Lecture 3 & Lab 3

update to tables

1. Insert statements

```
-- should specify all values
insert into {tab1} values ({val1}, {val2}, {val3})

-- furthermore, values match columns one by one
-- other column will be inserted NULL automatically
-- It may occur error for mandatory columns
insert into {tab1}({col1}, {col2}) values({val1}, {val2})

-- if we want to insert a string with a single quote
insert into student(name) values('Bob''t'); -- Bob't here
```

2. delete statements

```
-- remove all the tuples in the tab1
delete from {tab1}
```

Query -- select

1. format

```
select {col1}, {col2}, {col3}... from {tab1}, {tab2}, {tab3}
where {conditions}
```

colomn name don't need `' '`

2. query for all columns

```
select * from {tab1}
```

Remark: This query is frequently used when you don't remember column names. But it would flood the database if the data is large. (In the application programs)

select with restrictions

1. Reason:

   when you are instrested in only a small subset or only want to return some of the rows.

2. Filtering(rows)

    1. overview

        1. perform in the `where` clause

        2. conditions are usually expressed by a column name

3. only rows for which the condition is true will be returned
4. if you want to filter the columns, just filter it behind the "select"

2. example

```
select * from movies where country = 'us'
```

3. what you compare

```
a number
a string constant: must be quoted between single-quotes
another column(same table or another)
result of a function
```

3. select without From or Where
    1. property

       never alter any table

       might be a good choice to figure out an expression is true or false

    2. examples

```
select '437'
-- just return return a cell that contains '437'
-- quite useless
-- may be use in ensure whether an expression is true
select '437' as FOO
-- generate a column name "FOO"
-- do not alter any table
select 'abcd' from {tab1}
-- generate 1 column and n rows with 'abcd' in the cells
-- even if tab1 hasn't such cells
```

Arithmetic Expression

1. **as** clause

   just show with the column name you want, **never alter the table itself**

```
select {col} as {col_newname} from {tab}
```

2. `and, or`
    1. functions like other programme langugue
    2. precedence: and > or
    3. we can use parentheses to enforce the precedence

```
-- these 2 statement are different
select * from {tab}
where ({cond1} or {cond2}) and ({cond3} or {cond4})

select * from {tab}
where {cond1} or {cond2} and {cond3} or {cond4}
```

3. comparison operators
    1. basic comparison operators

    ```
    <, <=, >, >=, =, !=, <>
    ```

    2. `!=, <>`

        Both means not equal to

        Whether they are equivalent depends on the SQL. In postgreSQL, they are equivalent.

    3. remark on **bigger** and **smaller**

        means different for different **data types**

    ```
    2 < 10 --true
    '2' < '10' -- false, ASCII + lexicographical order
    '2-JUN-1883' > '1-DEC-2056' -- comapre as strings or dates? It differs in different products
    ```

4. `in()`

    It can be used as the restriction for some columns are discrete range

    a set is in the parentheses

    ```
    where (country = 'us' or country = 'cn')
    where country in('us', 'cn')
    ```

5. intervals
    1. format:

    ```
    where {col1} between {lowerbound} and {upperbound}
    ```

    2. remark

        `between {a} and {b}` indicates an interval `[a, b]`

    3. example

    ```
    -- query all 5-10 years old kids
    where age between 5 and 10
    -- query all kids whose name with a first character 'a' or 'b'
    where name between 'A' and 'C' and name not like 'c%'
    ```

6. negation
    1. claim: all comparison can be negated with `not`
    2. examples

```
where country not in ('us', 'gb') or year not between 1940 and 1949;
-- by De Morgan's law, it's equivalent to
where not (country in ('us', 'gb') an year between 1940 and 1949);
```

7. match with **like**
    1. With the usage of **like**, we can restrict a column of string to a specifi patten
    2. wildcard

```
% -- meaning any number of characters, including none

_ -- one and only one character
```

    3. skills

```
-- 1. we don't want any string in the {col} with a character 'A'
where {col} not like '%A%'

-- 2. we don;t want any string in the {col} with 'a' or 'A'
where {col} not like '%A%' and not like '%a%'
where upper({col}) not like '%A%' -- Function called. Slow down the
query
```

8. Date and Datetime
    1. explicit cast is needed to avoid bad surprise when **comparing**

```
where post_date >= '2018-03-12'; -- bad habit
where post_date >= date('2018-03-12'); -- ok
```

    2. More than one patten of string can be converted to same date

```
'YYYY-MM-DD'
'YYYY/MM/DD'
'Mon DD, YYYY'
'Month DD, YYYY'
date('2018-3-12')
date('2018/3/12')
date('Mar 12, 2018')
date('March 12, 2018')
```

    3. better way to convert a string to **date** or **timestamp**
        0. Reference:

> PostgreSQL: Documentation: 12: 9.8. Data Type Formatting Functions

1. syntax

Function	Return Type	Description	Example
`to_date(text, text)`	`date`	convert string to date	`to_date('05 Dec 2000', 'DD Mon YYYY')`
`to_timestamp(text, text)`	`timestamp with time zone`	convert string to time stamp	`to_timestamp('05 Dec 2000', 'DD Mon YYYY')`

2. Common format specifiers

   date:

Pattern	Description
`YYYY`	year (4 or more digits)
`Month`	full capitalized month name (blank-padded to 9 chars)
`Mon`	abbreviated capitalized month name (3 chars in English, localized lengths vary)
`MM`	month number (01-12)
`DD`	day of month (01-31)

   timestamp:

Pattern	Description
`HH12`	hour of day (01-12)
`HH24`	hour of day (00-23)
`MI`	minute (00-59)
`SS`	second (00-59)
`MS`	millisecond (000-999)

3. More syntax and format supported by postgreSQL

   Reference:

> PostgreSQL: Documentation: 12: 9.8. Data Type Formatting Functions

4. Comparing **date** and **timestamp**

   A **date** will be automatically converted to **timestamp**, which has higher precision.

Its **time** will be automatically set to **'00:00:00'**