COMP9311:
Database Systems

Term 3 2021

Week 5 (Extending SQL with PLpgSQL)

By Helen Paik, CSE UNSW

Textbook: Chapters 6, 7, 8 and 10

Disclaimer: the course materials are sourced from

- previous offerings of COMP9311 and COMP3311
- Prof. Werner Nutt on Introduction to Database Systems (http://www.inf.unibz.it/~nutt/Teaching/IDBs1011/)

SQL as a Programming Language

SQL is a powerful language for manipulating relational data.

But it is *not* a powerful *programming language*.

At some point in developing complete database applications

- we need to implement user interactions
- we need to control sequences of database operations
- we need to process query results in complex ways, or enforce some business rules

and SQL cannot do any of these.

What's wrong with SQL?

Consider the problem of withdrawal from a bank account:

If a bank customer attempts to withdraw more funds than they have in their account, then indicate "Insufficient Funds", otherwise update the account

An attempt to implement this in SQL:

```
select 'Insufficient Funds'
from   Accounts
where  acctNo = AcctNum and balance < Amount;
update Accounts
set    balance = balance - Amount
where  acctNo = AcctNum and balance >= Amount;
select 'New balance: '||balance
from   Accounts
where  acctNo = AcctNum;
```

What's wrong with SQL?

Two possible evaluation scenarios:

- displays "Insufficient Funds", UPDATE has no effect, displays unchanged balance
- UPDATE occurs as required, displays changed balance

Some problems:

- SQL doesn't allow parameterisation (e.g. *AcctNum*)
- always attempts UPDATE, even when it knows it's invalid
- need to evaluate balance test twice (balance < *Amount, balance* >= *Amount*)
- always displays balance, even when not changed

To accurately express the "business logic", we need facilities like conditional execution and parameter passing.

Database Programming

Database programming requires a combination of

- manipulation of data in DB (via SQL)
- conventional programming (via procedural code)

This combination is realised in a number of ways:

- passing SQL commands via a "call-level" interface
  (prog. lang. is decoupled from DBMS; most flexible; e.g. Java/JDBC, Python/DB-API)
- embedding SQL into augmented programming languages
  (requires pre-processor for language; typically DBMS-specific; e.g. SQL/C)
- special-purpose programming languages in the DBMS
  (closely integrated with DBMS; enable extensibility; e.g. PL/SQL, PLpgSQL)

Database Programming

Combining SQL and procedural code solves the "withdrawal" problem:

```
create function
    withdraw(acctNum text, amount integer) returns text
declare bal integer;
begin
    set bal = (select balance
               from   Accounts
               where  acctNo = acctNum);
    if (bal < amount) then
        return 'Insufficient Funds';
    else
        update Accounts
        set    balance = balance - amount
        where  acctNo = acctNum;
        set bal = (select balance
                   from   Accounts
                   where  acctNo = acctNum);
        return 'New Balance: ' || bal;
    end if
end;
```

(This example is actually a stored procedure, using SQL/PSM syntax)

Stored Procedures

Stored procedures

- procedures/functions that are stored in DB along with data
- written in a language combining SQL and procedural ideas
- provide a way to extend operations available in database
- executed within the DBMS (close coupling with query engine)

Benefits of using stored procedures:

- code executed inside DBMS is fast with large data
- user-defined functions can be nicely integrated with SQL
- procedures are managed like other DBMS data
- procedures and the data they manipulate are held together

Stored Procedures – SQL/PSM

SQL/PSM is a 1996 standard for SQL stored procedures.

(PSM = **P**ersistent **S**tored **M**odules)

Syntax for PSM procedure/function definitions:

```
CREATE PROCEDURE ProcName ( Params )
[ local declarations ]
procedure body ;

CREATE FUNCTION FuncName ( Params )
RETURNS Type
[ local declarations ]
function body ;
```

Parameters have three modes: IN, OUT, INOUT

Stored Procedures – SQL/PSM

Example: Find the cost of Toohey's New beer at a specified bar

Default behaviour: return price charged for Toohey's New at that bar.

```
function CostOfNew(string) returns float;
```

How to deal with the case: New is not sold at that bar?

- i.e., exception-handling (e.g. Java)
- return null or negative value to indicate error
- return two values: price and/or status

In PSM, could use return-value *plus* OUT-mode parameter.

Stored Procedures – SQL/PSM

Example: Find cost of Toohey's New beer at a specified bar -> return price charged for New at that bar.

```
CREATE FUNCTION
    CostOfNew(IN pub VARCHAR)
    RETURNS FLOAT
DECLARE cost FLOAT;
BEGIN
    SET cost = (SELECT price FROM Sells
                WHERE  beer = 'New' and
                       bar = pub);
    -- cost is null if not sold in bar
    RETURN cost;
END;
```

Using NULL return value …

Stored Procedures – SQL/PSM

Using an OUT parameter

```
CREATE FUNCTION
    CostOfNew(IN pub VARCHAR,
              OUT status BOOLEAN)
    RETURNS FLOAT
DECLARE cost FLOAT;
BEGIN
    SET cost = (SELECT price FROM Sells
                WHERE  beer = 'New' and
                       bar = pub);
    SET status = (cost IS NOT NULL);
    RETURN cost;
END;
```

Stored Procedures – SQL/PSM

How the function is used …

```
DECLARE myCost FLOAT;
...
SET myCost = CostOfNew('The Regent',ok);
IF (myCost is not null) THEN
    ... do something with the cost ...
ELSE
    ... handle not having a cost ...
END IF;
```

SQL/PSM in REAL database systems

Unfortunately, the PSM standard was developed after most DBMSs had their own stored procedure language

⇒ no DBMS implements the PSM standard exactly.

IBM's DB2 and MySQL implement the SQL/PSM closely (but not exactly)

Oracle's PL/SQL is moderately close to the SQL/PSM standard

- syntax differences e.g. EXIT vs LEAVE, DECLARE only needed once, ...
- extra programming features e.g. packages, exceptions, input/output

PostgreSQL's PLpgSQL is close to PL/SQL (95% compatible)

- has only functions (but can return void); limited exceptions; no i/o
- PLpgSQL function bodies are defined within a string
- PLpgSQL is just one of a number of languages for stored procedures

PLpgSQL

PLpgSQL = **P**rocedural **L**anguage extensions to **P**ost**g**re**SQL**

A PostgreSQL-specific language integrating features of

- procedural programming and SQL programming

Functions are stored in the database with the data.

Provides a means for *extending DBMS functionality*, e.g.

- implementing constraint checking (triggered functions)
- complex query evaluation (e.g. recursive)
- complex computation of column values
- detailed control of displayed results

PLpgSQL

PLpgSQL functions are created (and inserted into db) via:

```
CREATE OR REPLACE
    funcName(param1, param2, ....)
    RETURNS rettype
AS $$
DECLARE
    variable declarations
BEGIN
    code for function
END;
$$ LANGUAGE plpgsql;  -- or 'plpgsql'
```

Note: the entire function body is a single SQL string ($$ … $$)

LANGUAGE plpgsql -> the function body is written in … (specify the language!)

currently... PL/Perl (Chapter 42), and PL/Python(Chapter 43), with more possibilities.

PLpgSQL

add ('abc', 'def') returns the string "abc'def"

```
CREATE OR REPLACE FUNCTION
    add(x text, y text) RETURNS text
AS $add$
DECLARE
    result text;      -- local variable
BEGIN
    result := x||''''||y;
    return result;
END;
$add$ LANGUAGE 'plpgsql';
```

Beware: *never give parameters the same names as attributes.*

One strategy: start all parameter names with an underscore (e.g., _x, _y)

PLpgSQL functions

```
CREATE OR REPLACE FUNCTION
   withdraw(acctNum text, amount real) RETURNS text AS $$
DECLARE
   current REAL;  newbalance REAL;
BEGIN
   SELECT INTO current balance
   FROM Accounts WHERE  acctNo = acctNum;
   IF (amount > current) THEN
      return 'Insufficient Funds';
   ELSE
      newbalance := current - amount;
      UPDATE Accounts
      SET    balance = newbalance
      WHERE  acctNo = acctNum;
      return 'New Balance: '||newbalance;
   END IF;
END;
$$ LANGUAGE plpgsql;
```

PLpgSQL functions

If a PLpgSQL function definition is syntactically correct

- the function is stored in the database
- but is not completely checked until executed

Common errors:

- using a variable with same name as some attribute
  (the variable hides the attribute, so queries using the attribute fail "inexplicably")

Warning: PLpgSQL's error messages can sometimes be obscure.

However, the PLpgSQL parser and error messages have improved *considerably* in recent versions.

PLpgSQL functions

Example: Adding two integers:

```
CREATE OR REPLACE FUNCTION
    add(x int, y int) RETURNS int
AS $add$
DECLARE
    sum integer;      -- local variable
BEGIN
    sum := x + y;
    return sum;       -- return result
END;
$add$ LANGUAGE plpgsql;
```

PLpgSQL function return types

A PostgreSQL function can return a value which is

- an atomic data type (e.g. integer, float, boolean, ...)
- a tuple (e.g. table record type or tuple type)
- a set of atomic values (like a table column)
- a set of tuples (i.e. a table)

A function returning a set of tuples is similar to a view.

Examples of different function return types:

```
create function factorial(int) returns int ...
create function EmployeeOfMonth(date) returns Employee ...
create function allSalaries() returns setof int ...
create function OlderEmployees returns setof Employee
```

The OlderEmployees function returns an instance of the Employee table.

PLpgSQL function return types

```
create function factorial(int) returns int ...
create function EmployeeOfMonth(date) returns Employee ...
create function allSalaries() returns setof int ...
create function OlderEmployees returns setof Employee
```

Different kinds of functions are invoked in different ways:

- function fD() returning a single atomic data value

```
select fD();  -- like an attribute called fD
```

- function fT() returning a single tuple (record)

```
select fT();  -- like a (x,y,z) tuple-value
select * from fT() ... -- like a 1-row table
```

- function fS() returning set of atomic values or records

```
select * from fS() ... -- like a table called fS
```

Using PLpgSQL

PLpgSQL functions can be invoked in several contexts:

- as part of a `SELECT` statement

```
select myFunction(arg1,arg2);
select * from myTableFunction(arg1,arg2);
```

(either on the command line or within another PLpgSQL function)

- as part of the execution of another PLpgSQL function

```
PERFORM myVoidFunction(arg1,arg2);
result := myOtherFunction(arg1);
```

- automatically, via an insert/delete/update trigger

```
create trigger T before update on R
  for each row execute procedure myCheck();
```

INSERT RETURNING … PLpgSQL

INSERT … RETURNING -> Can capture values from tuples inserted into DB:

Useful for recording id values generated for serial PKs:

```
declare newid integer;
...
insert into T(id,a,b,c) values (default,2,3,'red')
returning id into newid;
```

Exceptions… PLpgSQL

Handling Exceptions …

```
-- table T contains one tuple ('Tom','Jones')
declare
    x integer := 3;
begin
    update T set firstname = 'Joe' where lastname = 'Jones';
    -- table T now contains ('Joe','Jones')
    x := x + 1;
    y := x / 0;
exception
    when division_by_zero then
        -- update on T is rolled back to ('Tom','Jones')
        raise notice 'caught division_by_zero';
        return 0;
end;
```

list of exception names, e.g. division_by_zero.
A list of exceptions is in Appendix A of the PostgreSQL Manual.

The server log for your PostgreSQL server is located in /srvr/*YOU*/$PGDATA/log

Function returning tables

PLpgSQL functions can return tables by using a return type

CREATE OR REPLACE *funcName*(*arg1type*, *arg2type*, ...)

RETURNS SETOF *rowType*

Example:

```
CREATE OR REPLACE FUNCTION
   valuableEmployees(REAL) RETURNS SETOF Employees
AS $$
DECLARE
   e RECORD;
BEGIN
   FOR e IN SELECT * FROM Employees WHERE salary > $1
   LOOP
      RETURN NEXT e;  -- accumulates tuples
   END LOOP;
   RETURN;  -- returns accumulated tuples
END; $$ language plpgsql;
```

Function returning tables

Functions returning SETOF *rowType* are used like tables.

SETOF functions look similar to views.

Example:

```
select * from valuableEmployees(50000);
 id |  name   | salary
----+---------+--------
  1 | David   |  75000
  2 | John    |  70000
  3 | Andrew  |  75000
  4 | Peter   |  55000
  8 | Wendy   |  60000
(5 rows)
```

Function returning tables

A difference between views and functions returning a SETOF:

- CREATE VIEW produces a "virtual" table definition
  (table definitions induce a row type with same name as the virtual table)
- SETOF functions require an existing tuple type

In examples before, we used existing Employees tuple type.

You could also define a new tuple return type beforehand via:

```
CREATE TYPE NewTupleType AS (
    attr1  type1,
    attr2  type2,
    ...
    attrn  typen
);
```

Function returning tables

Example of using tuple types ... valuableEmployees() revisited:

```
CREATE TYPE EmpInfo as
   name  varchar(50),
   pay   integer
);
CREATE OR REPLACE FUNCTION
   valuableEmployees(REAL) RETURNS SETOF EmpInfo
AS $$
DECLARE
   emp RECORD;
   inf EmpInfo%ROWTYPE;
BEGIN
   FOR emp IN SELECT * FROM Employees WHERE salary > $1
   LOOP
      inf.name := emp.name;  inf.pay := emp.salary;
      RETURN NEXT inf;  -- accumulates tuples
   END LOOP;
   RETURN;  -- returns accumulated tuples
END; $$ LANGUAGE plpgsql;
```