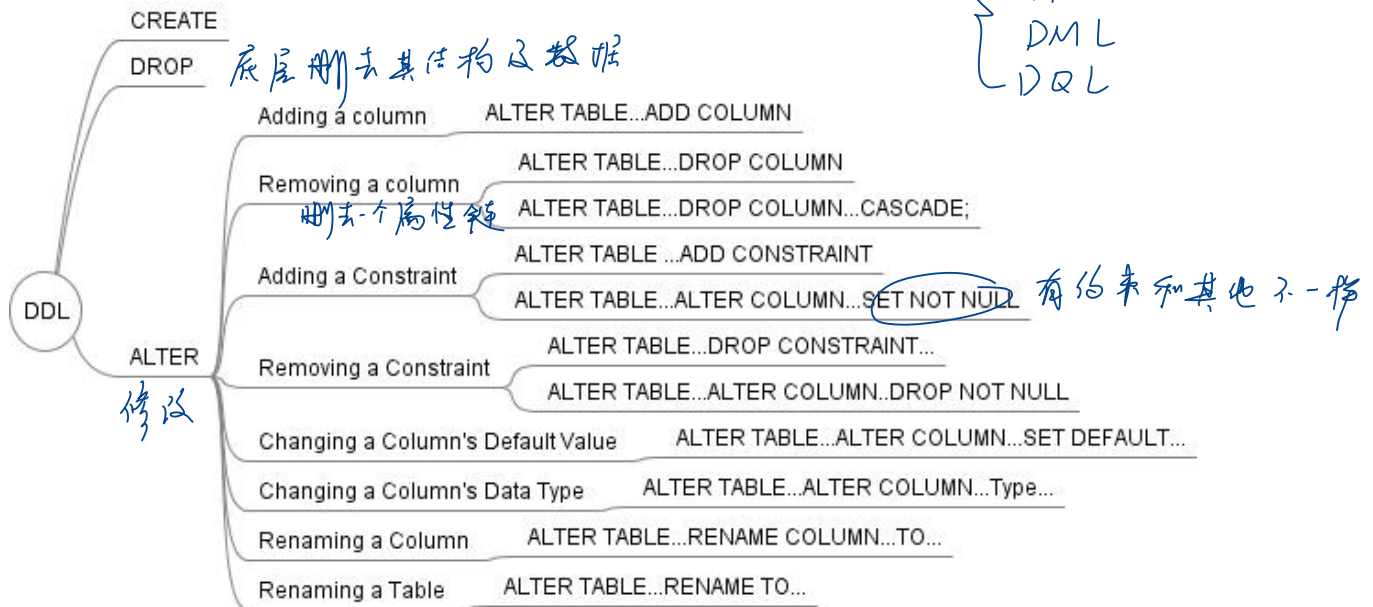


DDL

In a relational database, the raw data is stored in tables, so the majority of this chapter is devoted to explaining how tables are created and modified and what features are available to control what data is stored in the tables

Data definition language: **drop, create, alter**



Here we use following tables for example

```
create table dept(  
  id int primary key,  
  name varchar(40)  
);  
  
insert into dept values(1, '开发部');  
insert into dept values(2, '测试部');  
  
create table emp(  
  id int primary key,  
  name varchar(30),  
  salary numeric(9,2),  
  deptId int references dept(id)  
);
```

Alter

Experiment 1:

Adding a column

```
ALTER TABLE table_name ADD COLUMN description text;
```

```
ALTER TABLE emp ADD column phone varchar(30);  
ALTER TABLE emp ADD column office varchar(30) check (office <> '');
```

带上约束

Tips: The new column is initially filled with whatever default value is given (null if you don't specify a `DEFAULT` clause).

Experiment 2-1:

Removing a column

```
ALTER TABLE table_name DROP COLUMN description;
```

```
ALTER TABLE emp DROP column phone;
```

删除属性列:

名称不加单引号,
可以认为是变量

```
ALTER TABLE emp DROP column deptId;
```

Tips: Whatever data was in the column disappears. Table constraints involving the column are dropped, too. However, if the column is referenced by a foreign key constraint of another table, PostgreSQL will not silently drop that constraint.

Experiment 2-2:

Removing a column using CASCADE

→ 删外键

```
ALTER TABLE table_name DROP COLUMN description CASCADE;
```

```
ALTER TABLE emp DROP column deptId CASCADE;
```

Tips: You can authorize dropping everything that depends on the column by adding `CASCADE`:

Experiment 3:

Adding a Constraint

加约束的语法

```
ALTER TABLE table_name ADD CHECK (name <> '');  
ALTER TABLE table_name ADD CONSTRAINT some_name UNIQUE (product_no);  
ALTER TABLE table_name ADD FOREIGN KEY (fk) REFERENCES product_groups;
```

.....

```
ALTER TABLE dept ADD PRIMARY KEY(id);
ALTER TABLE emp ADD FOREIGN KEY (deptId) REFERENCES dept(id);
ALTER TABLE emp ADD CONSTRAINT phone_unique UNIQUE (phone);
ALTER TABLE dept ADD COLUMN id int;
```

Tips: To add a not-null constraint, which cannot be written as a table constraint, use this syntax:

```
ALTER TABLE table_name ALTER COLUMN col_name SET NOT NULL;
```

```
ALTER TABLE emp ALTER COLUMN name SET NOT NULL;
```

特殊的方法

Experiment 4-1:

Removing a Constraint

```
ALTER TABLE table_name DROP CONSTRAINT some_name;
```

```
ALTER TABLE emp DROP CONSTRAINT some_name; -- some_name key name
```

Experiment 4-2:

Removing a Constraint Not Null

```
ALTER TABLE table_name ALTER COLUMN col_name DROP NOT NULL;
```

```
ALTER TABLE emp ALTER COLUMN name DROP NOT NULL;
```

Tips: Not-null constraints do not have names.

Experiment 5:

Changing a Column's Default Value

```
ALTER TABLE table_name ALTER COLUMN col_name SET DEFAULT val;
```

```
ALTER TABLE emp ALTER COLUMN salary SET DEFAULT 0.00;
ALTER TABLE emp ALTER COLUMN salary DROP DEFAULT;
```

Tips: Note that this doesn't affect any existing rows in the table, it just changes the default for future `INSERT` commands.

Experiment 6:

Changing a Column's Data Type

```
ALTER TABLE table_name ALTER COLUMN col_name Type col_type;
```

```
ALTER TABLE emp ALTER COLUMN salary TYPE numeric(9,4);
```

Experiment 7:

Renaming a Column

```
ALTER TABLE table_name RENAME COLUMN old_name TO new_name;
```

```
ALTER TABLE emp RENAME COLUMN phone TO Tel_No;
```

Experiment 8:

Renaming a Table

```
ALTER TABLE table_old RENAME TO table_new;
```

```
ALTER TABLE emp RENAME TO employee;
```

DML

Data manipulation language: **insert, update, delete**

Using the following table

```
drop table emp;
drop table dept;

create table dept(
  id int primary key,
  name varchar(40)
);

create table emp(
  id int primary key,
  name varchar(30),
  salary numeric(9,2),
  deptId int references dept(id)
);
```

Experiment 9:

Insert— create new rows in a table

```
INSERT INTO table(...) VALUES(..);
```

<https://www.postgresql.org/docs/current/sql-insert.html>

```
INSERT INTO dept(id, name) values(1, '开发部');
INSERT INTO dept values('测试部', 2);
INSERT INTO dept(id, name) values
(3, '财务部'),
(4, '销售部'); \--insert multi-records
```

默认顺序操作

如果使用DDL 加行
被及顺序插对地方

Tips: When inserting a lot of data at the same time, consider using the [COPY](#) command. It is not as flexible as the INSERT command, but is more efficient.

Experiment 10:

Update— update rows of a table

```
UPDATE table SET ...;
```

赋值及修改

<https://www.postgresql.org/docs/current/sql-update.html>

```
UPDATE dept SET name='市场部' where id=4;
```

where 后为空则不更新. 但也不会报错

Tips: To update existing rows, use the [UPDATE](#) command. This requires three pieces of information:

1. The name of the table and column to update
2. The new value of the column
3. Which row(s) to update

Experiment 11:

Delete— delete rows of a table

```
DELETE FROM table WHERE ...;
```

<https://www.postgresql.org/docs/current/sql-delete.html>

```
DELETE FROM dept WHERE name='市场部';
DELETE FROM dept;
```

清空表

Lab2 truncate

Tips: You can also remove groups of rows matching a condition, or you can remove all rows in the table at once

DQL

SQL: structured query language

Here we use shenzhen_metro.sql, please download first

The general syntax:

```
[WITH with_queries] SELECT select_list FROM table_expression [sort_specification]
```

<https://www.postgresql.org/docs/current/sql-select.html>

Experiment 12:

Select

```
create table1(  
    a integer,  
    b integer,  
    c integer  
);  
insert into table1 values(1,2,3);  
  
SELECT * FROM table1;  
SELECT a, b + c FROM table1;  
SELECT 3 * 4; -- as a calculator  
SELECT random();
```

Tips: "*" present for all attribute or any attribute.

Experiment 13:

Select...(as)...from...where ... or, and, not

```
WHERE condition
```

```
select station_id , english_name , chinese_name  
from stations  
where district = 'Nanshan' or district = 'Bao'an';  
  
select station_id as sid, english_name as e_n, chinese_name c_n  
from stations  
where district = 'Nanshan' and latitude >22.54;  
  
select station_id, english_name, chinese_name  
from stations  
where not(district = 'Nanshan' or district = 'Bao'an');
```

Tips: where *condition* is any expression that evaluates to a result of type `boolean`. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns true when the actual row values are substituted for any variable references.

Experiment 14:

>, <, >=, <=, != or <>

<https://www.postgresql.org/docs/13/sql-syntax-lexical.html#SQL-SYNTAX-IDENTIFIERS> Table 4.2
Operator Precedence

Experiment 15:

in, not in

```
select station_id, english_name, chinese_name
from stations
where district in ('Nanshan' , 'Bao'an');

select station_id, english_name, chinese_name
from stations
where district not in ('Nanshan' , 'Bao'an');
```

Experiment 16:

like, %, _

```
select station_id, english_name, chinese_name
from stations
where chinese_name like '%湾%';

select station_id, english_name, chinese_name
from stations
where chinese_name like '湾%';

select station_id, english_name, chinese_name
from stations
where chinese_name like '___湾';
```

Experiment 17:

NULL, NOT NULL

```
select station_id, english_name, chinese_name
from stations
where latitude is NULL;

select station_id, english_name, chinese_name
from stations
where latitude is not NULL;
```

Tips: "NULL" is not a value, so we use "is" connect with NULL/NOT NULL.

a whole formed by combining
several (typically disparate)
elements


Aggregate Functions

Aggregate functions compute a single result from a set of input values. The built-in general-purpose aggregate functions are listed in <https://www.postgresql.org/docs/13/functions-aggregate.html> Table 9.55

Experiment 18:

count

```
select count(*)  
from stations  
where latitude is NULL;
```



| | count |
|---|-------|
| 1 | 20 |

Tips: Computes the number of input rows in which the input value is not null.

Experiment 19:

max, min, avg

```
select avg(latitude), min(latitude), max(latitude)  
from stations  
where latitude is not NULL;
```

| | avg | min | max |
|---|--------------------|----------|----------|
| 1 | 22.576341321556882 | 22.47944 | 22.78889 |

Experiment 20:

group by

```
select count(district), district  
from stations  
where district <> ''  
group by district;
```

| | count | district |
|---|-------|----------|
| 1 | 25 | Bao'an |
| 2 | 23 | Luohu |
| 3 | 51 | Futian |
| 4 | 22 | Longgang |
| 5 | 49 | Nanshan |
| 6 | 9 | Longhua |

make (something) denser or
more concentrated

Tips: GROUP BY will condense into a single row all selected rows that share the same values for the grouped expressions.

<https://www.postgresql.org/docs/13/sql-select.html#SQL-GROUPBY>

Experiment 21:

having

The HAVING statement can only be used in an aggregate function. An aggregate function is when the values of multiple rows are grouped together to form a single summary value

```
select count(district), district
from stations
group by district
having district <> '';
```

*can NOT change the order
MUST group THEN having*

| | count | district |
|---|-------|----------|
| 1 | 25 | Bao'an |
| 2 | 23 | Luohu |
| 3 | 51 | Futian |
| 4 | 22 | Longgang |
| 5 | 49 | Nanshan |
| 6 | 9 | Longhua |

Tips: After passing the WHERE filter, the derived input table might be subject to grouping, using the GROUP BY clause, and elimination of group rows using the HAVING clause.

<https://www.postgresql.org/docs/13/sql-select.html#SQL-HAVING>

Experiment 22:

distinct

If something is distinct from something else of the same type, it is different or separate from it.

```
select count(distinct (district))
from stations
where latitude > 22.6;
```

| count |
|-------|
| 3 |

```
select distinct (district)
from stations
where latitude > 22.6;
```

| district |
|----------|
| Bao'an |
| Longgang |
| Longhua |

Experiment 23:

order by...asc/dec

ascending descending

```
select station_id, english_name, chinese_name, latitude
from stations
order by latitude asc; --default is asc
```

```
select station_id, english_name, chinese_name, latitude
from stations
order by latitude desc;
```

```
select station_id, english_name, chinese_name, latitude
from stations
order by latitude desc nulls last;
```

Tips: null, order by...desc/asc...nulls last/first

Experiment 24:

limit , offset

```
select station_id, english_name, chinese_name, latitude
from stations
order by latitude;
```

| station_id | english_name | chinese_name | latitude |
|------------|----------------|--------------|----------|
| 1 | 49 Chiwan | 赤湾 | 22.47944 |
| 2 | 50 Shekou Port | 蛇口港 | 22.47944 |
| 3 | 51 Sea World | 海上世界 | 22.485 |
| 4 | 53 Dongjiaotou | 东角头 | 22.48583 |
| 5 | 52 Shuiwan | 水湾 | 22.48833 |
| 6 | 54 Wanxia | 湾厦 | 22.49333 |
| 7 | 55 Haiyue | 海月 | 22.5 |
| 8 | 56 Dengliang | 登良 | 22.50917 |
| 9 | 73 Yitian | 益田 | 22.51639 |

```

select count(district),district
from stations
where district <> ''
group by district
order by count(district) limit 3;

select count(district),district
from stations
where district <> ''
group by district
order by count(district) limit 3 offset 2;

```

deal with the select data

Experiment 25:

||

```

select '南山区地铁站平均经纬度为:' || avg(latitude) || '::~' || avg(longitude)
from stations
where latitude is not NULL and longitude is not null and district='Nanshan';

```

Experiment 26:

as

```

select '南山区地铁站平均经纬度为:' || avg(latitude) || '::~' || avg(longitude) as 输出信息
from stations
where latitude is not NULL and longitude is not null and district='Nanshan';

```

Experiment 27:

case...when...then...else...end

```

select district,avg(longitude) ,
case
when avg(longitude)>114
then 'North'
else 'West'
end as sign
from stations
where longitude is not null
group by district;

```

some other functions

- upper,lower
- trim:trim(' Oops ') return 'Oops'
- substr:substr('Nanshan', 3, 2) return 'ns'
- replace:replace('Sheep', 'ee', 'i') return 'Ship'
- length
- round, trunc

Subquery

SQL statements that use the EXISTS condition in PostgreSQL are very inefficient since the sub-query is RE-RUN for EVERY row in the outer query's table. There are more efficient ways to write most queries, that do not use the EXISTS condition.

Experiment 28:

exist

The argument of EXISTS is an arbitrary SELECT statement, or subquery.

The subquery is evaluated to determine whether it returns any rows.

If it returns at least one row, the result of EXISTS is “true”; if the subquery returns no rows, the result of EXISTS is “false”.

```
select station_id, english_name,chinese_name from stations where exists
(select district from stations where district='Nanshan');

select station_id, english_name,chinese_name from stations where exists
(select district from stations where district='Nansha');
```

Experiment 29:

in

```
select station_id, english_name,chinese_name,district from stations where
district in
(select distinct (district)from stations where latitude>22.6);
```

Tips:

SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is “true” if any equal subquery row is found. The result is “false” if no equal row is found (including the case where the subquery returns no rows).

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.