PostgreSQL SELECT 语句用于从数据库中选取数据。

**select语句**

**语法**

[ WITH ]

SELECT [ DISTINCT ]

FROM from\_item [, ...]

WHERE condition

GROUP BY grouping\_element [, ...]

HAVING condition

ORDER BY expression

LIMIT count OFFSET start

**单表查询**

select \* from t1;--查看全部字段

select id,name,age from t1;

select id,name from t1;--查看部分字段

select id,name,age+10 from t1;--字段中包含表达式

select id,name,age,5+10 from t1;--表达式与字段无关

select 2+3;

select 'hello world';

**别名使用**

select id i,name n from t1; --列级别的别名

select id as i,name as n from t1;

select 2+3 as int;

select \* from t1 as t11;--表级别的别名

select t11.id,t11.name,t11.age from t1 as t11;

select t11.id as i,t11.name as n,t11.age as a from t1 as t11;

**等值查询**

select \* from t1 where id=5;--数字类型等值查询

select \* from t1 where name='1\_user'; --字符类型等值查询

**范围查询**

select \* from t1 where id>=15;--范围查询

select \* from t1 where id between 10 and 20;--范围查询

范围运算符：>、>=、<、<=、<>、!=

**去重查询**

SELECT DISTINCT age from t1;--单列去重

SELECT DISTINCT name,age from t1;--多列去重

注意：null值会被计算，null值被认为是相同的。

**排序**

select \* from t1 order by age;--按年龄排序

select \* from t1 where age>=25 order by age;--过滤条件后排序

select \* from t1 where age>=25 order by age,name;--多列排序

select \* from t1 where age>=25 order by age desc;--降序排列

select \* from t1 where age>=25 order by age desc,name;--多列排序

select \* from t1 order by age nulls first;--null值放前面

select \* from t1 order by age desc nulls last;--null放后面

create table points(p point);

insert into points values (point '(1,1)'), (point '(3,2)'), (point'(6,3)'),(point '(5,5)'), (point '(7,8)'), (point '(8,6)');

select \* from points order by p <-> point '5,5';--距离排序

**分组查询**

常用聚合函数有 count、sum、max、min、avg

select count(\*) from t1;--统计满足条件的总行数

select count(id) from t1;--统计指定字段的总行数

select count(distinct id) from t1;--计算非重复结果的数目

select age,count() from t1 group by age;--统计不同年龄的人数

select age,count() from t1 group by age having count(\*)>=2;--过滤分组后的行

select class\_id,age,count() from student group by grouping sets(class\_id,age,());--多字段统计

select class\_id,age,count() from student group by rollup(class\_id,age);

select class\_id,age,count(\*) from student group by cube(class\_id,age);

**分页**

语法：select … from … limit .. offset ..

select \* from t1 limit 5;--查询前5行数据

select \* from t1 limit 5 offset 2;--从第2行起，查询前5行数据

**多表查询**

多表查询支持内连接、外连接和交叉连接。

语法：t1 join\_type t2[on join\_condition|using join\_column]

**内连接**

内连接(inner join)：在表中存在至少一个匹配时返回行

语法：

t1 [inner] join t2 on join\_condition--常用

t1 [inner] join t2 using (jion\_column)--两张表的连接字段名相同时使用

t1 natural [inner] join t2

SELECT student\_name, class\_name FROM student inner join class on student.class\_id = class.class\_id;

SELECT student\_name, class\_name FROM student inner join class using(class\_id);--连接字段相同

SELECT student\_name, class\_name FROM student, class WHERE student.class\_id = class.class\_id;

如果两张表中有相同字段，要明确使用 表名.列名 来指定

SELECT student\_name, class\_name FROM student a, class b WHERE a.class\_id = b.id;

SELECT student\_name, class\_name FROM student a, class b WHERE a.class\_id = b.id AND a.age >= 14; --加过滤条件

**外连接**

外连接分为：

Left outer join(左外连接)：从左表(t1)返回所有的行，即使右表(t2)中没有匹配。如果右表中没有匹配，则结果为 NULL。

right outer join(右外连接)：从右表(t2)返回所有的行，即使左表(t1)中没有匹配。如果左表中没有匹配，则结果为 NULL。

full outer join(全外连接)：只要左表(t1)和右表(t2)其中一个表中存在匹配，则返回行。

left outer join(左外连接)

语法：

t1 left [outer] join t2 on join\_condition

t1 left [outer] join t2 using join\_column

SELECT student\_name, class\_name FROM student left join class on student.class\_id = class.id;

SELECT student\_name, class\_name FROM student left join class on student.class\_id = class.id and class\_name='one';

SELECT student\_name, class\_name FROM student left join class on student.class\_id = class.id where class\_name='one';

后两条执行结果不同，是因为on中的约束是在连接之前被处理，而where中的约束是在连接之后被处理。

right outer join(右外连接)

SELECT student\_name, class\_name FROM student right join class on student.class\_id = class.id;

full outer join(全外连接)

SELECT student\_name, class\_name FROM student full join class on student.class\_id = class.id;

**交叉连接(笛卡尔积)**

SELECT student\_name, class\_name FROM student cross join class;

SELECT student\_name, class\_name FROM student inner join class on true;

SELECT student\_name, class\_name FROM student,class;

**多表连接复合查询**

SELECT student\_name, class\_name FROM student inner join class on student.class\_id = class.class\_id where class.class\_id=1 and student\_name='zhangsan';

注意：不用关联太多的表，不然会影响SQL语句的执行性能。

**子查询**

子查询又称为内部查询、嵌套查询，指的是在 PostgreSQL 查询中的 WHERE 子句中嵌入查询语句。

一个 SELECT 语句的查询结果能够作为另一个语句的输入值。

子查询可以与 SELECT、INSERT、UPDATE 和 DELETE 语句一起使用，并可使用运算符如 =、<、>、>=、<=、IN、BETWEEN 等。

**子查询必须遵循以下规则：**

1.子查询必须用括号括起来。

2.子查询在 SELECT 子句中只能有一个列，除非在主查询中有多列，与子查询的所选列进行比较。

3.ORDER BY 不能用在子查询中，虽然主查询可以使用 ORDER BY。可以在子查询中使用 GROUP BY，功能与 ORDER BY 相同。

4.子查询返回多于一行，只能与多值运算符一起使用，如 IN 运算符。

5.BETWEEN 运算符不能与子查询一起使用，但是，BETWEEN 可在子查询内使用。

**SELECT 语句中的子查询使用**

子查询通常与 SELECT 语句一起使用。基本语法如下：

SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

WHERE column\_name OPERATOR

(SELECT column\_name [, column\_name ]

FROM table1 [, table2 ]

[WHERE])

OPERATOR有4种形式：

1. IN： expression [NOT] IN (statement)。

2. EXISTS： [NOT] EXISTS (statement)。

3.比较运算符： >,<,=,!=

4.ANY|ALL|SOME

SELECT \* FROM student WHERE class\_id in (1,2,3);

SELECT \* FROM student WHERE class\_id not in (1,2,3);

SELECT \* FROM student WHERE class\_id in (select class\_id FROM class where class\_name = 'one');

SELECT \* FROM student s WHERE EXISTS (SELECT 1 FROM class c WHERE s.class\_id=c.class\_id AND c.class\_name = 'one');

SELECT \* FROM student WHERE class\_id = (SELECT class\_id FROM class c WHERE class\_name = 'one');--标量子查询

SELECT \* FROM student WHERE class\_id = any(SELECT class\_id FROM class c WHERE class\_name = 'one' or class\_name='two');

**INSERT 语句中的子查询使用**

INSERT 语句使用子查询返回的数据插入到另一个表中。

INSERT INTO t2 SELECT \* FROM t1 WHERE ID IN (SELECT ID FROM t1 where age>=25) ;

**UPDATE 语句中的子查询使用**

当通过 UPDATE 语句使用子查询时，表中单个或多个列被更新。

UPDATE student SET score = score+10 WHERE score >= (SELECT avg(score) FROM student);

**DELETE 语句中的子查询使用**

DELETE FROM student WHERE score <= (SELECT avg(score) FROM student);

**模糊匹配查询**

like查询

like通常需要和通配符配合使用，PostgreSQL提供了两个通配符：% 和 \_， \_代表任意单个字符，%代表任意0到多个字符。

'abc' LIKE 'abc' true

'abc' LIKE 'a%' true

'abc' LIKE '\_b\_' true

'abc' LIKE 'c' false

对于需要匹配字符中 \_ 或 % 这俩字符的需求，可以使用ESCAPE来指定转义字符，比如!为转义字符，就可以在LIKE中用来转义。

select tablename from pg\_tables where tablename like 'tbl!\_%'ESCAPE '!';

select \* from student where student\_name like 'zh%';--查询姓名以 zh 打头的学生

select \* from student where student\_name like '%o%';--查询姓名中包含 o 的学生

select \* from student where student\_name like 'zhang\_\_\_';

select \* from student where student\_name ilike 'zh%';--忽略大小写

其他运算符：

~~：同like

~~\*：同ilike

!~~：同 not like

!~~\*：同not ilike

**similar to查询**

利用正则表达式匹配

'abc' SIMILAR TO 'abc' true

'abc' SIMILAR TO 'a' false

'abc' SIMILAR TO '%(b|d)%' true

'abc' SIMILAR TO '(b|c)%' false

select \* from student where student\_name similar to '(zh|sh)%';--匹配名字以zh或sh打头的学生

注意：

1.带前缀的模糊查询，如'zh%'，可以使用b-tree索引进行优化

2.带后缀的模糊查询，如'%u'，可以使用reverse()反转函数建立索引进行优化

create index idx\_student\_name on student(reverse(name) text\_pattern\_ops);

select \* from student where reverse(student\_name) like ‘nas%';

3.不带前缀及后缀的模糊查询和正则表达式查询，可以使用pg\_trgm的gin索引进行优化。

**pg\_trgm插件**

普通的B树索引，仅能用于like前缀查询，而实际的业务场景中，%xxx% 这种模糊查询才是使用最广泛的，在PostgreSQL中，恰好有针对这种情况的优化，就是trgm组件，结合gin/gist索引，达到实现模糊搜索的目的。

pg\_trgm的主要设计目的，是支持字符串的相似度检查，以及针对字符串的相似查询。其原理，是把字符串处理为三个字符长度的组（认为每个字符串自带两个前缀空格，和一个后缀空格）：比如字符串“cat”会被切割为 “ c”、 “ ca”、 “cat”以及 “at ”。

select show\_trgm(‘cat’);

而gin/gist索引，就是在这样的三元组之上创建，可以明显认识到，这将会是个庞大的索引。

pg\_trgm提供了全文本搜索能力，pg\_trgm是一个非常强的插件，对文本搜索场景性能提升非常有效，100万左右的数据量，性能提升有100倍以上。该插件被认为是“trusted”，它可以由对当前数据库具有CREATE权限的非超级用户安装。

创建插件

create extension pg\_trgm;

创建测试表并插入数据

create table t1 (id int,info1 text,info2 text);

insert into t1 select n,md5(n::text),md5(n::text) from generate\_series(1,1000000) as n;

CREATE INDEX ON t1 USING gin (info1 gin\_trgm\_ops);

CREATE INDEX ON t1 USING gist(info2 gist\_trgm\_ops);

执行全文本搜索

SELECT \* FROM t1 WHERE info1 LIKE '%xxx%';

SELECT \* FROM t1 WHERE info2 LIKE '%xxx%';

从数据上看，gin索引的sql响应时间，明显强于gist，但gin有个小的致命的问题：因为trgm切割出来的都是3长度的组，对于1，2这种长度的like内字符串，匹配速度慢!

SELECT \* FROM t1 WHERE info1 LIKE '%xx%‘;

**集合运算**

语法格式：

query1 UNION [ALL] query2--并

query1 INTERSECT [ALL] query2--交

query1 EXCEPT [ALL] query2--差

UNION操作符返回位于一个或者两 个结果集中的全部行。INTERSECT操作符返回同时 位于两个结果集中的所有行。EXCEPT操作符返回 位于第一个结果集但不在第二个结果集中的行。

UNION [ALL]

SELECT \* FROM student WHERE id = 1 UNION SELECT \* FROM student where id = 1;

SELECT \* FROM student WHERE id = 1 UNION ALL SELECT \* FROM student where id = 1;

INTERSECT [ALL]

SELECT \* FROM student WHERE age > 14 INTERSECT SELECT \* FROM student where age<16;

SELECT \* FROM student WHERE age > 14 INTERSECT ALL SELECT \* FROM student where age<16;

EXCEPT [ALL]

SELECT \* FROM student WHERE age > 14 EXCEPT SELECT \* FROM student where age>15;

SELECT \* FROM student WHERE age > 14 EXCEPT ALL SELECT \* FROM student where age>15;

**WITH查询**

WITH查询的特性被称为CTE(common table expressions)，它是在复杂查询中定义一个辅助语句(临时表)，常用于复杂查询或递归查询场景。

CTE优点：

1.CTE可以简化SQL代码，减少SQL嵌套层数，提供SQL代码的可读性。

2.CTE的复杂语句只需要计算一次，在主查询中可以多次使用。

3.当不需要共享查询结果时，相比视图更轻量。

**复杂查询使用CTE**

案列一：

with t as(

select generate\_series(1,10)

)

select \* from t;

案例二：

with t as (

select \* from t1

)

select \* from t;

案例三：

with a as (

select \* from t1 where id = 1

),

b as (

select \* from t2 where id = 2

)

select \* from a

union all

select \* from b;

**递归查询使用CTE**

WITH RECURSIVE t(n) AS (

SELECT 1

UNION ALL

SELECT n+1 from t where n< 5

)

SELECT n FROM t;

1.定义一个临时表t(n)

2.SELECT 1，这个是非递归项，决定临时表的列名为n，值为1

3.UNION ALL，求并集，不去重

4.SELECT n+1 FROM t WHERE n<5，这个是递归项(对查询自身输出的引用)，n<5作为条件,并将n+1作为结果填充临时表

5.使用SELECT n FROM t，查询临时表

这个是将临时表列名指定在第一行

WITH RECURSIVE t(a,b,c) AS

(

SELECT 1,1,1

UNION ALL

SELECT 1+a,2+b,3+c FROM t WHERE a<10

)

SELECT \* FROM t;

这个第一行没有指定列名，然后列名由第一个查询返回结果确定

WITH RECURSIVE t AS

(

SELECT 1 AS a,1 AS b,1 AS c

UNION ALL

SELECT 1+a,2+b,3+c FROM t WHERE a<10

)

SELECT \* FROM t;

定义下面这样的表，存储每个区域（省、市、区）的id，名字及上级区域的pid

CREATE TABLE area(id text, pid text, name text);

INSERT INTO area VALUES('002', 0, '浙江省');

INSERT INTO area VALUES('001', 0, '广东省');

INSERT INTO area VALUES('003', '002', '衢州市');

INSERT INTO area VALUES('004', '002', '杭州市');

INSERT INTO area VALUES('005', '002', '湖州市');

INSERT INTO area VALUES('006', '002', '嘉兴市');

INSERT INTO area VALUES('007', '002', '宁波市');

INSERT INTO area VALUES('008', '002', '绍兴市');

INSERT INTO area VALUES('009', '002', '台州市');

INSERT INTO area VALUES('010', '002', '温州市');

INSERT INTO area VALUES('011', '002', '丽水市');

INSERT INTO area VALUES('012', '002', '金华市');

INSERT INTO area VALUES('013', '002', '舟山市');

INSERT INTO area VALUES('014', '004', '上城区');

INSERT INTO area VALUES('015', '004', '下城区');

INSERT INTO area VALUES('016', '004', '拱墅区');

INSERT INTO area VALUES('017', '004', '余杭区');

INSERT INTO area VALUES('018', '011', '金东区');

INSERT INTO area VALUES('019', '001', '广州市');

INSERT INTO area VALUES('020', '001', '深圳市');

WITH RECURSIVE t AS (

SELECT \* FROM area WHERE id= '018'

UNION ALL

SELECT area.\* from area,t where area.id=t.pid

)

SELECT \* FROM t;

1.定义一个临时表t

2.  SELECT \* FROM area WHERE id= '018'，查询id为018的行填入临时表

3.  SELECT area.\* from area,t where area.id=t.pid，这个是递归查询，将上次查出来的行的pid带入连接查询

4.SELECT \* FROM t，查询临时表

WITH RECURSIVE t AS (

SELECT \* FROM area WHERE id= '018'

UNION ALL

SELECT area.\* from area,t where area.id=t.pid

)

SELECT string\_agg(name,' ') from (select name from t order by id) n;

**数据抽样**

select \* from t1 order by random() limit 1;

explain analyze select \* from t1 order by random() limit 1;--走全表扫描

抽样方法：SYSTEM、BERNOULLI

SYSTEM抽样方法：基于块级别的抽样

抽样因子：表示返回总数的的百分比，如1000000\*0.01%=100

select \* from t1 tablesample system(0.01);

注意：采样比例过低，可能查不到数据

BERNOULLI抽样方法：基于行级别的抽样

select \* from t1 tablesample bernoulli(0.01);

**窗口函数**

窗口函数多用于进行数据统计，PostgreSQL内置了大量的窗口函数。

语法格式：select … function\_name OVER ([partition by … [order by …]])

partition by：分组，如果没有此项，则当作一个集合来处理

**avg() OVER()：**

select age,count(\*) over(partition by age) from t2;--查看

partition by age：根据age分组

**row\_number() OVER()**：当前行在其窗口中的行号，从1开始

select row\_number() over ( partition by age order by id desc) ,\* from t2;

select row\_number() over (order by id desc) ,\* from t2;--显示表所有记录的行号

**rank() OVER()**：带间隙的当前行排名，值相同，行号重复

select rank() over ( partition by age order by id desc) ,\* from t2;

**dense\_rank() OVER()**：不带间隙的当前行排名，值相同，行号重复

select dense\_rank() over ( partition by age order by id desc) ,\* from t2;

**lag() OVER()**：返回偏移量值

语法：lag(value [,offset [,default]])

value：指定要返回记录的字段

offset：指定行偏移量，默认为1

default：不存在offset偏移的行时用默认值填充，默认为null

select lag(id,1) over () ,\* from t2;

select lag(id,2,500) over () ,\* from t2;

**frist\_value() OVER()**：取分组中的第一行的字段值

select first\_value(id) over (partition by age) ,\* from t2;

select first\_value(id) over (partition by age order by id desc) ,\* from t2;--取分组数据中的最大值或最小值

**lsat\_value()**：取分组中最后一行的字段值

select last\_value(id) over (partition by age) ,\* from t2;

**nth\_value()**：取分组中指定行的字段值

语法：nth\_value(value,nth)

value：指定表的字段

nth：指定第几行，不存在则返回空

select nth\_value(id,2) over (partition by age) ,\* from t2;

**窗口函数别名的使用**

如果要多次调用窗口函数，可以使用窗口函数别名

语法：select … from … WINDOW window\_name AS (window\_definition)

WINDOW指定表的别名为window\_name，可以给OVER引用

select max(id) over(w),first\_value(name) over(w) ,\* from t2 window w as (partition by age);

**SELECT INTO 语句**

select into--从一个表复制数据，然后创建一个新表，接着把数据插入到这个新表中

select \* into t3 from t2;--复制所有列

select name,age into t3 from t2;--复制指定的列

select \* into t3 from t2 where age>14;--复制年龄大于14岁的记录

select id,name,age,class\_id into t3 from student left join class on student.class\_id = class\_class\_id;--复制多个表中的数据插入新表

select \* into t3 from t2 where 1=0;--创建一个新的空表

CREATE TABLE class(class\_id int primary key, class\_name text);

INSERT INTO class VALUES(1,'one');

INSERT INTO class VALUES(2,'two');

INSERT INTO class VALUES(3,'three');

INSERT INTO class VALUES(4,'four');

INSERT INTO class VALUES(5,'five');

INSERT INTO class VALUES(6,'six');

CREATE TABLE student(id int primary key,student\_name text,age int, score int,class\_id int);

INSERT INTO student VALUES(1, 'zhangsan', 14,70, 1);

INSERT INTO student VALUES(2, 'lisi', 15,80, 1);

INSERT INTO student VALUES(3, 'wangwu', 13 ,90, 2);

INSERT INTO student VALUES(4, 'zhaoliu',15,76, 2);

INSERT INTO student VALUES(5, 'shunqi',15,68, 3);

INSERT INTO student VALUES(6, 'zhouba',14,58, 3);

INSERT INTO student VALUES(7, 'wujiu',15,100, 4);

INSERT INTO student VALUES(8, 'zhengshi',14,42, 4);

INSERT INTO student VALUES(9, 'yiyi',14,88, 8);

INSERT INTO student VALUES(10, 'erer',14,70, 10);