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

Exceptions

Syntax of exception

```
BEGIN
        Statements ...
EXCEPTION
        WHEN Exceptions1 THEN
                StatementsForHandler1
        WHEN Exceptions2 THEN
                StatementsForHandler2
        ...
END ;
```

Each exception could be an OR list of exception names, e.g.,

- division_by_zero OR floating_point_exception OR ...

Exceptions

Example:

```
-- 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 x ;
        -- value returned is 4
END ;
```

Exceptions

The **RAISE** operator generates server log entries, e.g.

- RAISE DEBUG 'Simple message';
- RAISE NOTICE ' User = % ' , user_id ;
- RAISE EXCEPTION ' Fatal : value was % ' , value ;

There are several levels of severity:

- DEBUG, LOG, INFO, NOTICE, WARNING, and EXCEPTION
- not all severities generate a message to the client

Cursors

A cursor is an object that retrieves rows from a result table

A cursor is linked to a query, cursors move sequentially from row to row of a result table

Useful for applications to retrieve each row sequentially from the result table.

What happen when the cursor reaches the end of a result table?

Employees

cursor ----➤

Id	Name	Salary
961234	John Smith	35000.00
954321	Kevin Smith	48000.00
912222	David Smith	31000.00

Cursors

Benefits of cursors:

- Save network bandwidth and time. We don't need to wait for whole result set to be retrieved/ processed.
- Since the cursor already stores the value of a row, other database processes can continue to update or delete other rows on the table,
- You can return a cursor in a pl/pgsql function.

Cursors

A FOR loop works with a built-in cursor. There are also explicit cursors in pl/pgsql.
Requires: **RECORD** variable or **Table%ROWTYPE** variable

```
Create Function totalSalary() Returns real As $$
Declare
        employee RECORD;
        totalSalary REAL:=0;
Begin
        FOR employee IN SELECT * FROM Employees
        Loop
            totalSalary:=totalSalary+employee.salary ;
        End Loop;
        Return total;
End ; $$ Language plpgsql;
```

This style accounts for 95% of cursor usage.

Opening and Closing Cursors

A cursor is usually bound to a specific query (i.e., a **bound cursor**)

```
<cursor_name_a> CURSOR FOR <query_b>;
OPEN <cursor_name_a>;
…
CLOSE <cursor_name_a>;
```

OR a cursor may be declared without reference to any query. A cursor that isn't bound to a query is an **unbound cursor**.

```
<cursor_name_c> REFCURSOR;
OPEN <cursor_name_c> FOR <query_d>; … CLOSE
<cursor_name_c>;
OPEN <cursor_name_c> FOR <query_e>; …
```

Either way, declaring a cursor creates an **explicit** cursor.

Fetching Cursors

The fetch operator retrieves the next row from the cursor into a target.

FETCH e INTO me;

FETCH e INTO my_id , my_name , my_salary ;

Note: the variables need to match the corresponding type form the return table.

You could use also fetch in the opposite direction if you specified SCROLL in the cursor declaration.

E.g., <cursor_name_a> SCROLL CURSOR FOR <query_b>;

Cursors

Example of operations on cursors:

```
DECLARE
   employee Employee%ROWTYPE;
   e CURSOR FOR Select * From Employees ;
         totalSalary REAL:=0;
Begin
   OPEN e;
   LOOP
         FETCH e INTO employee;
         EXIT WHEN NOT FOUND;
          totalSalary := totalSalary +employee.salary;
   END LOOP ;
   CLOSE e ;
End; …
```

Database Triggers

The event-condition-action rules was developed to support the need to react to different kinds of events occurring in active databases

Most relational DBMSs effectively support ECA rules by using triggers or procedures, and triggers are included in the SQL:1999 standard.

Event-condition-action rules approach:

- an event activates the trigger
- on activation, the trigger checks a condition
- if the condition holds, a procedure is executed (the action)

In short: a set of stored procedures to automatically executed in response to specified database events

Database Triggers in PostgreSQL

Syntax for PostgreSQL trigger definition:

```
CREATE TRIGGER TriggerName
AFTER/BEFORE Event1 [OR Event2 ...]
ON TableName
FOR EACH ROW/STATEMENT
EXECUTE PROCEDURE FunctionName(args...);
```

Once a trigger is defined, it is bound to one or more database events.

PostgreSQL triggers provide a mechanism for INSERT, DELETE or UPDATE events to automatically activate PL/pgSQL functions

Trigger Procedures

A trigger is defined, there needs to be a trigger procedure.

-- create a trigger

```
CREATE TRIGGER TriggerName
  …
  EXECUTE PROCEDURE function_name(args...);
```

-- follow with the trigger procedure

```
CREATE OR REPLACE FUNCTION function_name() RETURNS
TRIGGER
  ...
```

Types of Triggers

Row level triggers and **Statement-level triggers**

- Row-level triggers executes once for each row affected in the transaction
- Statement-level trigger is invoked once per statement/transaction

```
CREATE TRIGGER TriggerName
  AFTER/BEFORE Event1 ON TableName
  FOR EACH ROW
  EXECUTE PROCEDURE FunctionName(args...);
```

```
CREATE TRIGGER TriggerName
  AFTER/BEFORE Event1 ON TableName
  FOR EACH STATEMENT
  EXECUTE PROCEDURE FunctionName(args...);
```

Trigger Procedures

The trigger function also receives two variables **NEW** and **OLD** that contains the new and old row version, respectively.

Depending on the trigger, NEW and OLD variables can be accessed.

Trigger	NEW	OLD
Insert	Yes	No
Update	Yes	Yes
Delete	No	Yes

Possible usage: RETURN OLD or RETURN NEW (depending on which version of the tuple is to be used)

Trigger Example

Consider a database of people in the USA:

```
Create table Person (
        id integer primary key,
        ssn varchar(11) unique,
        state char(2), ... );
```

```
Create table States (
        id integer primary key,
        code char(2) unique,
        ... );
```

We want the state value Person.state ∈ (select code from States), or exists (select id from States where code=Person.state)

Note: we can use a trigger to help enforce this constraint.

Trigger Example

Create Trigger checkState before insert or update on Person for each row execute procedure **checkState()**;

```
Create Function checkState() returns trigger as $$
begin
        -- normalise the user-supplied value
        new.state = upper(trim(new.state));
        if (new.state !~ '^[A-Z][A-Z]$') then
                raise exception 'Code Must Be Two Alpha Chars';
        end if;
        -- implement referential integrity check
        select * from States where code=new.state;
        if (not found) then
                raise exception 'Invalid State Code %',new.state;
        end if;
        return new;
end;  $$ language plpgsql;
```

Trigger Example

Example Scenario:

- Employee(id, name, address, deptartment, salary)
- Department(id, name, manager, **totSal**)

Consider a **constraint** that we wish to enforce.

The value of Department.total_salary be equal to that of…

select sum(e.salary) from Employee e where e.dept = d.id;

Question: How can we keep the value of total_salary correct?

Trigger Example

Example Scenario:

- Employee(id, name, address, deptartment, salary)
- Department(id, name, manager, **totSal**)

These natural events that could affect the **validity of the database**

- a new employee beginning work in some department
- an employee getting a rise in salary
- an employee changing from one department to another
- an employee leaving the company

Trigger Example

Case 1: A new employees arrives

```
Create trigger TotalSalary1
after insert on Employees
for each row execute procedure totalSalary1();
```

```
Create function totalSalary1() returns trigger
as $$
begin
      if (new.dept is not null) then
         update Department
          set totSal = totSal + new.salary where Department.id = new.dept;
      end if;
      return new;
end; $$ language plpgsql;
```

Trigger Example

Case 2: An employees change departments/salaries

```
Create trigger TotalSalary2
after update on Employee
for each row execute procedure totalSalary2();
```

```
Create function totalSalary2() returns trigger
as $$
begin
        update Department
        set totSal = totSal + new.salary where Department.id = new.dept;
        update Department
        set totSal = totSal - old.salary  where Department.id = old.dept;
        return new;
end; $$ language plpgsql;
```

Trigger Example

Case 3: An employees leaves

```
Create trigger TotalSalary3
after delete on Employee
for each row execute procedure totalSalary3();
```

```
Create function totalSalary3() returns trigger
as $$
begin
      if (old.deptartment is not null) then
         update Department
         set totSal = totSal - old.salary  where Department.id =
old.deptartment;
      end if;
      return old;
end; $$ language plpgsql;
```

Database Triggers

General database trigger usage scenarios:

- To main a separate table for summary data
- Checking schema-level **constraints** (assertions) on update
- To perform updates across tables (to maintain assertions)

Trigger events

Database triggers invoke automatically when the defined event occurs:

We've seen the following in action in the Trigger Example slides

- After Delete?
- After Update?
- After Insert?

Think about situations where this is useful?

- Before Delete?
- Before Update?
- Before Insert?