

MySQL-SQL窗口函数

一、窗口函数有什么用

在日常工作中，经常会遇到需要在每组内排名，比如下面的业务需求：

排名问题：每个部门按业绩来排名

topN问题：找出每个部门排名前N的员工进行奖励

二、什么是窗口函数

窗口函数，也叫OLAP函数（**Online Analytical Processing**，联机分析处理），可以对数据库数据进行实时分析处理

窗口函数基本语法：

```
<窗口函数> over (partition by <用于分组的列名>  
  
                    order by <用于排序的列名>)
```

语法中的<窗口函数>都有哪些：

1) 专用窗口函数，包括后面要讲到的`rank`, `dense_rank`, `row_number` 等专用窗口函数。

2) 聚合函数，如`sum`, `avg`, `count`, `max`, `min` 等

三、如何使用

1. 专用窗口函数

下表是班级表内容

班级表

学号	班级	成绩
0001	1	86
0002	1	95
0003	2	89
0004	1	83
0005	2	86
0006	3	92
0007	3	86
0008	1	88

如果我们想在每个班级内按成绩排名，得到下面的结果。

学号	班级	成绩	ranking
0002	1	95	1
0008	1	88	2
0001	1	86	3
0004	1	83	4
0003	2	89	1
0005	2	86	2
0006	3	92	1
0007	3	86	2

以班级“1”为例，这个班级的成绩“95”排在第1位，这个班级的“83”排在第4位。

上面这个结果确实按我们的要求在每个班级内，按成绩排名了。

得到上面结果的sql语句代码如下：

```
select *,
rank() over (partition by 班级 order by 成绩 desc) as
ranking
from 班级表
```

解释下这个sql语句里的select子句。rank是排序的函数。

要求是“每个班级内按成绩排名”，这句话可以分为两部分：

1. 每个班级内：按班级分组partition by用来对表分组。在这个例子中，所以我们指定了按“班级”分组（partition by 班级）
2. 按成绩排名：order by子句的功能是对分组后的结果进行排序，默认是按照升序（asc）排列。

在本例中（order by 成绩 desc）是按成绩这一列排序，加了desc关键词表示降序排列。

通过下图，我们就可以理解partition by（分组）和order by（在组内排序）的作用了。

partition 分组（橘色）
order by 排序（蓝色）

	学号	班级	成绩	ranking	
}	0002	1	95	1	↓
	0008	1	88	2	
	0001	1	86	3	
	0004	1	83	4	
}	0003	2	89	1	↓
	0005	2	86	2	
}	0006	3	92	1	↓
	0007	3	86	2	

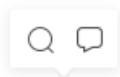
窗口函数具备了我们之前学过的group by子句分组的功能和order by子句排序的功能。

那为什么还要用窗口函数呢？

这是因为，group by分组汇总后改变了表的行数，一行只有一个类别。而partition by和rank函数不会减少原表中的行数。例如下面统计每个班级的人数。

group by 分组汇总改变行数

```
select 班级, count(学号)
from 班级表
group by 班级
order by 班级
```



班级	count(学号)
1	4
2	2
3	2

partition by 分组汇总行数不变

```
select 班级,
       count(学号) over (partition by 班级
                        order by 班级) as
       current_count
from 班级表
```

班级	current_count
1	4
1	4
1	4
1	4
2	2
2	2
3	2
3	2

https://blog.csdn.net/William_r

为什么叫“窗口”函数呢？

这是因为partition by 分组后的结果称为“窗口”，这里的窗口不是我们家里的门窗，而是表示“范围”的意思。

简单来说，窗口函数有以下功能：

- 同时具有分组和排序的功能
- 不减少原表的行数
- 语法如下：

<窗口函数> over (partition by <用于分组的列名>

order by <用于排序的列名>)

2. 其他专业窗口函数

专用窗口函数rank, dense_rank, row_number有什么区别呢？

```
select *,
       rank() over (order by 成绩 desc) as ranking,
       dense_rank() over (order by 成绩 desc) as dese_rank,
       row_number() over (order by 成绩 desc) as row_num
from 班级表
```

得到结果：

学号	班级	成绩	ranking	dese_rank	row_num
0002	1	95	1	1	1
0006	3	92	2	2	2
0003	2	89	3	3	3
0008	1	88	4	4	4
0001	1	86	5	5	5
0005	2	86	5	5	6
0007	3	86	5	5	7
0004	1	83	8	6	8

从上面的结果可以看出：

rank函数： 这个例子中是5位，5位，5位，8位，也就是如果有并列名次的行，会占用下一名次的位置。

比如正常排名是1，2，3，4，但是现在前3名是并列的名次，结果是：
1，1，1，4

dense_rank函数： 这个例子中是5位，5位，5位，6位，也就是如果有并列名次的行，不占用下一名次的位置。

比如正常排名是1，2，3，4，但是现在前3名是并列的名次，结果是：
1，1，1，2。

row_number函数： 这个例子中是5位，6位，7位，8位，也就是不考虑并列名次的情况。

比如前3名是并列的名次，排名是正常的**1，2，3，4。**

这三个函数的区别如下：

成绩	ranking	dese_rank	row_num
100	1	1	1
100	1	1	2
100	1	1	3
98	4	2	4

最后，需要强调的一点是：

在上述的这三个专用窗口函数中，函数后面的括号不需要任何参数，保持()空着就可以。

3. 聚合函数作为窗口函数

聚合窗口函数和上面提到的专用窗口函数用法完全相同，只需要把聚合函数写在窗口函数的位置即可，但是函数后面括号里面不能为空，需要指定聚合的列名。

我们来看一下窗口函数是聚合函数时，会出来什么结果：

```
select *,  
  
sum(成绩) over (order by 学号) as current_sum,  
  
avg(成绩) over (order by 学号) as current_avg,  
  
count(成绩) over order by 学号) as current_count,  
  
max(成绩) over (order by 学号) as current_max,  
  
min(成绩) over (order by 学号) as current_min  
  
from 班级表;
```

得到结果：

有发现什么吗？单独用sum举个例子：

如上图，聚合函数sum在窗口函数中，是对自身记录、及位于自身记录以上的数据进行求和的结果。

比如0004号，在使用sum窗口函数后的结果，是对0001，0002，0003，0004号的成绩求和，若是0005号，则结果是0001号~0005号成绩的求和，以此类推。

不仅是sum求和，平均、计数、最大最小值，也是同理，都是针对自身记录、以及自身记录之上的所有数据进行计算，现在再结合刚才得到的结果（下图），是不是理解起来容易多了？

比如0005号后面的聚合窗口函数结果是：学号0001~0005五人成绩的总和、平均、计数及最大最小值。

如果想要知道所有人成绩的总和、平均等聚合结果，看最后一行即可。

> 聚合函数作为窗口函数，可以在每一行的数据里直观的看到，截止到本行数据，统计数据是多少（最大值、最小值等）。同时可以看出每一行数据，对整体统计数据的影响。

四、注意事项

partition子句可是省略，省略就是不指定分组，结果如下，只是按成绩由高到低进行了排序：

```
```sql```
select *, rank() over (order by 成绩 desc) as ranking from
班级表
```

得到结果：

## 省略了partition语句

学号	班级	成绩	ranking
0002	1	95	1
0006	3	92	2
0003	2	89	3
0008	1	88	4
0001	1	86	5
0005	2	86	5
0007	3	86	5
0004	1	83	8

但是，这就失去了窗口函数的功能，所以一般不要这么使用。

## 五、总结

### 1.窗口函数的语法

<窗口函数> over (partition by <用于分组的列名> order by <用于排序的列名>)

<窗口函数>的位置，可以放以下两种函数：

- 专用窗口函数，比如rank, dense\_rank, row\_number 等
- 聚合函数，如sum, avg, count, max, min 等

## 2.窗口函数有以下功能：

- 同时具有分组（partition by）和排序（order by）的功能
- 不减少原表的行数，所以经常用来在每组内排名

## 3.注意事项

窗口函数 原则上 只能写在select子句中

## 4.窗口函数的使用场景

### 1.业务需求“在每组内排名”，比如：

排名问题：每个部门按业绩来排名

topN问题：找出每个部门排名前N的员工进行奖励

## 2. 窗口函数和普通聚合函数的区别：

①聚合函数是将多条记录聚合为一条；窗口函数是每条记录都会执行，有几条记录执行完还是几条。

窗口函数兼具之前我们学过的GROUP BY 子句的分组功能以及ORDER BY 子句的排序功能。但是，PARTITION BY子句并不具备GROUP BY 子句的汇总功能。因此，使用RANK 函数并不会减少原表中 记录的行数，结果中仍然包含8 行数据。

窗口函数兼具分组和排序两种功能。

②聚合函数也可以用于窗口函数。

原因就在于窗口函数的执行顺序（逻辑上的）是在FROM, JOIN, WHERE, GROUP BY, HAVING之后，在ORDER BY, LIMIT, SELECT DISTINCT之前。它执行时GROUP BY的聚合过程已经完成了，所以不会再产生数据聚合。

注:窗口函数是在where之后执行的，所以如果where子句需要用窗口函数作为条件，需要多一层查询，在子查询外面进行,例如:求30天内后一天比前一天平均时间差

```
**select user_id,avg(diff)
from
(
 select user_id,lead(log_time)over(partition by user_id
order by log_time) -log_time as diff
 from user_log**

**)
where datediff(now(),t.log_time)<=30
group by user_id**
```



作为窗口函数使用的聚合函数

所有的聚合函数都能用作窗口函数，其语法和专用窗口函数完全相同。但大家可能对所能得到的结果还没有一个直观的印象，所以我们还是通过具体的示例来学习。

product_id (商品编号)	product_name (商品名称)	product_type (商品种类)	sale_price (销售单价)	purchase_price (进货单价)	regist_date (登记日期)
0003	运动T恤	衣服	4000	2800	NULL
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500	NULL	2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100	NULL	2009-11-11

该列的最小值 该列的最大值

```
SELECT product_id, product_name, sale_price, SUM (sale_price) OVER
(ORDER BY product_id) AS current_sum FROM Product;
```

执行结果

product_id	product_name	sale_price	current_sum
0001	T恤衫	1000	1000
0002	打孔器	500	1500
0003	运动T恤	4000	5500
0004	菜刀	3000	8500
0005	高压锅	6800	15300
0006	叉子	500	15800
0007	擦菜板	880	16680
0008	圆珠笔	100	16780

https://blog.csdn.net/qq\_34089667

我们得到的并不仅仅是合计值，而是按照ORDER BY子句指定的product\_id的升序进行排列，计算出商品编号“小于自己”的商品的销售单价的合计值。因此，计算该合计值的逻辑就像金字塔堆积那样，一行一行逐渐添加计算对象。在按照时间序列的顺序，计算各个时间的销售额总额等的时候，通常都会使用这种称为累计的统计方法。使用其他聚合函数时的操作逻辑也和本例相同。例如，使用AVG来代替SELECT语句中的SUM:

```
SELECT product_id, product_name, sale_price, AVG (sale_price) OVER
(ORDER BY product_id) AS current_avg FROM Product;
```

执行结果

product_id	product_name	sale_price	current_avg
0001	T恤衫	1000	1000.0000000000000000
0002	打孔器	500	750.0000000000000000
0003	运动T恤	4000	1833.3333333333333333
0004	菜刀	3000	2125.0000000000000000
0005	高压锅	6800	3060.0000000000000000
0006	叉子	500	2633.3333333333333333
0007	擦菜板	880	2382.8571428571428571
0008	圆珠笔	100	2097.5000000000000000

https://blog.csdn.net/qq\_34089667

从结果中我们可以看到，current\_avg的计算方法确实是计算平均值的方法，但作为统计对象的却只是“排在自己之上”的记录。像这样以“自身记录（当前记录）”作为基准进行统计，就是将聚合函数当作窗口函数使用时的最大特征。

计算移动平均

窗口函数就是将表以窗口为单位进行分割，并在其中进行排序的函数。其实其中还包含在窗口中指定更加详细的汇总范围的备选功能，该备选功能中的汇总范围称为框架。

例：指定“最靠近的3行”作为汇总对象

```
SELECT product_id, product_name, sale_price,
 AVG(sale_price) OVER (ORDER BY product_id
 ROWS 2 PRECEDING) AS moving_avg
FROM Product;
```

执行结果（在DB2中执行）

product_id	product_name	sale_price	moving_avg
0001	T恤衫	1000	1000 ←(1000)/1
0002	打孔器	500	750 ←(1000+500)/2
0003	运动T恤	4000	1833 ←(1000+500+4000)/3
0004	菜刀	3000	2500 ←(500+4000+3000)/3
0005	高压锅	6800	4600 ←(4000+3000+6800)/3
0006	叉子	500	3433 .
0007	擦菜板	880	2726 .
0008	圆珠笔	100	493 .

- 指定框架（汇总范围）

我们将上述结果与之前的结果进行比较，可以发现商品编号为“0004”的“菜刀”以下的记录和窗口函数的计算结果并不相同。这是因为我们指定了框架，将汇总对象限定为了“最靠近的3行”。

这里我们使用了**ROWS**（“行”）和**PRECEDING**（“之前”）两个关键字，将框架指定为“截止到之前~行”，因此“**ROWS 2 PRECEDING**”就是将框架指定为“截止到之前2行”，也就是将作为汇总对象的记录限定为如下的“最靠近的3行”。

- 自身（当前记录）
- 之前1行的记录
- 之前2行的记录

也就是说，由于框架是根据当前记录来确定的，因此和固定的窗口不同，其范围会随着当前记录的变化而变化。

这样的统计方法称为移动平均（moving average）。由于这种方法在希望实时把握“最近状态”时非常方便，因此常常会应用在对股市趋势的实时跟踪当中。

使用关键字**FOLLOWING**（“之后”）替换**PRECEDING**，就可以指定“截止到之后~行”作为框架了。

将当前记录的前后行作为汇总对象

如果希望将当前记录的前后行作为汇总对象时，就可以同时使**PRECEDING**（“之前”）和**FOLLOWING**（“之后”）关键字来实现。

例：将当前记录的前后行作为汇总对象

```

SELECT product_id, product_name, sale_price,
 AVG (sale_price) OVER (ORDER BY product_id
 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS
 moving_avg
FROM Product;

```

执行结果 (在DB2中执行)

product_id	product_name	sale_price	moving_avg	
0001	T恤衫	1000	750	$\leftarrow (1000+500)/2$
0002	打孔器	500	1833	$\leftarrow (1000+500+4000)/3$
0003	运动T恤	4000	2500	$\leftarrow (500+4000+3000)/3$
0004	菜刀	3000	4600	$\leftarrow (4000+3000+6800)/3$
0005	高压锅	6800	3433	.
0006	叉子	500	2726	.
0007	擦菜板	880	493	.
0008	圆珠笔	100	490	.

在上述代码中，我们通过指定框架，将“1 PRECEDING”（之前1行）和“1 FOLLOWING”（之后1行）的区间作为汇总对象。具体来说，就是将如下3行作为汇总对象来进行计算：

- 之前1行的记录
- 自身(当前记录)
- 之后1行的记录

### 3. (面试考点) 序号函数:row\_number(),rank(),dense\_rank()的区别

ROW\_NUMBER():顺序排序——1、2、3

RANK():并列排序，跳过重复序号——1、1、3

DENSE\_RANK():并列排序，不跳过重复序号——1、1、2

### 4. 分布函数:percent\_rank(),cume\_dist()

**percent\_rank():**

每行按照公式 $(\text{rank}-1) / (\text{rows}-1)$ 进行计算。其中，rank为RANK()函数产生的序号，rows为当前窗口的记录总行数

```

1 --给窗口指定别名:WINDOW w AS (PARTITION BY stu_id ORDER BY score) rows = 5
2 mysql> SELECT
3 -> RANK() OVER w AS rk,
4 -> PERCENT_RANK() OVER w AS prk,
5 -> stu_id, lesson_id, score
6 -> FROM t_score
7 -> WHERE stu_id = 1
8 -> WINDOW w AS (PARTITION BY stu_id ORDER BY score)
9 -> ;
10 +-----+-----+-----+-----+
11 | rk | prk | stu_id | lesson_id | score |
12 +-----+-----+-----+-----+
13 | 1 | 0 | 1 | L003 | 79 |
14 | 2 | 0.25 | 1 | L002 | 86 |
15 | 3 | 0.5 | 1 | L004 | 88 |
16 | 4 | 0.75 | 1 | L005 | 98 |
17 | 4 | 0.75 | 1 | L001 | 98 |
18 +-----+-----+-----+-----+

```

### cume\_dist():

分组内小于、等于当前rank值的行数 / 分组内总行数 eg:查询小于等于当前成绩（score）的比例

```

1 --cd1: 没有分区, 则所有数据均为一组, 总行数为8
2 --cd2: 按照lesson_id分成了两组, 行数各为4
3 mysql> SELECT stu_id, lesson_id, score,
4 -> CUME_DIST() OVER (ORDER BY score) AS cd1,
5 -> CUME_DIST() OVER (PARTITION BY lesson_id ORDER BY score) AS cd2
6 -> FROM t_score
7 -> WHERE lesson_id IN ('L001','L002')
8 -> ;
9 +-----+-----+-----+-----+
10 | stu_id | lesson_id | score | cd1 | cd2 |
11 +-----+-----+-----+-----+
12 | 2 | L001 | 84 | 0.125 | 0.25 |
13 | 1 | L001 | 98 | 0.75 | 0.5 |
14 | 4 | L001 | 99 | 0.875 | 0.75 |
15 | 3 | L001 | 100 | 1 | 1 |
16 | 1 | L002 | 86 | 0.25 | 0.25 |
17 | 4 | L002 | 88 | 0.375 | 0.5 |
18 | 2 | L002 | 90 | 0.5 | 0.75 |
19 | 3 | L002 | 91 | 0.625 | 1 |
20 +-----+-----+-----+-----+

```

### 5.前后函数:lag(expr,n),lead(expr,n)

用途: 返回位于当前行的前n行 (LAG(expr,n)) 或后n行 (LEAD(expr,n)) 的expr的值

应用场景: 查询前1名同学的成绩和当前同学成绩的差值

```

1 mysql> SELECT stu_id, lesson_id, score, pre_score,
2 -> score-pre_score AS diff
3 -> FROM(
4 -> SELECT stu_id, lesson_id, score,
5 -> LAG(score,1) OVER w AS pre_score
6 -> FROM t_score
7 -> WHERE lesson_id IN ('L001','L002')
8 -> WINDOW w AS (PARTITION BY lesson_id ORDER BY score)) t
9 -> ;
10 +-----+-----+-----+-----+-----+
11 | stu_id | lesson_id | score | pre_score | diff |
12 +-----+-----+-----+-----+-----+
13 | 2 | L001 | 84 | NULL | NULL |
14 | 1 | L001 | 98 | 84 | 14 |
15 | 4 | L001 | 99 | 98 | 1 |
16 | 3 | L001 | 100 | 99 | 1 |
17 | 1 | L002 | 86 | NULL | NULL |
18 | 4 | L002 | 88 | 86 | 2 |
19 | 2 | L002 | 90 | 88 | 2 |
20 | 3 | L002 | 91 | 90 | 1 |
21 +-----+-----+-----+-----+-----+

```

## 6.头尾函数：FIRST\_VALUE(expr),LAST\_VALUE(expr)

用途：返回第一个（FIRST\_VALUE(expr)）或最后一个（LAST\_VALUE(expr)）expr的值

应用场景：截止到当前成绩，按照日期排序查询第1个和最后1个同学的成绩

```

1 mysql> SELECT stu_id, lesson_id, score, create_time,
2 -> FIRST_VALUE(score) OVER w AS first_score,
3 -> LAST_VALUE(score) OVER w AS last_score
4 -> FROM t_score
5 -> WHERE lesson_id IN ('L001','L002')
6 -> WINDOW w AS (PARTITION BY lesson_id ORDER BY create_time)
7 -> ;
8 +-----+-----+-----+-----+-----+-----+
9 | stu_id | lesson_id | score | create_time | first_score | last_score |
10 +-----+-----+-----+-----+-----+-----+
11 | 3 | L001 | 100 | 2018-08-07 | 100 | 100 |
12 | 1 | L001 | 98 | 2018-08-08 | 100 | 98 |
13 | 2 | L001 | 84 | 2018-08-09 | 100 | 99 |
14 | 4 | L001 | 99 | 2018-08-09 | 100 | 99 |
15 | 3 | L002 | 91 | 2018-08-07 | 91 | 91 |
16 | 1 | L002 | 86 | 2018-08-08 | 91 | 86 |
17 | 2 | L002 | 90 | 2018-08-09 | 91 | 90 |
18 | 4 | L002 | 88 | 2018-08-10 | 91 | 88 |
19 +-----+-----+-----+-----+-----+-----+

```

## 面试题

### 1.用户行为分析

表1——用户行为表tracking\_log，大概字段有（user\_id‘用户编号’,opr\_id‘操作编号’,log\_time‘操作时间’）如下所示：

user_id	opr_id	log_time
001	A	2020-01-01 12:01:44
001	B	2020-01-01 12:02:44
002	C	2020-02-01 11:01:44
002	A	2020-02-03 12:02:44

问题：

1.统计每天符合以下条件的用户数：A操作之后是B操作，AB操作必须相邻

分析：

1.统计每天，所以需要按天分组统计求和

2.A操作之后是B，且AB操作必须相邻，那就涉及一个前后问题，所以想到用窗口函数中的lag()或lead()。

解答：

sql

```
select date,count(*)
from(
 select user_id
 from(
 select user_id,convert(log_time,date) date,opr_id f,lag(opr_id,1) over(partition
by user_id,convert(log_time,date) order by log_time) l
 from tracking_log
) a
 where f='A' and l='B'
) b
group by date;
```

2.统计用户行为序列为A-B-D的用户数，其中：A-B之间可以有任何其他浏览记录(如C,E等)，B-D之间除了C记录可以有任何其他浏览记录(如A,E等)

```
```sql
select count(*)
from(
  select user_id,droup_concat(opr_id) ubp
  from tracking_log
  group by user_id
) a
where ubp like '%A%B%D%' and ubp not like '%A%B%C%D%'
```

学生成绩分析

表：Enrollments

```

±-----±-----+
| Column Name | Type |
±-----±-----+
| student_id  | int  |
| course_id   | int  |
| grade       | int  |
±-----±-----+
(student_id, course_id) 是该表的主键。

```

student_id	course_id	grade
001	1	60
001	2	75
002	2	33
002	3	97

1.查询每位学生获得的最高成绩和它所对应的科目，若科目成绩并列，取 course_id 最小的一门。查询结果需按 student_id 增序进行排序。

分析：因为需要最高成绩和所对应的科目，所以可采用窗口函数排序分组取第一个

按每位学生的成绩排名

```

SELECT student_id,course_id,grade
      RANK_NUMBR() OVER (PARTITION BY student_id order by
grade DESC) as Rank
FROM Enrollments
ORDER BY Rank, course_id

```

取其中最高的成绩

```

SELECT student_id,course_id,grade
FROM (SELECT student_id,course_id,grade
      RANK_NUMBR() OVER (PARTITION BY student_id order by
grade DESC) as Rank
      FROM Enrollments
      ORDER BY Rank, course_id) as A
where A.Rank = 1
order by student_id

```

解法2：IN 解法

先取最大成绩

```

SELECT student_id,max(grade)
FROM Enrollments
GROUP BY student_id

```

然后取成绩在最大成绩之中的学生的最小课程号的课程

```

select student_id,min(course_id)
from Enrollments
where (student_id,grade) in (
                                select student_id,max(grade)
                                from Enrollments
                                group by student_id)
group by student_id
order by student_id;

```

2.查询每一科目成绩最高和最低分数的学生,输出course_id,student_id,score
我们可以按科目查找成绩最高的同学和最低分的同学，然后利用union连接起来

```

select c_id,s_id
from(
        select *,row_number() over(partition by c_id order
by s_score desc) r
        from score
    ) a
where r=1
union
select c_id,s_id
from(
        select *,row_number() over(partition by c_id order
by s_score) r
        from score
    ) a
where r=1;

```

解法2: case-when

```

select c_id,
        max(case when r1=1 then s_id else null end) '最高分
学生',
        max(case when r2=1 then s_id else null end)
'最低分学生'
from(
        select *,row_number() over(partition by c_id order
by s_score desc) r1,
        row_number() over(partition by c_id order
by s_score) r2
        from score
    ) a
group by c_id;

```