# 函数

## 窗口函数

### 介绍

**窗口函数**可以进行排序、生成序列号等一般的聚合函数无法实现的高级操作；聚合函数将结果集进行计算并且通常返回一行。窗口函数也是基于结果集的运算。与聚合函数不同的是，窗口函数并不会将结果集进行分组合并输出一行；而是将计算的结果合并到基于结果集运算的列上。

排序用order by即可，但是要返回排名或者分组排序时，就需要用到窗口函数了。其结构如下：

select 排序函数/聚合函数 over(<partition by…> order by…)

如何理解窗口函数：

**窗口**：一个集合

**over (partition by a order by b) from T** 是指：把表T按照a列进行分组，然后，分别让每一个集合的记录按照b列进行排序。

于是，我们再使用一个排序函数，我们就可以得到一个新列，这一列的值就是每一条记录在它所在集合中的排序编号。

partiton by是可选的。如果不使用partition by，那么就是将整张表作为一个集合，最后使用排序函数得到的就是每一条记录根据b列的排序编号。

在SQL处理中，窗口函数都是最后一步执行，而且仅位于Order by字句之前.

### 排序窗口函数

有3种：

rank()

dense\_rank()

row\_number()

它们的区别是当碰到相同大小值的时候，编号方式不同，分别形如：

1，1，1，4，5，...

1，1，1，2，3，...

1，2，3，4，5，...

注意：这些序号是组内序号，和全局没关系

因此总结来说，窗口函数具有 group by 的分组功能和 order by 的排序功能。但是，它又与group by和order by同时使用的概念完全不同。

### 聚合窗口函数

在窗口函数中使用SUM、AVG、MAX、MIN计算的范围是排序后每条记录自身和排在它之前的所有记录。

比如按照时间排序，计算各个时间的销售总额，这种统计方法称为累计：

SUM(sale\_price)over(partition by 日期 order by 时间) as date\_sum

返回结果是，每一天截至每个时间点的销售总额。

另外，在聚合窗口函数中，可以指定汇总范围，这个功能称为框架。

SUM(sale\_price) over(order by product\_id rows 1 preceding)

自身+上1条记录

SUM(sale\_price) over(order by product\_id rows 1 following)

自身+下1条记录

SUM(sale\_price) over(order by product\_id rows between 1 preceding and 2 following)

自身+上1条记录+下2条记录

### 窗口函数中的order by

窗口函数中的order by只是决定窗口内的数据按什么顺序进行排序或计算，对最后返回结果的排列顺序并没有影响。例如：

rank() over(order by sale\_price) as ranking

返回结果是：

sale\_price ranking

3000 6

500 2

4000 7

1000 5

如果需要让结果按照ranking升序排列返回，还需要在语句末尾使用order by ranking。

只要掌握好窗口函数，再结合子查询或者case when的使用，就可以解决几乎所有在SQL中碰到的复杂问题。

### 例子

参考<https://blog.csdn.net/qq_28745235/article/details/78425162>

#### 分组limit

假设现有一批数据,字段中包含date\_id,数据格式yyyyMMdd. 现在需求为需要取出每天中前N条数据，那么sql来了(用到窗口函数):

create temp table dddd(id serial , date\_id int , name varchar(20));

insert into dddd(date\_id , name) values (20160301 , 'name1'),(20160301 , 'name2'),(20160301 , 'name3'),(20160302 , 'name4'),(20160302 , 'name5') ;

select id , date\_id , name

from

(

select \* , row\_number() over(partition by date\_id order by id ) as row\_id from dddd

) as t

where t.row\_id <= 2 ;

结果：

1;20160301;"name1"

2;20160301;"name2"

4;20160302;"name4"

5;20160302;"name5"

#### 数据准备

[参考](https://blog.csdn.net/qq_28745235/article/details/78425162)

数据准备

create table order1(name varchar(10),orderdate date,cost integer );

insert into order1 values('jack','2017-01-01',10);

insert into order1 values('tony','2017-01-02',15);

insert into order1 values('jack','2017-02-03',23);

insert into order1 values('tony','2017-01-04',29);

insert into order1 values('jack','2017-01-01',46);

insert into order1 values('jack','2017-04-06',42);

insert into order1 values('tony','2017-01-07',50);

insert into order1 values('jack','2017-01-08',55);

insert into order1 values('mart','2017-04-08',62);

insert into order1 values('mart','2017-04-09',68);

insert into order1 values('neil','2017-05-10',12);

insert into order1 values('mart','2017-04-11',75);

insert into order1 values('neil','2017-06-12',80);

insert into order1 values('mart','2017-04-13',94);

insert into order1 values('jack','2017-06-06',23);

insert into order1 values('jack','2017-09-06',42);

#### 聚合函数+over（明细和聚集）

注意：这里演示是使用postgresql执行的

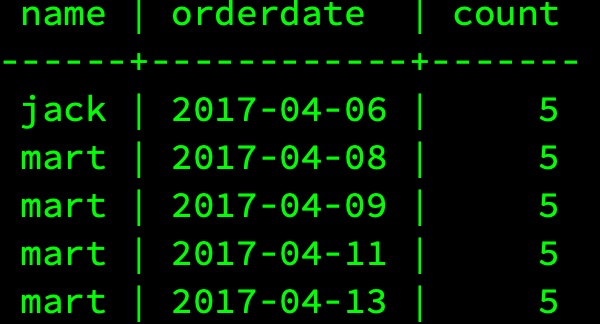
如果我们想要显示聚集前的数据和聚集后的数据，就适合使用窗口函数。

假如说我们想要查询在2017年4月份购买过的顾客及总人数,我们便可以使用窗口函数去去实现

select name,orderdate,count(\*) over() --over()里什么都不写，意味着查询结果集不需要分组，那么它前面的聚合就是在结果集中累积。

from order1

where substring(to\_char(orderdate, 'yyyy-MM-dd') from 1 for 7)='2017-04';



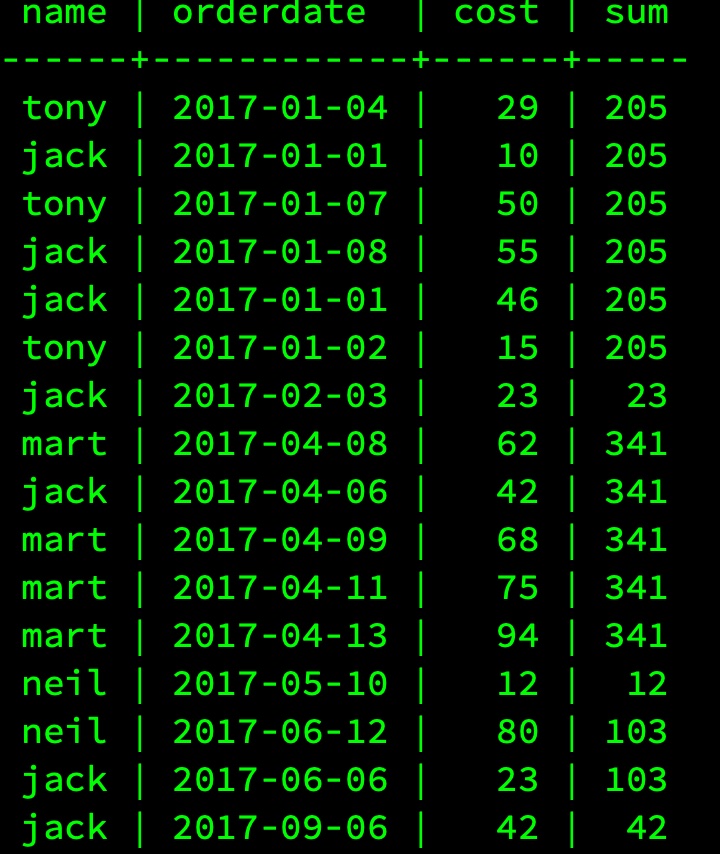
#### partition by子句（明细和分组聚集）

Partition By子句可以称为查询分区子句，非常类似于Group By，都是将数据按照边界值分组，而Over之前的函数在每一个分组之内进行，如果超出了分组，则函数会重新计算.意思是聚集函数在下一个分组中计算时从零开始。

我们想要去看顾客的购买明细及月购买总额（简单相加即可）,可以执行如下的sql

select name ,orderdate,cost,sum(cost) over(partition by extract(MONTH from orderdate))

from order1; （结果中每条记录都带有聚集函数聚集的结果）



#### order by子句（明细和分组累积）

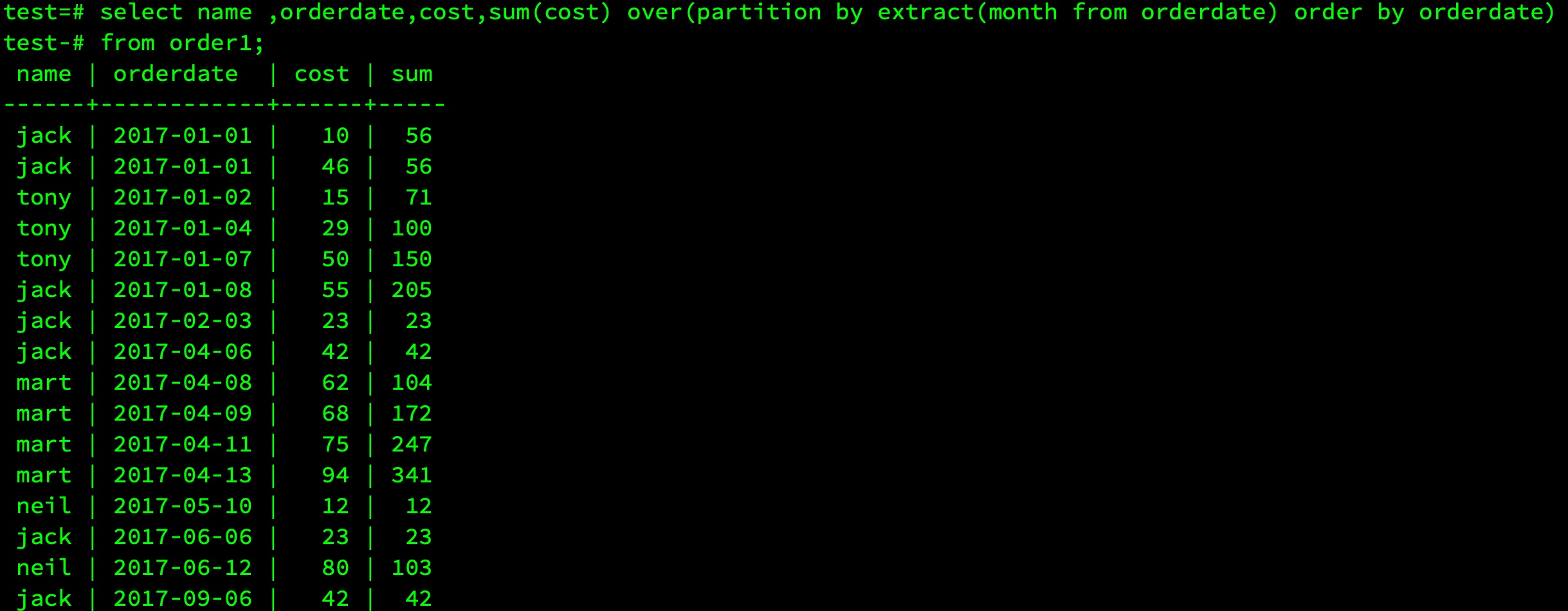
上述的场景,假如我们想要将cost按照月进行累加（注意累加和相加的区别）.这时我们引入order by子句.

order by子句会让输入的数据强制排序（文章前面提到过，窗口函数是SQL语句最后执行的函数，因此可以把SQL结果集想象成输入数据）。Order By子句对于诸如Row\_Number()，Lead()，LAG()等函数是必须的，因为如果数据无序，这些函数的结果就没有任何意义。因此如果有了Order By子句，则Count()，Min()等计算出来的结果就没有任何意义。

我们在上面的代码中加入order by

select name ,orderdate,cost,sum(cost) over(partition by extract(month from orderdate) order by orderdate)

from order1;



#### window子句

我们在上面已经通过使用partition by子句将数据进行了分组的处理.如果我们想要更细粒度的划分，我们就要引入window子句了.

我们首先要理解两个概念:

- 如果只使用partition by子句,未指定order by的话,我们的聚合是分组内的聚合.

当同一个select查询中存在多个窗口函数时,他们相互之间是没有影响的.

每个窗口函数应用自己的规则.

window子句：

- PRECEDING（preceding）：往前

- FOLLOWING(following)：往后

- CURRENT ROW(current row)：当前行

- UNBOUNDED(unbounded)：起点，UNBOUNDED PRECEDING 表示从前面的起点， UNBOUNDED FOLLOWING：表示到后面的终点

我们按照name进行分区,按照购物时间进行排序,做cost的累加.

如下我们结合使用window子句进行查询。

select name ,orderdate,cost,

sum(cost) over() as sample1,--所有行相加

sum(cost) over(partition by name) as sample2,--按name 分组，组内数据相加

sum(cost) over(partition by name order by orderdate) as sample3,--按name分组并按orderdate排序（这里注意，一次累加的是下一个orderdate的所有数据），组内数据累加

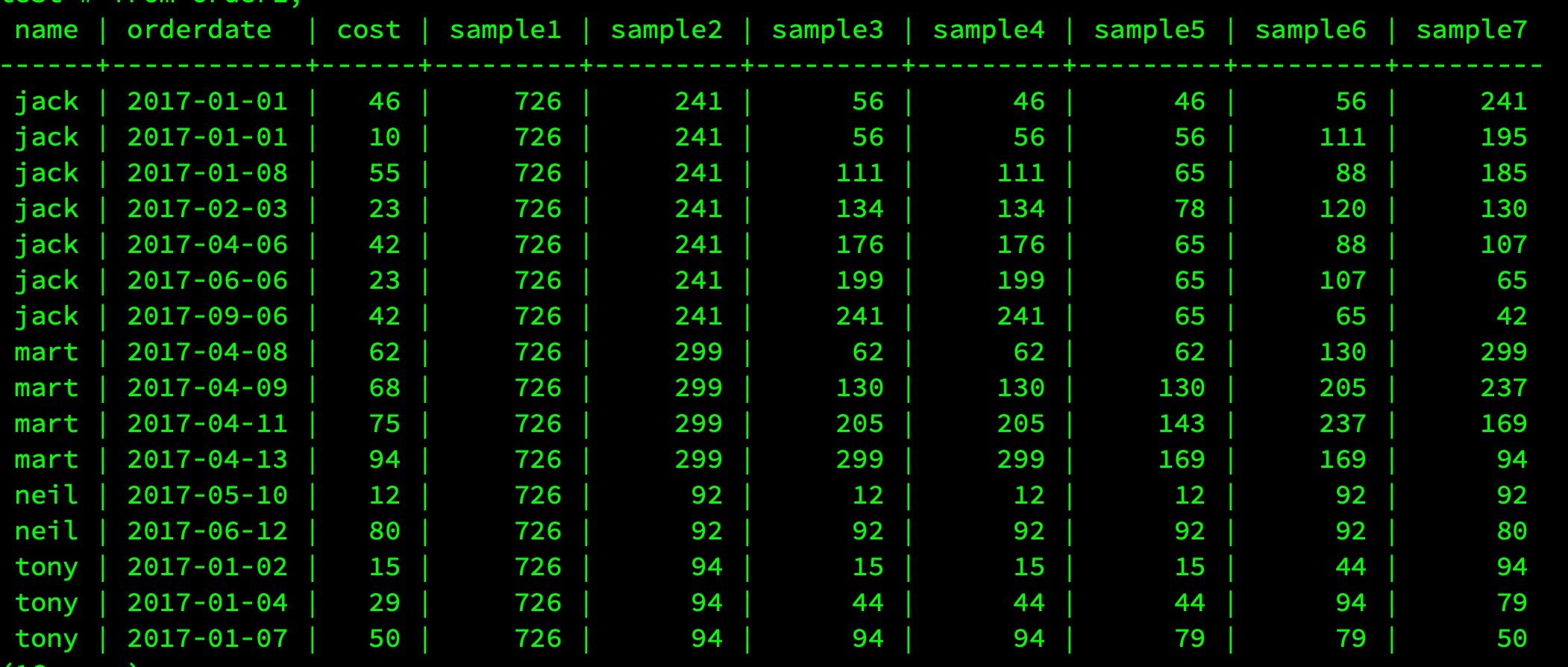
sum(cost) over(partition by name order by orderdate rows between unbounded preceding and current row) as sample4, --和sample3一样，由起点到当前行的聚合

sum(cost) over(partition by name order by orderdate rows between 1 preceding and current row) as sample5,--当前行和前面一行做聚合

sum(cost) over(partition by name order by orderdate rows between 1 preceding and 1 following )as sample6,--当前行和前一行及后面一行

sum(cost) over(partition by name order by orderdate rows between current row and unbounded following) as sample7 --当前行及后面所有行

from order1;



#### row\_number、rank、dense\_rank

这三个窗口函数的使用场景非常多

- row\_number()从1开始，按照顺序，生成分组内记录的序列,row\_number()的值不会存在重复,当排序的值相同时,按照表中记录的顺序进行排列

- RANK() 生成数据项在分组中的排名，排名相等会在名次中留下空位

- DENSE\_RANK() 生成数据项在分组中的排名，排名相等会在名次中不会留下空位

\*\*注意：

rank和dense\_rank的区别在于排名相等时会不会留下空位.\*\*

select name ,orderdate,cost,

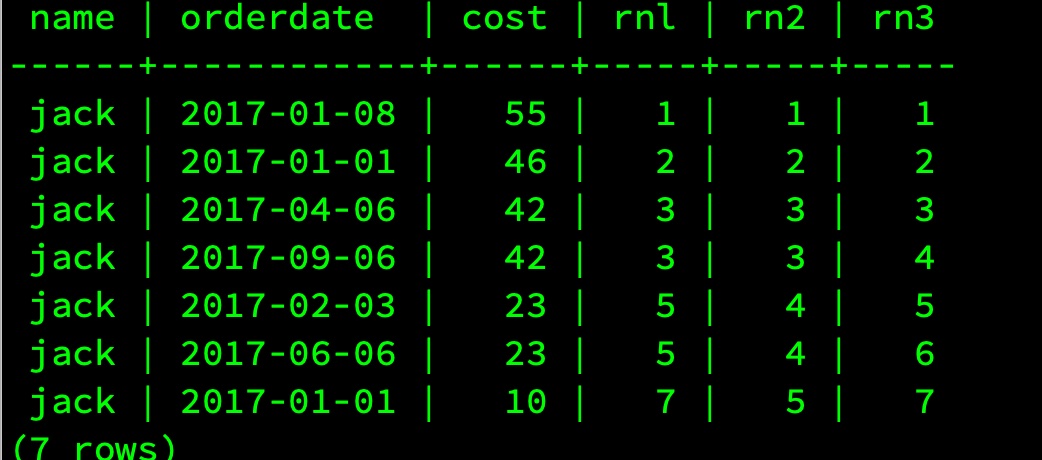
rank() over(partition by name order by cost desc) as rnl,--值同排名相同，同时不保留被占用的排名序号，即总排名号不连续

dense\_rank() over(partition by name order by cost desc) as rn2,--值同，排名相同，保留下一个的排名序列号，即总排名连续

row\_number() over(partition by name order by cost desc) as rn3--强制按罗列的结果排序，更像行号。

from order1

where name='jack';



#### 分组topN

create table SC(

stdid int,

clazzid int,

course VARCHAR,

score int

)

insert into SC VALUES(1,1,'Eng',89);

insert into SC VALUES(2,2,'Eng',79);

insert into SC VALUES(3,1,'Eng',69);

insert into SC VALUES(4,2,'Eng',39);

insert into SC VALUES(5,1,'Eng',99);

insert into SC VALUES(1,1,'yuwen',86);

insert into SC VALUES(2,2,'yuwen',76);

insert into SC VALUES(3,1,'yuwen',64);

insert into SC VALUES(4,2,'yuwen',32);

insert into SC VALUES(5,1,'yuwen',91);

insert into SC VALUES(1,1,'shuxue',11);

insert into SC VALUES(2,2,'shuxue',52);

insert into SC VALUES(3,1,'shuxue',55);

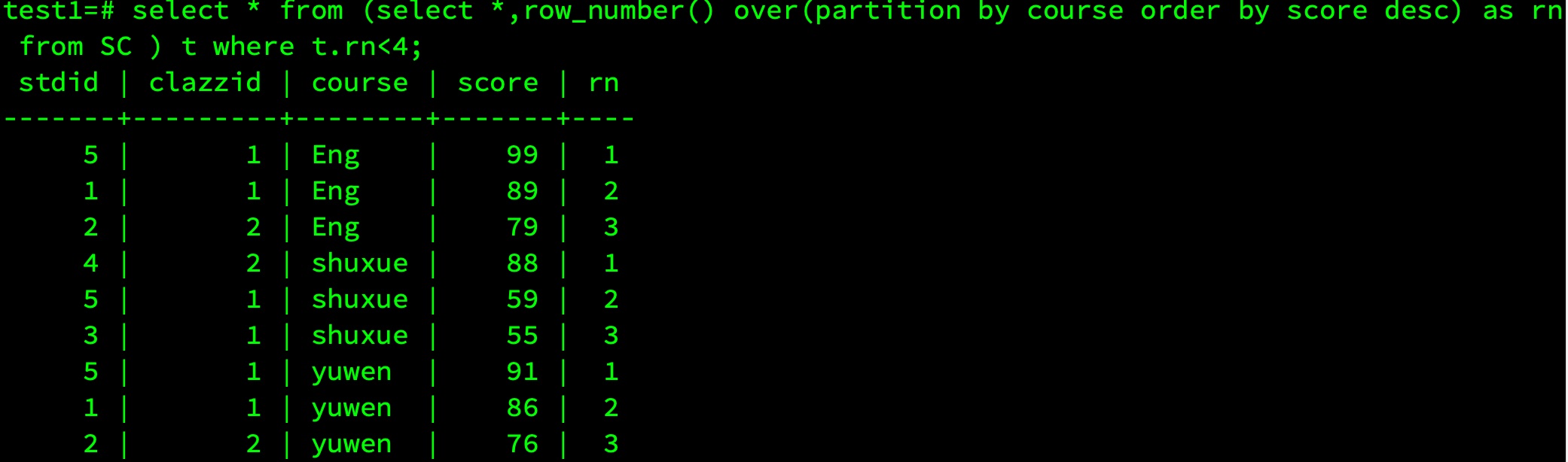
insert into SC VALUES(4,2,'shuxue',88);

insert into SC VALUES(5,1,'shuxue',59);

取每门课的前3名

select \* from (select \*,row\_number() over(partition by course order by score desc) as rn from sc ) t where t.rn< 4;

这里要注意下，只能用外层查询取到行号，子查询where取不到rn



## nvl

NVL函数是一个空值转换函数  
NVL（表达式1，表达式2）  
如果表达式1为空值，NVL返回值为表达式2的值，否则返回表达式1的值。 该函数的目的是把一个空值（null）转换成一个实际的值。其表达式的值可以是数字型、字符型和日期型。但是表达式1和表达式2的数据类型必须为同一个类型。

# 高级子句

## Having子句

使用having子句的原因是，where关键字无法与聚合函数一起使用，having子句可以让我们筛选分组后的各组数据。

having可以对分组后的数据进一步聚合筛选。

### 用法

SELECT column\_name, aggregate\_function(column\_name)  
FROM table\_name  
WHERE column\_name operator value  
GROUP BY column\_name  
HAVING aggregate\_function(column\_name) operator value;

#### 连续3天有交易记录的客户

利用表的自关联查询

表A

CUS\_ID TXN\_DT ID

1 20180101 1

2 20180101 2

3 20180101 3

1 20180102 4

2 20180102 5

2 20180102 6

1 20180103 7

3 20180103 8

select

cus\_id

from a m

inner join a n

on m.cus\_id = n.cus\_id

and m.txn\_dt <= n.txn\_dt +2

group by 1

having count(distinct txn\_dt) = 3；

with t as (

select stack(

7,

1,20180101,1,

2,20180101,2,

3,20180101,3,

1,20180102,4,

2,20180102,5,

2,20180102,6,

1,20180103,7

)as (cus\_id,txn\_dt,id)

),

a as (

select

\*,

row\_number() over(partition by cus\_id order by txn\_dt) rn)

select cus\_id

from a

group by date\_sub(txn\_dt,rn)

having count(1)>=3

## full. join

只要其中某个表存在匹配，FULL JOIN 关键字就会返回行。

用法：

SELECT column\_name(s)

FROM table\_name1

FULL JOIN table\_name2

ON table\_name1.column\_name=table\_name2.column\_name