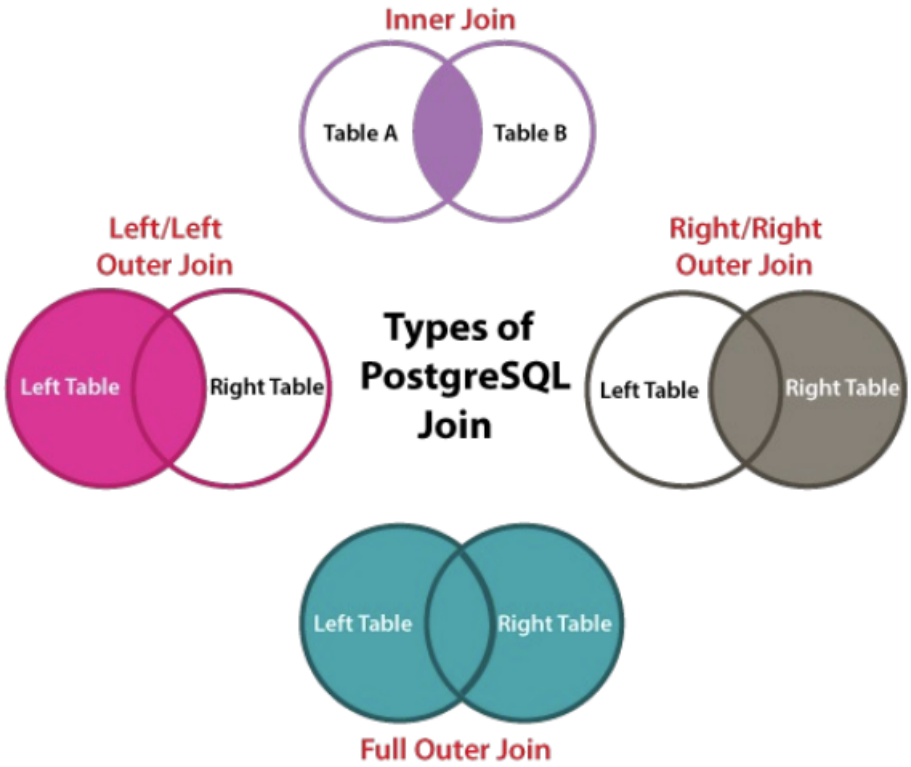
Lecture 5 & Lab 5

Cross Join, Inner and Outer Joins

1. types of PostgreSQL join



2. **Inner join:**

   • The default join type

   • Acturally, all examples before are considered inner joins

   • Only joined rows with matching values are selected

   • cartesian product will be filtered in both **left** and **right** table

3. **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

   • cartesian product will be filtered in both **left** and **right** table, but add all rows containing the key **only** in the left table, and the the columns of the right table in that special rows will display **null**

```
-- before left join, check the left table to ensure what you want to do will meet your aim
select columns
from {tab1} left [outer] join {tab2}
on ...
-- outer can be omitted in the postgres
```

   example:

```
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>

4. **Right outer join, full 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

5. **cross join**

   - return the cartesian product of all rows in the left table and right table

6. **Applications: different set**

   t1, t2 is tables, find all the rows that the join key is in the t1 but not in the t2

```sql
select *
from t1 left join t2
on t1.join_key = t2.join_key
where t1.join_key != t2.join_key
```

   t1, t2 is tables, find all the rows that the join key is in **one and excatly one** table.

```sql
select *
from t1 full join t2
on t1.join_key = t2.join_key
where t1.join_key != t2.join_key
```

7. `and` , `where`

   - `and` restrict the join_key during the joining, while `where` filter the rows after joining the table.
   - No difference in inner join. However, there may be some difference in the outer join.

   example:

```sql
select * from T1 left join T2 on T1.A = T2.A and T1.C=3;
select * from T1 left join T2 on T1.A = T2.A where T1.C=3;
```

T1

a	b	c
1	hello	3
2	world	8
2	hi	3

T2

a	b	c
3	database	4
2	hello	8
4	cs307	3

Result of upper one:

	t1.a	t1.b	t1.c	t2.a	t2.b	t2.c
1	1	hello	3	<null>	<null>	<null>
2	2	world	8	<null>	<null>	<null>
3	2	hi	3	2	hello	8

Result of lower one:

	t1.a	t1.b	t1.c	t2.a	t2.b	t2.c
1	2	hi	3	2	hello	8
2	1	hello	3	<null>	<null>	<null>

Set Operation

1. difference from the **Join**

   Join concatenate different colums from different tables, while **Set operation** concatenate the different rows from different tables.

2. **Union**

   - Restrictions

     They must return the same number of columns

     The data types of corresponding columns must match

   - features

     return the rows appear at least one time in some specified table

- syntax

```
select {col1}, {col2} from {tab1} where {conditions}
union
select {col3}, {col4} from {tab2} where {conditions}
```

- warning:

  `Union` will **remove** duplicated rows. To avoid this, `union all` is more recommended.

  Rename the columns to have same name is highly recommended

3. `intersect`

   - Return the rows that appears in both tables
   - **dedulplicate the rows**

4. `except`

   - Return the rows that appear in the first table but not the second one
   - Sometimes written as `minus` in some database products
   - **dedulplicate the rows**

Subquery

1. subquery after `from`

   we have mention before

   ```
   select *
   from (
       select * from {tab1}
   )
   where {conditions}
   ```

2. subquery after `select`

   - when we want to query the columns in another table, obviously, a method is to use `join`. However, another method is using subquery.
   - **replace** a specify column **with** another column in another table
   - How to match?

     ```
     -- Originally
     select m.title, m.year_released, m.country
     from movies m
     where m.country != 'us'

     -- we want to show the country name respectively
     select m.title, m.year_released, (
         select c.country_name
         from countries c
         where c.country_code = m.country_code -- found the country
     respectively
     ) -- normally, we join the country_code
     from movies m
     where country != 'us';
     -- replace the country code with the country name, and we should
     guarantee that their is a map: country_code -> country_name, or it will
     throw exception, for example, delete the where cause will throw
     exception
     ```

   - warning

     It will be less effective than the `join`, thus, `join` is more recommended.

3. subquery after `where`

   - when we want to restrict the values **in** some values appear in another table, what should we do?
   - syntax

     ```
     select * from {tabl1}
     where col1 in (
         select col2
     ```

```
    from {tab2}
    where {conditions}
)

/*
-- whether this can be used depends on the database
-- postgreSQL support this feature
select * from {tabl1}
where {col1, col2} in (
    select {col3, col4}
    from {tab2}
    where {conditions}
)
*/
```

- Some important points for `in()`
  - `in()` means an **implicit distinct** in the subquery
  - **null** values in `in()`: Be extremely cautious if you are using `not in(…)` with a **null** value in it.
  - machanism

    both `value = null` and `value != null` are always not true

```
not in (val1, val2, null)
=> not (value = val1 or value = val2 or value = null)
-- though it's not equivalent, it will be converted like this
=> (value != val1 and value != val2 and value != null)
=> true/false and true/false and null
=> false or null
-- this will perform normally
in (val1, val2, null)
=> (value = val1 or value = val2 or value = null)
=> (value = val1 or value = val2)
```

- `exists` and `in`
  - syntax

```
select * from {tab1} {alias1}
where exists (
    select null from {tab2} {alias2}
    where {conditions}
)

select * from {tab1} {alias1}
where {col1} in (
    select {col2} from {tab2} {alias2}
    where {conditions}
)
```

`exists`	`in`
Scan each row in the outer-query and judge whether it match the condition.	Scan each row in the subquery and filter rows match the condition in the outer-query
`exists` is faster if sub-query result is larger than the outer query	`in` is faster if sub-query result is smaller than the outer query

Update and Delete

1. 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`
    - When you are doing any experiments with writing operations (update, delete), **backup the data first**
    - syntax

```
update {tab}
set col1 = val1, col2 = val2...
where ...

-- It's more meaningful
update {tab}
set col1 = func1{col1}, col2 = func2{col2}
where ...
```

    - The update operation may not be successful when **constraints are violated**

      For example, update the **primary key** but with **duplicated** values

      This is why we need constraints when creating tables: **avoid unacceptable writing operations that break the integrity of the tables**
2. Delete
    - delete removes rows from tables(If you omit the `WHERE` clause, you will **end up with an empty table**)
    - One important point with constraints (**foreign keys** in particular) is that they **guarantee that data remains consistent**


```
[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".
```