CS307 Database

Lab1

1 创建一个数据库并指定编码方式

```
create database cs307 encoding='utf8';
```

2 找出所有数据库

```
select datname from pg_database;
```

3 创建用户

```
create user checker superuser password '123456';    //创建超级用户
create user checker password '123456'; //创建普通用户
```

4 valid identifiers

1. 字母数字字符：包括**大小写字母、数字和下划线（_）字符。**标识符必须以字母或下划线字符开头，不能以数字开头。
2. 引号括起来的标识符：如果标识符包**含非字母数字字符**，可以使用引号将其括起来。在引号中可以使用任何字符，包括空格和特殊字符，但引号本身需要用两个单引号表示。
3. 保留字：不能使用保留字作为标识符，否则会导致语法错误。在PostgreSQL中，保留字包括SQL标准的保留字、PostgreSQL扩展的保留字和SQL2003新增的保留字。

   (by chatgpt)

Lecture 2 & Lab 2

SQL: Struture query language

1. expalnation: SQL是一种声明性语言，用户只需要指定需要的数据，而不需要指定如何获取数据。SQL的语法规则和语句可以用于对关系型数据库进行增删改查等操作，包括创建表、插入数据、更新数据、删除数据、查询数据等。(by chatgpt)
2. basic syntax in SQL

```
select ...; --followed by the names of columns you want to select(列筛选)
from ...; --followed by the name of tables(表的名字)
where ..; --filtering condition(一个单条件或组合条件语句，允许对行进行过滤)
```

DDL: Data Definition Language($\subset SQL$)

1. expalnation: a main component for a query language(是用于定义数据库对象的SQL语言部分)
2. basic syntax:

```
create ...; --创建表、视图、索引等。
alter ...; --修改表结构、视图、索引、约束等
drop ...; --删除表、视图、索引等
truncate ...; --清空表的数据
rename ...; --重命名表，图标等
```

(by chatgpt)

DML：Data Manipulation Language($\subset SQL$)

1. explanation: 操纵表中的数据
2. basic syntax:

```
select ...; --检索数据
insert ...; --插入数据
update ...; --更新数据
delete ...; --删除数据
```

Create Tables:

1. characteristics:
    1. case-insensitive: 关键字，标识符等都是大小写不敏感的
    2. make identifier case-sensitive: double quotes, not recommended

```
-- two tables are created in the following
CREATE TABLE "myTable";
create table "mytable";
```

    3. 命名惯例：使用小写
2. basic syntax:

```
-- if no exist 可选择不写，表示如果这个表不存在就进行创建
create table if no exist {table_name1}(
    {attribute_name1} {data_type1},
    {attribute_name2} {data_type2},
    ...
    (
        {integrity_constraint1},
        {integrity_constraint2},
        ...
    )
)
```

Data Types

1. text data types

```
/*
当使用CHAR存储字符串时，如果字符串的长度小于length，则在字符串后面补充空格，使其达到指定长度。CHAR类型的字段始终会占用指定长度的存储空间
*/
char(length) --fix-length string

/*
使用 VARCHAR 存储字符串时，它只会占用存储实际字符串所需的空间
*/
varchar(max length) --non-fix-length text
varchar2(max length) --non-fix-length text(Oracles's)

clob -- very very long text(GB level)
text -- very very long text(GB level)
```

2. numerical type

```
int -- a finite subset of the intergers, machine-dependent
float(n) -- Floating point number, precision: at least n digits
real -- 单精度浮点数(4 bytes)
double -- 双精度浮点数(8 bytes)
double precision -- 双精度浮点数(8 bytes)
numeric(p, d) -- 数字总长为p, 小数部分长度为d
```

3. date types

```
date -- format: YYYY-MM-DD
datetime -- format: YYYY-MM-DD HH:mm:SS
timestamp -- format: YYYY-MM-DD HH:mm:SS, in UNIX system, have 2038 problem
```

4. binary data type

```
raw(max length) -- 定长二进制数据，超过会报错，但是一定占用{max length}个字节的空间
varbinary(max length) -- 变长二进制数据，超过也会报错，但是所需存储空间会变
blob -- 存储较大的二进制数据，变长
bytea -- postgresSql中使用，可存储任意类型的的二进制数据，变长，
```

Constraints

1. DBMS(Database Management System) will check the constraints or declarative rules every time when data is added, changed, deleted.
2. NOT-NULL constraints

```
/*
    we don't want some colomns with no element in some cells
*/
create table notnull_example(
    peopleid int not null -- 这一列一定不能为空
)
```

better:

```
create table notnull_example(
    peopleid int constraint nn not null -- 这一列一定不能为空
)
```

由于not null是两个关键字，所以不能有如下操作：

```
create table notnull_example(
    peopleid int,
    ...
    constraint nn not null(peopleid)
)
```

3. Primary-Key constraints
    1. the value is mandatory(该字段必须)
    2. the value is unique
    3. Only 1 column can be set as primary key
    4. format:

        ```
        create table prime_example(
            peopleid int primary key
        )
        ```

        可以使用复合列作为主键

        ```
        create table prime_example(
            peopleid int,
            name_ varchar(40),
            ...
            primary key(peopleid, name_)
        )
        ```

        better: declare a constraint name explicitly

        ```
        create table prime_example(
            peopleid int constraint id_pm primary key
        )
        ```

an equivalent format

```
create table prime_example(
    peopleid int,
    ...
    constraint id_pm primary key(peopleid)
)
```

4. Unique
    1. the **value** of a column or a **combination** of several column cannot be the same for 2 rows
    2. format:

```
create table unique_example(
    peopleid int unique,
    ...
    unique (first_name, second_name)
)
```

    better: declare a constraint name explicitly

```
create table unique_example(
    peopleid int constraint id_uni unique,
    ...
    constraint name_uni unique (first_name, second_name)
)
```

    3. 区分Unique和primary key

    primary key只有一个且要求not-null

    unique可以有多个，不要求not-null

5. Check
    1. check(condition), 只有满足条件表达式时，数据才会被插入或更新
    2. 在check约束中，可以使用的条件表达式包括比较操作符、逻辑操作符、函数调用等。（by chatgpt)
    3. format

```
create table check_example(
    firstname varchar(100),
    lastname varchar(100),
    age int,
    ...
    check(lastname = upper(lastname)), -- 保证大写
    check(firstname = lower(firstname)), -- 保证小写
    constraint age_check check(age >= 0)
    -- 前面constraint age_check部分可加可不加
)
```

6. foreign key（外键）

    1. format

```
create table fk_example(
    {column1} {type1},
    ...
    constraint id_fk {column1}
        references {outer_table1}({outer_column1})
)
```

    2. remark：

    outer_column1一定要是unique或者primary key

    column1的值一定出自outer_column1，但是column1，outer_column1不一定完全相同

Alter

1. set or drop not-null constraint

    1. syntax

```
/*
    [...]表示可有可无
    {|} 表示任选其一
    {} 表示标识符
*/
alter table [if exist] [only] {table_name}
alter {column_name} {set | drop} not null -- 要保证不含null值
-- 不需要add constraint，因为有drop方法，不需要自定义约束的标识符
```

    2. example

```
alter table customer
alter passw set not null,
add constraint nn check(passw is not null)
```

2. add/drop **unique, primary key, foreign key, check**

    1. syntax

    adding:

```
-- unique
alter table {table_name}
add constraint cons_name unique (column1, column2, ...);

-- primary key
alter table {table_name}
add constraint cons_name primary key (column1, column2, ...);

-- foreign key
```

```
alter table {table_name}
add constraint cons_name foreign key({inner_col})
    references {outer_table} ({outer_table});

-- check
alter table {table_name}
add constraint cons_name check ({condition});
```

droping:

```
alter table {table_name}
drop constraint cons_name
-- that's why declaring the constraint name explicitly matters
```

3. change data type
    1. syntax

    ```
    alter table {ta_name}
    alter column {col_name} type {new_type}
    ```

    2. example

    ```
    alter table customer
    alter column phone_number type varchar(2);
    ```

4. add/drop column
    1. syntax

    ```
    -- adding
    alter table {ta_name}
    add column {col_name} {type_name};

    -- droping
    alter table {ta_name}
    drop column {col_name};
    ```

    2. example

    ```
    alter table table2
    add column age int;

    alter table table2
    drop column age;
    ```

5. rename
    1. syntax

```
-- rename the whole table
alter table {ta_name}
rename to {new_table};

-- rename a column
alter table {ta_name}
rename column {col_name} to {new_col}
```

2. example

```
alter table table1
rename to table2

alter table table2
rename column age to ages
```

6. check constraint

```
select tc.constraint_name, tc.constraint_type, tc.table_name
from information_schema.table_constraints tc
where tc.constraint_schema="current_schema"();
```

7. drop table

```
drop table {ta_name};
```

如果存在外部键指向表内键，则无法删除，解决方法是alter其他表格，将外部键全部删除

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

   modified the **display** name:

   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. `=`

      means **equals**

      never use `==` as **equals** in postgreSQL

4. **NULL** judgement

```
-- never use {col1} = null as a filter
where {col1} is null;
where {col2} is not null
```

5. 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')`

Function	Return Type	Description	Example
`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'**

Transforming displayed colomn in Select

1. concatenate strings

   `||` are use to concatenate strings

   syntax:

   ```
   select [{col1} | {str1}] || [{col2} | {str2}] || [{col3} | {str3}]
   from {tab1}
   ```

   example:

   ```
   select 'The code of '||c.name||' is '|| c.hex
   from color_names;
   ```

   display colomns

?column?
The code of AliceBlue is #F0F8FF
The code of AntiqueWhite is #FAEBD7
The code of Aqua is #00FFFF
The code of Aquamarine is #7FFFD4

   use `as` to modified the column name displayed

   ```
   select 'The code of ' || c.name || ' is ' || c.hex as 'combine'
   from color_names;
   ```

?column?
The code of AliceBlue is #F0F8FF
The code of AntiqueWhite is #FAEBD7
The code of Aqua is #00FFFF
The code of Aquamarine is #7FFFD4

2. case-end

   aim: **maps** the **current column cells** to a **enumerate set** and generate a new column

   syntax1: for multi-columns condition

```
case when {cond1} then {result1}
    when {cond2} then {result2}
    ......
    else {resultn}
    end as {new_name}
from [table]
where {conditions}
-- the results will automatically form a column
```

```
-- example
-- two column will be shown in the console
select s.english_name,
case
    when s.latitude is null then 'missing'
    when s.district in ('Luohu','Futian','Nanshan') then 'Inside Area'
    else 'Outside Area' end as area
from stations s;
```

syntax2: for single-column condition

```
case {col}
    when {form1} then {result1}
    when {form2} then {result2}
    ......
    else {resultn}
    end as {new_name}
from [table]
where {conditions}
-- use '=' to match the col and form1, form2...
-- the results will automatically form a column
```

```
-- example
-- only one column will be shown in the console
select '=' ||
case s.district
    when 'Luohu' then 1
    when 'Futian' then 1
    when 'Nanshan' then 1
else 0 end as Inside_or_not
from stations s
```

3. Useful Functions

upper and lower

```
-- convert a column or a string to uppercase or lowercase
upper([{col} | {str}])
lower([{col} | {str}])
```

trim

```
-- delete some sybolic strings in the head or in the tail
select trim([both | leading | trailing] {str_pattern} from [{col},
{str_text}]) from {tab1};
-- 'from {tab1}' can be omitted when no columns appear behind the SELECT

--example
selecet trim(trailing '.' from '10.00.....'); -- 10.00
```

substr

```
-- from the begin_ position intercept string with a length of span_
-- the first character in the string is 1, not 0
substr([{col} | {str_text}], {begin_}, {span_})

--example
select substr('hello world', 5, 3); -- 'o w'
```

replace

```
-- replace all the str_pattern with new_pattern in the text
replace([{col} | {text}], {str_pattern}, {new_pattern})

--example
select replace('sheep', 'ee', 'i');
```

length

```
-- compute the number of bytes of a string
select length('hello'); -- 5
select length('你好');  -- 2 in postgreSQL, depends on the product
```

round and trunc

```
-- Round to retain specified significant figures
select round(50.123, 2); -- 50.12
select round(50.125, 2); -- 50.13

-- truncates a value to the specified precision
select trunc(123.456, 2); -- 123.45
select trunc(123.456, 0); -- 123
select trunc(123.456, -2); -- 100
```