COMP3311 22T3 Database Systems

Week 07
Constraints, Triggers, and Aggregates

1. Consider a schema for an organisation

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

Where `works_in` is a foreign key indicating which Department an Employee works in, and `manager` is a foreign key indicating which Employee is the manager of a Department.

A manager must work in the Department they manage. Can this be checked by standard SQL table constraints? If not, why not?

If it cannot be checked by standard SQL constraints, write an assertion to ensure that each manager also works in the department they manage. Define using a standard SQL assertion statement like:

```
create assertion manager_works_in_department
check  ...
```

Answer:

Except for foreign key constraints, the standard SQL table constraints cannot be used to define conditions which involve multiple tables. SQL assertions allow this, but are not typically implemented because of the overhead of checking them on each change to the relevant tables. Triggers provide a way of implementing this kid of checking, but also allow updates to *ensure* that the assertion conditions hold.

The following would be a suitable assertion:

```
create assertion manager_works_in_department
check  ( not exists (
           select *
           from   Employee e
                  join Department d on (d.manager = e.id)
           where  e.works_in <> d.id
         )
       );
```

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  ...
```

Answer:

The following is suitable:

```
create assertion employee_manager_salary
check  ( not exists (
           select *
           from   Employee emp
                  join Department dept on (dept.id = emp.works_in)
                  join Employee mgr on (dept.manager = mgr.id)
           where  emp.salary > mgr.salary
         )
       );
```

3. What is the SQL command used in PostgreSQL to define a trigger? And what is the SQL command to remove it?

Answer:

Trigger functions are defined using the SQL `CREATE FUNCTION` command. A trigger function should be defined as accepting no arguments, and returns a value of the special `TRIGGER` data type. The syntax for defining a PLpgSQL function is:

```
CREATE FUNCTION function_name () RETURNS trigger
AS $$
   DECLARE
      declarations;
      [...]
   BEGIN
      statements;
      [...]
   END;
$$ LANGUAGE plpgsql;
```

A trigger is defined with the SQL CREATE TRIGGER command with syntax as:

```
CREATE TRIGGER trigger_name
               BEFORE operation
               ON table_name FOR EACH ROW
               EXECUTE PROCEDURE function_name();
```

or

```
CREATE TRIGGER trigger_name
               AFTER operation
               ON table_name FOR EACH ROW
               EXECUTE PROCEDURE function_name();
```

where *operation* is one of INSERT or DELETE or UPDATE

You can assign one trigger to several operations via

```
(BEFORE|AFTER) op1 OR op2 OR ...
```

Triggers are removed via the SQL statement:

```
DROP TRIGGER trigger_name ON table_name;
```

Note that this does not remove any past effects caused by the trigger.

4. Trigers can be defined as BEFORE or AFTER. What exactly are they *before* or *after*?

   Answer:

   Changes are made to the database as follows:

   - fire all BEFORE triggers, possibly modifying any new tuple
   - do all standard SQL constraint checking (e.g. FKs, domains)
   - file all AFTER triggers, possibly updating other tables

   Note that trigger functions can raise exceptions, which would would cause the change to be aborted and rolled back.

5. Give examples of when you might use a trigger BEFORE and AFTER ...

   a. an insert operation

Answer:

- a trigger *before* an insert might check for valid values of the fields (e.g. referential integrity checks), or perhaps generate additional values, such as timestamps, to be included in the newly-inserted tuple
- a trigger *after* an insert might perform additional database updates to ensure semantic consistency of the database, such as enforcing inter-table dependencies (e.g. installing a count of tuples in one relation into another)

b. an update operation

Answer:

- a trigger *before* an update might check for valid values of the modified fields, or generate a new timestamp to be included in the modified tuple
- a trigger *after* an update might do similar maintenance of database consistencies as an insert trigger

c. a delete operation

Answer:

- a trigger *before* a delete might check referential integrity constraints (e.g. can't delete a tuple because it has tuples in other relations referring to it)
- a trigger *after* a delete might do similar maintenance of database consistencies as an insert/update trigger

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`

Answer:

Note: in the solution below, TG_OP is a special variable that tells the trigger function which operation caused it to be invoked. This is useful when a trigger is defined to act on more than one type of operation (as in the triggers below).

```
create trigger R_pk_check before insert or update on R
for each row execute procedure R_pk_check();

create function R_pk_check() returns trigger
as $$
begin
        if (new.a is null or new.b is null) then
                raise exception 'Partially specified primary key for R';
        end if;
        if (TG_OP = 'UPDATE' and old.a=new.a and old.b=new.b) then
                return;
        end if;
    -- not UPDATE, so must be INSERT
        select * from R where a=new.a and b=new.b;
        if (found) then
                raise exception 'Duplicate primary key for R';
        end if;
end;
$$ language plpgsql;
```

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

Answer:

```
create trigger T_fk_check before insert or update on T
for each row execute procedure T_fk_check();

create function T_fk_check() returns trigger
as $$
begin
        select * from S where x=new.k;
        if (not found) then
                raise exception 'Non-existent S.x key in T';;
        end if;
end;
$$ language plpgsql;
```

```
-- assuming that we do *not* want "on delete cascade" semantics

create trigger S_refs_check before delete or update on S
for each row execute procedure S_refs_check();

create function S_refs_check() returns trigger
as $$
begin
        if (TG_OP = 'UPDATE' and old.x=new.x) then
                return;
        end if;
        select * from T where k=old.x;
        if (found) then
                raise exception 'References to S.x from T';;
        end if;
end;
$$ language plpgsql;
```

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;`

Answer:

There is no effective difference in the action of the two triggers for this case. The update only effects one tuple, and so each trigger is activated once for that tuple.

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

Answer:

Trigger `updateS1` will cause `updateS()` to be executed on each of the affected tuples (the ones with primary key values (6,7,8,9)).

Trigger `updateS2` will cause `updateS()` to be executed once, after all of the affected tuples have been modified, but before the updates have been committed.

In both cases, if the function fails, none of the updates will take place (i.e. they are not committed).

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;
```

Answer:

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.

9. Given a table:

```
Emp(empname:text, salary:integer, last_date:timestamp, last_usr:text)
```

Define a trigger that ensures that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. The trigger should also ensure that an employee's name is given and that the salary has a positive value.

The two PostgreSQL builtin functions `user()` and `now()` will provide the values that you need for the "stamp".

Answer:

```
drop if exists table emp;
create table emp (
    empname text primary key,
    salary integer,
    last_date timestamp,
    last_user text

create or replace function emp_stamp () returns trigger
as $$
begin
    -- Check that empname and salary are given
    if new.empname is null then
        raise exception 'empname cannot be NULL value';
    end if;
    if new.salary is null then
        raise exception '% cannot have NULL salary', new.empname;
    end if;

    -- Who would works if they had to pay to do it?
    if new.salary < 0 then
        raise exception '% cannot have a negative salary', new.empname;
    end if;

    -- Remember who changed the payroll when
    new.last_date := now();
    new.last_user := user();
    return new;
end;
$$ language plpgsql;

create trigger emp_stamp before insert or update on emp
    for each row execute procedure emp_stamp();
```

If these were used in a sample database called `mydb`, with an initially empty `emp` table, the effect would appear like this:

```
mydb=> insert into emp values ('John',50000);
INSERT 478005 1
mydb=> select * from emp;
 empname | salary |           last_date           | last_user
---------+--------+-------------------------------+-----------
 John    |  50000 | 2002-09-16 17:30:38.613018+10 | jas
(1 row)

mydb=> update emp set salary=60000 where empname='John';
UPDATE 1
mydb=> select * from emp;
 empname | salary |           last_date           | last_user
---------+--------+-------------------------------+-----------
 John    |  60000 | 2002-09-16 17:31:33.141904+10 | jas
(1 row)
```

10. Consider the following relational schema:

```
   Enrolment(course:char(8), sid:integer, mark:integer)
   Course(code:char(8), lic:text, quota:integer, numStudes:integer);
```

Define triggers which keep the `numStudes` field in the `Course` table consistent with the number of enrolment records for that course in the `Enrolment` table, and also ensure that new enrolment records are rejected if they would cause the quota to be exceeded.

Answer:

```
create or replace function ins_stu() returns trigger as $$
begin
        update course set numStudes=numStudes+1 where code=new.course;
        return new; -- return value is ignore for AFTER triggers
end;
$$ language plpgsql;

create or replace function del_stu() returns trigger as $$
begin
        update course set numStudes=numStudes-1 where code=old.course;
        return old; -- return value is ignore for AFTER triggers
end;
$$ language plpgsql;
```

```
create or replace function upd_stu() returns trigger as $$
begin
        update course set numStudes=numStudes+1 where code=new.course;
        update course set numStudes=numStudes-1 where code=old.course;
        return new; -- return value is ignore for AFTER triggers
end;
$$ language plpgsql;

create or replace function chk_quo() returns trigger as $$
declare
        quota_filled boolean;
begin
        select into quota_filled (numStudes >= quota)
        from Course where code = new.course;
        if (quota_filled) then
                raise exception 'Class % is full', new.course;
        end if;
        return new;
end;
$$ language plpgsql;

create trigger ins_stu after insert on enrolment
    for each row execute procedure ins_stu();

create trigger del_stu after delete on enrolment
    for each row execute procedure del_stu();

create trigger upd_stu after update on enrolment
    for each row execute procedure upd_stu();

create trigger chk_quo before insert or update on enrolment
    for each row execute procedure chk_quo();
```

11. Consider the following (partial) relational schema:

```
   Shipments(id:integer, customer:integer, isbn:text, ship_date:timestamp)
   Editions(isbn:text, title:text, publisher:integer, published:date, ...)
   Stock(isbn:text, numInStock:integer, numSold:integer)
   Customer(id:integer, name:text, ...)
```

Define a PLpgSQL trigger function `new_shipment()` which is called after each `INSERT` or `UPDATE` operation is performed on the `Shipments` table.

The `new_shipment()` function should check to make sure that each added shipment contains a valid customer ID number and a valid ISBN. It should then update the stock information:

- for an `INSERT`, subtract one from the total amount of stock and add one to the number sold
- for an `UPDATE`, if the change involves the book, then restore the `Stock` entry for the old book and update the `Stock` entry for the new book

It should also calculate a new shipment ID (one higher than the previous highest) and ensure that it is placed in the `shipment_id` field of the new tuple. It should also set the time-stamp for the new tuple to the current time.

Under this scheme, tuples would be inserted into the `Shipments` table as:

```
insert into Shipments(customer,isbn) values (9300035,'0-8053-1755-4');
```

Answer:

```
create function new_shipment() returns trigger as $$
declare
    cust_id     integer;    -- customer ID
    book_isbn   text;       -- isbn of new book
    shipment_id integer;    -- shipment ID number
    right_now   timestamp;  -- current time
begin
    -- If there is no matching customer id number, raise an exception.
    select into cust_id id from customers where id = new.customer_id;
    if not found then
      raise exception 'Invalid customer ID number.';
    end if;

    -- If there is no matching ISBN, raise an exception.
    select into book_isbn isbn from editions where isbn = new.isbn;
    if not found then
      raise exception 'Invalid ISBN.';
    end if;

    -- If the previous checks succeeded, update the stock amount
```

```
    -- for INSERT commands.
    if TG_OP = 'INSERT' then
       update Stock set numInStock=numInStock-1 where isbn = new.isbn;
    elsif new.isbn <> old.isbn THEN
       update Stock set numInStock=numInStock+1 where isbn = OLD.isbn;
       update Stock set numInStock=numInStock-1 where isbn = new.isbn;
    end if;

     -- Set the current time variable to current time.
    right_now := now();

     -- Generate a new shipment_id
    select into shipment_id max(id) from shipments order by id desc;
    shipment_id := shipment_id + 1;

    -- Set the values in the new tuple
    new.id := shipment_id;
    new.ship_date := right_now;

    return new;
end;
$$ language plpgsql;

create trigger check_shipment before insert or update
on Shipments for each row execute procedure shipment_addition();
```

12. Suggest a PostgreSQL `CREATE TABLE` definition that would ensure that all of the effects in the previous question happened automatically.

    Answer:

    The above triggers implement referential integrity checks and introduce default values. The same effect could be achieved via:

```
create table Shipments (
    id        serial    primary key,
    -- or id  integer   default netxval('Shipments_id_seq') primary key
    customer  integer   references Customer(id),
    isbn      text      references Editions(id),
    ship_date timestamp default now()
```

13. Consider the following typical components of a PostgreSQL user-defined aggregate definition:

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

Explain what each one does, and how they work together to produce the aggregation result.

Answer:

An aggregate maps a column of values of *BaseType* to a single value of type *ResultType*. The *ResultType* is often the same as the *BaseType*, but this is not required.

Aggregates work using a *state* data structure which carries through the entire calculation. The state could be a single atomic value, or could be a tuple. The `stype` parameter gives the precise data type of the state.

The `initcond` component gives the initial value of the state variable as a string. The `sfunc` function takes a state and a value, and returns a new state, based on incorporating the value into the state. The `finalfunc` function takes a state and returns the final return value of the aggregate. Note that `finalfunc` is optional; if it is not supplied the final result is just the final state.

Type signatures for the above:

```
aggregate Agg : setof BaseType → ResultType
sfunc : StateType,BaseType → StateType
finalfunc : StateType → ResultType
```

How they work together:

```
S : StateType
S = initcond
for each value V in column A of relation R {
   S = sfunc(S, V)
}
return finalfunc(S)
```

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`.

    Answer:

```
create type StateType as ( sum numeric, count numeric );

create function include(s StateType, v numeric) returns StateType
as $$
begin
   if (v is not NULL) then
      s.sum := s.sum + v;
      s.count := s.count + 1;
   end if;
   return s;
end;
$$ language plpgsql;

create or replace function compute(s StateType) returns numeric
as $$
begin
   if (s.count = 0) then
      return null;
   else
      return s.sum::numeric / s.count;
   end if;
end;
$$ language plpgsql;

create aggregate mean(numeric) (
    stype     = StateType,
    initcond  = '(0,0)',
    sfunc     = include,
    finalfunc = compute
);
```

15. How could you get the mean of a column of values without having to write your own aggregation operation? Assume a simple table with schema: `R(...,a:integer,...)`.

Answer:

Using existing aggregates:

```
select sum(a)::numeric/count(a) from R;
```

You need to cast to `numeric` in case the column contains `integer` values; if you don't it will do integer division with truncation. Note that `sum(a)` and `count(a)` correctly ignore null values in the column. If you used `count(*)`, you would count all tuples in the table and produce an average lower than it should be; you would effectively be treating the null values as `0`.