COMP3311 Week 7

By Manhua Lu

Adapted from Kyu-Sang Kim

Announcements

- Quiz 4 due Friday 27 October @11:59pm AEST
- Assignment 1 provisional performance marks already out!
  - Style mark not included yet
- This is the last real week of SQL and PostgreSQL content
  - Future weeks will return back to Database Theory
  - Next week we will be using Python

Learning Objectives

01 → Constraints (Q2)

02 → Triggers (Q6, Q7)

03 → Aggregates (Q14)

01

Constraints

01 Constraints

```
create table Students(
    zid         serial,
    familyName  varchar(40) not NULL,
    givenName   varchar(30) not NULL,
    dob         date not NULL,
    gender      char(1) check (gender in ('M', 'F', 'X')),
    degree      integer,
    primary key (zid),
    foreign key (degree) references Degrees(dID)
);
```

- Simple constraints can be done with a **check** (covered in Wk3)
- **Assertions** are **schema level constraints**
  - Involve multiple tables
  - Some assertion that must always hold true
  - A bit more complex than a constraint

(Assertion needs to be checked after every change to either Courses or Enrolments)

```
create assertion ClassSizeConstraint check (
    not exists (
        select      c.id
        from        Courses c
        join        Enrolments e on (c.id = e.course)
        group       by c.id
        having      count(e.student) > 9999
    )
);
```

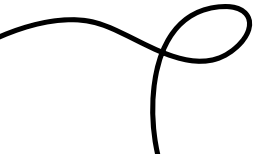
#students in course must be < 10 000

01 → Constraints Question

```
Employee(id:integer, name:text, works_in:integer, salary:integer, ...)
Department(id:integer, name:text, manager:integer, ...)
```

2. Using the same schema from the previous question, write an assertion to ensure that no employee in a department earns more than the manager of their department. Define using a standard SQL assertion statement like:

```
create assertion employee_manager_salary
check  ...
```



02

Triggers

02 → Triggers

- How do we check assertions?
  - After **every change to a table, which is expensive**
- Triggers are like a function that is triggered **after some specific event occurs**
- Triggers are used:
  - To enforce assertions rather than checking them
  - For other side effects that happen when you need to insert /delete / update things
    - Modifying data before insert/update especially if you need data from another table
    - Updating data in another table
    - and many more...

02 → SQL Trigger Syntax

SQL "standard" syntax for defining triggers:

```
CREATE TRIGGER TriggerName
{AFTER|BEFORE}  Event1 [ OR Event2 ... ]
[ FOR EACH ROW ]
ON TableName
[ WHEN ( Condition ) ]
Block of Procedural/SQL Code ;
```

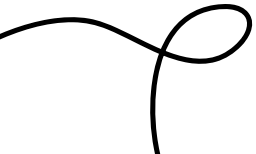
If there is the **'for each row'** clause then code is **executed on each modified tuple**

Else, code is executed **once after all tuples are modified**, just before changes are finally committed

02 → PL/pgSQL Trigger Syntax

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

- ROW: trigger executed for each row
- STATEMENT: trigger executed once



02 → PL/pgSQL Trigger Function Syntax

- For triggers in PL/pgSQL:
  - Triggers can be **activated BEFORE** or **AFTER** the event.
  - If activated AFTER, the effects of the event are visible:
    - **NEW** contains the current value of the altered tuple, is NULL for DELETE operations
    - **OLD** contains the previous value of the altered tuple, is NULL for INSERT operations

Note: The words in capital letters (NEW, OLD) are special variables that will store a current value.

02 → PL/pgSQL Trigger Function Syntax

Order of operation with Triggers

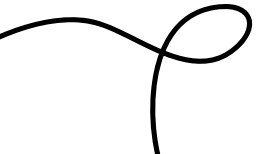
1. BEFORE trigger
2. Constraint checking
3. Operation performed (insert, delete, update etc.)
4. AFTER trigger
5. Constraint checking
6. Commit changes

If anything in steps 1 to 5 inclusive fails, changes are rolled back.

02 → PL/pgSQL Trigger Special Variables

Common special variables:

- NEW
- OLD
- FOUND (boolean)
- TG_OP (stores the string insert, update, delete, truncate telling which of the operation caused the trigger to be fired)
- ...etc.



02 → PL/pgSQL Trigger Function Syntax

```
-- PostgreSQL 7.3 and later
CREATE OR REPLACE FUNCTION name() RETURNS TRIGGER ...
-- PostgreSQL 7.2
CREATE OR REPLACE FUNCTION name() RETURNS OPAQUE ...
```

There is no restriction on what code can go in the function.

However it must contain one of:

```
RETURN old;     or     RETURN new;
```

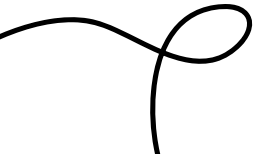
depending on which version of the tuple is to be used.

If an exception is raised in the function, no change occurs.

← This is not exactly accurate. You can also return NULL or a record/row value having the same structure of the table the trigger was fired for.

02 → PL/pgSQL Trigger Overall Process

1. Create function that returns a trigger
2. Then create the trigger that calls that function



02 → PL/pgSQL Trigger Overall Process

6. Consider the following relational schema:

```
create table R(a int, b int, c text, primary key(a,b));
create table S(x int primary key, y int);
create table T(j int primary key, k int references S(x));
```

State how you could use triggers to implement the following constraint checking (hint: revise the material on Constraint Checking from the Relational Data Model and ER-Relational Mapping extended notes)

a. primary key constraint on relation `R`

b. foreign key constraint between `T.j` and `S.x`

02 → PL/pgSQL Trigger Overall Process

7. Explain the difference between these triggers

```
create trigger updateS1 after update on S
for each row execute procedure updateS();

create trigger updateS2 after update on S
for each statement execute procedure updateS();
```

when executed with the following statements. Assume that `S` contains primary keys (1,2,3,4,5,6,7,8,9).

a. `update S set y = y + 1 where x = 5;`

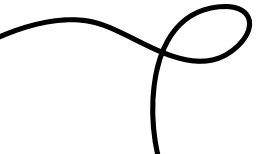
b. `update S set y = y + 1 where x > 5;`

03

Aggregates

03 → Aggregates

- Takes a column of data and collapses it into a single value
- Similar to the reduce() function from other languages
- Example aggregates provided by PostgreSQL
  - avg()
  - min()
  - max()
  - count()
  - sum()
- Users can create their own aggregates too!



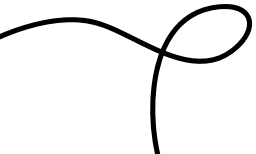
03 → Aggregates (User Supplied) Syntax

```
CREATE AGGREGATE AggName(BaseType) (
    stype     = ...,
    initcond  = ...,
    sfunc     = ...,
    finalfunc = ...,
);
```

- BaseType = **type** of data we're **aggregating over** e.g. float for sum(salary)
- stype = **type** of the state which is **carried throughout the 'calculation'** e.g a single atomic value, or could be a tuple.
- initcond = initial condition e.g. for sum it would start at 0
- sfunc = function that **takes a state and a value**, and **returns a new state**, based on **incorporating the value into the state**.
- finalfunc = **takes a state** and **returns the final return value** of the aggregate. It is optional; if it is not supplied the final result is just the final state.

03 → Aggregates Question

14. Imagine that PostgreSQL did not have an `avg()` aggregate to compute the mean of a column of `numeric` values. How could you implement it using a PostgreSQL user-defined aggregation definition (called `mean`)? Assume that it ignores `null` values. If the column is empty (has no values) return `null`.



02 → Triggers Question (if there is time left)

8. What problems might be caused by the following pair of triggers?

```
create trigger T1 after insert on Table1
for each row execute procedure T1trigger();

create trigger T2 after update on Table2
for each row execute procedure T2trigger();

create function T1trigger() returns trigger
as $$
begin
update Table 2 set Attr1 = ...;
end; $$ language plpgsql;

create function T2trigger() returns trigger
as $$
begin
insert into Table1 values (...);
end; $$ language plpgsql;
```

02 → Triggers Question (if there is time left)

8. What problems might be caused by the following pair of triggers?

```
create trigger T1 after insert on Table1
for each row execute procedure T1trigger();

create trigger T2 after update on Table2
for each row execute procedure T2trigger();

create function T1trigger() returns trigger
as $$
begin
update Table 2 set Attr1 = ...;
end; $$ language plpgsql;

create function T2trigger() returns trigger
as $$
begin
insert into Table1 values (...);
end; $$ language plpgsql;
```

The problem with these triggers **occurs on either an insert on Table1 or an update on Table2**.

When either trigger is activated, it unconditionally executes an SQL statement that will **activate the other trigger**, leading to an **infinite sequence of trigger activations**.