

Principles of Database Systems (CS307)

Lecture 5: More on Join; Set Operators; Subqueries

Yuxin Ma

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.

More on Join

The Old Way of Writing Joins

- Use commas to separate the tables
 - Example: The solution for the same question in the previous slide
- A little bit history:
 - join was introduced in SQL-1999 (later than this original way)
- Relationship to the relational algebra
 - Filtering based on the Cartesian product
 - $\text{movies} \times \text{credits} \times \text{people}$



```
select m.title, c.credited_as,  
       p.first_name, p.surname  
from movies m,  
     credits c,  
     people p  
where c.movieid = m.movieid  
     and p.peopleid = c.peopleid  
     and m.country = 'cn'
```

The Old Way of Writing Joins

- Problems in the old way:
 - If you forget a comma, it will still work sometimes (interpreted as “renaming”)
 - The semantic meaning of the **where** clause here is a little bit different from the **where** we introduced before
 - (join key vs. filtering condition)
 - If you forget **where**, the query **will not return an error** but to **end up with HUGE** amount of rows
 - `#movies * #credits * #people`

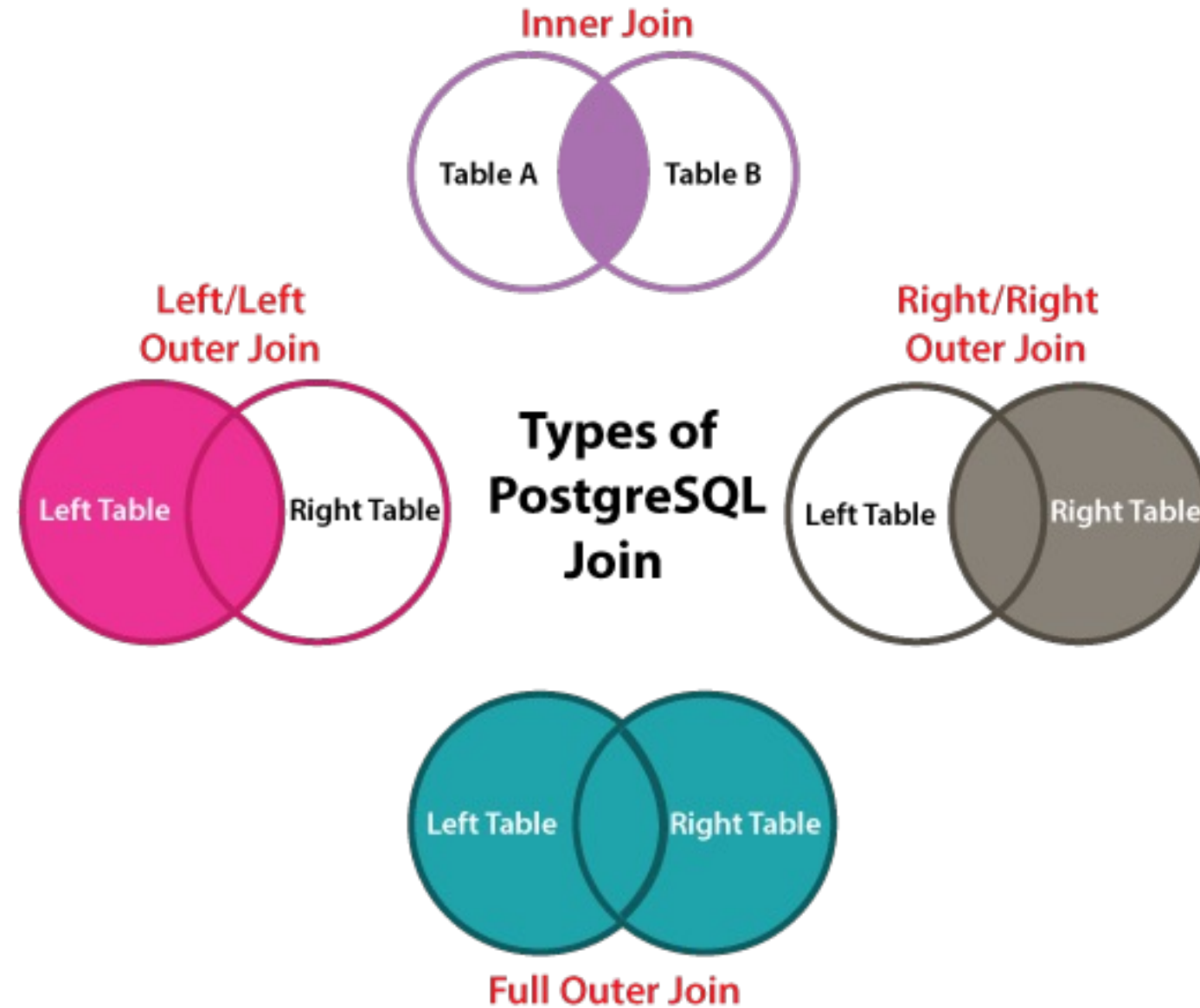


```
select m.title, c.credited_as,  
       p.first_name, p.surname  
from movies m,  
     credits c,  
     people p  
where c.movieid = m.movieid  
     and p.peopleid = c.peopleid  
     and m.country = 'cn'
```

Inner and Outer Joins

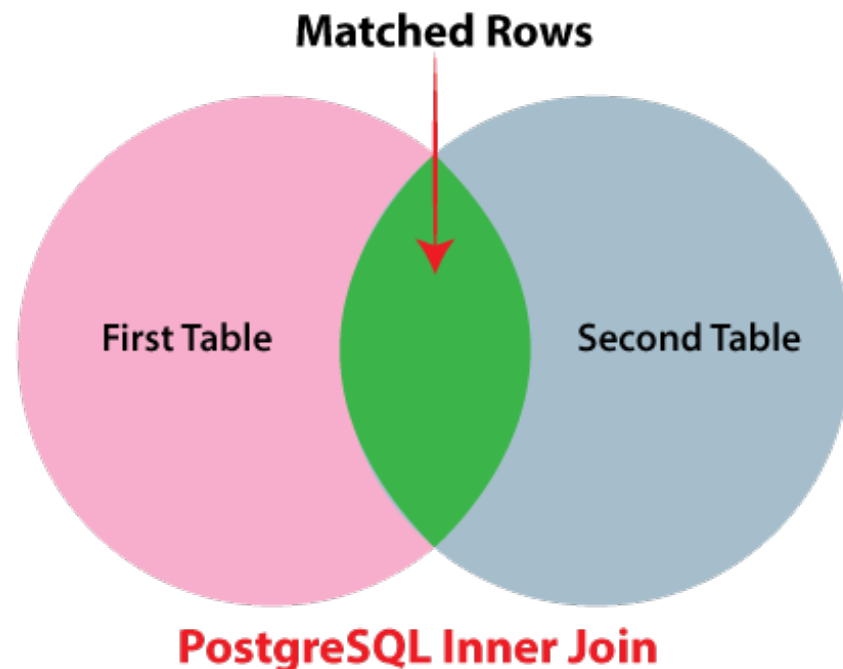
- So far, we only consider the rows with matching values on the corresponding columns
 - However, there are more things you can do with join

Inner and Outer Joins



Inner and Outer Joins

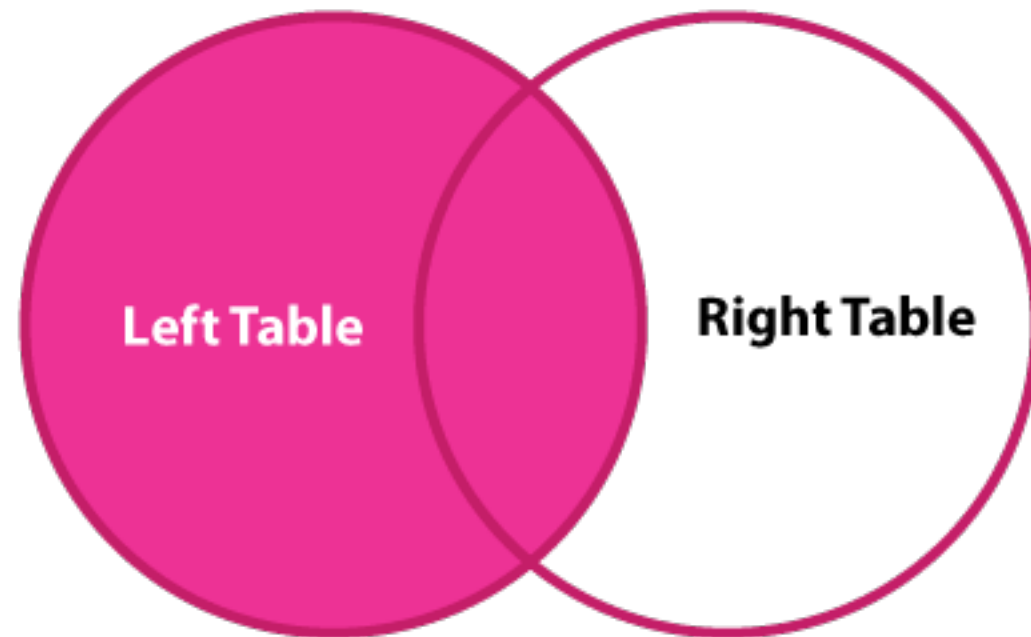
- Inner join
 - The default join type
 - Acturally, all examples before are considered inner joins
 - Only joined rows with matching values are selected



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

Inner and Outer Joins

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



PostgreSQL Left Join



```
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)

```
✓ select * from movies where movieid = 9203;
```

	movieid	title	country	year_released	runtime
1	9203	A Wrinkle in Time	us	2018	109

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
 - 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
 - But, left (outer) join can give you a record for the movie (in the left table) where all right-table columns are null

Pay attention to the syntax:

- `left join` or `left outer join`
- But some databases recognize the `outer` keyword, some do not. Refer to the database manual if you meet any error.

```
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

- Left outer join
 - **Why?** Why should we show the records in the left table with no matches?
 - **Scenario: Movie Website (Douban, for example)**
 - We cannot just ignore the movies with no credit information
 - Instead, we should list them and also show that credit information is missing
 - All things can be done in a single query
 - And we can distinguish between them by checking the values in the right-table columns

Inner and Outer Joins

- Left outer join
 - Another example: let's count how many movies we have per country (again)

Inner and Outer Joins

- Left outer join
 - Another example: let's count how many movies we have per country (again)



```
select c.country_name, number_of_movies
from countries c join (
    select country as stat_country_code,
           count(*) as number_of_movies
    from movies
    group by country
) stat
on c.country_code = stat_country_code;
```

	country_name	number_of_movies
1	Algeria	2
2	Burkina Faso	2
3	Egypt	11
4	Ghana	1
5	Guinea-Bissau	1
6	Kenya	1
7	Libya	2
8	Mali	2
9	Morocco	2
10	Namibia	1
11	Niger	
12	Nigeria	
13	Senegal	

“85 rows”:

- Problem
 - We have ~200 countries in total
 - How can we show the other countries?

Inner and Outer Joins

- Left outer join
 - All countries are here now
 - In addition, **how can we replace nulls?**



```
select c.country_name, number_of_movies
from countries c left join (
    select country as stat_country_code,
           count(*) as number_of_movies
    from movies
    group by country
) stat
on c.country_code = stat_country_code;
```

	country_name	number_of_movies
1	Algeria	2
2	Angola	<null>
3	Benin	<null>
4	Botswana	<null>
5	Burkina Faso	2
6	Burundi	<null>
7	Cameroon	<null>
8	Central African Republic	<null>
9	Chad	<null>
10	Comoros	<null>
11	Congo Brazzaville	<null>
12	Congo Kinshasa	<null>
13	Cote d'Ivoire	<null>
14	Djibouti	<null>
15	Egypt	11
16	Equatorial Guinea	<null>
17	Eritrea	<null>
18	Ethiopia	11

Inner and Outer Joins

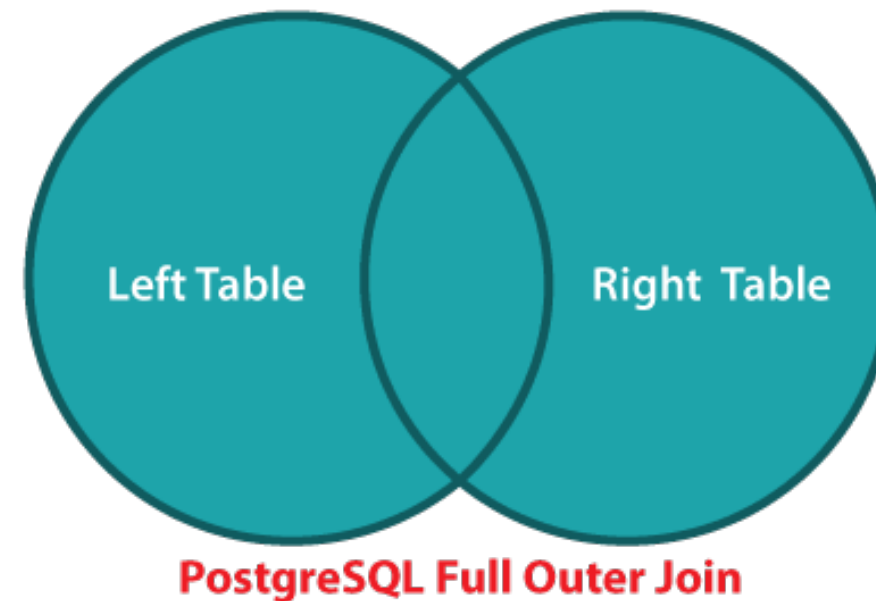
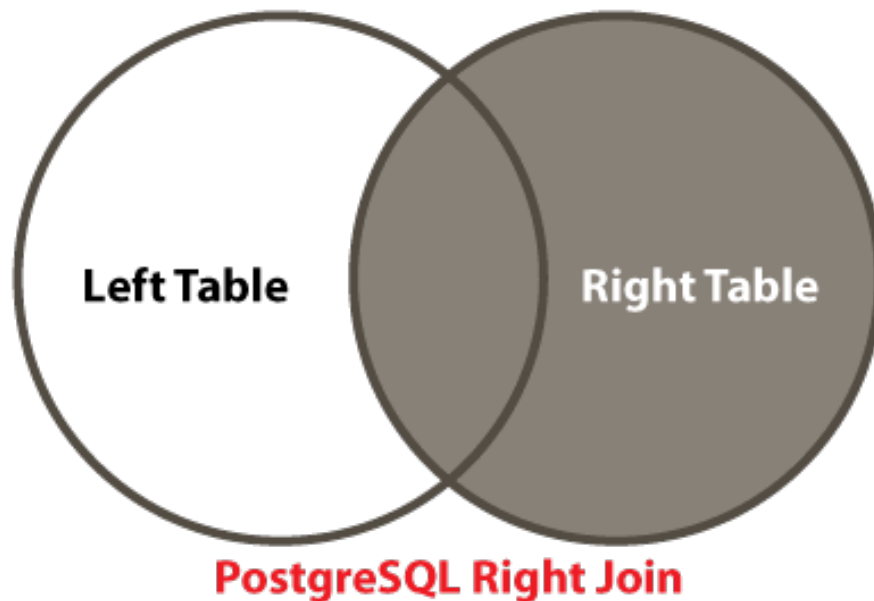
- Left outer join
 - All countries are here now
 - In addition, **how can we replace nulls?**
 - Add another CASE condition

```
select c.country_name,
       case
         when stat.number_of_movies is null then 0
         else stat.number_of_movies
       end
from countries c left join (
  select country as stat_country_code,
         count(*) as number_of_movies
  from movies
  group by country
) stat
on c.country_code = stat_country_code;
```

	country_name	number_of_movies
1	Algeria	2
2	Angola	0
3	Benin	0
4	Botswana	0
5	Burkina Faso	2
6	Burundi	0
7	Cameroon	0
8	Central African Republic	0
9	Chad	0
10	Comoros	0
11	Congo Brazzaville	0
12	Congo Kinshasa	0
13	Cote d'Ivoire	0
14	Djibouti	0
15	Egypt	11
16	Equatorial Guinea	0
17	Eritrea	0
18	Ethiopia	0

Inner and Outer Joins

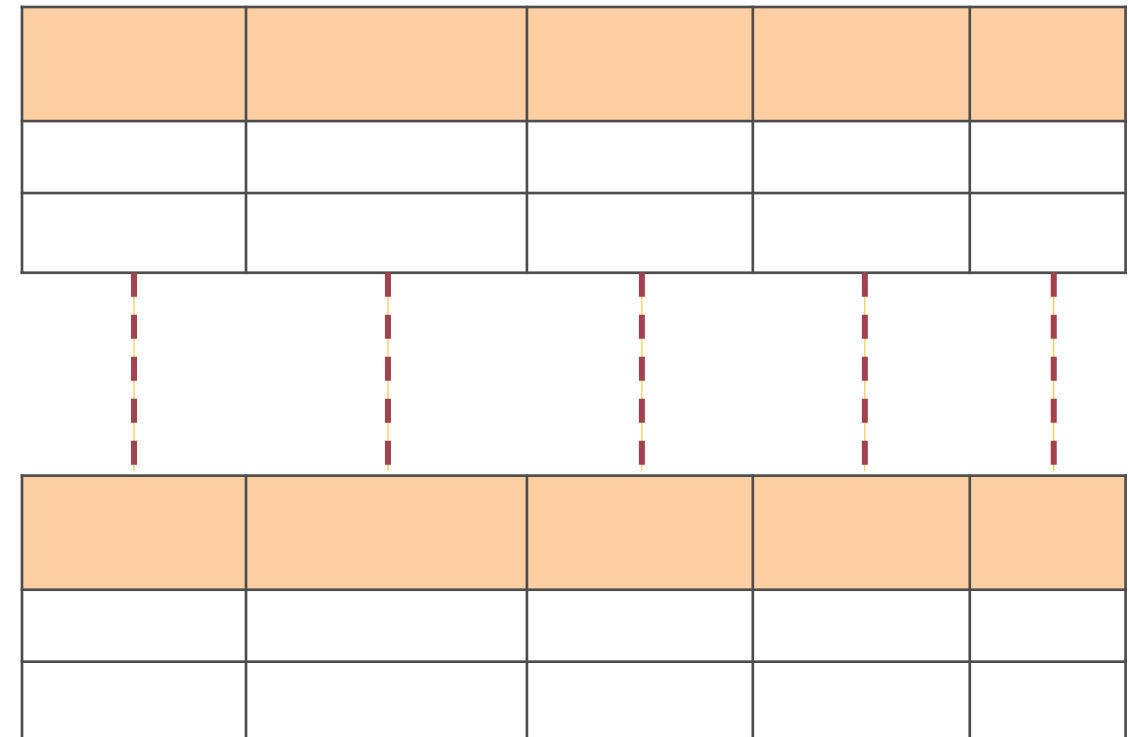
- Right outer join, full outer join
 - Books always refer to three kinds of outer joins. Only one is useful and we can forget about anything but the LEFT OUTER JOIN
 - A right outer join can ALWAYS be rewritten as a left outer join (by swapping the order of tables in the join list)
 - A full outer join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



Set Operators

Set Operators

- Union
 - Takes two result sets and combines them into a single result set
- Union requires two (commonsensical) conditions:
 - They must return the same number of columns
 - The data types of corresponding columns must match.



Set Operators

- Union
 - Example: Stack US and GB movies together



```
select movieid, title, year_released, country
from movies
where country = 'us'
and year_released between 1940 and 1949
```

union

```
select movieid, title, year_released, country
from movies
where country = 'gb'
and year_released between 1940 and 1949;
```

	movieid	title	year_released	country
1	3840	The Secret Life of Walter Mitty	1947	us
2	678	The Ox-Bow Incident	1943	us
3	3174	The Red House	1947	us
4	5152	Minesweeper	1943	us
5	1487	Kiss of Death	1947	us
6	3408	Ministry of Fear	1944	us
7	2543	The Way to the Stars	1945	gb
8	5341	All Through the Night	1942	us
9	1435	They Live by Night	1948	us
10	2644	Criminal Court	1946	us
11	7250	The Seventh Veil	1945	gb
12	7341	Mr. Lucky	1943	us

Set Operators

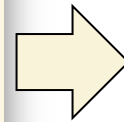
- Union
 - Usage scenario: combine movies from two tables, one for standard accounts and one for VIP accounts
 - We don't want to miss the “standard movies” for the VIP accounts

Set Operators

- Union
 - Warning: **union** will remove duplicated rows
 - Instead, you can use **union all**



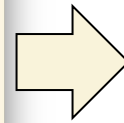
```
(select movieid, title, year_released, country
from movies limit 5 offset 0)
union
(select movieid, title, year_released, country
from movies limit 5 offset 3);
```



	movieid	title	year_released	country
1	1	12 stulyev	1971	ru
2	5	Ardh Satya	1983	in
3	2	Al-mummia	1969	eg
4	6	Armaan	2003	in
5	7	Armaan	1966	pk
6	3	Ali Zaoua, prince de la rue	2000	ma
7	8	Babettes gæstebud	1987	dk
8	4	Apariencias	2000	ar



```
(select movieid, title, year_released, country
from movies limit 5 offset 0)
union all
(select movieid, title, year_released, country
from movies limit 5 offset 3);
```



	movieid	title	year_released	country
1	1	12 stulyev	1971	ru
2	2	Al-mummia	1969	eg
3	3	Ali Zaoua, prince de la rue	2000	ma
4	4	Apariencias	2000	ar
5	5	Ardh Satya	1983	in
6	4	Apariencias	2000	ar
7	5	Ardh Satya	1983	in
8	6	Armaan	2003	in
9	7	Armaan	1966	pk
10	8	Babettes gæstebud	1987	dk

Set Operators

- Intersect (**intersect**)
 - Return the rows that appears in both tables
- Except (**except**)
 - Return the rows that appear in the first table but not the second one
 - Sometimes written as **minus** in some database products
- However, they are not used as much as union
 - intersect -> inner join
 - except -> left outer join with an “is null” condition

Subquery

Subquery

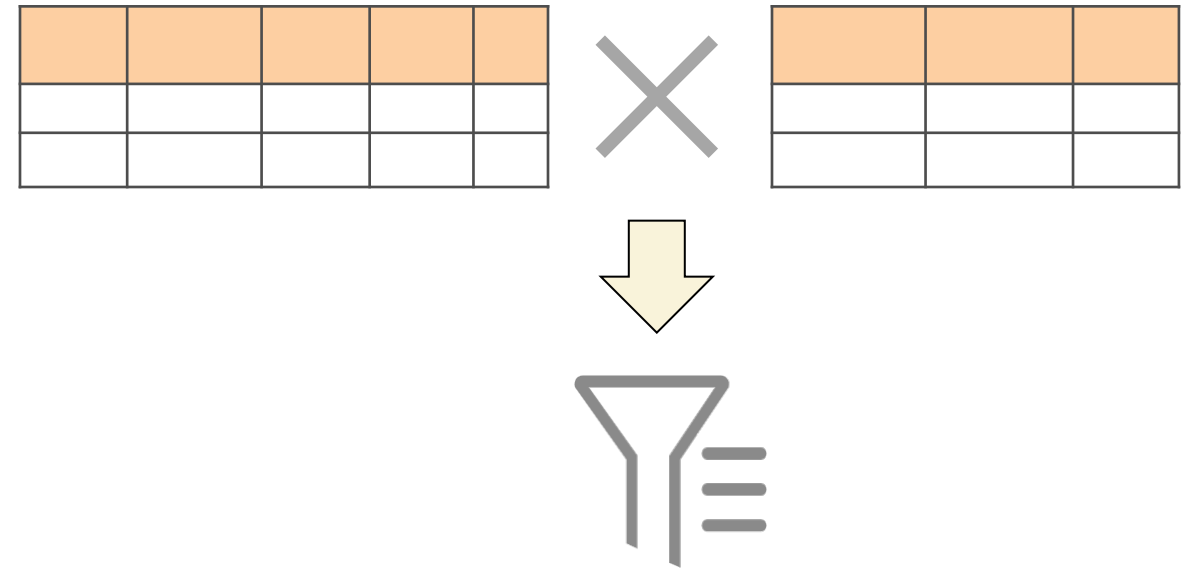
- We have used subqueries after **from** before
 - ... in order to build queries upon a query result
- And, we can add subqueries after **select** and **where** as well

Subquery after Select

- Example: show titles, released years, and country names for non-US movies
 - Solution 1: Join




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



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';
```

... still a country code though

- How can we replace it with the country name?

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

- Recall: the `in()` operator
 - It can be used as the equivalent for a series of equalities with OR (it has also other interesting uses)
 - It may make a comparison clearer than a parenthesized expression



```
where (country = 'us' or country = 'gb')  
      and year_released between 1940 and 1949  
  
where country in ('us', 'gb')  
      and year_released between 1940 and 1949
```


Subquery after Where

- ... But `in()` is far more powerful than this
 - What is between parentheses may be, **not only an explicit list**, but also **an implicit list of values generated by a query**

```
in (select col  
    from ...  
    where ...)
```

Subquery after Where

- Example: Select all European movies
 - How can we specify the filtering condition?



```
select country,  
        year_released,  
        title  
from movies  
where [?]
```

Subquery after Where

- Example: Select all European movies
 - A horrible solution: list all European countries with **or**



```
select country,  
       year_released,  
       title  
from movies  
where country = 'fr' or country = 'de' or ...
```



er的一下死掉了

Subquery after Where

- Example: Select all European movies
 - A (slightly better) solution: list all European countries in an **in** operator



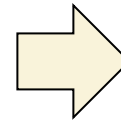
```
select country,  
       year_released,  
       title  
from movies  
where country in('fr', 'de', ...)
```

Subquery after Where

- Example: Select all European movies
 - A (slightly better) solution: list all European countries in an **in** operator

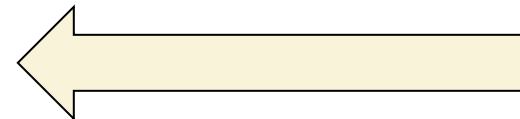


```
select country,  
       year_released,  
       title  
from movies  
where country in('fr', 'de', ...)
```



```
✓ select * from countries where continent = 'EUROPE';
```

40 rows ▾



Subquery after Where

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



```
select country,
       year_released,
       title
from movies
where country in(
    select country_code
    from countries
    where continent = 'EUROPE'
);
```



```
select country,
       year_released,
       title
from movies
where country in('fr', 'de', ...)
```

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

Subquery after Where

- Some products (Oracle, DB2, PostgreSQL with some twisting) even allow comparing a set of column values (the correct word is "tuple") to the result of a subquery.

```
(col1, col2) in  
    (select col3, col4  
     from t  
     where ...)
```

Subquery after Where

- Some important points for `in()`
 - `in()` means an implicit distinct in the subquery
 - `in('cn', 'us', 'cn', 'us', 'us')` is equal to `in('cn', 'us')`

Subquery after Where

- Some important points for `in()`
 - `in()` means an implicit distinct in the subquery
 - `in('cn', 'us', 'cn', 'us', 'us')` is equal to `in('cn', 'us')`
 - null values in `in()`
 - Be extremely cautious if you are using `not in(...)` with a null value in it

Subquery after Where

- Some important points for `in()`
 - `in()` means an implicit distinct in the subquery
 - `in('cn', 'us', 'cn', 'us', 'us')` is equal to `in('cn', 'us')`
 - null values in `in()`
 - Be extremely cautious if you are using `not in(...)` with a null value in it

`value not in(2, 3, null)`

\Rightarrow `not (value=2 or value=3 or value=null)`

\Rightarrow `value<>2 and value<>3 and value<>null`

\Rightarrow `false -- always false or null, never true`

...however, `value=null` and `value<>null` are always not true:

- We should use `is [not] null` instead

Thus, the `not in()` expression always returns false, and hence no row will be selected and returned.

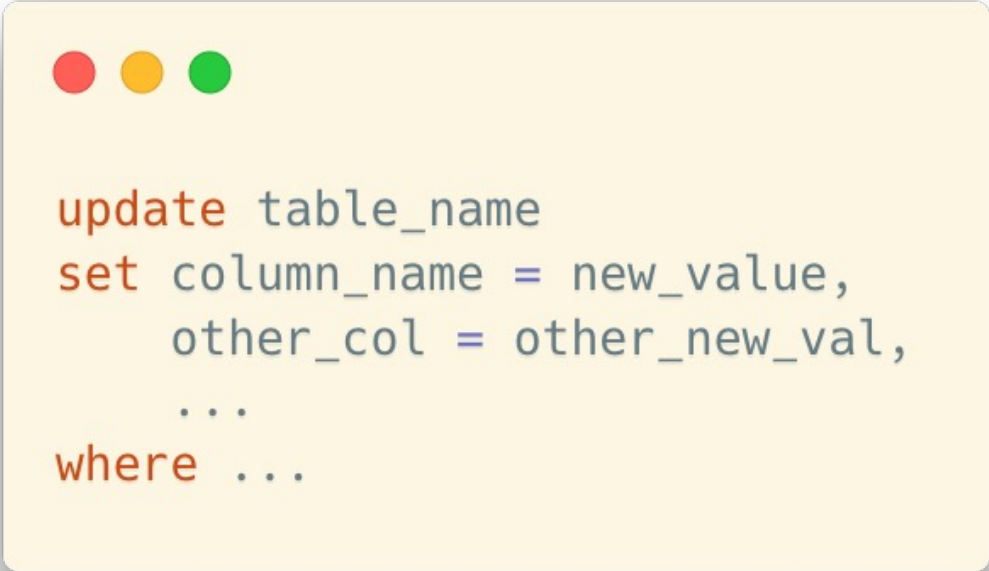
Update and Delete

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: A nobiliary particle is used in a surname or family name in many Western cultures to signal the nobility of a family.
 - We may want to modify some names in such a way as they sort as they should.

	peopleid	first_name	surname	born	died	gender
1	16439	Axel	von Ambesser	1910	1988	M
2	16440	Daniel	von Bargaen	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: A nobiliary particle is used in a surname or family name in many Western cultures to signal the nobility of a family.
 - We may want to modify some names in such a way as they sort as they should.
- First, how can we find these names?

Update

- Example: A nobiliary particle is used in a surname or family name in many Western cultures to signal the nobility of a family.
 - We may want to modify some names in such a way as they sort as they should.
- First, how can we find these names?
 - Wildcards

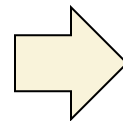


```
select * from people_1 where surname like 'von %';
```

Update

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

This could be used to postfix all surnames starting by 'von' with '(von)' and turn for instance 'von Stroheim' into 'Stroheim (von)'

```
-- 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
 - However, you can use update without **where**, where the updates 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
 - For example, 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

- **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

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

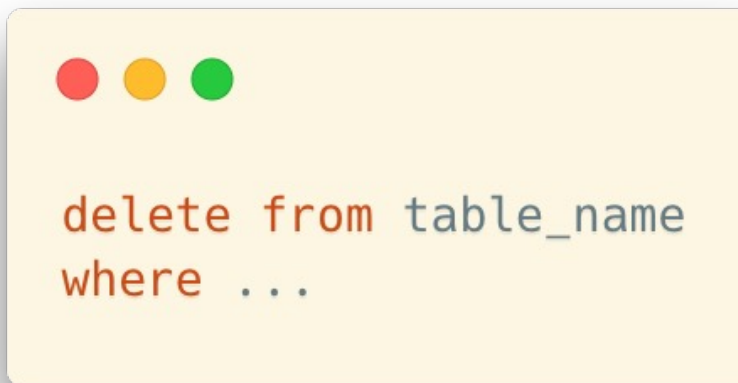
```
update people p

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



```
delete from table_name  
where ...
```

- 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)

Delete

- One important point with constraints (foreign keys in particular) is that **they guarantee that data remains consistent**
 - They don't only work with **insert**, but with **update** and **delete** as well.
- Example: Try to delete some rows in the country table

```
❗ 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".
```

- Foreign-key constraints are especially useful in controlling **delete** operations

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.

