

Tutorial of Function in PostgreSQL

Improved by ZHU Yueming in 2021. April 23th.

All of queries are from the teaching materials of Stephane Faroult.

Experimental-Objective

1. Learn how to create before trigger
2. Learn how to create after trigger

Trigger

When are triggers fired? "During the change" is not a proper answer. In fact, depending on what the trigger is designed to achieve, it may be fired by various events and at various possible precise moments.

If we use an INSERT ... SELECT ... statement, we have ONE statement that inserts SEVERAL rows. If we activate a procedure, what will happen?

```
insert into movies
select *
from films_francais
where year_released = 2010;
```

One thing you can sometimes do is fire the procedure only once for the statement, either BEFORE the first row is inserted, or AFTER the last row is inserted.

OR (and it's sometimes the only option) you can call the procedure before or after you insert EACH row, in which case it will be executed a far greater number of times.

Before Trigger

```
create trigger trigger_name
before insert or update or delete
on table_name
for each row
.....
execute procedure procedure_name();
```

After Trigger

```
create trigger trigger_name
  after insert or update or delete
  on table_name
  for each row
  .....
execute procedure procedure_name();
```

Before trigger example

This example illustrates how sometimes you can still use programs for which you haven't the source code after you have modified the structure of the database.

I must emphasize that this is a "work around", the proper solution is of course to modify the programs, not to use triggers that inflict a severe performance penalty. BUT sometimes you have little choice and this may save the day.

Step1: Create a Students table using studnets.sql

```
create table if not exists students
(
  studentid varchar(10) not null primary key,
  name      varchar(50) not null
);
```

Step2: Import data

It only has two columns.

You can load `CS307SA.txt` into it using the program `SimpleLoader.class` (no source code available).

To use `SimpleLoader` you must:

1. Edit `loader.cnf` and set your username and password into it.
The host and database info is correct for the database used during labs, but it can be changed too.
2. Check the exact location of the Postgres JDBC driver and launch the program from a console like this:

(For MacOS or Linux)

```
java -cp .:<full path to the .jar driver> SimpleLoader -v CS307SA.txt
```

(For Windows)

```
java -cp .;<full path to the .jar driver> SimpleLoader -v CS307SA.txt
```

(perhaps that the : is a ; on Windows). -v means "verbose", it displays informational messages that aren't displayed otherwise.

You can check that table students contains data.

Step3: Separate "name" (implicitly Chinese name) from "English name".

```
alter table students
add column english_name varchar(50);
```

Step4: Create a trigger when insert a row, it can separate English name from "name"

Of course, if you empty the table and load again, `english_name` will remain empty. But we can create a trigger to load the table with the new design. The script `new_design_trigger.sql` creates a trigger that takes what we want to insert, modifies it, and builds a new row suitable for the new table design. The most complicated part is for international students, because sometimes you find the name both in Eastern and Western order, and sometimes the name contains the name only once. Basically I split on the comma, and check whether what I have on one side and the other are the same names or not, which is done with a set operator.

```
create or replace function new_design()
returns trigger
as $$
declare
    n_count int;
    s_name students.name%type;
begin
    s_name := new.name;
    if ascii(s_name) between 19968 and 40959
    then
        -- Chinese name
        new.name := trim(split_part(s_name, ',', 1));
        new.english_name := trim(split_part(s_name, ',', 2));
    else
        -- International student.
        -- Two cases : Same thing in Eastern/Western order
        -- or first name and surname
        with q as
        (select
            new.studentid,
            new.name,
            split_part(s_name, ',', 1) as part1,
            split_part(s_name, ',', 2) as part2)
        select count(*)
        into n_count
        from (select
```

```

        studentid,
        name,
        split_part(part1, ' ', n) as part
    from q
    cross join generate_series(1, 20) n
except
select
    studentid,
    name,
    split_part(part2, ' ', n) as part
    from q
    cross join generate_series(1, 20) n) x;
if n_count = 0
then
    -- Comma separates the same name in different order
    -- Same case as Chinese name
    new.name := trim(split_part(s_name, ',', 1));
    new.english_name := trim(split_part(s_name, ',', 2));
else
    -- Don't change name, modify for English name
    new.english_name := trim(upper(trim(split_part(s_name, ',', 1)))
        || ' ' || trim(split_part(s_name, ',', 2)));
end if;
end if;
return new; -- modified
end;
$$
language plpgsql;

```

create_trigger.sql

creates the trigger. Note that if we insert something into english_name, it won't fire.

```

create trigger students_trg
before insert
on students
for each row
when (new.english_name is null) -- Only for insert statements
-- unaware of the new table structure
execute procedure new_design();

```

Step 5. Execute the java command again.

You can also check following query:

```
insert into students(studentid,name) values('99999999','Chinese,English');
```

```
insert into students(studentid,name) values('99999990', '中国,English');
```

After trigger example

This one is based on what is done with titles in Latin characters. It demonstrates a few interesting things:

- functions that return a table (like generate_series())
- functions written in pure SQL.

This is very Postgres specific, but comparable features exist in Oracle or SQL Server.

There are several scripts to run, they are numbered.

The idea is for every Chinese title to split it into sequences of 1, 2 or 3 characters. When we search a title, the same process is applied to the Chinese string that is searched, and the film(s) with the greatest number of matches will be returned.

Step1: chinese_search0.sql

Creates a table based on alt_titles but only containing Chinese titles. Note that as only the first character is checked, some Japanese titles slip in. There is also a row number added, because some films (Mainland/Hong Kong or Taiwan co-productions) may have both a title in simplified and traditional Chinese.

```
drop table if exists chinese_titles ;
create table chinese_titles
as
select movieid,
row_number() over (partition by movieid order by title) as rn, title
from alt_titles
where ascii(title) between 19968 and 40959;
```

Result:

movieid	rn	title
2906	1	广西电影制片厂
2906	2	廣西電影製片廠
2906	3	黃土地
2906	4	黃土地

Step2: chinese_search1.sql

Intermediate query that shows how we are going to split titles.

```
with t as (select cast('遭遇大王奇遇记' as varchar) as title) select title,
substring(title, n, 1) as one_char, substring(title, n, 2) as two_chars,
substring(title, n, 3) as three_chars
from t
cross join generate_series(1, 200) n
where length(coalesce(substring(title, n, 1), '')) > 0 order by n
;
```

Result:

title	one_char	two_chars	three_chars
邈邈大王奇遇记	邈	邈邈	邈邈大
邈邈大王奇遇记	邈	邈大	邈大王
邈邈大王奇遇记	大	大王	大王奇
邈邈大王奇遇记	王	王奇	王奇遇
邈邈大王奇遇记	奇	奇遇	奇遇记
邈邈大王奇遇记	遇	遇记	遇记
邈邈大王奇遇记	记	记	记

Step3: chinese_search2.sql

Built upon the previous one, query that returns all the bits from a Chinese string.

```
with t as (select cast('邈邈大王奇遇记' as varchar) as title)
select distinct case n
                  when 1
                    then one_char
                  when 2
                    then two_chars
                  else three_chars
                  end
from (select
      substring(title, n, 1) as one_char,
      substring(title, n, 2) as two_chars,
      substring(title, n, 3) as three_chars
from t
  cross join generate_series(1, 200) n
 where length(coalesce(substring(title, n, 1), '')) > 0) x
cross join generate_series(1, 3) n;
```

Result:

three_chars	↑ n
邈	1
记	1
遇	1
奇	1
王	1
大	1
遇	1
王奇	2
遇大	2
大王	2
奇遇	2
邈遇	2
遇记	2
记	2
大王奇	3

Step4: chinese_search3.sql

Pure SQL function returning a table, based on the previous query. The ..._example.sql script show it in action.

It is an example to exercise pure SQL function before you exercise chinese_search3.sql. The return type must be one column

```
create or replace function split_function(p_chinese_text text)
  returns table(char_block varchar(10))
as $$
select substr(t.title, n, 1)
from
  (select cast(p_chinese_text as varchar) as title) t
  cross join generate_series(1, length(t.title)
    ) n;
$$
language sql;
```

chinese_search3.sql:

```
create or replace function chinese_split(p_chinese_text text)
  returns table(char_block varchar(3))
as $$
with t as (select p_chinese_text as chinese_text)
select distinct case n
  when 1
    then one_char
  when 2
    then two_chars
```

```

        else three_chars
        end
from (select
    substring(chinese_text, n, 1) as one_char,
    substring(chinese_text, n, 2) as two_chars,
    substring(chinese_text, n, 3) as three_chars
from t
    cross join generate_series(1, 200) n
    where length(coalesce(substring(chinese_text, n, 1), '')) > 0) x
    cross join generate_series(1, 3) n;
$$
language sql;

```

Test

```
select movieid, chinese_split(title) from chinese_titles ;
```

Result:

movieid	chinese_split
9	春
9	晚春
9	晚
17	闹天
17	宫
17	天
17	大
17	大闹
17	闹

Step5: chinese_search4.sql






Table to hold the bits (note the ON DELETE CASCADE - as it's populated by trigger, it's also automatically deleted) and trigger. At the end of the script it should be properly populated.

Run this one before chinese_search4.sql.

```

alter table chinese_titles
add constraint chinese_titles_pk primary key (movieid, rn);

```


▼	chinese_titles
	movieid integer
	rn bigint
	title varchar(250)
	chinese_titles_pk (movieid, rn)
	chinese_titles_pk (movieid, rn) UNIQUE

chinese_search4.sql

```
create table chinese_blocks
(
  movieid int          not null,
  rn       int          not null,
  block    varchar(3) not null,
  constraint chinese_blocks_pk
  primary key (block, movieid, rn),
  constraint chinese_blocks_fk
  foreign key (movieid, rn)
  references chinese_titles (movieid, rn) on delete cascade
);
```

```
create or replace function chinese_title_split()
  returns trigger
as $$
begin
  if tg_op = 'update'
  then
    delete from chinese_blocks
    where movieid = old.movieid
          and rn = old.rn;
  end if;
  insert into chinese_blocks (movieid, rn, block)
  select
    new.movieid,
    new.rn,
    bl
  from chinese_split(new.title) as bl;
  return null;
end;
$$
language plpgsql;
```

```
create trigger chinese_titles_trg
  after insert or update
  on chinese_titles
  for each row
execute procedure chinese_title_split();
```

Delete all rows in chinese_titles first, and then test what will happen when insert a row in the table chinese_titles

```
delete from chinese_titles;
```

If we execute following query

```
insert into chinese_titles values (9,1,'晚春');
```

in the table chinese_titles

movieid	rn	title
9	1	晚春

in the table chinese_blocks

movieid	rn	block
9	1	晚春
9	1	春
9	1	晚

We can insert all data. Before you do it, don't forget to delete all rows again.

```
insert into chinese_titles
select
  movieid,
  row_number()
  over (
    partition by movieid
    order by title ) as rn,
  title
from alt_titles
where ascii(title) between 19968 and 40959;
```

Step6: chinese_search5.sql

Another SQL table-returning function that finds suitable candidates for a film the (approximate) title of which was supplied as parameter.

```

create or replace function chinese_candidates(p_user_input text)
returns table(movieid int)
as $$
    select movieid
    from (select movieid, rank() over (order by hits desc) rnk
          from (select cb.movieid, count(*) as hits
                from chinese_split(p_user_input) searched
                join chinese_blocks cb
                on cb.block = searched
                group by cb.movieid) x) y
    where rnk = 1;
$$ language sql
;

```

Test:

```

select movieid,title from chinese_titles
where movieid in (select chinese_candidates('故事'));

```

Result:

movieid	title
173	警察故事
7150	民警故事
9025	警察故事4之简单任务
9070	北京爱情故事

Step7: chinese_search6.sql

Search in action. Not sure that his example is the best one.

```

select title,
       also_known_as,
       origin,
       string_agg(case c.credited_as
                   when 'D' then trim(p.surname
                                       || ' ' || coalesce(p.first_name, ''))
                   else null
                   end, ',') directors,
       string_agg(case c.credited_as
                   when 'A' then trim(p.surname
                                       || ' ' || coalesce(p.first_name, ''))
                   else null
                   end, ',') actors
from (select cm.title,
            string_agg(at.title, ',') also_known_as,
            co.country_name || ', ' || m.year_released origin,

```

```

        m.movieid
    from (select ct.movieid, ct.title
          from chinese_candidates('施普灵河') cc
               join chinese_titles ct
                   on ct.movieid = cc.movieid
          where ct.rn = 1) cm
    join movies m
        on m.movieid = cm.movieid
    join countries co
        on co.country_code = m.country
    left join alt_titles at
        on at.movieid = cm.movieid
        and at.title <> cm.title
    group by cm.title,
             co.country_name,
             m.year_released,
             m.movieid) x
    left join credits c
        on c.movieid = x.movieid
    left join people p
        on p.peopleid = c.peopleid
    group by title,
             also_known_as,
             origin
    order by origin
;

```

Result:

	title	also_known_as	origin	directors	actors
1	湄公河行动	湄公河行動,Operation Mekong	Thailand, 2016	Lam Dante	Zhang Hanyu,Peng Eddie