Principles of Database Systems (CS307)

Function; Procedure; Trigger

Ran Cheng

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.

Function

Built-in Functions

- Most DBMS provides a series of built-in functions
  - E.g., Scalar function, aggregation function, window function

```
round(3.141592, 3)  -- 3.142
trunc(3.141592, 3)  -- 3.141
```

```
upper('Citizen Kane')
lower('Citizen Kane')
substr('Citizen Kane', 5, 3)  -- 'zen'
trim('   Oops  ')  -- 'Oops'
replace('Sheep', 'ee', 'i')  -- 'Ship'
```

```
count(*)/count(col), min(col), max(col), stddev(col), avg(col)
```

```
<function> over (partition by <col_p> order by <col_o1, col_o2, …>)
```

- <function>: we can apply (1) ranking window functions, or (2) aggregation functions
- partition by: specify the column for grouping
- order by: specify the column(s) for ordering in each group

Self-defined Function

- Sometimes the built-in functions cannot fulfill our requirements
  - And the power of declarative language (SQL) is not enough

- Most DBMS implement a built-in, SQL-based programming language
  - A procedural extension to SQL

Procedural vs. Declarative

- Two different programming paradigms
  - Imperative (命令式): Describe the algorithm step-by-step (How to do)
    - Procedural: C (and many other legacy languages)
    - Object-oriented: Java
  - Declarative (过程式): Describe the result without specifying the steps (What to do)
    - (Pure) declarative: SQL, Regular Expressions, Markup (HTML, XML), CSS
    - Functional: Scheme, Haskell, Scala, Erlang
    - Logic programming: Prolog

Procedural vs. Declarative

- E.g., How can we get a cup of tea?
  - In a procedural way:

  - In a declarative way:

Procedural vs. Declarative

- E.g., How can we get a cup of tea?
  - In a procedural way:

1. Get a cup
2. Get some tea
3. Get some hot water
4. Put tea into the cup
5. Pour hot water into the cup
6. `return tea;`

  - In a declarative way:

Procedural vs. Declarative

- E.g., How can we get a cup of tea?
  - In a procedural way:

1. Get a cup
2. Get some tea
3. Get some hot water
4. Put tea into the cup
5. Pour hot water into the cup
6. `return tea;`

  - In a declarative way:

```
<a cup of tea/>
```

    - You don't really need to know how to make a cup of tea
    - The system can do it in a black-box manner

大佬喝茶

Procedural vs. Declarative

- E.g., Find all Chinese movies before 1990 in the `movies` table?
  - In a procedural way:

1. Read the `movies` table into the memory
2. For each row `i` in the table, repeat:
   2.1 In row `i`, read the value of the column “country”
   2.2 if …

  - In a declarative way:

```
select * from movies where country = ‘cn’ and year_released < 1990
```

    - You don’t really need to know how to filter the table
    - The DBMS system can do it in a black-box manner

Procedural vs. Declarative

- Benefits in declarative languages
  - No need to understand the details
    - The systems take in charge of all the details
  - Easier to use than imperative programming
    - More user-friendly

- Problem in declarative languages?
  - Cannot specify the control flow of a program
    - “If there is no such command as <a cup of tea/>, you need to create it by yourself”

Procedural Extension to SQL

- Many DBMS products provide a proprietary procedural extension to the standard SQL
  - Transact-SQL (T-SQL) 

  - PL/SQL 

  - PL/PGSQL 

  - (No specific name) 

  - (Not supported) 

… well, sometimes SQLite is even not considered as a DBMS, you can write your own SQL based on it

Function in (Postgre)SQL

- Example: Display the full name for people with “von”
  - When introducing `update`, we have modified the names starting with “von” into “… (von)” for ordering

	peopleid	first_name	surname	born	died	gender
1	16439	Axel	Ambesser (von)	1910	1988	M
2	16440	Daniel	Bargen (von)	1950	2015	M
3	16441	Eduard	Borsody (von)	1898	1970	M
4	16442	Suzanne	Borsody (von)	1957	<null>	F
5	16443	Tomas	Brömssen (von)	1943	<null>	M
6	16444	Erik	Detten (von)	1982	<null>	M
7	16445	Theodore	Eltz (von)	1893	1964	M
8	16446	Gunther	Fritsch (von)	1906	1988	M
9	16447	Katja	Garnier (von)	1966	<null>	F
10	16448	Harry	Meter (von)	1871	1956	M
11	16449	Jenna	Oÿ (von)	1977	<null>	F
12	16450	Alicia	Rittberg (von)	1993	<null>	F
13	16451	Daisy	Scherler Mayer (von)	1966	<null>	F
14	16452	Gustav	Seyffertitz (von)	1862	1943	M

Function in (Postgre)SQL

- If we simply concatenate the first name and the last name, it looks like this:
  - A little bit weird format (a trailing “von”)

```
select first_name || ' ' || surname
from people
where surname like '%(von)';
```

	?column?
1	Axel Ambesser (von)
2	Daniel Bargen (von)
3	Eduard Borsody (von)
4	Suzanne Borsody (von)
5	Tomas Brömssen (von)
6	Erik Detten (von)
7	Theodore Eltz (von)
8	Gunther Fritsch (von)
9	Katja Garnier (von)
10	Harry Meter (von)
11	Jenna Oÿ (von)
12	Alicia Rittberg (von)
13	Daisy Scherler Mayer (von)
14	Gustav Seyffertitz (von)

Function in (Postgre)SQL

- Question: How can we restore the format into “first_name von surname”?
  - String operations

Function in (Postgre)SQL

- Question: How can we restore the format into “first_name von surname”?
  - String operations

```
select case
    when first_name is null then ''
    else first_name || ' '
end || case position('(' in surname)
    when 0 then surname
    else trim(')' from substr(surname, position('(' in surname) + 1))
        || ' '
        || trim(substr(surname, 1, position('(' in surname) - 1))
end
from people
where surname not like '%(von)';
```

Function in (Postgre)SQL

- Question: How can we restore the format into “first_name von surname”?
  - String operations

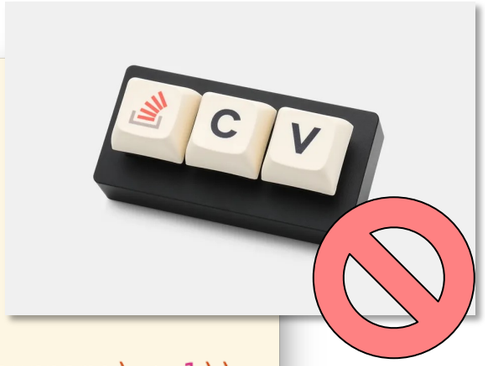
Then, how can we store this part to reuse it in the future?

```
        case
    when first_name is null then ''
    else first_name || ' '
end || case position('(' in surname)
    when 0 then surname
    else trim(')' from substr(surname, position('(' in surname) + 1))
        || ' '
        || trim(substr(surname, 1, position('(' in surname) - 1))
end
from people
where surname not like '%(von)';
```

Function in (Postgre)SQL

- “Copy and Caste” is NOT a good habit
  - Whenever you have painfully written something as complicated, which is pretty generic, you'd rather not copy and paste the code every time you need it

```
case
    when first_name is null then ''
    else first_name || ' '
end || case position('(' in surname)
    when 0 then surname
    else trim(')' from substr(surname, position('(' in surname) + 1))
        || ' '
        || trim(substr(surname, 1, position('(' in surname) - 1))
end
```



Function in (Postgre)SQL

- Store for Reuse
  - In PostgreSQL, we can store the expression and reuse it in another context
- Self-defined Function
  - `create function`

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $variable_name$
   DECLARE
      declaration;
      [...]
   BEGIN
      < function_body >
      [...]
      RETURN { variable_name | value }
   END; LANGUAGE plpgsql;
```

...or, a simpler version

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
    | sql_body
  } ...
```

https://www.postgresql.org/docs/current/sql-createfunction.html

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Function name and the parameter list

- Format for variables and parameters: [name] [type]

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Return type

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

Body

```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

A very simple body: return the value of an expression

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position
        || ' '
        || trim(substr(p_sname, 1, position('(' in
    end;
end;
$$ language plpgsql;
```

A very simple body: return the value of an expression

Procedural extensions provide all the bells and whistles in a true (procedural) programming languages, such as:

- Variables
- Conditions
- Loops
- Arrays
- Error management
- ...

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

```
create function full_name(p_fname varchar, p_sname varchar)
returns varchar
as $$
begin
    return case
        when p_fname is null then ''
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

Language Type

PostgreSQL supports 4 procedural languages:
PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python

- Tcl, Perl, and Python are famous scripting languages in case you don't know

Function in (Postgre)SQL

- How do we rewrite the name conversion expression into a function?

```
create function full_name(p_
returns varchar
as $$
begin
    return case
        when p_fname is null
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr( sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1 osition('(' in p_sname) - 1))
    end;
end;
$$ language plpgsql;
```

```
create function append_test(p_code varchar)
returns varchar
as $$
    if p_code == 'cn':
        return 'China'
    else:
        return 'not China'
$$ language plpython3u;
```

Yes, we can even use Python to write functions

Language Type

PostgreSQL supports 4 procedural languages:
PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python

- Tcl, Perl, and Python are famous scripting languages in case you don’t know

Function in (Postgre)SQL

- Once your function is created, you can use it as if it were any built-in function.

```
select full_name(first_name, surname)
from people
where surname like '%(von)';
```

Function in (Postgre)SQL

- We can run `select` queries in functions
  - Example: design a function “get_country_name” to transform the country codes into country names based on the `countries` table

Function in (Postgre)SQL

- We can run `select` queries in functions
  - Example: design a function “get_country_name” to transform the country codes into country names based on the `countries` table

```
create function get_country_name(p_code varchar)
returns countries.country_name%type
as $$
declare
    v_name countries.country_name%type;
begin
    select country_name
    into v_name
    from countries
    where country_code = p_code;
    return v_name;
end;
$$ language plpgsql;
```

```
select get_country_name(country) from movies;
```

Function in (Postgre)SQL

- We can run `select` queries in functions
  - Example: design a function “get_country_name” to transform the country codes into country names based on the `countries` table

```
create function get_country_name(p_code varchar)
returns countries.country_name%type
as $$
declare
    v_name countries.country_name%type;
begin
    select country_name
    into v_name
    from countries
    where country_code = p_code;
    return v_name;
end;
$$ language plpgsql;
```

```
select get_country_name(country) from movies;
```

… seems to be an easy way to get rid of `join` operations?

```
select c.country_name
from countries c join movies m
on c.country_code = m.country;
```

Function in (Postgre)SQL

- “Cultural Mismatch”
  - Here we have a problem, because there is a big cultural gap between the relational mindset and procedural processing.

  - A “look-up function” forces a “one-row-at-a-time join” which in most cases will be dreadful

```
select get_country_name(country) from movies;
```

For each row in movies, the `select` query in `get_country_name()` is executed once

Comment on Procedural SQL (PL/SQL)

- **Tom Kyte**, who is a Senior Technology Architect at Oracle, says that his suggestion is:
  - You should do it in a single SQL statement if at all possible.
  - If you cannot do it in a single SQL statement, then do it in PL/SQL (as little PL/SQL as possible!)

More to Read

- We may not cover all the details in functions in the theoretical session, so here are some more materials on procedural programming in PostgreSQL:
  - Lab tutorial on Functions
    - Please read it before your next lab sessions
  - Chapter 5.2 “Functions and Procedures,” Database System Concepts (7th Edition)
  - Chapter 43 “PL/pgSQL,” PostgreSQL Documentation
    - https://www.postgresql.org/docs/current/plpgsql.html

Procedure

Functions vs. Procedures

- Generally,
  - “Function” comes from mathematics
    - … which calculates a value with a given input (or to say, map a value to another)
    - Thus, functions always have a return value
  - ”Procedure” comes from programming
    - … which is used to describe a set of instructions that will be executed in order
    - … and does NOT (necessarily) have a return value
- However,
  - Sometimes, the two terms are used for representing the same thing
    - e.g., procedures are called functions as well
  - You need to be careful when meeting both terms
    - Always identify the exact meaning of each term and see whether they have different or the same meaning(s)

Functions and Procedures in (Postgre)SQL

- It follows the general definition of functions and procedures
  - Function: return a value
  - Procedure: return NO value

- However,
  - For some historical reasons, PostgreSQL actually has no implementation specifically for procedures
    - It shares the same mechanism with functions and treats the procedures as “void functions”
  - * But for some other database systems, there are separate implementations for functions and procedures

When to Use Procedures

- For business logics
  - One requirement may need a series of SQL querys and statements
  - Transactions may be used

  - Example: Insert a new movie into the databases
    - `movies` table: basic information for the movie
    - `countries` table: transformation between country names and codes
    - `people` table: new actors / directors
    - `credits` table: new credit information

    - Problem: Update all the tables? Input validation? Code reuse? Security?

When to Use Procedures

- For the requirement of adding a movie:
  - We may have a series queries to execute when inserting only one movie
  - How about one call for all the processes?
    - Benefit 1: Network overhead
      - When running multiple queries, you are going to waste time chatting over the network with the remote serve
    - Benefit 2: Security
      - You can prevent users from modifying data otherwise than by calling carefully written and well tested procedures

Example: Adding a New Movie

- The information provided for a new movie:
  - Title - Movies Table
  - Year - Movies Table
  - Country Name - Countries Table
    - Note: Name, not code. Country codes are not user-friendly
      - E.g. Which country does “at” represent? ”al”? “ma”? “li”?
  - Director - Credits Table
  - Actor 1 - Credits Table
  - Actor 2 - Credits Table
    - Let’s only consider a situation where only one director and at most two actors are allowed
    - It will be more difficult when the number of people are flexible

A Typical Process

- Insert and check the values, constraints, existence, duplicates, etc

- A series of inter-related statements

```
select country_code from countries
... -- Look up the country code

insert into movies
... -- Insert a row in the movies table

select peopleid from people
...
insert into credits
... -- Director

select peopleid from people
...
insert into credits
... -- Actor 1

select peopleid from people
...
insert into credits
... -- Actor 2
```

A Typical Process

- How can we pack them into a single execution unit?
  - Minimize the communication between the client program we are using and the database server
    - Client program = DataGrip, `psql`, ...

```
select country_code from countries
... -- Look up the country code

insert into movies
... -- Insert a row in the movies table

select peopleid from people
...
insert into credits
... -- Director

select peopleid from people
...
insert into credits
... -- Actor 1

select peopleid from people
...
insert into credits
... -- Actor 2
```

`insert into select`

- One thing to optimize: `insert into ... select`

```sql
-- when the arities of table 1 and 2 are the same:
insert into table2
select * from table1
where condition;

-- only insert specific columns
insert into table2 (column1, column2, column3, ...)
select column1, column2, column3, ...
from table1
where condition;
```

Optimize the Insertion

```
insert into ...
select ...
from ...
```

```
insert into movies ...
select country_code, ...
from countries
...

-- insert the director
-- by looking up the people table
insert into credits ...
select peopleid, 'D', ...
from people
...

-- insert the first actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...

-- insert the second actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...
```

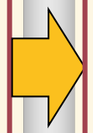
Further Optimize the Insertion

```
insert into movies ...
select country_code, ...
from countries
...

-- insert the director
-- by looking up the people table
insert into credits ...
select peopleid, 'D', ...
from people
...

-- insert the first actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...

-- insert the second actor
-- by looking up the people table
insert into credits ...
select peopleid, 'A', ...
from people
...
```



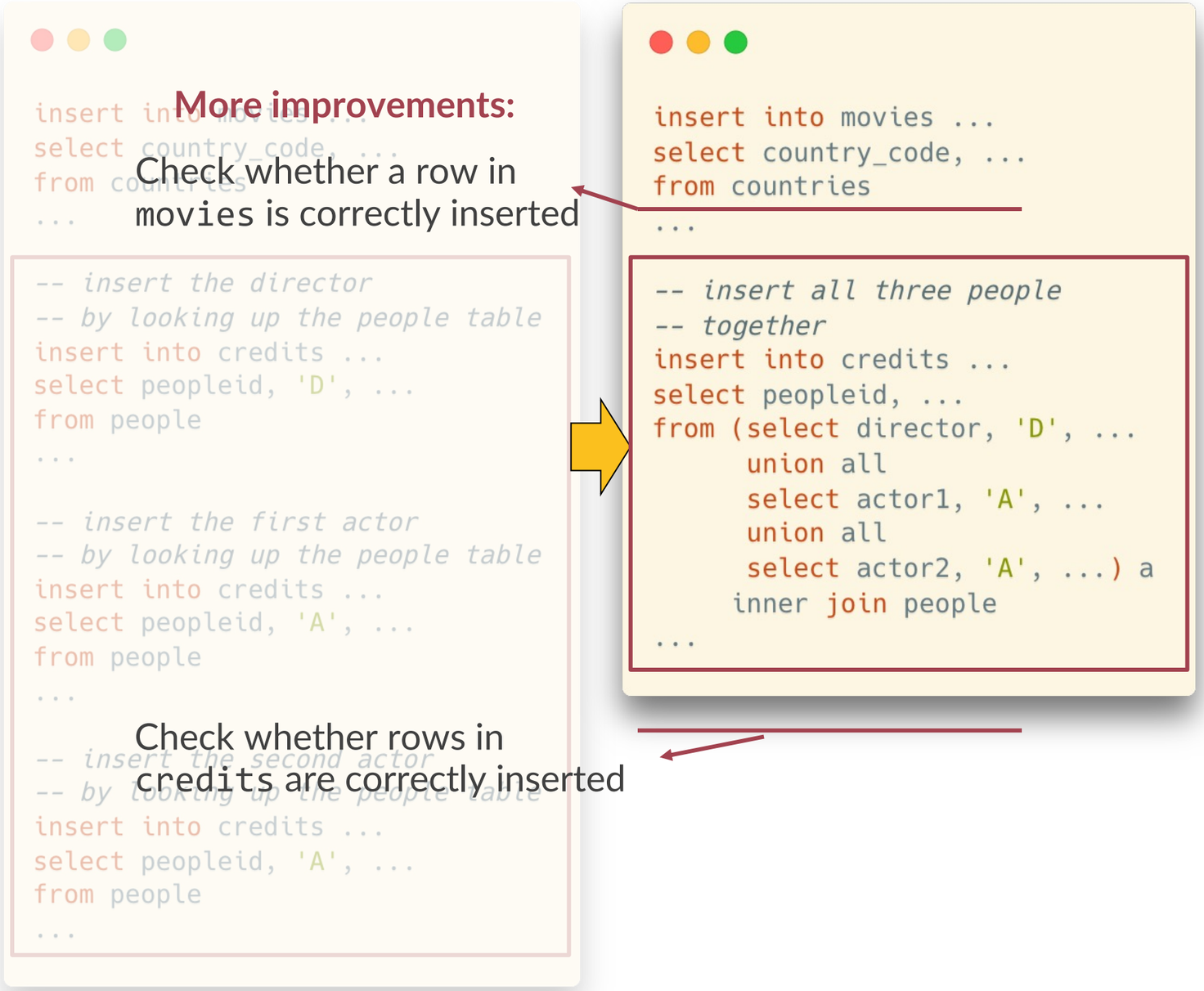
```
insert into movies ...
select country_code, ...
from countries
...

-- insert all three people
-- together
insert into credits ...
select peopleid, ...
from (select director, 'D', ...
      union all
      select actor1, 'A', ...
      union all
      select actor2, 'A', ...) a
     inner join people
...
```

Further Optimize the Insertion

- Use `insert into select`

- Combine the queries of `people`



The Procedure

```
create function movie_registration
           (p_title        varchar,
            p_country_name varchar,
            p_year         int,
            p_director_fn  varchar,
            p_director_sn  varchar,
            p_actor1_fn    varchar,
            p_actor1_sn    varchar,
            p_actor2_fn    varchar,
            p_actor2_sn    varchar)
returns void
as $$
declare
  n_rowcount int;
  n_movieid int;
  n_people int;
begin
    insert into movies(title, country, year_released)
        select p_title, country_code, p_year
        from countries
        where country_name = p_country_name;
    get diagnostics n_rowcount = row_count;

    if n_rowcount = 0
    then
        raise exception 'country not found in table COUNTRIES';
    end if;

    n_movieid := lastval();
    select count(surname)
    into n_people
    from (select p_director_sn as surname
        union all
        select p_actor1_sn as surname
        union all
        select p_actor2_sn as surname) specified_people
    where surname is not null;

    insert into credits(movieid, peopleid, credited_as)
        select n_movieid, people.peopleid, provided.credited_as
        from (select coalesce(p_director_fn, '*') as first_name,
            p_director_sn as surname,
            'D' as credited_as
            union all
            select coalesce(p_actor1_fn, '*') as first_name,
                p_actor1_sn as surname,
                'A' as credited_as
            union all
            select coalesce(p_actor2_fn, '*') as first_name,
                p_actor2_sn as surname,
                'A' as credited_as) provided
            inner join people
        on people.surname = provided.surname
            and coalesce(people.first_name, '*') = provided.first_name
        where provided.surname is not null;

    get diagnostics n_rowcount = row_count;
    if n_rowcount != n_people
    then
        raise exception 'Some people couldn''t be found';
    end if;
end;
$$ language plpgsql;
```

The Procedure

```
create function movie_registration
           (p_title        varchar,
            p_country_name varchar,
            p_year         int,
            p_director_fn  varchar,
            p_director_sn  varchar,
            p_actor1_fn    varchar,
            p_actor1_sn    varchar,
            p_actor2_fn    varchar,
            p_actor2_sn    varchar)
returns void
as $$
declare
  n_rowcount int;
  n_movieid int;
  n_people int;
begin
    insert into movies(title, country, year_released)
        select p_title, country_code, p_year
        from countries
        where country_name = p_country_name;
    get diagnostics n_rowcount = row_count;

    if n_rowcount = 0
    then
        raise exception 'country not found in table COUNTRIES';
    end if;
```

Check whether a row in `movies` is correctly inserted

```
    n_movieid := lastval();
    select count(surname)
    into n_people
    from (select p_director_sn as surname
        union all
        select p_actor1_sn as surname
        union all
        select p_actor2_sn as surname) specified_people
    where surname is not null;

    insert into credits(movieid, peopleid, credited_as)
        select n_movieid, people.peopleid, provided.credited_as
        from (select coalesce(p_director_fn, '*') as first_name,
            p_director_sn as surname,
            'D' as credited_as
            union all
            select coalesce(p_actor1_fn, '*') as first_name,
                p_actor1_sn as surname,
                'A' as credited_as
            union all
            select coalesce(p_actor2_fn, '*') as first_name,
                p_actor2_sn as surname,
                'A' as credited_as) provided
            inner join people
        on people.surname = provided.surname
            and coalesce(people.first_name, '*') = provided.first_name
        where provided.surname is not null;

    get diagnostics n_rowcount = row_count;
    if n_rowcount != n_people
    then
        raise exception 'Some people couldn''t be found';
    end if;
end;
$$ language plpgsql;
```

Check whether rows in `credits` are correctly inserted

Calling Procedures

- (In PostgreSQL) We can call the procedure interactively by calling it from a SELECT statement (that will return nothing)

```
select movie_registration('The Adventures of Robin Hood',
                          'United States', 1938,
                          'Michael', 'Curtiz',
                          'Errol', 'Flynn',
                          null, null);
```

- We can also call a procedure from another procedure

```
perform movie_registration('The Adventures of Robin Hood',
                           'United States', 1938,
                           'Michael', 'Curtiz',
                           'Errol', 'Flynn',
                           null, null);
```

Trigger

Trigger - Actions When Changing Tables

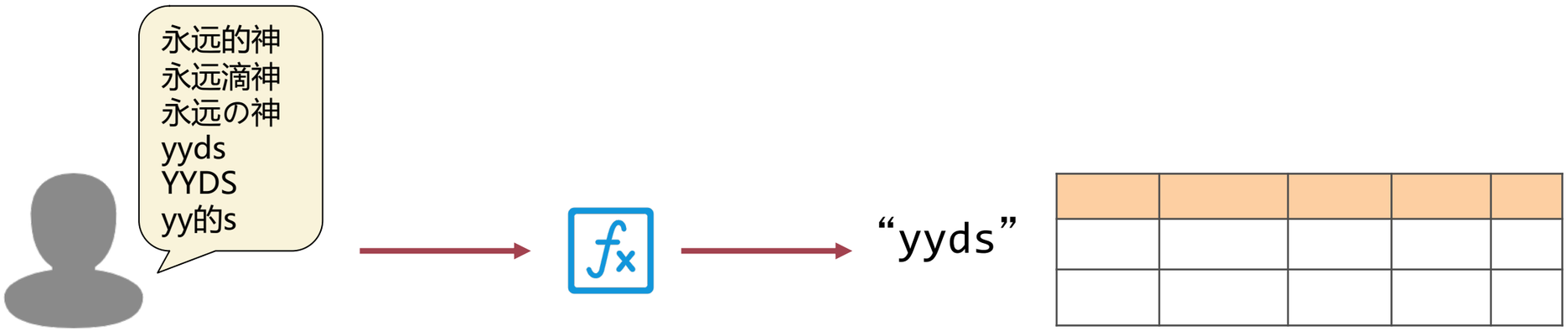
> A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed.
>
> -- Chapter 39, PostgreSQL Documentation
>
> A trigger is a statement that the system executes automatically as a side effect of a modification to the database.
>
> -- Chapter 5.3, Database System Concepts, 7th

- We can attach “actions” to a table
  - They will be executed automatically whenever the data in the table changes
- Purpose of using triggers
  - Validate data
  - Checking complex rules
  - Manage data redundancy

Purpose of Using Triggers

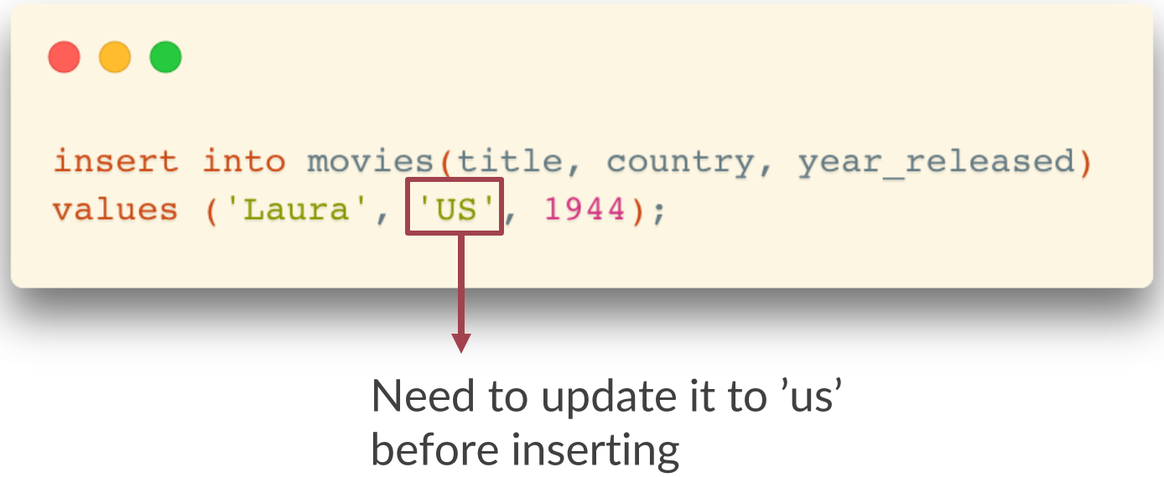
- Validate data
  - Some data are badly processed in programs before sending to the database
  - We need to validate such data before inserting them into the database

  - “On-the-fly” modification
    - Change the input directly when the input arrives



Purpose of Using Triggers

- Validate data
  - Example: insert a row in the movies table
    - In the JDBC program, an `insert` request is written like the following:



  - Although,
    - Such validation or transformation should be better handled by the application programs

Purpose of Using Triggers

- Check complex rules
  - Sometimes, the business rules are so complex that it cannot be checked via declarative constraints

Purpose of Using Triggers

- Manage data redundancy
  - Some redundancy issues may not be avoided by simply adding constraints

  - For example: We inserted the same movie but in different languages

```sql
-- US
insert into movies(title, country, year_released)
values ('The Matrix', 'us', 1999);

-- China (Mainland)
insert into movies(title, country, year_released)
values ('黑客帝国', 'us', 1999);

-- Hongkong
insert into movies(title, country, year_released
values ('22世紀殺人網絡', 'us', 1999);
```

It satisfies the constraint of uniqueness on (`title, country, year_released`)
- ... but they represent the same movie

Trigger Activation

- Two key points:
  - When to fire a trigger?
  - What (command) fires a trigger?

Trigger Activation

- When to fire a trigger?
  - In general: “During the change of data”
    - … but we need a detailed discussion

--- Note: “During the change” means `select` queries won’t fire a trigger.

Trigger Activation: When

- Example: Insert a set of rows with “`insert into select`”
  - One statement, multiple rows

```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais;
```

Trigger Activation: When

- Example: Insert a set of rows with "`insert into select`"
  - One statement, multiple rows

```
insert into movies(title, country, year_released)
select titre, 'fr', annee
from films_francais;
```

  - Option 1: Fire a trigger only once for the statement
    - Before the first row is inserted, or after the last row is inserted
  - Option 2: Fire a trigger for each row
    - Before or after the row is inserted

Trigger Activation: When

- Different options between DBMS products

- Before statement
  - Before each row
  - After each row
- After statement

- ~~Before statement~~
  - Before each row
  - After each row
- ~~After statement~~

- ~~Before statement~~
  - ~~Before each row~~
  - ~~After each row~~
- After statement

Trigger Activation: What

- What (command) fires a trigger?
  - `insert`
  - `update`
  - `delete`

Example of Triggers

- A (Toy) Example
  - For the following `people_1` table, count the number of movies when updating a person and save the result in the `num_movies` column

```sql
-- auto-generated definition
create table people_1
(
    peopleid   integer,
    first_name varchar(30),
    surname    varchar(30),
    born       integer,
    died       integer,
    gender     bpchar,
    num_movies integer
);
```

	peopleid	first_name	surname	born	died	gender	num_movies
1	13	Hiam	Abbass	1960	<null>	F	<null>
2	559	Aleksandr	Askoldov	1932	<null>	M	<null>
3	572	John	Astin	1930	<null>	M	<null>
4	585	Essence	Atkins	1972	<null>	F	<null>
5	598	Antonella	Attili	1963	<null>	F	<null>
6	611	Stéphane	Audran	1932	<null>	F	<null>
7	624	William	Austin	1884	1975	M	<null>
8	637	Tex	Avery	1908	1980	M	<null>
9	650	Dan	Aykroyd	1952	<null>	M	<null>
10	520	Zackary	Arthur	2006	<null>	M	<null>
11	533	Oscar	Asche	1871	1936	M	<null>
12	546	Elizabeth	Ashley	1939	<null>	F	<null>

Example of Triggers

- Create a trigger

```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

Example of Triggers

- Create a trigger

Name of the trigger

```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

Example of Triggers

- Create a trigger

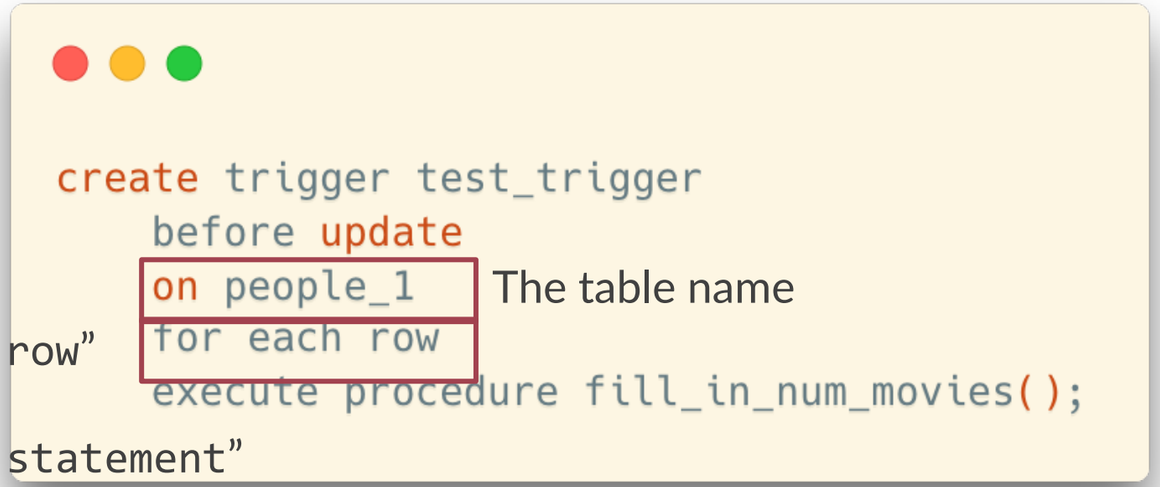
```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

{ BEFORE | AFTER | INSTEAD OF } { event [OR ...] }

- Specify when the trigger will be executed
  - `before` | `after`
- … and on what operations the trigger will be executed
  - `insert [or update [or delete]]`

Example of Triggers

- Create a trigger



Example of Triggers

- Create a trigger

```
create trigger test_trigger
    before update
    on people_1
    for each row
    execute procedure fill_in_num_movies();
```

The actual procedure for the trigger

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

"trigger" is the return type

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

”new” and “old” are two internal variables that represents the row before and after the changes

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well

```
create or replace function fill_in_num_movies()
    returns trigger
as
$$
begin
    select count(distinct c.movieid)
    into new.num_movies
    from credits c
    where c.peopleid = new.peopleid;
    return new;
end;
$$ language plpgsql;
```

Remember to return the result which will be used in the `update` statement

Example of Triggers

- Create a trigger
  - Besides, a corresponding procedure should be created as well
    - Remember to create the procedure before creating the trigger

- Run test updates

```
-- create the procedure fill_in_num_movies() first

-- then, create the trigger

-- finally, we can run some test update statements
update people_1 set num_movies = 0 where people_1.peopleid <= 100;
```

Before and After Triggers

- Differences between before and after triggers
  - “Before” and “after” the operation is done (`insert`, `update`, `delete`)
  - If we want to update the incoming values in an `update` statement, the “before trigger” should be used since the incoming values have not been written to the table yet

Before and After Triggers

- Typical usage scenarios for trigger settings
  - Modify input on the fly
    - before insert / update
    - for each row
  - Check complex rules
    - before insert / update / delete
    - for each row
  - Manage data redundancy
    - after insert / update / delete
    - for each row

Example: Auditing

- One good example of managing some data redundancy is keeping an audit trail
  - It won't do anything for people who steal data
    - (remember that `select` cannot fire a trigger – although with the big products you can trace all queries)
  - … but it may be useful for checking people who modify data that they aren't supposed to modify

Example: Auditing

- Trace the insertions and updates to employees in a company

```
create table company(
   id int primary key     not null,
   name           text    not null,
   age            int     not null,
   address        char(50),
   salary         real
);


create table audit(
   emp_id int not null,
   change_type char(1) not null,
   change_date text not null
);
```

Example: Auditing

- Trace the insertions and updates to employees in a company

```
create trigger audit_trigger
    after insert or update
    on company
    for each row
execute procedure auditlogfunc();


create or replace function auditlogfunc() returns trigger as
$example_table$
begin
    insert into audit(emp_id, change_type, change_date)
    values (new.id,
            case
                when tg_op = 'UPDATE' then 'U'
                when tg_op = 'INSERT' then 'I'
                else 'X'
            end,
            current_timestamp);
    return new;
end ;
$example_table$ language plpgsql;
```

Example: Auditing

- Trace the insertions and updates to employees in a company

```
insert into company (id, name, age, address, salary)
values (2, 'Mike', 35, 'Arizona', 30000.00);
```

company

	id	name	age	address	salary
1	2	Mike	35	Arizona	30000

audit

	emp_id	change_type	change_date
1	2	I	2022-04-25 18:37:35.515151+00