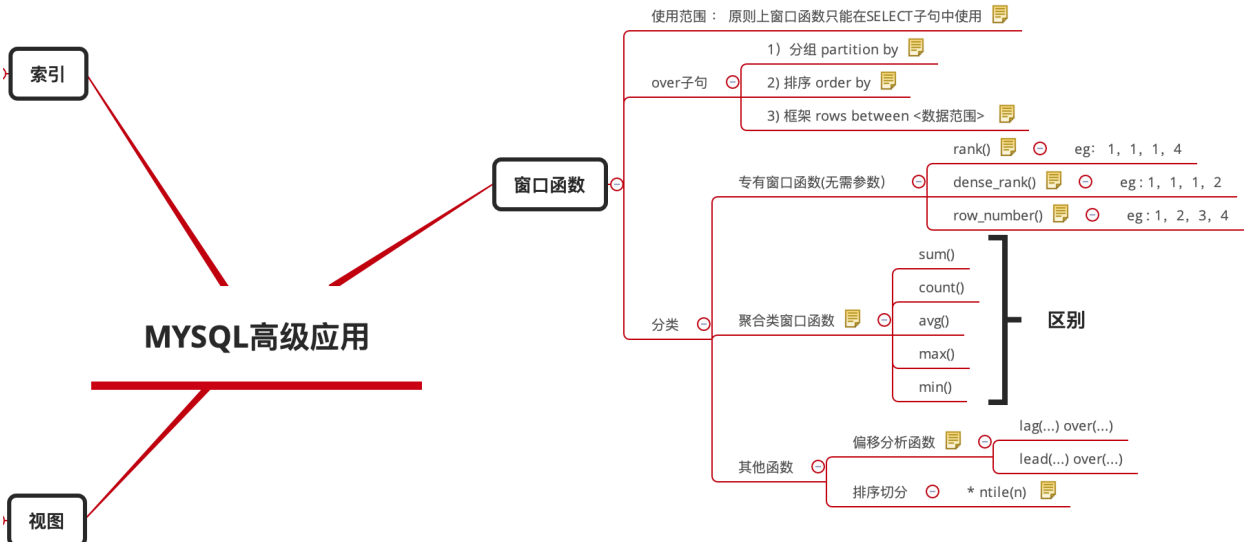
笔记摘要

本文主要介绍窗口函数。第一部分介绍窗口函数定义，第二部分介绍窗口函数的语法，包括窗口函数以及over子句。其中窗口函数部分介绍了专用窗口函数、聚合函数、排序函数ntitile以及偏移函数lean与lag。over子句部分介绍partition by,order by 及rows三个参数，在partition by 和order by部分介绍了他们和group by 以及group by 子句、order by子句的区别。每一个参数均举了例子说明。第三部分对内容作了总结。第四部分是一些SQL的练习题以及答案，来源于拉钩数据分析训练营教学笔记。

1 思维导图



2 窗口函数定义

> 窗口函数也称为OLAP(Online Analytical Processing)函数， 意思是可以对数据库数据进行实时分析分析处理的函数。

窗口函数主要用来处理相对复杂的报表统计分析场景。举个例子，现在有三个月的销售数据，如下表所示：

日期	销售额
2019-01	100
2019-02	200
2019-03	300

如何在每个月实现所有月份销售额的统计，如下表：

日期	销售额	累积销售额
2019-01	100	100
2019-02	200	300
2019-03	300	600

再比如，如何实现移动累积销售额的计算？比如我要计算最近三个月的累积销售额。

以上这些需求可以通过窗口函数来实现。

3 窗口函数语法

```
<窗口函数> OVER子句
```

可以看到，窗口函数的语法包含两个大的部分，分别是：**<窗口函数>**、**over 子句**。现在分别对这两个概念进行说明。

3.1 窗口函数

能够作为窗口函数使用的函数有以下几种：

1. 专用窗口函数
    1. **RANK()** ：计算排序时，如果存在相同位次的记录，则会跳过该记录。例如有2条记录的值为100，1条记录的值为200，对这3条记录进行排序时，结果为：1，1，3
    2. **DENSE_RANK()**：计算排序时，即使存在相同位次的记录，也不会跳过之后的位次。例如有2条记录的值为100，1条记录的值为200，对这3条记录进行排序时，结果为：1，1，2
    3. **ROW_NUMBER()**：赋予唯一的连续位次。例如有2条记录的值为100，1条记录的值为200，对这3条记录进行排序时，结果为：1，2，3
2. 聚合函数

    SUM、AVG、COUNT、MAX、MIN等。窗口函数的聚合方式和GROUP BY的聚合方式有些区别，在后面会详细说明。

3. 其他常用函数：
    1. 排序函数：ntitle(n)
    2. 偏移函数：lag(column_name, offset, defval)，lead (olumn_name, offset, defval)

运用了窗口函数的语法如下:

```
RANK() OVER子句
```

rank（）这里可以替换为其他的窗口函数与聚合函数，例如替换为sum这个聚合函数:

```
SUM(列) OVER子句
```

可以看到，在上面两个语法中，聚合函数sum的括号中我加入了参数“列”，而rank窗口函数这里没有加。这也是两类窗口函数在语法上一个区别：

> 专用窗口函数可以不加任何参数直接使用。如 RANK(), DENSE_RANK()。而聚合函数后面必须加上参数。

窗口函数(专有窗口函数+聚合类窗口函数)和普通场景下的聚合函数也很容易混淆，会在over子句中详细说明。

另外，常用窗口函数还有以下几个：

> 1. 排序函数：ntitle(n)
> 2. 偏移函数：lag(), lead ()

ntitile（n) 可以在排序时按照排序顺序，将数据切分为N组。其适用的场景比如有：

> 找出销量排前30%的店铺。

偏移函数可以应用在需要前后数据一起计算的场景，比如计算环比，或者计算用户相邻订单间差了几天。现对偏移函数参数继续说明：

> lag(exp_str, offset, defval) ： 取出目标字段前面第N行的数据作为独立列放在目标字段的后一列。
>
> 1. exp_str: 是该函数作用的字段。
> 2. offset：偏移量。即在当前字段，在当前行的基础上向前取多少行数据。如目前的行是第3行，offset=2，那么就取出该列第一行的数据。
> 3. defval: 用来处理特殊情况。即当offset移动的行数超出了表的范围，lag函数返回defval这个值。如果不指定该参数，则默认返回NULL。

举例，现有以下表格：

name	value
A	10
A	30
A	40
A	60
A	90

我们要把value这列的值，每一行都下移一格，即第二行30移到第三行。这个时候我们就用lag函数。

> 1.exp_str 应该是value这个字段
>
> 2.offset 应该是1 （因为只下移一行）

> 3.defval 也是value，即如果下移超过了表格范围，那么那一行的值和Value这列的值一样。

函数如下：

```
lag(value,1,value) over(order by value)
```

产出的结果如下：

name	value	lag()
A	10	10
A	30	10
A	40	30
A	60	40
A	90	60

需要注意，第一行前面没有数值，如过lag就超过了表格的范围，所以结果是=value这一列，即等于10。lean 函数的逻辑与lag相同。lag指取出同一字段的前N行，lean指取出同一字段的后N行。

3.2 OVER 子句

over 子句包含三个部分：

> 1. partition by （分组功能）
> 2. order by （排序功能）
> 3. rows （指定框架：所选的数据范围）

分别对这三个参数进行说明。

3.2.1 PARTITION BY

partition by 可以设定分组的对象范围，即按照什么进行分组，以便后面用order by进行排序。**需要注意partition by 和sql 基础操作中group by 的区别：**

> GROUP BY是一种聚合操作，它将分组后该组中的内容聚合为一条汇总的信息。即使该组有N条信息，其group by 后返回的结果也只有一条。但是partition by 不同，如果在窗口函数中用了partition by这个参数，那么分组的组内有几条信息，partition by 之后还是返回几条信息。

对于以上这种区别，我们可能会产生这样的疑惑：

> partition分组之后为什么结果是多条而不是一条?
>
> Group by 分组聚合后结果为一条是容易理解的。比如用sum函数，那么就是把每组的结果相加，最后肯定只有一个汇总的值。但是partition by 也是分组，为什么分组后返回的不是一行结果，而是和汇总之前的行数相同?

这里就要说到窗口函数进行聚合操作和sql一般操作的不同之处： 窗口函数的操作有些类似于金子塔堆砌。还是以sum函数为例，**如果用partition by，在分组之前每一组有10条信息，那么partition by 返回的结果还是10条。对于第一条结果，其sum的值对其自身的值；对于第二条结果，其返回第一条+第二条的值，对于第三条结果，其返回第一+第二+第三条的值。**

举个更形象的例子，如果我们有二个组别A、B，每组有两个值1、2、3，那么呈现的原始表格为：

组别	值
A	1
B	1
A	2
B	2
A	3
B	3

如果用group by 对组别进行分组，然后进行sum，那么其返回结果为：

组别	SUM(值)
A	6
B	6

如果用窗口函数，参数选择了 0ver (partition by 组别,order by..., rows) [后面两个参数先不管]，那么其结果应该是：

组别	值	sum(值)
A	1	1
A	2	3
A	3	6
B	1	1
B	2	3
B	3	6

可以看到，返回的行数和原来一样多，sum 是按照组进行划分，然后每一组都分别按照金字塔堆砌的方式进行分步累加。这就是为什么partition by 也是分组，但是可以返回和原始数据一样多少行，而不是汇总成一行。

> 为了区分partition by 之后的结果 和 group by之后的结果，我们通常将partition by 分割后的记录集合称为**“窗口”**，而将group by 分割之后的记录集合称为**“组”**。这个**“窗口”**代表的是范围，即要对哪些组别的数据进行order by 和 row 操作。

3.2.2 ORDER BY

order by，顾名思义，就是指定 **数据** 按照哪一列、以何种顺序进行排序（升序、降序）。该参数发生在partition by 后面，意味着该操作是对各个窗口（窗口：partition by 之后，每一组的数据集合可以看作一个窗口）分别进行排序操作。

需要注意的是，over子句中的ORDER BY 和 SQL基本操作中 SELECT 语句后面的ORDER BY 不太一样。

> 在over子句中，order by 只是用来决定窗口函数按照什么顺序进行计算，对结果呈现的排列顺序没有影响。

例如，现在有A、B、C三组，值如下所示：

组别	值
A	100
A	500
B	200
B	300
B	400
C	600

如果在over语句中，我们用了order by 对所有值进行排序，我们 ***期望*** 的结果可能是这样：

组别	值	排序结果
A	100	1
B	200	2
B	300	3
C	400	4
A	500	5
C	600	6

但实际上，有时候返回的结果可能是：

组别	值	排序结果
C	600	6
A	100	1
A	500	5
C	400	4
B	300	3
B	200	2

上面提到过“在over子句中，order by 只是用来决定窗口函数按照什么顺序进行计算。”在我们的例子中，over子句要求窗口函数按照值的大小进行排序，其确实做到了（在排序结果这栏）。但是，在视觉呈现上，并没有按照我们的预期来。

如果要按照我们的预期来，那么我们需要在SELECT 语句之后的ORDER BY 语句中，对排序结果进行升序排序。

3.2.3 ROWS

通过上面两个参数，我们已经了解到：

> 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。

其实除了这两个功能外，窗口函数还有一个备选功能：**在窗口中指定更加详细的汇总范围**。

这一功能可以通过row这个参数实现，而通过row参数指定的这个范围可以称之为 **框架**。

总体看起来是这样：

```
<窗口函数> OVER (partition by ... order by... rows ...)
```

在rows后面，常用的参数有以下几个：

> 1. PRECEDING (之前)： 其将框架指定为“截止到之前~行” 如 rows 3 preceding 表示自身（当前记录）这一行 + 该记录之前的3行，共四行数据。
> 2. FOLLOWING(之后)：其将框架指定为“截止到之后~行” 如 rows 3 following 表示自身（当前记录）这一行 + 该记录之后的3行，共四行数据。

当然，可以同时使用preceding 和following，用以选取当前记录前后的行作为汇总对象：

> rows between 3 preceding and 3 following

上面参数表示选取的数据范围是：当前记录前的三行 —— 当前记录后的三行，共7行数据。

除此之外，我们也可以选取当前记录之前的所有行，或者当前记录之后的所有行。“所有行”可以用以下单词实现：

> unbounded .它的意思是无限的，极大的

因此：

> rows between unbounded preceding and current row 表示选取本行以及本行之前的所有行
>
> rows between unbouded following and current row 表示选取本行和之后的所有行

通过rows 参数，我们可以创建滑动窗口，实现移动平均等需求的计算，比如计算近三个月的总销售额，近两个月的支付金额等。

3.2.4 其他知识

三个参数中，order by相对比较重要，在一些操作中，partition by 和 row都可以不写，但是order by一般要写。即采用：

```
函数名 () over (order by...)
```

如果参数像上面这样，那么partition by 是默认对所有结果进行排序（即不划分窗口了）。

参数中没有指定rows，默认选取**当前行及当前行之前的所有行**。即 rows between unbounded preceding and current row。

4 窗口函数内容总结

窗口函数兼具排序和分组两组功能，它有两个重要概念：一个是窗口，一个是函数。窗口指通过over子句的partition by分组后的**记录集合**，简单理解就是分组后由不同的组所组成的一个表格称为窗口。这个窗口可以是固定的静态窗口，也可以是滑动的窗口（通过over 子句的 row参数实现）。

另一个概念**函数**指**在窗口内执行的函数**。即函数分别作用于窗口内不同组的每条记录。举例理解，如果partition by将窗口内数据分为三个组，而函数sum()作用于窗口内的三个组，那么sum函数在三个组内分别进行累加计算，但组与组之间不会累加。函数分为两种，一种是专用窗口函数，如rank()，一种是聚合函数，如sum()。窗口函数的基本语法为：

```
<函数名称> over(partition by <要分列的组> order by <要排序的列> rows between <数据范围>)
```

5 窗口函数练习及答案

现有电商平台订单表格user_trade，结果如下：

列名	解释
user_name	用户名
piece	购买数量
price	价格
pay_amount	支付金额
goods_category	商品品类
pay_time	支付日期

5.1 练习

请用mysql分别实现以下需求：

> 1. 查询出2019年每月的支付总额和当年累积支付总额
> 2. 查询出2018-2019年每月的支付总额和当年累积支付总额
> 3. 查询出2019年每个月的近三月移动平均支付金额
> 4. 查询出每四个月的最大月总支付金额
> 5. 2020年1月，购买商品品类数的用户排名
> 6. 查询出将2020年2月的支付用户，按照支付金额分成5组后的结果
> 7. 查询出2020年支付金额排名前30%的所有用户
> 8. 查询出支付时间间隔超过100天的用户数
> 9. 查询出每年支付时间间隔最长的用户

5.2 参考答案

5.2.1 查询出2019年每月的支付总额和当年累积支付总额

```
SELECT
  a.MONTH,-- 月份
  a.pay_amount,-- 当月总支付金额
  sum( a.pay_amount ) over ( ORDER BY a.MONTH ) -- 就是2019年的数据，所以
  不用分组
-- --此时没有使用rows指定窗口数据范围，默认当前行及其之前的所有行
FROM
    (
    SELECT MONTH
        ( pay_time ) MONTH,
        sum( pay_amount ) pay_amount
    FROM
        user_trade
    WHERE
        YEAR ( pay_time ) = 2019
    GROUP BY
    MONTH ( pay_time ) )a
```

5.2.2 查询出2019年每月的支付总额和当年累积支付总额

```
SELECT a.year,
       a.month,
       a.pay_amount,
       sum(a.pay_amount) over(partition by a.year order by a.month)
FROM
       (SELECT year(pay_time) year,
               month(pay_time) month,
               sum(pay_amount) pay_amount
       FROM user_trade
       WHERE year(pay_time) in (2018,2019)
       GROUP BY year(pay_time),
                month(pay_time))a;
```

5.2.3 查询出2019年每个月的近三月移动平均支付金额

```
SELECT a.month,
       a.pay_amount,
       avg(a.pay_amount) over(order by a.month rows between 2 preceding and
current row)
FROM
       (SELECT month(pay_time) month,
               sum(pay_amount) pay_amount
        FROM user_trade
        WHERE year(pay_time)=2019
        GROUP BY month(pay_time))a;
```

5.2.4 查询出每四个月的最大月总支付金额

```
SELECT a.month,
       a.pay_amount,
       max(a.pay_amount) over(order by a.month rows between 3 preceding and
current row)
FROM
       (SELECT substr(pay_time,1,7) as month,
               sum(pay_amount) as pay_amount
        FROM user_trade
        GROUP BY substr(pay_time,1,7))a;
```

5.2.5 2020年1月，购买商品品类数的用户排名

```
SELECT
    user_name,
    count( DISTINCT goods_category )  category_count,
    row_number() over(order by count( DISTINCT goods_category ) )  order1,
-- 没有说明何种方式排名，因此row_number(),dense_rank(),rank()都可以。
FROM
    user_trade
WHERE
substring( pay_time, 1, 7 ) = '2020-01'
GROUP BY
    user_name;
```

5.2.6 查询出将2020年2月的支付用户，按照支付金额分成5组后的结果

```
SELECT user_name,
       sum(pay_amount) pay_amount,
       ntile(5) over(order by sum(pay_amount) desc) level
FROM user_trade
WHERE substr(pay_time,1,7)='2020-02' GROUP BY user_name;
```

5.2.7 查询出2020年支付金额排名前30%的所有用户

```
SELECT a.user_name,
      a.pay_amount,
      a.level
FROM
      (SELECT user_name,
              sum(pay_amount) pay_amount,
              ntile(10) over(order by sum(pay_amount) desc) level
      FROM user_trade
      WHERE year(pay_time)=2020
      GROUP BY user_name) a
WHERE a.level in (1,2,3);
```

5.2.8 查询出支付时间间隔超过100天的用户数

```
SELECT count(distinct user_name)
FROM
      (SELECT user_name,
              pay_time,
              lead(pay_time) over(partition by user_name order by
pay_time) lead_dt
        FROM user_trade)a
WHERE a.lead_dt is not null
and datediff(a.lead_dt,a.pay_time)>100;
```

5.2.9 查询出每年支付时间间隔最长的用户

```
SELECT
      years,
      b.user_name,
      b.pay_days
FROM
    (SELECT
        years,
        a.user_name,
        datediff(a.pay_time,a.lag_dt) pay_days, # datediff用来计算两个日期之间的差
值
        rank() over(partition by years order by datediff(a.pay_time,a.lag_dt)
desc) rank1
     FROM
        (SELECT
                year(pay_time) as years,
                user_name,
                pay_time,
                lag(pay_time) over(partition by user_name,year(pay_time)
order by pay_time) lag_dt
         FROM user_trade)a
      WHERE a.lag_dt is not null)b
WHERE b.rank1=1;
```

参考资料

1. 《SQL基础教程 第2版》
2. 拉钩数据分析训练营阶段二模块二 MySQL高级应用课程笔记