Principles of Database Systems (CS307)

Lecture 5: Advanced SQL

Zhong-Qiu Wang

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

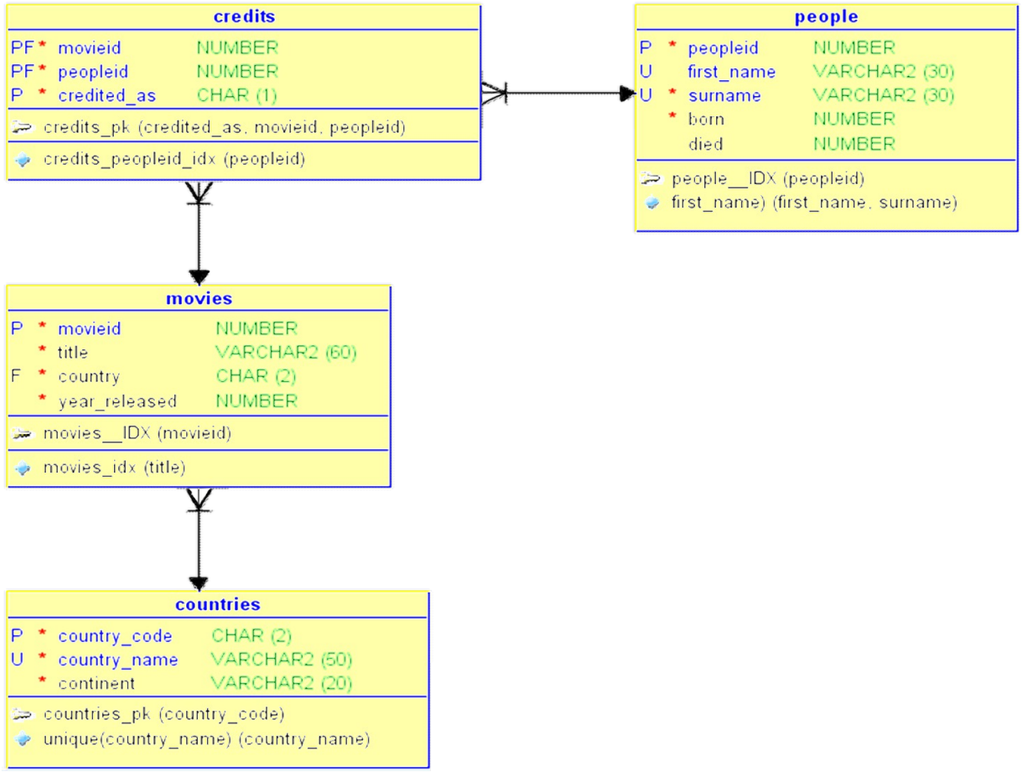
- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

Announcements

- University requires us to check attendance
- Project I is out, due date: 23:59 on November 10th 2024, Beijing Time
  - First find a teammate, and form a team
    - Each team should have two members
  - Publication management system
    - Design a database with relational tables
    - Provide the E-R diagram
    - Import provided data into the database
    - Compare select, update, delete & insert operations between using database and file I/O
  - Project II will be based on the database you created in Project I

Entity and Relationship

- Starring -> Actor table
- Country -> Country and Region table
  - You can also link the movies with corresponding actors, countries/regions, etc.
- Entity Relationship Diagram (E/R Diagram, ER Diagram, ERD)
  - A way of representing entity tables and their relationships (relationship tables)
  - Connect tables via foreign keys and relationship tables



Create Tables

- An SQL relation is defined using the create table command:

```
create table r (
    A1 D1, A2 D2, ..., An Dn,
    (integrity-constraint1),
    ...,
    (integrity-constraintk)
)
```

  - Relation schema: r is the name of the relation
  - Attribute: each Ai is an attribute name in the schema of relation r
  - Data type: Di is the data type of values in the domain of attribute Ai
  - Constraints: not null, unique, primary key, check function, referential integrity & foreign key

Select

- `select * from` [tablename]
  - The `select` clause lists the attributes desired in the result of a query
  - To display the full content of a table, you can use select *
    - * : all columns

```
select A1, A2, ..., An
 from r1, r2, ..., rm
       where P
```

Some Functions

- When to use functions
  - An example of showing a result that isn't stored as such is computing an age
    - You should never store an age; it changes all the time!
    - If you want to display the age of people who are alive, you must compute their age by subtracting the year when they were born from the current year.

  - In the table people:
    - Alive – died is `null`
    - Age: `<this year> - born`

```
select peopleid, surname,
       date_part('year', now()) - born as age
from people
where died is null;
```

7	7	Caroline	Aaron	1952	<null>	F
8	8	Quinton	Aaron	1984	<null>	M
9	9	Dodo	Abashidze	1924	1990	M

Aggregate Functions

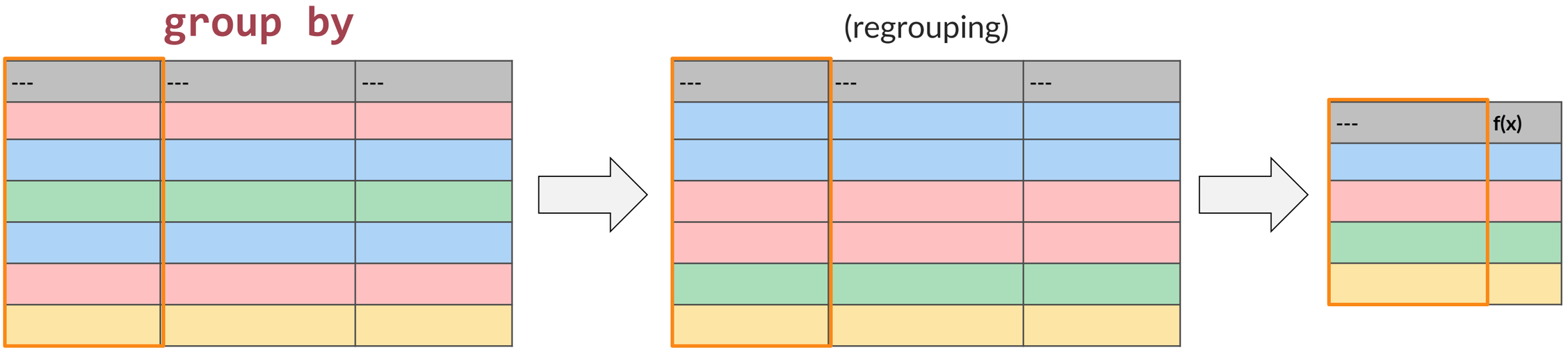
`count(*)/count(col), min(col), max(col), stddev(col), avg(col)`

- These aggregate functions are supported in almost every DBMS
  - Most DBMS implement other more advanced functions
  - Some functions work with any datatype, others only work with numerical columns

  - It is strongly recommended to refer to user manual for details
    - For example, SQLite doesn't have `stddev()`

Aggregate Functions

- Beware of some performance implications
  - With a `group by`, you must regroup rows before you can aggregate them and return results.
    - In other words, you have a preparatory phase that may take time, even if you return few rows in the end.
    - In interactive applications, end-users don't always understand it well.



Retrieving Data from Multiple Tables

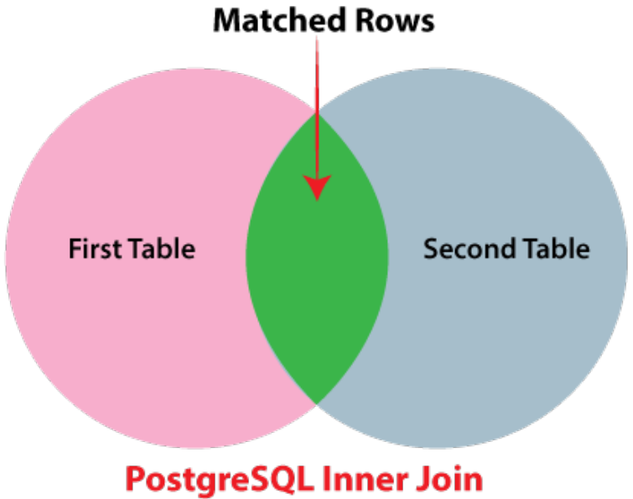
- This is done with this type of query. We retrieve, and display as a single set, pieces of data coming from two different tables.

```
select title,
    country_name,
    year_released
    from movies
    join countries
    on country_code = country;
```

title	country_name	year_released
12 stulyev	Russia	1971
Al-mummia	Egypt	1969
Ali Zaoua, prince de la rue	Morocco	2000
Apariencias	Argentina	2000
Ardh Satya	India	1983
Armaan	India	2003
Armaan	Pakistan	1966
Babettes gæstebud	Denmark	1987
Banshun	Japan	1949
Bidaya wa Nihaya	Egypt	1960
Variety	United States	2008
Bon Cop, Bad Cop	Canada	2006
Brilliantovaja ruka	Russia	1969
C'est arrivé près de chez vous	Belgium	1992
Carlota Joaquina - Princesa do Brasil	Brazil	1995
Cicak-man	Malaysia	2006
Da Nao Tian Gong	China	1965
Das indische Grabmal	Germany	1959
Das Leben der Anderen	Germany	2006
Den store gavtyv	Denmark	1956

Inner and Outer Joins

- Inner join
  - Join type in default
  - All examples so far are inner joins
  - Only select joined rows with matching values
    - Typically, a subset of rows in left table are joined with a subset of rows in right table
    - Some rows in left table may NOT be joined with any rows in right table
      - E.g., for movies released by a country that is not in the countries table




```
select title,
    country_name,
    year_released
    from movies
    join countries
    on country_code = country;
```

Inner and Outer Joins

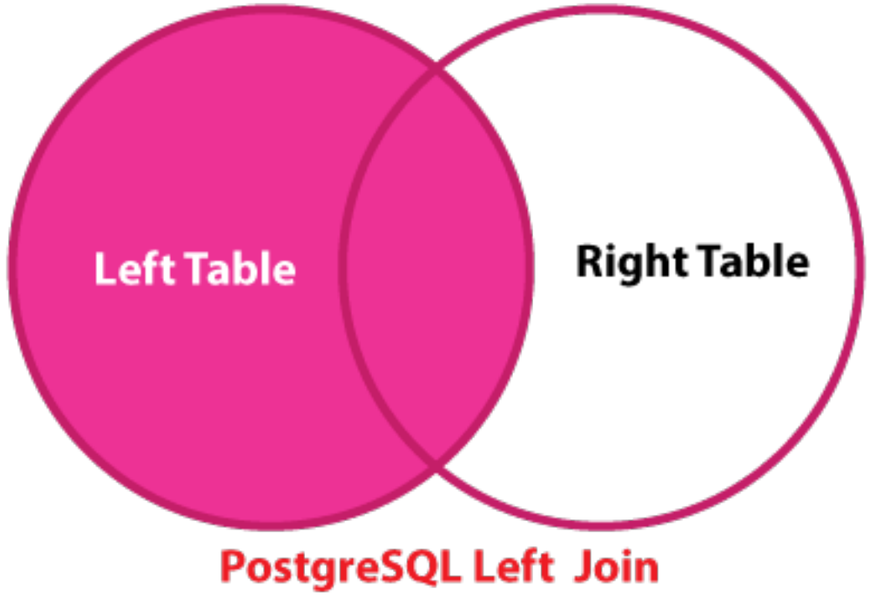
- Left outer join
  - Example: there is a movie in 2018 where there is no credit information
    - #9203 (A Wrinkle in Time)
    - Inner join of all 2018 movies will not show any matching results for that movie

```
select *
from movies m join credits c
on m.movieid = c.movieid
where m.year_released = 2018;
```

	m.movieid	title	country	year_released	runtime	c.movieid	peopleid	credited_as
1	8987	Red Sparrow	us	2018	145	8987	4062	A
2	8987	Red Sparrow	us	2018	145	8987	6711	A
3	8987	Red Sparrow	us	2018	145	8987	8308	D
4	8987	Red Sparrow	us	2018	145	8987	8310	A
5	8987	Red Sparrow	us	2018	145	8987	11247	A
6	8987	Red Sparrow	us	2018	145	8987	12048	A
7	8987	Red Sparrow	us	2018	145	8987	13071	A
8	8988	Ready Player One	us	2018	0	8988	2934	A
9	8988	Ready Player One	us	2018	0	8988	9819	A
10	8988	Ready Player One	us	2018	0	8988	9971	A
11	8988	Ready Player One	us	2018	0	8988	11390	A
12	8988	Ready Player One	us	2018	0	8988	12758	A
13	8988	Ready Player One	us	2018	0	8988	13421	A
14	8988	Ready Player One	us	2018	0	8988	13850	D
15	8989	Guernsey	gb	2018	0	8989	1864	A
16	8989	Guernsey	gb	2018	0	8989	5280	A
17	8989	Guernsey	gb	2018	0	8989	6523	A
18	8989	Guernsey	gb	2018	0	8989	6836	A
19	8989	Guernsey	gb	2018	0	8989	10643	D
20	8989	Guernsey	gb	2018	0	8989	11261	A
21	8989	Guernsey	gb	2018	0	8989	11733	A
22	8989	Guernsey	gb	2018	0	8989	15708	A
23	8990	A Star Is Born	us	2018	0	8990	2431	A
24	8990	A Star Is Born	us	2018	0	8990	2759	A
25	8990	A Star Is Born	us	2018	0	8990	2939	A
26	8990	A Star Is Born	us	2018	0	8990	2939	D
27	8990	A Star Is Born	us	2018	0	8990	4158	A
28	8990	A Star Is Born	us	2018	0	8990	8105	A
29	8992	Mary Queen of Scots	us	2018	0	8992	272	A
30	8992	Mary Queen of Scots	us	2018	0	8992	2879	A
31	8992	Mary Queen of Scots	us	2018	0	8992	3056	A
32	8992	Mary Queen of Scots	us	2018	0	8992	3365	A
33	8992	Mary Queen of Scots	us	2018	0	8992	8892	A
34	8992	Mary Queen of Scots	us	2018	0	8992	11371	A
35	8992	Mary Queen of Scots	us	2018	0	8992	12435	A
36	8992	Mary Queen of Scots	us	2018	0	8992	12563	A
37	8992	Mary Queen of Scots	us	2018	0	8992	12636	D
38	8993	The Girl in the Spider's Web	se	2018	0	8993	4696	A
39	8993	The Girl in the Spider's Web	se	2018	0	8993	5543	A
40	8993	The Girl in the Spider's Web	se	2018	0	8993	16462	D
41	9202	Black Panther	us	2018	134	9202	3933	A
42	9202	Black Panther	us	2018	134	9202	5588	A
43	9202	Black Panther	us	2018	134	9202	15870	A

Inner and Outer Joins

- Left outer join
  - All the matching rows will be selected
  - … and the rows in **left table** with **no matches** will be **selected** as well




```
select columns
from table1
LEFT [OUTER] join table2
on table1.column = table2.column;
```

Inner and Outer Joins

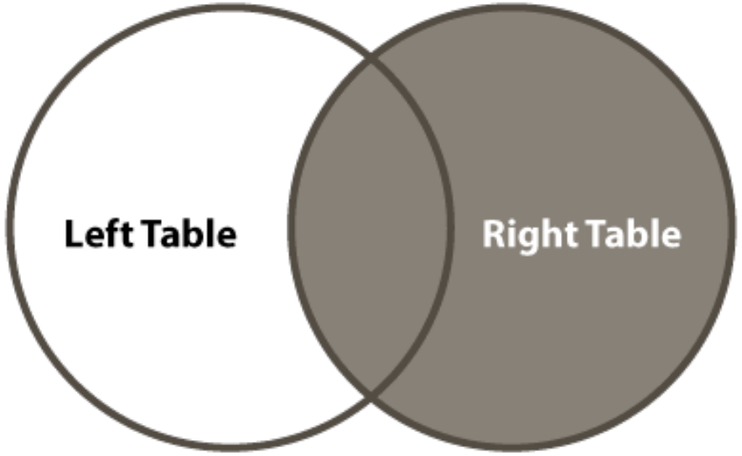
- Left outer join
  - Example: there is a movie in 2018 where there is no credit information
    - #9203 (A Wrinkle in Time)
    - Inner join of all 2018 movies will not show any matching results for that movie
    - Left (outer) join can give you a row for the movie (in the left table) where all right-table columns are null

```
select * from movies m left join credits c on m.movieid = c.movieid
where m.year_released = 2018;
```

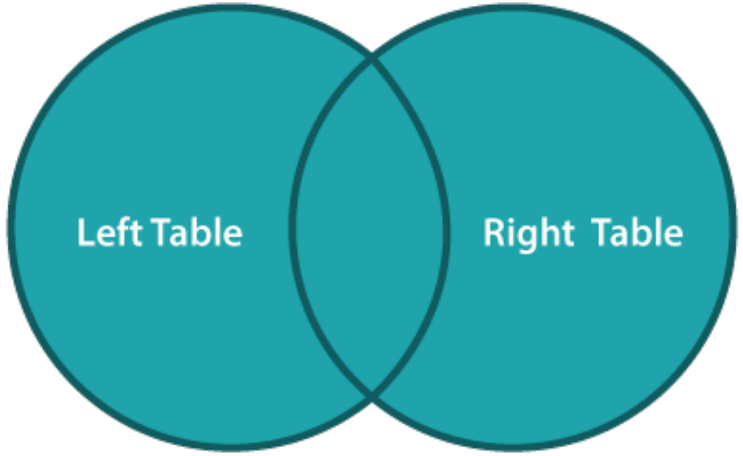
	m.movieid	title	country	year_released	runtime	c.movieid	peopleid	credited_as
41	9202	Black Panther	us	2018	134	9202	3933	A
42	9202	Black Panther	us	2018	134	9202	5588	A
43	9202	Black Panther	us	2018	134	9202	15870	A
44	9203	A Wrinkle in Time	us	2018	109	<null>	<null>	<null>

Inner and Outer Joins

- Right outer join, full outer join
  - Books always refer to three kinds of outer joins. Only one is useful and we can forget about anything but the LEFT OUTER JOIN
    - A right outer join can ALWAYS be rewritten as a left outer join
      - by swapping the order of tables in the join list
    - A full outer join is seldom used



PostgreSQL Right Join



PostgreSQL Full Outer Join

Subquery

- We have used subqueries after the keyword `from` before
  - … in order to build queries upon a query result

- And, we can add subqueries after `select` and `where` as well

```
select A1, A2, ..., An
 from r1, r2, ..., rm
      where P
```

Subquery after Select

- Example: show titles, released years, and country names for non-US movies
  - Solution 2: Nested selection

```
select m.title,
       m.year_released,
       m.country
from movies m
where m.country <> 'us';
```

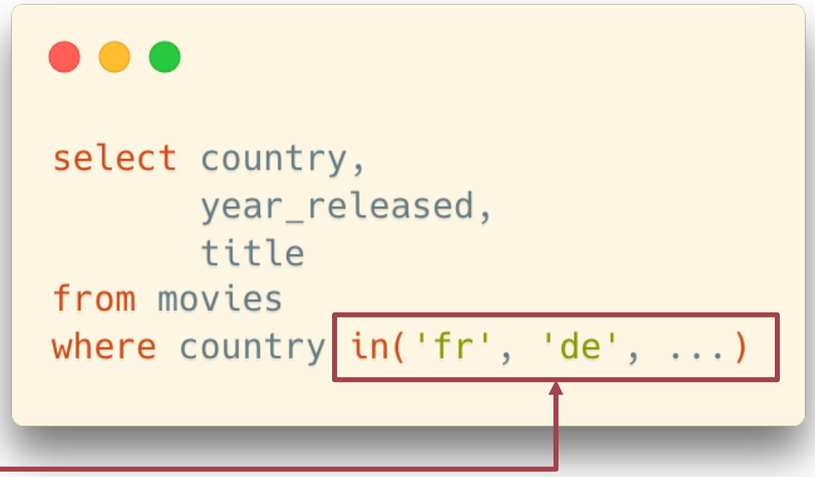
```
select m.title,
       m.year_released,
       (
           select c.country_name
           from countries c
           where c.country_code = m.country
       ) country_name
from movies m
where m.country <> 'us';
```

A subquery after select:

- For each selected row in the outer query, find the corresponding country name in the `countries` table

Subquery after Where

- Example: Select all European movies
  - A proper solution: (dynamically) fill in the list of country codes in an `in` operator



The same results (if you fill in all European country codes on the right side)

- But you can automatically generate this list
- Especially useful when the table in the subquery changes often

Advanced Ordering

- Multiple columns
  - The result set will be order by `col1` first
  - Rows with the same value on `col1` will be ordered by `col2`

```
order by col1, col2, ...
```

- Ascending or descending order
  - Add `desc` or `asc` after the column
    - `asc` is the default option and thus always omitted

```
-- Order col1 descendingly
order by col1 desc

-- Order based on col1 first, then col2.
-- col1 will be in the descending order, col2 ascending.
order by col1 desc, col2 asc, ...
```

Window Function

Scalar Functions and Aggregation Functions

- Scalar function
  - Functions that operate on values in the current row

```
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'
```

- Aggregation function
  - Functions that operate on sets of rows and return an aggregated value

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

Issues with Aggregate Functions

- A Problem: In aggregated functions, the details of the rows are vanished
  - For example: if we ask for the year of the oldest movie per country
    - We get a country, a year, and nothing else.

```
select country,
       min(year_released) earliest_year
from movies
group by country
```

country	oldest_movie
fr	1896
ke	2008
si	2000
eg	1949
nz	1981
bg	1967
ru	1924
gh	2012
pe	2004
hr	1970
sg	2002
mx	1933
cn	1913
ee	2007
sp	1933
cl	1926
ec	1999
cz	1949
dk	1910
vn	1992
ro	1964
mn	2007
gb	1916
se	1913
tw	1971
ie	1970
ph	1975
ar	1945
th	1971

Issues with Aggregate Functions

- A Problem: In aggregated functions, the details of the rows are vanished
  - For example: if we ask for the year of the oldest movie per country
    - We get a country, a year, and nothing else

If we want some more details, like the title of the oldest movies for each country

- We can only use self-join to get the title information
- The join conditions include (1) same country code; and (2) same release year

```
select m1.country,
       m1.title,
       m1.year_released
from movies m1
     inner join
     (select country,
        min(year_released) minyear
        from movies
        group by country) m2
on m2.country = m1.country and m2.minyear = m1.year_released
order by m1.country
```

Issues with Aggregate Functions

- A Problem: In aggregated functions, the details of the rows are vanished
  - For example: if we ask for the year of the oldest movie per country
    - We get a country, a year, and nothing else

If we want some more details, like the title of the oldest movies for each country

- What if the title of the second oldest movie is what we want?

```
select m1.country,
       m1.title,
       m1.year_released
from movies m1
     inner join
     (select country,
        min(year_released) minyear
        from movies
        group by country) m2
on m2.country = m1.country and m2.minyear = m1.year_released
order by m1.country
```

Issues with Aggregate Functions

- A Problem
  - In aggregated functions, the details of the rows are vanished
  - Another example
    - How can we rank the movies in each country separately based on the released year?
    - “order by” for subgroups
  - One more example
    - Get the top-3 oldest movies for each country
    - How can we implement it?

Window Function

- Syntax:

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

  - `<function>`
    - Ranking window functions
    - Aggregation functions
  - `partition by`
    - Specify the column for grouping
  - `order by`
    - Specify the column(s) for ordering in each group

Ranking Window Function

- Example
  - How can we rank the movies in each country separately based on the released year?
    - "order by" for subgroups

```
select country,
       title,
       year_released,
       rank() over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

Partitioned by country
- i.e., a country in a group

country	title	year_released	oldest_movie_per_country
ar	some title	1948	1
ar	some title	1959	2
ar	some title	1980	3
cn	some title	1987	1
cn	some title	2002	2
uk	some title	1985	1
uk	some title	1992	2
uk	some title	2010	3

`rank()`
- A function to say that "I want to order the rows in each partition"
- No parameters in the parentheses

`order by year_released`
- In each group (partition), the rows will be ordered by the column "`year_released`"

An order value is computed for each row in a partition.
- Only inside the partition, not across the entire result set

Ranking Window Function

- Example
  - How can we rank the movies in each country separately based on the released year?
    - “order by” for subgroups

```
select country,
       title,
       year_released,
       rank() over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

country	title	year_released	oldest_movie_per_country
ar	some title	1948	1
ar	some title	1959	2
ar	some title	1980	3
cn	some title	1987	1
cn	some title	2002	2
uk	some title	1985	1
uk	some title	1992	2
uk	some title	2010	3

Note: partition functions can only be used in the `select` clause

- … since it is designed to work on the query result

Ranking Window Function

- Example
  - How can we rank the movies in each country separately based on the released year?
    - “order by” for subgroups

```
select country,
       title,
       year_released,
       rank() over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

You can also add “desc” here, similar to the “order by” we introduced before

country	title	year_released	oldest_movie_per_country
ar	some title	1948	1
ar	some title	1959	2
ar	some title	1980	3
cn	some title	1987	1
cn	some title	2002	2
uk	some title	1985	1
uk	some title	1992	2
uk	some title	2010	3

Ranking Window Function

- Why window function, not group by?
  - “Group by” reduces the rows in a group (partition) into one result, which is the meaning of “aggregation”
    - Then, the values in non-aggregating columns are vanished
  - Window functions do not reduce the rows
    - Instead, they attach computed values next to the rows in a group (partition) and keep the details
    - Actually, the partition here means “window”: an affective range for statistics

Ranking Window Function

- Some more ranking window functions
  - Besides `rank()`, we also have `dense_rank()` and `row_number()`
  - The difference is about how they treat rows with the same rank

```
select country,
       title,
       year_released,

       rank() over (
           partition by country order by year_released
       ) rank_result,

       dense_rank() over (
           partition by country order by year_released
       ) dense_rank_result,

       row_number() over (
           partition by country order by year_released
       ) row_number_result
from movies;
```

country	title	year_released	rank_result	dense_rank_result	row_number_result
cn	some title	1948	1	1	1
cn	some title	1959	2	2	2
cn	some title	1959	2	2	3
cn	some title	1987	4	3	4
cn	some title	2002	5	4	5
uk	some title	1985	1	1	1
uk	some title	1992	2	2	2
uk	some title	2010	3	3	3

Aggregation Functions as Window Functions

- Aggregation functions can be used with window functions
  - `min/max(col)` , `sum(col)`, `count(col)`, `avg(col)`, `stddev(col)`, etc.
  - **If `order by` is specified,** return aggregation value from the first row to current row in each partition
  - **If not**, fill all rows with a single aggregation value computed on all rows in each partition

```
select country,
       title,
       year_released,
       sum(runtime) over (
           partition by country order by year_released
       ) total_runtime_till_this_row
from movies;
```

Need to specify a column in the parameter list

	country	title	year_released	total_runtime_till_this_row
1	am	Sayat Nova	1969	78
2	ar	Pampa bárbara	1945	98
3	ar	Albéniz	1947	308
4	ar	Madame Bovary	1947	308
5	ar	La bestia debe morir	1952	494
6	ar	Las aguas bajan turbias	1952	494
7	ar	Intermezzo criminal	1953	494
8	ar	La casa del ángel	1957	570
9	ar	Bajo un mismo rostro	1962	695
10	ar	Las aventuras del Capitán Piluso	1963	785
11	ar	Savage Pampas	1966	897
12	ar	La hora de los hornos	1968	1157
13	ar	Waiting for the Hearse	1985	1354
14	ar	La historia oficial	1985	1354

Pay attention to the behavior on rows with the same rank:

- They are “treated like the same row” here

Aggregation Functions as Window Functions

- Aggregation functions can be used with window functions
  - `min/max(col)` , `sum(col)`, `count(col)`, `avg(col)`, `stddev(col)`, etc.
  - **If `order by` is specified,** return aggregation value from the first row to current row in each partition
  - **If not**, fill all rows with a single aggregation value computed on all rows in each partition

```
select country,
       title,
       year_released,
       min(year_released) over (
           partition by country order by year_released
       ) oldest_movie_per_country
from movies;
```

	cou...	title	year_released	oldest_movie_per_country
1	am	Sayat Nova	1969	1969
2	ar	Pampa bárbara	1945	1945
3	ar	Albéniz	1947	1945
4	ar	Madame Bovary	1947	1945
5	ar	La bestia debe morir	1952	1945
6	ar	Las aguas bajan turbias	1952	1945
7	ar	Intermezzo criminal	1953	1945
8	ar	La casa del ángel	1957	1945
9	ar	Bajo un mismo rostro	1962	1945
10	ar	Las aventuras del Capitán Piluso	1963	1945
11	ar	Savage Pampas	1966	1945
12	ar	La hora de los hornos	1968	1945
13	ar	Waiting for the Hearse	1985	1945
14	ar	La historia oficial	1985	1945

Exercise

- Question: How can we get the top-5 most recent movies for each country?
  - Hint: Use a subquery in the "`from`" clause

```
select x.country,
       x.title,
       x.year_released
from (
 select country,
        title,
        year_released,
        row_number()
        over (partition by country
              order by year_released desc) rn
 from movies) x
where x.rn <= 5
```

Update and Delete

So Far...

- We have learned:
  - How to access existing data in tables (select)
  - How to create new rows (insert)

- CRUD/CURD
  - create, read, update, delete
    - In SQL: insert, select, update, delete
    - In RESTful API: Post, Get, Put, Delete
  - Necessary operations for persistent storage

Update

- Make changes to the existing rows in a table
- `update` is the command that changes column values
  - You can even set a non-mandatory column to NULL
  - The change is applied to all rows selected by the `where`

```
update table_name
set column_name = new_value,
    other_col = other_new_val,
    ...
where ...
```

Update

- Remember
  - When you are doing any experiments with writing operations (update, delete), **backup the data first**
    - E.g., copy the tables

Update

- Example: In many Western cultures, a nobiliary particle is used in a surname or family name to signal the nobility of a family
  - We may want to modify some names in such a way as they sort as they should
    - `von Neumann -> Neumann (von)`

23 rows | Tx: Auto | DDL

	peopleid	first_name	surname	born	died	gender
1	16439	Axel	von Ambesser	1910	1988	M
2	16440	Daniel	von Bargen	1950	2015	M
3	16441	Eduard	von Borsody	1898	1970	M
4	16442	Suzanne	von Borsody	1957	<null>	F
5	16443	Tomas	von Brömssen	1943	<null>	M
6	16444	Erik	von Detten	1982	<null>	M
7	16445	Theodore	von Eltz	1893	1964	M
8	16446	Gunther	von Fritsch	1906	1988	M
9	16447	Katja	von Garnier	1966	<null>	F
10	16448	Harry	von Meter	1871	1956	M
11	16449	Jenna	von Oÿ	1977	<null>	F
12	16450	Alicia	von Rittberg	1993	<null>	F
13	16451	Daisy	von Scherler Mayer	1966	<null>	F
14	16452	Gustav	von Seyffertitz	1862	1943	M
15	16453	Josef	von Sternberg	1894	1969	M

John von Neumann

Update

- Example: In many Western cultures, a nobiliary particle is used in a surname or family name to signal the nobility of a family
  - We may want to modify some names in such a way as they sort as they should
    - `von Neumann -> Neumann (von)`

- First, how can we find these names?

Update

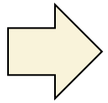
- Example: In many Western cultures, a nobiliary particle is used in a surname or family name to signal the nobility of a family
  - We may want to modify some names in such a way as they sort as they should
    - **`von Neumann -> Neumann (von)`**
- First, how can we find these names?
  - Wildcards
  - Strings starting with “von ”

```
select * from people_1 where surname like 'von %';
```

Update

- Example: In many Western cultures, a nobiliary particle is used in a surname or family name to signal the nobility of a family
  - We may want to modify some names in such a way as they sort as they should.
    - `von Neumann -> Neumann (von)`
- Then, how should we update the names?

```
(first_name) John
(surname) von Neumann
```



```
(first_name) John
(surname) Neumann (von)
```

  - Try the transformation with select:

```
select replace('von Neumann', 'von ', '') || ' (von)';
```

	?column?
1	Neumann (von)

Update

- Example: In many Western cultures, a nobiliary particle is used in a surname or family name to signal the nobility of a family
  - We may want to modify some names in such a way as they sort as they should.
    - **`von Neumann -> Neumann (von)`**
- Finally, the `update` statement:

```
-- Specify the table
update people

-- Set the update rule
set surname = replace(surname, 'von ', '') || ' (von)'

-- Find the rows that need to be updated
where surname like 'von %';
```

Update

- The `where` clause specifies the affected rows
  - If `where` is not provided, the update will be applied to all rows in the table
    - Use with caution!
    - Sometimes, there will be a warning in IDEs such as DataGrip

Update

- The update operation may not be successful when constraints are violated
  - E.g., update the primary key but with duplicated values

```
update people set peopleid = 1 where peopleid < 10;
```

```
[23505] ERROR: duplicate key value violates unique constraint "people_pkey"
Detail: Key (peopleid)=(1) already exists.
```

  - This is why we need constraints when creating tables
    - Avoid unacceptable writing operations that break the integrity of the tables

Update

- Use Subqueries in update
  - Complex update operations where values are based on a query result
- Example: Add a column in people table to record the number of movies one has joined (either directed or played a role in)

```
update table_name
set column_name = new_value,
    other_col = other_new_val,
    ...
where ...
```

Update

- Example: Add a column in people table to record the number of movies one has joined (either directed or played a role in)
  - First, count the movies for a person
    - Used as the subquery part in the update statement

```
select count(*) from credits c where c.peopleid = [some peopleid];
```

Update

- Example: Add a column in people table to record the number of movies one has joined (either directed or played a role in)
  - First, count the movies for a person
    - Used as the subquery part in the update statement
  - Then, update

```
update people p

set num_movies = (
    select count(*) from credits c where c.peopleid = p.peopleid
)

where peopleid < 500;
-- This where is only for testing purpose;
-- You should change it (or remove it) when in actual use.
```

Delete

- As the name shows, `delete` removes rows from tables

  - If you omit the WHERE clause, then (as with UPDATE) the statement affects all rows and you **end up with an empty table!**
  - Well,
    - Many database products provide a roll-back mechanism when deleting rows
    - Transactions can also protect you (to some extent)

Constraints

- One important point with constraints (esp., foreign keys) is that they guarantee that data remains consistent
  - They not only work with `insert`, but with `update` and `delete`
  - Example: Try to delete some rows in the `country` table
    - Foreign-key constraints are especially useful in controlling `delete` operations

```
delete from countries where country_code = 'us';
```

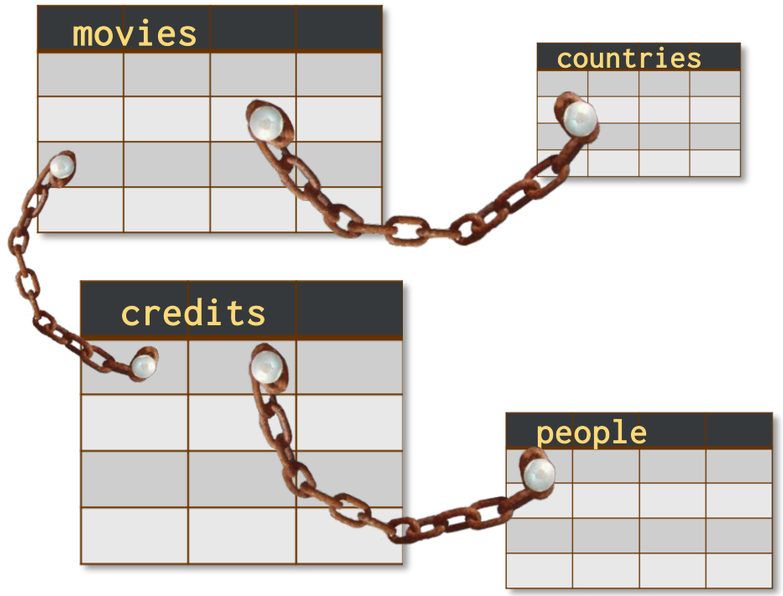
```
[23503] ERROR: update or delete on table "countries" violates foreign key constraint "movies_country_fkey" on table "movies"
Detail: Key (country_code)=(us) is still referenced from table "movies".
```

movieid	title	country	year_released
1	Casab	us	1942
2	Goodfellas	us	1990
3	Bronenosets Potyomkin	ru	1925

country_code	country_name	continent
ru	Russia	Europe
us	United States	America

Constraints

- This is why constraints are so important:
  - They ensure that whatever happens, you'll always be able to make sense of ALL pieces of data in your database
  - For any changes made to the tables, all declared constraints need to be satisfied



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 strong enough

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

Procedural vs. Declarative

- Two different programming paradigms
  - Imperative programming （命令式编程）
    - Describe the algorithms step-by-step (i.e., how to do)
    - Procedural （过程式）: C (and many other legacy languages)
    - Object-oriented: Java
  - Declarative programming（声明式编程）
    - Describe the result without specifying the detailed steps (i.e., 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:

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
- Problems 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) 

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
    - **`von Neumann -> Neumann (von)`**

	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
  - `i.e., Neumann (von) -> von Neumann`

Function in (Postgre)SQL

- Question: How can we restore the format into “**first_name von surname**”?
  - String operations
  - `i.e., Neumann (von) -> von Neumann`

```
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     like '%(von)';
```

Function in (Postgre)SQL

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

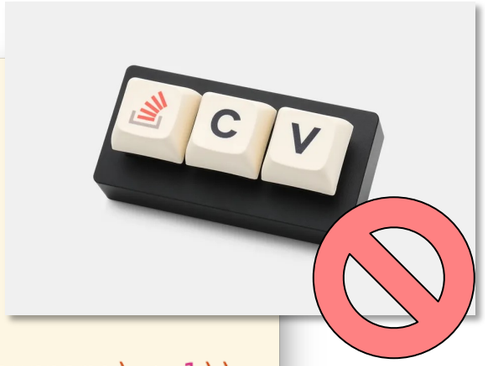
Then, how can we store this part to reuse it in the future?

```
       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     like '%(von)';
```

Function in (Postgre)SQL

- Copy and paste 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

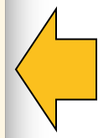
```
case
    when first_name is null then ''
    else first_name || ' '
end || case position('(' in surname)
    when 0 then surname
    else trim(')' from substr(surname, position('(' in surname) + 1))
        || ' '
        || trim(substr(surname, 1, position('(' in surname) - 1))
end
```



Function in (Postgre)SQL

- Stored 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;
```



```
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

...or, a simpler version

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 features 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 $$
begin
    return case
        when p_fname is null
        else p_fname || ' '
    end || case position('(' in p_sname)
        when 0 then p_sname
        else trim(')' from substr(p_sname, position('(' in p_sname) + 1))
        || ' '
        || trim(substr(p_sname, 1, position('(' in p_sname) - 1))
    end;
end;
$$ 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

```
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;
```

i.e., same type as countries.country_name

```
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

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

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

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

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
  - Be careful when seeing 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
    - Treats 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 server
    - Benefit 2: Security
      - Prevent users from modifying data otherwise than by calling carefully written and well tested procedures
      - Users can only modify data via carefully written and well tested procedures

Example: Adding a New Movie

- The information provided for a new movie:
  - Title
  - Year
  - Country Name
    - Note: Name, not code. Country codes are not user-friendly
      - E.g. Which country does “at” represent? ”al”? “ma”? “li”?
  - Director
  - Actor 1
  - Actor 2
    - Let’s assume that 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 communication between client program and 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

- Use `insert into select`

```
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

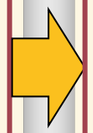
- Use `insert into select`
- Combine the queries of `people`

```
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
...
```



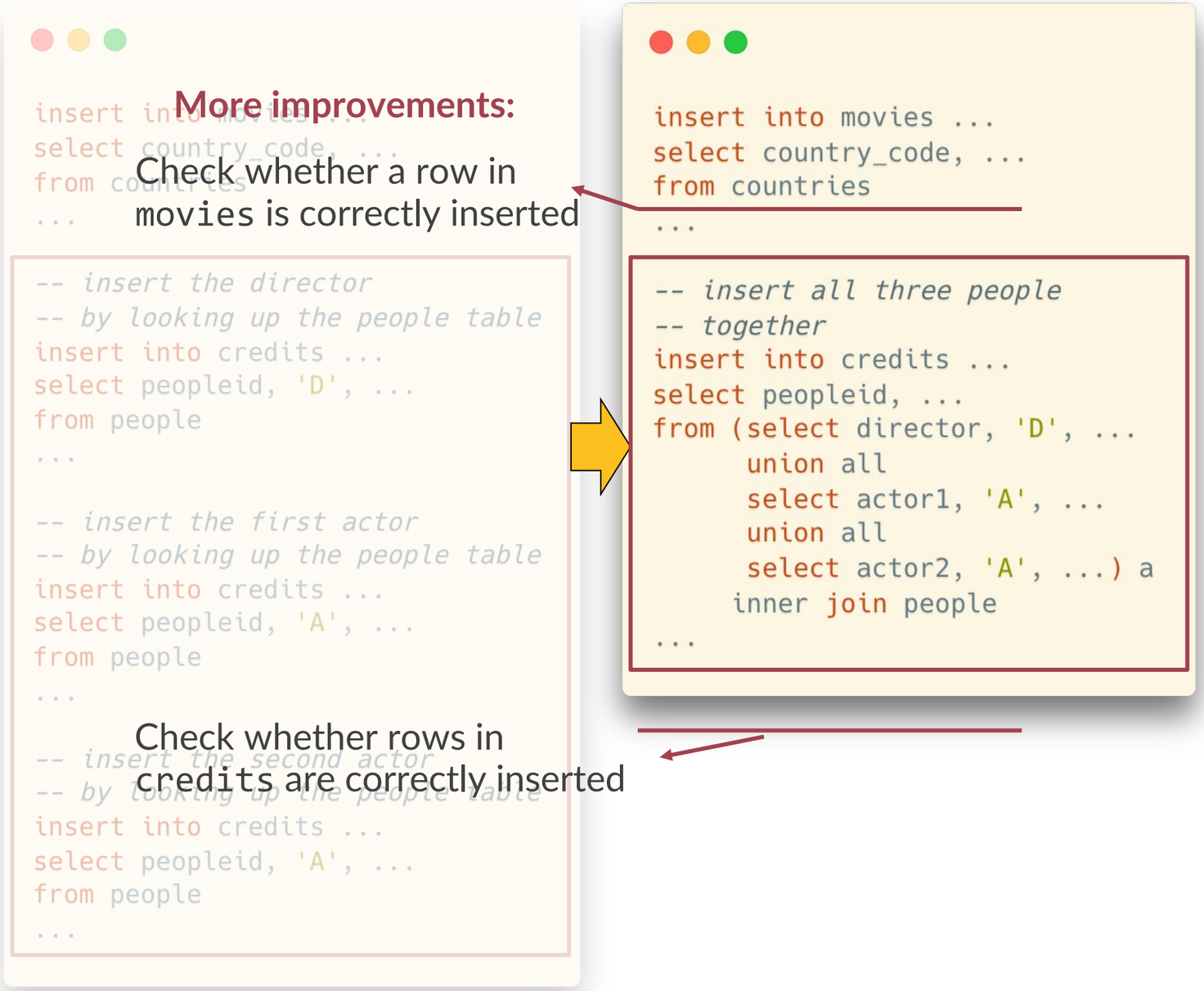
```
insert into movies ...
select country_code, ...
from countries
...

-- 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

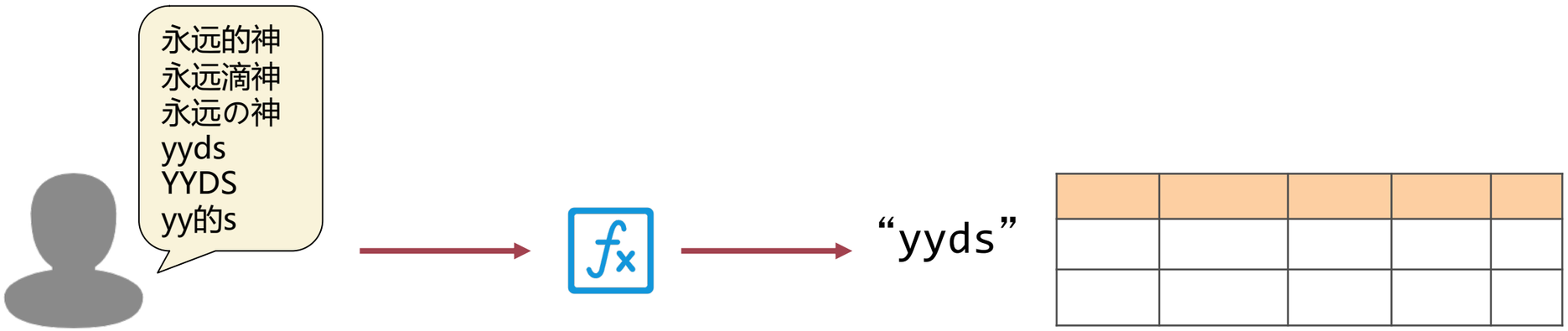
> A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed.
>
> -- Chapter 39, PostgreSQL Documentation
>
> A trigger is a statement that the system executes automatically as a side effect of a modification to 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
  - Validating data
  - Checking complex rules
  - Managing data redundancy

Purpose of Using Triggers

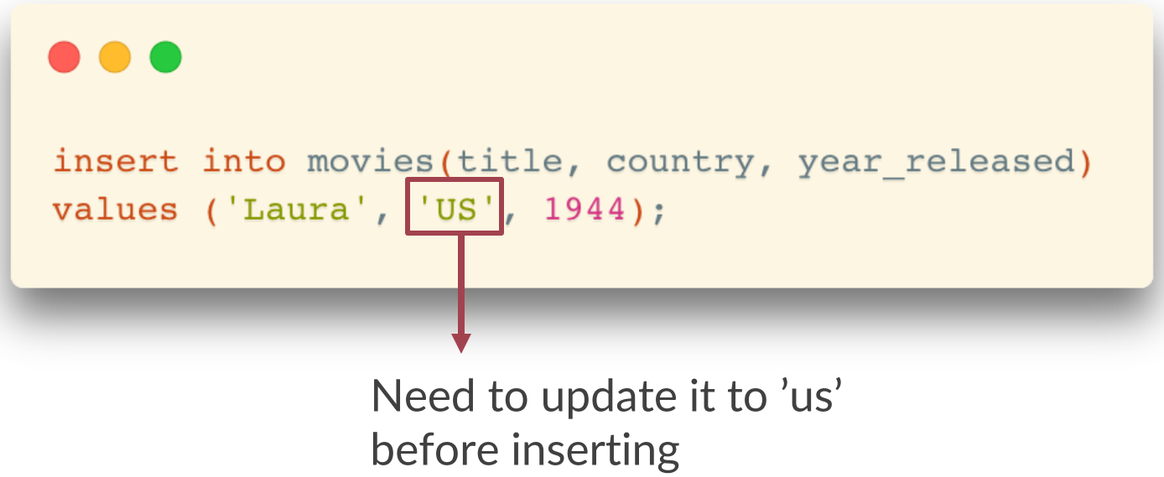
- Validating data
  - Some data are badly processed in programs before sending to the database
  - We need to validate such data before inserting them into the database

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



Purpose of Using Triggers

- Validating 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

- Checking complex rules
  - Sometimes, the business rules are so complex that they CANNOT be checked via declarative integrity constraints

Purpose of Using Triggers

- Managing 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 unique constraint 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 `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

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

The table name

“for each row”
or
“for each statement”
(default)

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

View

Recall: Function

- Used for returning numbers, strings, dates, etc.
  - Or, return a single value

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

... returns a string of the full name

View

- Reuse of the relational operations (as we reuse codes in a program)
  - ... which returns relations (tables) instead of simple values

View

- Example: Create a view like this:

```
create view vmovies as
select m.movieid,
       m.title,
       m.year_released,
       c.country_name
from movies m join countries c
    on c.country_code = m.country;
```

View

- Example: Create a view like this:
  - … where the result of the inner `select` query looks like this:

```
create view vmovies as
select m.movieid,
       m.title,
       m.year_released,
       c.country_name
from movies m join countries c
    on c.country_code = m.country;
```

	movieid	title	year_released	country_name
1	1	12 stulyev	1971	Russia
2	2	Al-mummia	1969	Egypt
3	3	Ali Zaoua, prince de la rue	2000	Morocco
4	4	Apariencias	2000	Argentina
5	5	Ardh Satya	1983	India
6	6	Armaan	2003	India
7	7	Armaan	1966	Pakistan
8	8	Babettes gæstebud	1987	Denmark
9	9	Banshun	1949	Japan
10	10	Bidaya wa Nihaya	1960	Egypt
11	11	Variety	2008	United States
12	12	Bon Cop, Bad Cop	2006	Canada
13	13	Brilliantovaja ruka	1969	Russia
14	14	C'est arrivé près de chez vous	1992	Belgium
15	15	Carlota Joaquina - Princesa d…	1995	Brazil
16	16	Cicak-man	2006	Malaysia
17	17	Da Nao Tian Gong	1965	China
18	18	Das indische Grabmal	1959	Germany
19	19	Das Leben der Anderen	2006	Germany
20	20	Den store gavtyv	1956	Denmark

View vs. Table

- In practice, there isn't much to a view
  - It’s basically a named query
  - And, why not just consider it as a “virtual table”?

```
create view vmovies(v_id, v_title, v_year, v_country) as
select m.movieid,
       m.title,
       m.year_released,
       c.country_name
from movies m
         join countries c
                    on c.country_code = m.country;
```

View vs. Table

- In practice, there isn't much to a view
  - It’s basically a named query
  - And, why not just consider it as a “virtual table”?

```
create view vmovies(v_id, v_title, v_year, v_country) as
select m.movieid,
       m.title,
       m.year_released,
       c.country_name
from movies m
         join countries c
                    on c.country_code = m.country;
```

The columns can be renamed

- Otherwise, the original names in the tables will be used

View vs. Table

- In practice, there isn't much to a view
  - It’s basically a named query
  - And, why not just consider it as a “virtual table”?

The columns can be renamed

- Otherwise, the original names in the tables will be used

```
create view vmovies(v_id, v_title, v_year, v_country) as
select m.movieid,
       m.title,
       m.year_released,
       c.country_name
from movies m
         join countries c
                      on c.country_code = m.country;
```

Drop the existing view before creating a new one with the same name:

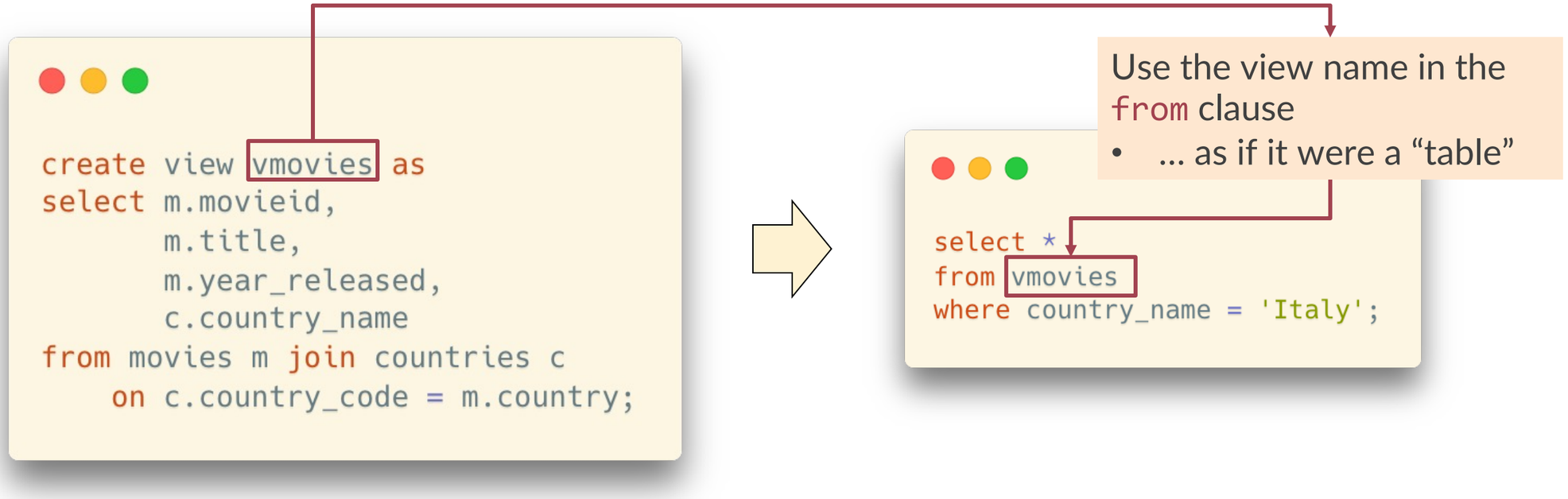
```
drop view vmovies;
```

View vs. Table

- A view can be as a complicated query as you want
  - It will usually return something that isn't as normalized as your tables, but easier to understand.
  - Imagine: a function of a `select` query
  - From a programming language's view:
    - Tables -> variables
    - Rows -> values

View vs. Table

- Once the view is created,
  - … we can query the view exactly as if it were a table



Dynamic Content in Views

- When the rows change in the tables where a view relies on, the result of a view will change as well
  - Think it like a table variable, or a query result
  - * Or, think it like a relational function which returns a relation

Dynamic Content in Views

- Beware that columns are the one in tables when the view was created
  - Columns added later to tables in the view won't be added even if the view was created with "`select *`"
    - And, "`select *`" is a bad practice in view. We will not know what exact columns are selected once the columns in the tables are changed.

```
-- Create the view first
create view vmovies as
select *  -- IT IS A BAD PRACTICE OF USING * HERE
from movies m join countries c
    on c.country_code = m.country;

-- Alter the table then
alter table movies add column test_col int not null default 0;

-- There won't be a column "test_col" in the query result
select * from vmovies;
```

View vs. Table (Cont.)

- View looks like Table, tastes like Table
  - Sometimes it is used as a table

View vs. Table (Cont.)

- View looks like Table, tastes like Table
  - Sometimes it is used as a table

  - Usage Scenario 1: Simplify Complex Queries
    - Simplify a complicated query result into a single named query
    - E.g. Many business reports are based on the same set of joins, with just variations on the columns that you aggregate or order by
      - One command, same query

View vs. Table (Cont.)

- View looks like Table, tastes like Table
  - Sometimes it is used as a table
  - Usage Scenario 2: An alternative way to implement E-R models
    - Sometimes, we may not be able to use tables to model entities and relationships
      - Access control
      - Dirty and messy original data
    - Since views look like tables, we can use views to represent entities or relationships
      - … based on some existing objects (like tables)

Reduction to Relation Schemas

- Entity sets and relationship sets can be expressed uniformly as relation schemas that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of schemas.
  - For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set.
  - Each schema has a number of columns (generally corresponding to attributes), which have unique names.

There are more than one way to implement E-R models

Drawback of Views

- View looks like Table, tastes like Table
  - But it is still not a table.
  - In some cases, it may cause performance issues or unnecessary operations
    - * … since the users of the views don’t know the details of the views

Drawback of Views

- View looks like Table, tastes like Table
  - Example: A refined way to display the credit information with a view

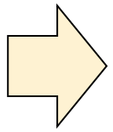
```
create view vmovie_credits
as
select m.title,
       m.year_released release,
       case c.credited_as
           when 'A' then 'Actor'
           when 'D' then 'Director'
           else '?'
           end duty,
       full_name(p.first_name, p.surname) name
from movies m
         inner join credits c
                    on c.movieid = m.movieid
         inner join people p
                    on p.peopleid = c.peopleid;
```

	title	release	duty	name
1	"Erogotoshitachi" yori Jinruigaku nyūmon	1966	Director	Shohei Imamura
2	"Erogotoshitachi" yori Jinruigaku nyūmon	1966	Actor	Shoichi Ozawa
3	"Erogotoshitachi" yori Jinruigaku nyūmon	1966	Actor	Sumiko Sakamoto
4	'76	2016	Actor	Ramsey Nouah
5	'76	2016	Actor	Ibinabo Fiberesima
6	(T)Raumschiff Surprise	2004	Actor	Michael Herbig
7	(T)Raumschiff Surprise	2004	Director	Michael Herbig
8	(T)Raumschiff Surprise	2004	Actor	Til Schweiger
9	(T)Raumschiff Surprise	2004	Actor	Anja Kling
10	(T)Raumschiff Surprise	2004	Actor	Rick Kavanian
11	(T)Raumschiff Surprise	2004	Actor	Christian Tramitz
12	(T)Raumschiff Surprise	2004	Actor	Sky du Mont
13	002 operazione luna	1965	Actor	Linda Sini
14	002 operazione luna	1965	Director	Lucio Fulci

Drawback of Views

- View looks like Table, tastes like Table
  - Example: A refined way to display the credit information with a view
    - However, if we only want to get all distinct movie titles, it will return the correct result, but there are a lot of useless works in the internal query of the view

```
select distinct title
from vmovie_credits
```



```
create view vmovie_credits
as
select m.title,
       m.year_released release,
       case c.credited_as
           when 'A' then 'Actor'
           when 'D' then 'Director'
           else '?'
           end duty,
       full_name(p.first_name, p.surname) name
from movies m
         inner join credits c
                    on c.movieid = m.movieid
         inner join people p
                    on p.peopleid = c.peopleid;
```

Do we really need the joins?

View vs. Materialized View

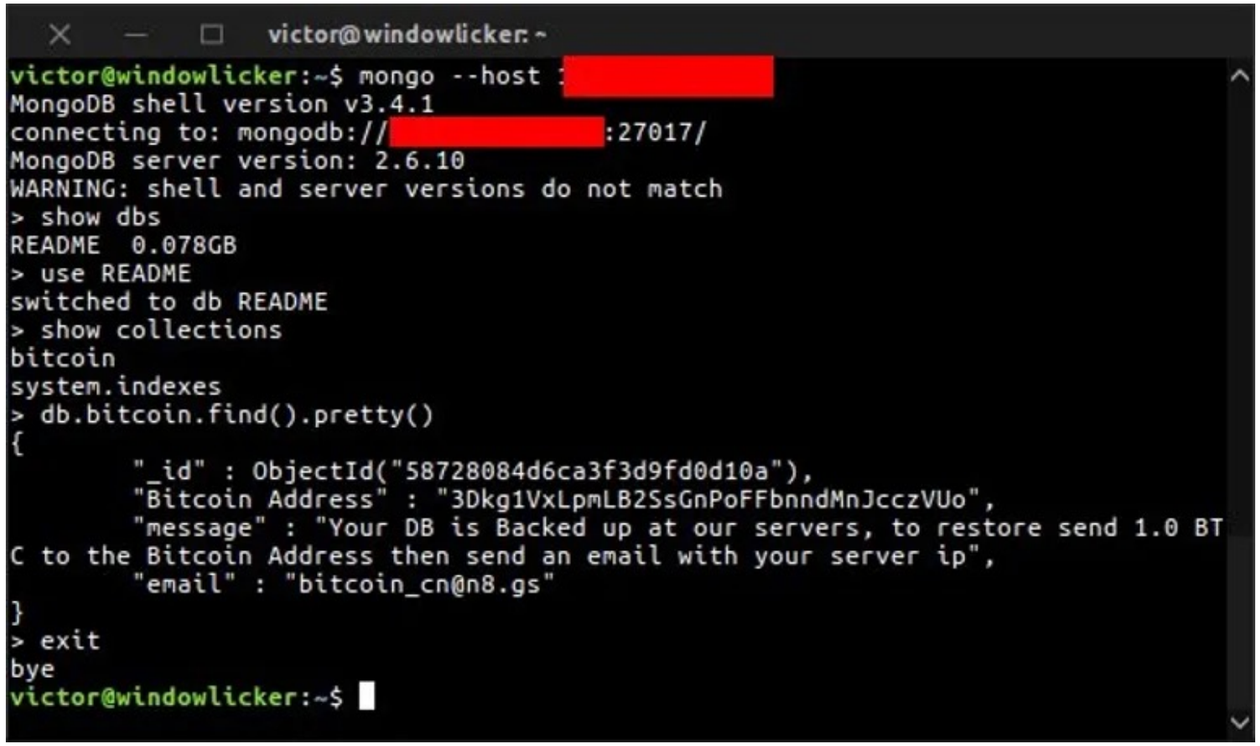
- Normal views dynamically retrieve data from the original tables
  - However, the performance won't be good

- Materialized View: A cached result of a view
  - When the corresponding tables are not changing rapidly, we can cache the view results (i.e., materialize the results)
    - Better performance than dynamic retrieval
  - The results are not updated each time the view is used
    - Manual update, or use the help of triggers

Access Control

Authentication

- To access a database, you must be authenticated, which often means entering a username and a password.
  - If you don't set a proper password, you will put your database system in danger



https://www.zdnet.com/article/mongodb-ransacked-now-27000-databases-hit-in-mass-ransom-attacks/

Privileges

- Besides, DBMS usually provide another layer of access control on objects in the system
  - Operations (select, update, insert, delete, etc.)
  - Objects (table, database, views, trigger, etc.)

Grant and Revoke Access

- `grant` and `revoke`
  - Can be used on the table or the database level

```
-- grant <right> to <account>
grant select on movies to test_user;

-- revoke <right> from <account>
revoke select on movies from test_user;
```

The possible privileges are:

```
SELECT
INSERT
UPDATE
DELETE
TRUNCATE
REFERENCES
TRIGGER
CREATE
CONNECT
TEMPORARY
EXECUTE
USAGE
```

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

How Can Views Help

- User access control
  - By creating a view that only returns rows associated with the user name, we can control the privileges for a specific user in this table

```
create view my_stuff
as
select * from stuff
where username = user
```

```
-- E.g., restrict the users to only access
-- the movies assigned to their user names
create view user_view as
select movieid,
       title,
       country,
       year_released,
       runtime
from movies
where user_name = user;
```

How Can Views Help

- User access control
  - By creating a view that only returns rows associated with the user name, we can control the privileges for a specific user in this table

```
create view my_stuff
as
select * from stuff
where username = user
```

```
-- E.g., restrict the users to only access
-- the movies assigned to their user names
create view user_view as
select movieid,
       title,
       country,
       year_released,
       runtime
from movies
where user_name = user;
```

`user`: An internal variable that returns the current user name

- Remember `old` and `new` in triggers?