Lecture 4 & Lab 4

Distinct

1. **No duplicated** identifier.

   adding `distinct` will help us to dedulplicate the columns we select, i.e. eliminate the identical rows for the columns

   NULL is treated as a distinct element

   used whem we are insterested in different keys

2. syntax

   ```
   select distinct {col} from {tab}
   ```

   then we weill see the counts of some columns

   ```
   select distinct {col1}, {col2}, ... from {tab}
   ```

   the tuble `(col1, col2, ...)` is distinct

3. Remark

   We cannot use `distinct` to select both distinct and not distinct columns. Once distinct is used, the combination of the following columns should be distinct and there is no exception.

   We can use `group by` to deal with this problem.

Aggregate Functions

1. Aggregate functions will **aggregate all rows** that share a feature and **return a characteristic** of each group of aggregated rows.
2. **count**

   0. reference

      > PostgreSQL COUNT Function: Counting Rows That Satisfy a Condition (postgresqltutorial.com)

   1. syntax

      Summary: It **won't** consider the **NULL if and only if** you do the **single-column counting**

```
-- to count the number of rows including NULLs
select count(*) from {tab1}

-- to count the number of rows of col1 ignoring NULL
select count({col1}) from {tab1}

-- to count the number of different rows of col1 ignoring NULL
select count(distinct {col1}) from {tab1}

-- to count the number of different rows of (col1, col2, ...) including NULLs
select count(distinct({col1}, {col2}, ...)) from {tab1};
```

2. example

num1	num2	num3
1	2	null
1	null	null
null	2	1
null	null	null
null	null	1

```
select count(*) from table1; -- 5
select count(num3) from table1; -- 2
select count(distinct num3) from table1; -- 1
select count(distinct (num2, num3)) from table1; -- 4
```

3. **group by**

1. Aggregate all rows to several rows with the specified criteria.

   Then some other aggregate functions can aggregate each group and return a specific value

2. syntax

   NULL is also a group in postgreSQL

```
select {col1} -- columns must also appear after group by
from {tab}
where {cond}
group by {col1}

select {col1}, {col2} -- columns must also appear after group by
from {tab}
where {cond}
group by {col1}, {col2}

-- or we can use aggregate functions
```

```
-- as can be ommited without changing the meaning
select {col1} aggregate_func({col2}) as {new_name}
from {tab}
where {cond}
group by {col1} -- not compulsory to appear col2 here
```

3. example

num1	num2
null	1
null	1
1	2
1	1
null	null

```
select num1, count(num2) from table1 group by num1;
/*
null 2
1    2
*/
select num1, count(*) from table1 group by num1;
/*
null 3
1    2
*/
select num1, num2 from table1 group by num1, num2;
/*
null 1
1    2
1    1
null null
*/
```

4. other aggregate functions

   they all ignore **NULL**

```
min(col)
max(col)
stddev(col) -- find the standard deviation of a column
avg(col)
```

5. composite relation

```
-- for the more complicated composite relation, use alias.col1 to specify the exact col
select {alias1.col1} from
{
    select ...
    from ...
    where...
} {alias1} -- give the intermediate results an alias
where {cond}
```

examples:

num1	num2
null	1
null	1
1	2
1	1
null	null

```
select newta.std from(
    select num1, stddev(num2) std
    from testable1
    group by num1
) newta where newta.std > 0.58;
```

6. having
    1. From actual meaning, rows firstly filtered by `where`, then grouped through `group by`, finally filtered by `having`
    2. syntax

        ```
        select {} from {} where {cond1} group by {cols} having {cond1_}
        ```

    3. examples

num1	num2
null	1
null	1
1	2
1	1

num1	num2
null	null

```
select num1, stddev(num2) std
    from testable1
    where num2 is not null
    group by num1
    having stddev(num) > 0.58
    -- having std > 0.58 is wrong
    -- having num2 > 1 is wrong
    -- the expression must be aggregated or specify in the GROUP;
```

Join

1. aim: we want to retrieve data from **multiple** table and display them in **one** table
2. machanism:

   make a **cartesian product** for the keys on tabel1 and table2,

   then filter those `ta1_keyCol != ta2_keyCol`

   rows with **key_Col is null** is ignored
3. syntax:

```
select {ta1_col1}, {ta1_col2}...
    {ta2_col1}, {ta2_col2}...
    from {ta1} (alias1) join {ta2} (alias2)
    on {alias1}.{ta1_keyCol} = {alias2}.{ta2_keyCol}
    where {conditions}
    ...;
-- though alias is not compulsory, it's a good habit
-- it's vital when ta1_keyCol have the same name as ta2_keyCol
```

4. natural join:(not recommended)

   what if we don't specify the column?

```
select * from people natural join credits;

-- will automatically join by recognized key(same column name)
-- but sometimes we don't guarantee the same name means same semantic

-- The same as
select *
from people join credit
on people.peopleid = credits.peopleid
-- note that peopleid is a column in both people and credits
```

   what if we want to simplify the query

```sql
-- better
select *
from people join credits using(peopleid)
```

5. self join:

   join the same table together:

   For example: How can we find how many **unordered pairs** of people with the same first name? (An inner pairing problem)

```sql
-- recall the machanism: cartesian product

-- wrong solution 1:
select count(*)
from people p1 join people p2
on p1.first_name = p2.first_name
-- someone matching himself is counted

-- wrong solution 2:
select count(*)
from people p1 join people p2
on p1.first_name = p2.first_name
where p1.id != p2.id
-- A matches C and then C will matches A, that is, for one pair (A, C) will be counted twice

-- correct answer
select count(*) / 2
from people p1 join people p2
on p1.first_name = p2.first_name
where p1.id != p2.id
```

6. join in a subquery

   syntax:

```sql
select {col1}, {col2}, {col3}...
from
    {tab1} ({alias1})
    join {tab2} ({alias2}) on {alias1}.{ta1_keyCol} = {alias2}.{ta2_keyCol}
    join {tab3} ({alias3}) on {alias1}.{ta1_keyCol} = {alias3}.{ta3_keyCol}
    ...
where {conditions}...
```

   machanism is similar, just filter more than one time.

Quickly seach some characteristics in a table

1. search the column names of a table

```
SELECT column_name FROM information_schema.columns WHERE table_name =
{tab1};
```

2. search the possible values for a column

```
select distinct {col} from {tab1}
```

3. advanced: check the column names, types, not null, reference, primary key, foreign key.

   Command line or **powershell** is needed in windows

   Ensure that you add the **\bin** to the path

```
# [optional]
chcp 437 # switch to English
chcp 936 # switch to Chinese

# firstly log in to connect the database
psql -U {user_name} -d {database_name}

# input your password
{your_password}

# check charateristics of a table
\d {tabble_name}

# search
select
distinct {col} from {tab}

# terminate query
ctrl+c
```