-SET

```
CREATE OR REPLACE FUNCTION empset() RETURNS SETOF employee AS $$
DECLARE
        e employee%rowtype;
BEGIN
        FOR e IN SELECT * FROM employee
        LOOP
                IF e.salary > 50000 THEN  -- could be more complex filter
                        RETURN NEXT e;
                        -- return basically appends a row to result-set, and builds it for return
                END IF;
        END LOOP;
        RETURN;         -- this is actual return
END $BODY$ LANGUAGE 'plpgsql';
```

```
Call:  SELECT * FROM empset();
Note: * FROM used only when functions returns a set of rows or records
```

Catching and handling Errors

An exception catching handling blocks in PL/PgSQL is analogous to try-catch block in Java or C++; code below should be explanatory.

```
BEGIN
        statements
        EXCEPTION
                WHEN cond_1 THEN
                        handler_statements_1
                [WHEN cond_2 THEN
                        handler_statements_2
                ... ]
END;
```

```
Analogous to
try {
  statements
}
catch ( cond_1 ) {
  handler_statements_1
}
catch ( cond_2 ) {
  handler_statements_2
}
```

In PL/PgSQL **cond_x** are pre-declared set of constants; for example:

- NULL_VALUE_NOT_ALLOWED
- NO_DATA_FOUND
- TOO_MANY_ROWS
- INSUFFICIENT_PRIVILEGE
- TOO_MANY_CONNECTIONS
- SQL_ROUTINE_EXCEPTION

Exhaustive list of exceptions at:
http://intranet.daiict.ac.in/~pm_jat/postgres/html/errcodes-appendix.html

Dynamic Query in PL/pgSQL :

Dynamic queries are built at run-time.

For example consider following PL/pgSQL code fragments

```
query := 'UPDATE EMPLOYEE SET salary = salary + salary * ' || percent ||
        ' WHERE dno = ' || mdno;
```

Where query, mdno, and percent are host variables. Since values of percent and mdno is not known till the run-time, could is not available at compile time.

Dynamic query is executed by following command-

```
EXECUTE query;
```

Few more examples

```
CREATE OR REPLACE FUNCTION compute_avg_salary()
  RETURNS SETOF avg_type AS $BODY$
DECLARE
        sum_sal int4; count_emp int4; avg_sal numeric;
        dep department%rowtype; emp employee%rowtype;
        rec avg_type;
BEGIN
        FOR dep IN SELECT * from department LOOP
                sum_sal := 0; count_emp := 0;
                FOR emp IN SELECT * FROM employee WHERE dno = dep.dno LOOP
                        IF emp.salary IS NOT NULL THEN
                                sum_sal := sum_sal + emp.salary;
                                count_emp := count_emp + 1;
                        END IF;
                END Loop;
                rec.dno := dep.dno;
                rec.dname := dep.dname;
                IF count_emp > 0 THEN
                        avg_sal := sum_sal / count_emp;
                ELSE
                        avg_sal := 0;
                END IF;
                rec.avg_sal := avg_sal; RETURN NEXT rec;
        END LOOP;
        RETURN;
END $BODY$ LANGUAGE 'plpgsql';
```

Call the function as: SELECT * FROM compute_avg_salary();

Partial code for ComputeSPI

```
FOR stud IN SELECT DISTINCT student_id from registers WHERE AcadYr = 2010 and Semester=1
Loop
    sum_credit := 0;
    sum_points := 0;
    FOR course_taken IN SELECT * FROM registers WHERE AcadYr = 2010 and Semester=1
                                                AND student_id = stud.student_id
    Loop
    IF course.grade = 'AA' THEN
        p := 10;
    ELSEIF course.grade = 'AB' THEN
        p := 9;
    ...
    END IF;

    SELECT credit into cr FROM course WHERE courseno = course_taken.courseno;
        sum_credit := sum_credit + cr;
        sum_points := sum_point + cr * p;
    end Loop;
    mspi = sum_points/sum_credits;
    UPDATE result set spi = mspi WHERE AcadYr = 2010 and Semester=1 AND
                                       student_id = stud.student_id;
end Loop;
```