COMP9311:
DATABASE SYSTEMS

Term 1 2024

Week 4 – PLpgSQL

By Xiaoyang Wang, CSE UNSW

Disclaimer: the course materials are sourced from previous offerings of COMP9311 and COMP3311

Assignment 1: Fri 16:59:59 8th March (Sydney Time)

Late Submission Penalty

- 5% of the max mark (24) will be deducted for each additional day
- Submissions that are more than five days late will not be marked

Special Consideration

- https://www.student.unsw.edu.au/special-consideration

SQL

SELECT

- Single Table
- Multiple Tables

Aggregation

- GROUP BY
- HAVING

Data definition

- CREATE TABLE

Modification

- INSERT/DELETE/UPDATE

Change schemas

- ALTER

Views

Can SQL do this?

Consider the **scenario:**

- Withdraw money at an ATM
- A bank customer attempts to withdraw funds in their account.
- An ATM interacts with a secure database with your banking details.

What can SQL do?

Example: say a person with acctNum 1 is trying to withdraw 50 dollars

Imagine that this is the implementation for the bank withdraw scenario:

```
Select 'Insufficient Funds'
from Accounts
where acctNo = 1 and balance < 50;

Update Accounts
set balance = balance - 50
where acctNo = 1 and balance >=  50;

Select 'New balance:' || balance
from Accounts
where acctNo = 1;
```

What can SQL do?

We can feel that it implicitly defines two evaluation scenarios:

- Display ‘Insufficient Funds’, UPDATE has no effect, displays unchanged balance
- UPDATE occurs as required, displays changed balance

i.e. If there is not enough funds, the ATM should indicate ‘Insufficient Funds’; otherwise, it should allow the withdrawal and update the account balance.

What can SQL do?

```
Select 'Insufficient Funds'
from Accounts
where acctNo = 1 and balance < 50;

Update Accounts
set balance = balance - 50
where acctNo = 1 and balance >=
50;

Select 'New balance:' || balance
from Accounts
where acctNo = 1;
```

Some issues:

- There is no parameterisation (e.g. acctNum, amount)
- Will always attempt UPDATE, even when it knows it's invalid
- Will always display “new” balance, even if it’s unchanged

To accurately express the “business logic“ of withdrawing money, we need facilities like **conditional controls**.

The Limitation of SQL

What we have seen from SQL:

- Data definition (create table(...))
- Query (select...from...where...)
- Constraints on values (domain, key, referential integrity)

And some useful functionalities..

- Views (giving names to SQL queries)

But this is not enough to support real applications. Therefore, more **extensibility** and **programmability** needed.

SQL as a Programming Language

SQL is a powerful language for manipulating relational data, but it is not meant to be a powerful programming language.

What if at some point in developing complete database applications

- We will need to consider implement user interactions
- we need to control sequences of database operations
- we need to process query results in additional ways

How would SQL be able to handle these?

Extending SQL by PostgreSQL

Ways that SQL could be extended

- new data types (incl. constraints, I/O, indexes, ...)
- more powerful constraint checking
- parameterizing queries
- more functions/aggregates for use in queries
- event-based triggered actions

All are required to assist application development

Database Programming

(Let's return to the example of withdrawing money)

To return one of the two possible text results

- If try to withdraw too much => return 'Insufficient funds'
- If withdrawal ok => return 'New balance: newAmount'

Requires a combination of

- **SQL code** to access the database
- **procedural code** to control the process

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**

  (PL is decoupled from DBMS; most flexible; e.g. Java/JDBC)
- **Embedding SQL** into augmented programming languages

  (requires PL pre-processor; DBMS-specific; e.g. SQL/C)
- …

A Stored Procedure Approach

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:

- minimal data transfer cost SQL ↔ procedural code
- user-defined functions can be nicely integrated with SQL
- procedures are managed like other DBMS data (ACID)
- procedures and the data they manipulate are held together

SQL/PSM

SQL/PSM is a **1996 standard for SQL** stored procedures. (PSM = Persistent Stored Modules)

Syntax for PSM procedure/function dentitions:

```
CREATE PROCEDURE ProcName (<ParamList>)
[ local declarations ]
procedure body ;
```

```
CREATE FUNCTION FuncName (<ParamList>)
RETURNS Type
[ local declarations ]
function body ;
```

Parameters have three modes: IN, OUT, INOUT

Parameters

- **IN :** A variable passed in this mode is of **read-only** nature.
- **OUT :** In this mode, a variable is **write-only** and can be passed back to the calling program. It cannot be read inside the procedure and needs to be assigned a value.
- **INOUT :** This procedure has features of **both** IN and OUT mode. The procedure can also read the variables value and can also change it to pass it to the calling function.

SQL/PSM

Example: Defining a procedure:

```
CREATE PROCEDURE AddNewPerson (
IN name CHAR(20),
IN id INTEGER)
INSERT INTO People VALUES(name, id);
```

Example: Invoking a procedure using the SQL/PSM statement CALL

```
CALL AddNewPerson('Codd', 000001);
```

Status of PSM in Modern DB

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

PostgreSQL

We can pass SQL commands via a "call-level" interface

(PL is decoupled from DBMS; most flexible; e.g. Java/JDBC)

We can embed SQL into augmented programming languages

(requires PL pre-processor; DBMS-specific; e.g. SQL/C)

Database programming can also be realised via special-purpose programming language **in the DBMS**

- integrated with DBMS;
- enables extensibility;
- e.g. PL/SQL, PL/pgSQL.

User-defined Data Types

SQL data definition language provides

- atomic types: integer, float, character, Boolean
- ability to define tuple types (create table)

PostgreSQL also provides mechanisms to define new types

- basic types: **CREATE DOMAIN**
- tuple types: **CREATE TYPE**

User-defined Data Types

Syntax for defining a new atomic type (as specialisation of existing type):

```
CREATE DOMAIN DomainName [ AS ] DataType
[ DEFAULT expression ]
[ CONSTRAINT ConstrName constraint ]
```

Example

```
Create Domain UnswCourseCode as text
check (value ~ '[A-Z]{4}[0-9]{4}' );
```

which can then be used like other SQL atomic types

```
Create Table Course (
        id integer,
        code UnswCourseCode, ...
);
```

~ is POSIX Regular Expressions

POSIX regular expressions provide a more powerful means for pattern matching than LIKE and SIMILAR TO.

User-defined Data Types

Syntax for defining a new tuple type

```
CREATE TYPE TypeName AS
(AttrName1 DataType1, AttrName2 DataType2, ...)
```

Example

```
Create type ComplexNumber as (r float, i float);

Create type CourseInfo as (
        course UnswCourseCode ,
        syllabus text ,
        lecturer text
);
```

If attributes need constraints, can be supplied by using a DOMAIN.

User-defined Data Types

CREATE TYPE is different from CREATE TABLE:

- does not create a new (empty) table
- does not provide for key constraints
- does not have explicit specification of domain constraints

Used for **specifying return types of functions** that return tuples or sets.

PostgreSQL: SQL Functions

PostgreSQL allows users to define functions to be defined in SQL

```
CREATE OR REPLACE FUNCTION
      funcName(arg1type, arg2type, ....)
      RETURNS rettype
AS $$
      SQL statements
$$ LANGUAGE sql;
```

PostgreSQL: SQL Functions

Function arguments: accessed as $1, $2, ...

Return value: result of **the last** SQL statement.

- rettype can be any PostgreSQL data type.
- rettype can be a table: returns set of TupleType

PostgreSQL: SQL Functions

Example1:

```
-- max price of specified beer
create or replace function
      maxPrice(text) returns float
as $$
      select max(price) from Sells where beer = $1;
$$ language sql;
```

PostgreSQL: SQL Functions

```
-- usage examples
select maxPrice('New');
```

```
maxprice
----------
2.8
```

```
select bar, price from sells
where beer='New' and price=maxPrice('New');
```

```
bar             price
--------------- -------
Marble Bar      2.8
```

PostgreSQL: SQL Functions

Example2:

```
-- set of Bars from specified suburb
create or replace function
      hotelsIn(text) returns setof Bars
as $$
      select * from Bars where addr = $1;
$$ language sql;
```

PostgreSQL: SQL Functions

```
-- usage examples
select * from hotelsIn('The Rocks');
```

name	addr	license
Australia Hotel	The Rocks	123456
Lord Nelson	The Rocks	123888

PL/pgSQL

Procedural **L**anguage extensions to **P**ost**g**re**SQL**

A PostgreSQL-specific language integrating features of:

- procedural programming
- SQL programming

PL/pgSQL Function

PLpgSQL functions are created in the db

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

Note: the entire function body is a single SQL string.

PL/pgSQL Function Parameters

All parameters are passed by value in PL/pgSQL.

Within a function, parameters can be referred:

- using positional notation ($1, $2, ...)

OR

- via aliases, supplied either

  - as part of the function header (e.g. f(a int, b int))

  - as part of the declarations (e.g. a alias for $1; b alias for $2)

PL/pgSQL Function Parameters

Example: new-style function

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

Beware: never give aliases the same names as attributes.

PL/pgSQL Function Parameters

Example: old-style function exists

```
CREATE OR REPLACE FUNCTION
        cat(text, text) RETURNS text
AS ‘
DECLARE
        x alias for $1; -- alias for parameter
        y alias for $2; -- alias for parameter
        result text; -- local variable
BEGIN
        result := x||""""||y;
        return result;
END;
‘ LANGUAGE ‘plpgsql’;
```

Beware: never give aliases the same names as attributes.

PL/pgSQL Function Parameters

Restrictions: requires x and y to have values of the same “addable" type.

```
CREATE OR REPLACE FUNCTION
      add ( x anyelement, y anyelement) RETURNS anyelement
AS $$
BEGIN
            return x + y ;
END ;
$$ LANGUAGE plpgsql ;
```

PL/pgSQL Function Parameters

PLpgSQL allows **function overloading** (i.e. same name, different arg types)

```
CREATE FUNCTION add ( int , int ) RETURNS int AS
$$ BEGIN return $1 + $2 ; END ; $$ LANGUAGE plpgsql ;

CREATE FUNCTION add ( int , int , int ) RETURNS int AS
$$ BEGIN return $1 + $2 + $3 ; END ; $$ LANGUAGE plpgsql ;

CREATE FUNCTION add ( char (1) , int ) RETURNS int AS
$$ BEGIN return ascii ( $1 )+ $2 ; END ; $$ LANGUAGE plpgsql ;
```

But must differ in arg types, so cannot also define:

```
CREATE FUNCTION add ( char (1) , int ) RETURNS char AS
$$ BEGIN return chr ( ascii ( $1 )+ $2 ); END ; $$ LANGUAGE plpgsql ;
```

i.e. cannot have two functions that look like add(char(1), int).

Function Return Types

A PostgreSQL function can return a value which is

- an atomic data type (e.g. integer, text, ...)
- 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)
- void (i.e. no return value)

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

Function Return Types

Examples of different function return types:

```
create type Employee as (id integer, name text, salary float, ...);
```

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

Function Return Types

Different kinds of functions are invoked in different ways:

```
select factorial(5);
        -- returns one integer
select EmployeeOfMonth('2008-04-01');
        -- returns (x,y,z,...)
```

```
select * from EmployeeOfMonth('2008-04-01');
        -- one-row table
select * from allSalaries();
        -- single-column table
select * from OlderEmployees();
        -- subset of Employees
```

Using PL/pgSQL Functions

PLpgSQL functions can be invoked in several ways:

as part of a SELECT statement

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

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 ();
```

Declaring Data Types

Variables can also be defined in terms of

- the type of an existing variable or table column
- the type of an existing table row (implicit RECORD type)

Declaring Data Types

The variable of a composite type is called a row-type variable.
A row-type variable can hold one row from a SELECT query result.

You can declare a variable to have the same type as a row from an table using <table_name>**%ROWTYPE**, e.g.

```
account Accounts%ROWTYPE ;
```

You may also refer to an attributes type using and specifying <table_name>**.** <column_name>**%TYPE**, e.g.

```
account.branchName%TYPE
```

Declaring Data Types

Examples of declaring data types (in a pl/pgsql function)

- quantity INTEGER ;
- start_quantity quantity%TYPE ;
- employee Employees%ROWTYPE ;
- name Employees.name%TYPE ;

Control Structures in PL/pgSQL

Assignment

variable := expression;

Example:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

Conditionals

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSIF ... THEN ... ELSE

Example

```
IF v_user_id > 0 THEN
UPDATE users SET email = v_email WHERE user_id =
v_user_id; END IF;
```

Control Structures

Iteration

```
LOOP
        Statement
END LOOP ;
```

Example

```
LOOP
        -- some computations
        EXIT WHEN count > 0;
END LOOP;
```

Control Structures

Iteration

```
FOR int_var IN low .. high LOOP
    Statement
END LOOP ;
```

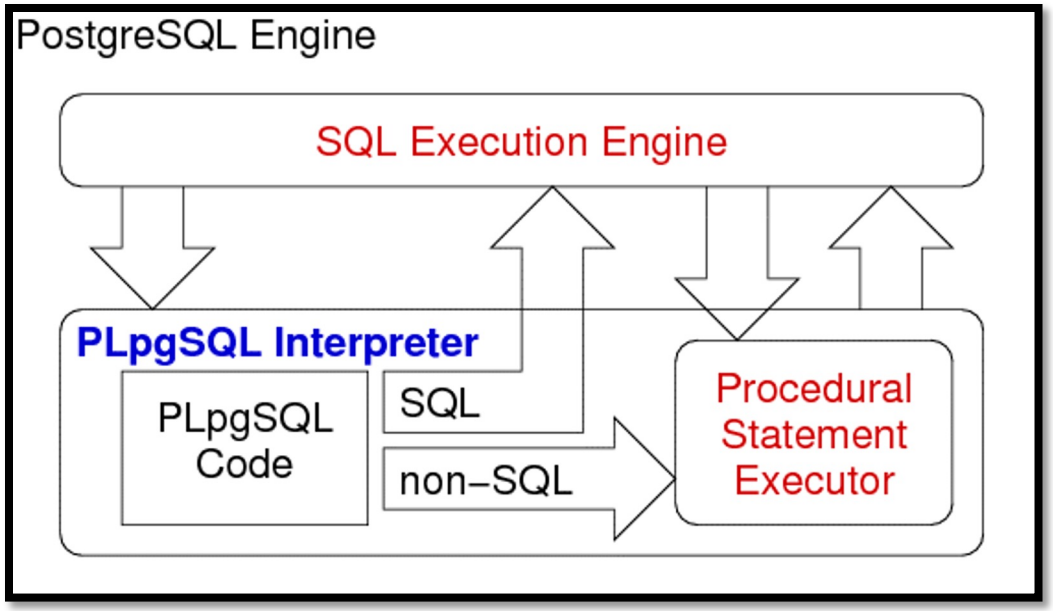
Example

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;
```

PL/pgSQL

The PL/pgSQL interpreter

- executes procedural code and manages variables
- calls PostgreSQL engine to evaluate SQL statements



PL/pgSQL

Provided 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

PL/pgSQL Function

Stored-procedure approach (PLpgSQL):

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

SELECT ... INTO

Can capture query results via

```
SELECT Exp1, Exp2, ..., Expn
INTO   Var1, Var2, ..., Varn
FROM   tablelist
Where  condition
```

The semantics

- execute the query as usual
- return 'projection list' (Exp1, Exp2, ...) as usual
- assign each Expi to corresponding Vari

SELECT ... INTO

Assigning a simple value via SELECT ... INTO:

```
-- cost is local var, price is attr

SELECT price INTO cost

FROM StockList

WHERE item = 'Cricket Bat ';

cost := cost * (1 + tax_rate );

total := total + cost ;
```