说明: 1)Oracle 开发专题 99%收集自: http://www.blogjava.net/pengpenglin/(偶补充了一点点 1%);
2) PLSQL 开发笔记和小结收集自 http://www.blogjava.net/cheneyfree/
3)分析函数简述收集自 http://space.itpub.net/7607759/

昆明小虫 http://ynlxc.cnblogs.com/ 收集,并补充了一点点 1%

Oracle 开发专题之：分析函数(OVER)

目录：

==========

一、Oracle 分析函数简介：

在日常的生产环境中，我们接触得比较多的是 OLTP 系统(即 Online Transaction Process)，这些系统的特点是具备实时要求，或者至少说对响应的时间多长有一定的要求；其次这些系统的业务逻辑一般比较复杂，可能需要经过多次的运算。比如我们经常接触到的电子商城。

在这些系统之外，还有一种称之为 OLAP 的系统(即 Online Aanalyse Process)，这些系统一般用于系统决策使用。通常和数据仓库、数据分析、数据挖掘等概念联系在一起。这些系统的特点是数据量大，对实时响应的要求不高或者根本不关注这方面的要求，以查询、统计操作为主。

我们来看看下面的几个典型例子：
①查找上一年度各个销售区域排名前 10 的员工
②按区域查找上一年度订单总额占区域订单总额 20%以上的客户
③查找上一年度销售最差的部门所在的区域
④查找上一年度销售最好和最差的产品

我们看看上面的几个例子就可以感觉到这几个查询和我们日常遇到的查询有些不同，具体有：

①需要对同样的数据进行不同级别的聚合操作
②需要在表内将多条数据和同一条数据进行多次的比较
③需要在排序完的结果集上进行额外的过滤操作

分析函数语法:
FUNCTION_NAME(<argument>,<argument>...)
OVER
(<Partition-Clause><Order-by-Clause><Windowing Clause>)

例:
sum(sal) over (partition by deptno order by ename) new_alias
sum 就是函数名
(sal)是分析函数的参数,每个函数有 0~3 个参数,参数可以是表达式,例如:sum(sal+comm)
over 是一个关键字,用于标识分析函数,否则查询分析器不能区别 sum()聚集函数和 sum()分析函数
partition by deptno 是可选的分区子句,如果不存在任何分区子句,则全部的结果集可看作一个单一的大区
order by ename 是可选的 order by 子句,有些函数需要它,有些则不需要.依靠已排序数据的那些函数,如:用于访问结果集中前一行和后一行的 LAG 和 LEAD,必须使用,其它函数,如 AVG,则不需要.在使用了任何排序的开窗函数时,该子句是强制性的,它指定了在计算分析函数时一组内的数据是如何排序的.

1)FUNCTION 子句
ORACLE 提供了 26 个分析函数,按功能分 5 类
分析函数分类
等级(ranking)函数:用于寻找前 N 种查询
开窗(windowing)函数:用于计算不同的累计,如 SUM,COUNT,AVG,MIN,MAX 等,作用于数据的一个窗口上
例:
sum(t.sal) over (order by t.deptno,t.ename) running_total,
sum(t.sal) over (partition by t.deptno order by t.ename) department_total
制表(reporting)函数:与开窗函数同名,作用于一个分区或一组上的所有列
例:
sum(t.sal) over () running_total2,
sum(t.sal) over (partition by t.deptno) department_total2
制表函数与开窗函数的关键不同之处在于 OVER 语句上缺少一个 ORDER BY 子句!
LAG,LEAD 函数:这类函数允许在结果集中向前或向后检索值,为了避免数据的自连接,它们是非常用用的.
VAR_POP,VAR_SAMP,STDEV_POPE 及线性的衰减函数:计算任何未排序分区的统计值

2)PARTITION 子句
按照表达式分区(就是分组),如果省略了分区子句,则全部的结果集被看作是一个单一的组

3)ORDER BY 子句
分析函数中 ORDER BY 的存在将添加一个默认的开窗子句,这意味着计算中所使用的行的集合是当前分区中当前行和前面所有行,没有 ORDER BY 时,默认的窗口是全部的分区 在 Order by 子句后可以添加 **nulls last**,如:order by comm desc nulls last 表示排序时忽略 comm 列为空的行.

4)WINDOWING 子句
用于定义分析函数将在其上操作的行的集合
Windowing 子句给出了一个定义变化或固定的数据窗口的方法,分析函数将对这些数据进行操作

默认的窗口是一个固定的窗口,仅仅在一组的第一行开始,一直继续到当前行,要使用窗口,必须使用 ORDER BY 子句
根据 2 个标准可以建立窗口:数据值的范围(RANGES)或与当前行的行偏移量.

5)Rang 窗口
Range 5 preceding:将产生一个滑动窗口,他在组中拥有当前行以前 5 行的集合
ANGE 窗口仅对 NUMBERS 和 DATES 起作用,因为不可能从 VARCHAR2 中增加或减去 N 个单元
另外的限制是 ORDER BY 中只能有一列,因而范围实际上是一维的,不能在 N 维空间中
例:
avg(t.sal) over(order by t.hiredate asc range 100 preceding) 统计前 100 天平均工资

6)Row 窗口
利用 ROW 分区,就没有 RANGE 分区那样的限制了,数据可以是任何类型,且 ORDER BY 可以包括很多列

7)Specifying 窗口
UNBOUNDED PRECEDING:这个窗口从当前分区的每一行开始,并结束于正在处理的当前行
CURRENT ROW:该窗口从当前行开始(并结束)
Numeric Expression PRECEDING:对该窗口从当前行之前的数字表达式(Numeric Expression)的行开始,对 RANGE 来说,从从行序值小于数字表达式的当前行的值开始.
Numeric Expression FOLLOWING:该窗口在当前行 Numeric Expression 行之后的行终止(或开始),且从行序值大于当前行 Numeric Expression 行的范围开始(或终止)
range between 100 preceding and 100 following:当前行 100 前,当前后 100 后

注意:分析函数允许你对一个数据集进排序和筛选,这是 SQL 从来不能实现的.除了最后的 Order by 子句之外,分析函数是在查询中执行的最后的操作集,这样的话,就不能直接在谓词中使用分析函数,即不能在上面使用 where 或 having 子句!!!

二、Oracle 分析函数简单实例:

下面我们通过一个实际的例子:按区域查找上一年度订单总额占区域订单总额 20%以上的客户,来看看分析函数的应用。

【1】测试环境:

```
SQL> desc orders_tmp;
 Name                          Null?    Type
 ----------------------------- -------- ----------------
 CUST_NBR                      NOT NULL NUMBER(5)
 REGION_ID                     NOT NULL NUMBER(5)
 SALESPERSON_ID       NOT NULL NUMBER(5)
 YEAR                          NOT NULL NUMBER(4)
 MONTH                         NOT NULL NUMBER(2)
 TOT_ORDERS                NOT NULL NUMBER(7)
 TOT_SALES                   NOT NULL NUMBER(11,2)
```

【2】测试数据:

```
SQL> select * from orders_tmp;
```

```
 CUST_NBR  REGION_ID SALESPERSON_ID      YEAR     MONTH TOT_ORDERS  TOT_SALES
---------- ---------- -------------- ---------- ---------- ---------- ----------
      11          7          11          2001          7          2      12204
       4          5           4          2001         10          2      37802
       7          6           7          2001          2          3       3750
      10          6           8          2001          1          2      21691
      10          6           7          2001          2          3      42624
      15          7          12          2000          5          6         24
      12          7           9          2000          6          2      50658
       1          5           2          2000          3          2      44494
       1          5           1          2000          9          2      74864
       2          5           4          2000          3          2      35060
       2          5           4          2000          4          4       6454
       2          5           1          2000         10          4      35580
       4          5           4          2000         12          2      39190

13 rows selected.
```

【3】测试语句：

```
SQL> select o.cust_nbr customer,
  2         o.region_id region,
  3         sum(o.tot_sales) cust_sales,
  4         sum(sum(o.tot_sales)) over(partition by o.region_id) region_sales
  5    from orders_tmp o
  6   where o.year = 2001
  7   group by o.region_id, o.cust_nbr;

 CUSTOMER     REGION CUST_SALES REGION_SALES
---------- ---------- ---------- ------------
       4          5      37802        37802
       7          6       3750        68065
      10          6      64315        68065
      11          7      12204        12204
```

三、分析函数 OVER 解析：

请注意上面的绿色高亮部分，group by 的意图很明显：将数据按区域 ID，客户进行分组，那么 Over 这一部分有什么用呢？假如我们只需要统计每个区域每个客户的订单总额，那么我们只需要 group by o.region_id,o.cust_nbr 就够了。但我们还想在每一行显示该客户所在区域的订单总额，这一点和前面的不同：需要在前面分组的基础上按区域累加。很显然 group by 和 sum 是无法做到这一点的(因为聚集操作的级别不一样，前者是对一个客户，后者是对一批客户)。

这就是 over 函数的作用了！它的作用是告诉 SQL 引擎：按区域对数据进行分区，然后累积每个区域每个客户的订单总额(sum(sum(o.tot_sales)))。

现在我们已经知道 2001 年度每个客户及其对应区域的订单总额，那么下面就是筛选那些个人订单总额占到区域订单总额 20%以上的大客户了

```
SQL> select *
  2    from (select o.cust_nbr customer,
  3                 o.region_id region,
  4                 sum(o.tot_sales) cust_sales,
  5                 sum(sum(o.tot_sales)) over(partition by o.region_id) region_sales
  6            from orders_tmp o
  7           where o.year = 2001
  8           group by o.region_id, o.cust_nbr) all_sales
  9   where all_sales.cust_sales > all_sales.region_sales * 0.2;

  CUSTOMER     REGION CUST_SALES REGION_SALES
---------- ---------- ---------- ------------
         4          5      37802        37802
        10          6      64315        68065
        11          7      12204        12204

SQL>
```

现在我们已经知道这些大客户是谁了！哦，不过这还不够，如果我们想要知道每个大客户所占的订单比例呢？看看下面的 SQL 语句，只需要一个简单的 Round 函数就搞定了。

```
SQL> select all_sales.*,
  2         100 * round(cust_sales / region_sales, 2) || '%' Percent
  3    from (select o.cust_nbr customer,
  4                 o.region_id region,
  5                 sum(o.tot_sales) cust_sales,
  6                 sum(sum(o.tot_sales)) over(partition by o.region_id) region_sales
  7            from orders_tmp o
  8           where o.year = 2001
  9           group by o.region_id, o.cust_nbr) all_sales
 10   where all_sales.cust_sales > all_sales.region_sales * 0.2;

  CUSTOMER     REGION CUST_SALES REGION_SALES PERCENT
---------- ---------- ---------- ------------ ----------------------------------------
         4          5      37802        37802 100%
        10          6      64315        68065 94%
        11          7      12204        12204 100%

SQL>
```

总结：

①Over 函数指明在那些字段上做分析，其内跟 Partition by 表示对数据进行分组。注意 Partition by 可以有多个字段。

②Over 函数可以和其它聚集函数、分析函数搭配，起到不同的作用。例如这里的 SUM，还有诸如 Rank，Dense_rank

等。

Oracle 开发专题之：分析函数 2(Rank, Dense_rank, row_number)

目录

==========

一、使用 rownum 为记录排名：

在前面一篇《Oracle 开发专题之：分析函数》，我们认识了分析函数的基本应用，现在我们再来考虑下面几个问题：

①对所有客户按订单总额进行排名
②按区域和客户订单总额进行排名
③找出订单总额排名前 13 位的客户
④找出订单总额最高、最低的客户
⑤找出订单总额排名前 25%的客户

按照前面第一篇文章的思路，我们只能做到对各个分组的数据进行统计，如果需要排名的话那么只需要简单地加上 rownum 不就行了吗？事实情况是否如此想象般简单，我们来实践一下。

【1】测试环境：

```
SQL> desc user_order;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 REGION_ID                                          NUMBER(2)
 CUSTOMER_ID                                NUMBER(2)
 CUSTOMER_SALES                             NUMBER
```

【2】测试数据：

```
SQL> select * from user_order order by customer_sales;

 REGION_ID CUSTOMER_ID CUSTOMER_SALES
---------- ----------- --------------
         5           1         151162
        10          29          903383
         6           7         971585
```

```
        10         28          986964
         9         21         1020541
         9         22         1036146
         8         16         1068467
         6          8         1141638
         5          3         1161286
         5          5         1169926
         8         19         1174421
         7         12         1182275
         7         11         1190421
         6         10         1196748
         6          9         1208959
        10         30         1216858
         5          2            1224992
          9         24           1224992
          9         23           1224992
          8         18          1253840
         7         15         1255591
         7         13         1310434
        10         27         1322747
         8         20         1413722
         6          6         1788836
        10         26         1808949
         5          4         1878275
         7         14         1929774
         8         17         1944281
         9         25         2232703

30 rows selected.
```

注意这里有 3 条记录的订单总额是一样的。假如我们现在需要筛选排名前 12 位的客户，如果使用 rownum 会有什么样的后果呢？

```
SQL> select rownum, t.*
  2    from (select *
  3            from user_order
  4           order by customer_sales desc) t
  5   where rownum <= 12
  6   order by customer_sales desc;

    ROWNUM  REGION_ID CUSTOMER_ID CUSTOMER_SALES
---------- ---------- ----------- --------------
         1          9          25        2232703
         2          8          17        1944281
         3          7          14        1929774
         4          5           4        1878275
         5         10          26        1808949
```

```
         6          6              6       1788836
         7          8             20       1413722
         8         10             27       1322747
         9          7             13       1310434
        10          7             15       1255591
        11          8             18       1253840
        12          5              2       1224992

12 rows selected.
```

很明显假如只是简单地按 rownum 进行排序的话，我们漏掉了另外两条记录(参考上面的结果)。

二、使用分析函数来为记录排名：

针对上面的情况，Oracle 从 8i 开始就提供了 3 个分析函数：rand，dense_rank，row_number 来解决诸如此类的问题，下面我们来看看这 3 个分析函数的作用以及彼此之间的区别：

Rank，Dense_rank，Row_number 函数为每条记录产生一个从 1 开始至 N 的自然数，N 的值可能小于等于记录的总数。这 3 个函数的唯一区别在于当碰到相同数据时的排名策略。

①ROW_NUMBER：

Row_number 函数返回一个唯一的值，当碰到相同数据时，排名按照记录集中记录的顺序依次递增。

②DENSE_RANK：

Dense_rank 函数返回一个唯一的值，除非当碰到相同数据时，此时所有相同数据的排名都是一样的。

③RANK：

Rank 函数返回一个唯一的值，除非遇到相同的数据时，此时所有相同数据的排名是一样的，同时会在最后一条相同记录和下一条不同记录的排名之间空出排名。

这样的介绍有点难懂，我们还是通过实例来说明吧，下面的例子演示了 3 个不同函数在遇到相同数据时不同排名策略：

```
SQL> select region_id, customer_id, sum(customer_sales) total,
  2        rank() over(order by sum(customer_sales) desc) rank,
  3        dense_rank() over(order by sum(customer_sales) desc) dense_rank,
  4        row_number() over(order by sum(customer_sales) desc) row_number
  5   from user_order
  6  group by region_id, customer_id;

 REGION_ID CUSTOMER_ID      TOTAL       RANK DENSE_RANK ROW_NUMBER
---------- ----------- ---------- ---------- ---------- ----------

 ... ...

         8          18        1253840         11         11         11
         5           2        1224992         12         12         12
         9          23        1224992         12         12         13
```

```
         9         24            1224992         12         12         14
        10         30            1216858         15         13         15
 ... ...

30 rows selected.
```

请注意上面的绿色高亮部分，这里生动的演示了 3 种不同的排名策略：

①对于第一条相同的记录，3 种函数的排名都是一样的：12

②当出现第二条相同的记录时，Rank 和 Dense_rank 依然给出同样的排名 12；而 row_number 则顺延递增为 13，依次类推至第三条相同的记录

③当排名进行到下一条不同的记录时，可以看到 Rank 函数在 12 和 15 之间空出了 13,14 的排名，因为这 2 个排名实际上已经被第二、三条相同的记录占了。而 Dense_rank 则顺序递增。row_number 函数也是顺序递增

比较上面 3 种不同的策略，我们在选择的时候就要根据客户的需求来定夺了：

①假如客户就只需要指定数目的记录，那么采用 row_number 是最简单的，但有漏掉的记录的危险

②假如客户需要所有达到排名水平的记录，那么采用 rank 或 dense_rank 是不错的选择。至于选择哪一种则看客户的需要，选择 dense_rank 或得到最大的记录

三、使用分析函数为记录进行分组排名：

上面的排名是按订单总额来进行排列的，现在跟进一步：假如是为各个地区的订单总额进行排名呢？这意味着又多了一次分组操作：对记录按地区分组然后进行排名。幸亏 Oracle 也提供了这样的支持，我们所要做的仅仅是在 over 函数中 order by 的前面增加一个分组子句：partition by region_id。

```
SQL> select region_id, customer_id,
           sum(customer_sales) total,
  2        rank() over(partition by region_id
                 order by sum(customer_sales) desc) rank,
  3        dense_rank() over(partition by region_id
                 order by sum(customer_sales) desc) dense_rank,
  4        row_number() over(partition by region_id
                 order by sum(customer_sales) desc) row_number

  5   from user_order
  6  group by region_id, customer_id;

 REGION_ID CUSTOMER_ID      TOTAL       RANK DENSE_RANK ROW_NUMBER
---------- ----------- ---------- ---------- ---------- ----------
         5           4    1878275          1          1          1
         5           2    1224992          2          2          2
         5           5    1169926          3          3          3
```

```
     6       6          1788836       1       1       1
     6       9          1208959       2       2       2
     6      10          1196748       3       3       3
... ...

30 rows selected.
```

现在我们看到的排名将是基于各个地区的，而非所有区域的了！Partition by 子句在排列函数中的作用是将一个结果集划分成几个部分，这样排列函数就能够应用于这各个子集。

前面我们提到的 5 个问题已经解决了 2 个了(第 1,2)，剩下的 3 个问题(Top/Bottom N，First/Last, NTile)会在下一篇讲解。

Oracle 开发专题之：分析函数 3(Top/Bottom N、First/Last、NTile)

目录

==========

一、带空值的排列：

在前面《Oracle 开发专题之：分析函数 2(Rank、Dense_rank、row_number)》一文中，我们已经知道了如何为一批记录进行全排列、分组排列。假如被排列的数据中含有空值呢？

```
SQL> select region_id, customer_id,
  2       sum(customer_sales) cust_sales,
  3       sum(sum(customer_sales)) over(partition by region_id) ran_total,
  4       rank() over(partition by region_id
  5              order by sum(customer_sales) desc) rank
  6   from user_order
  7  group by region_id, customer_id;

 REGION_ID CUSTOMER_ID CUST_SALES  RAN_TOTAL       RANK
---------- ----------- ---------- ---------- ----------
     10       31            6238901       1
     10       26   1808949  6238901       2
     10       27   1322747  6238901       3
     10       30   1216858  6238901       4
```

```
        10         28    986964    6238901         5
        10         29    903383    6238901         6
```

我们看到这里有一条记录的 CUST_TOTAL 字段值为 NULL，但居然排在第一名了！显然这不符合情理。所以我们重新调整完善一下我们的排名策略，看看下面的语句：

```
SQL> select region_id, customer_id,
  2       sum(customer_sales) cust_total,
  3       sum(sum(customer_sales)) over(partition by region_id) reg_total,
  4       rank() over(partition by region_id
                 order by sum(customer_sales) desc NULLS LAST) rank
  5      from user_order
  6     group by region_id, customer_id;

 REGION_ID CUSTOMER_ID CUST_TOTAL  REG_TOTAL       RANK
---------- ----------- ---------- ---------- ----------
        10         26    1808949    6238901          1
        10         27    1322747    6238901          2
        10         30    1216858    6238901          3
        10         28     986964    6238901          4
        10         29     903383    6238901          5
        10         31    6238901                     6
```

绿色高亮处，NULLS LAST/FIRST 告诉 Oracle 让空值排名最后后第一。

注意是 NULLS，不是 NULL。

二、Top/Bottom N 查询：

在日常的工作生产中，我们经常碰到这样的查询：找出排名前 5 位的订单客户、找出排名前 10 位的销售人员等等。现在这个对我们来说已经是很简单的问题了。下面我们用一个实际的例子来演示：

【1】找出所有订单总额排名前 3 的大客户：

```
SQL> select *
SQL>   from (select region_id,
SQL>                customer_id,
SQL>                sum(customer_sales) cust_total,
SQL>                rank() over(order by sum(customer_sales) desc NULLS LAST) rank
SQL>           from user_order
SQL>          group by region_id, customer_id)
SQL>  where rank <= 3;

 REGION_ID CUSTOMER_ID CUST_TOTAL       RANK
---------- ----------- ---------- ----------
         9          25    2232703          1
         8          17    1944281          2
```

```
      7       14   1929774       3

SQL>
```

【2】找出每个区域订单总额排名前 3 的大客户：

```
SQL> select *
  2    from (select region_id,
  3                 customer_id,
  4                 sum(customer_sales) cust_total,
  5                 sum(sum(customer_sales)) over(partition by region_id) reg_total,
  6                 rank() over(partition by region_id
                          order by sum(customer_sales) desc NULLS LAST) rank
  7            from user_order
  8           group by region_id, customer_id)
  9   where rank <= 3;

 REGION_ID CUSTOMER_ID CUST_TOTAL  REG_TOTAL       RANK
---------- ----------- ---------- ---------- ----------
      5        4   1878275   5585641       1
      5        2   1224992   5585641       2
      5        5   1169926   5585641       3
      6        6   1788836   6307766       1
      6        9   1208959   6307766       2
      6       10   1196748   6307766       3
      7       14   1929774   6868495       1
      7       13   1310434   6868495       2
      7       15   1255591   6868495       3
      8       17   1944281   6854731       1
      8       20   1413722   6854731       2
      8       18   1253840   6854731       3
      9       25   2232703   6739374       1
      9       23   1224992   6739374       2
      9       24   1224992   6739374       2
     10       26   1808949   6238901       1
     10       27   1322747   6238901       2
     10       30   1216858   6238901       3

18 rows selected.
```

三、First/Last 排名查询：

想象一下下面的情形：找出订单总额最多、最少的客户。按照前面我们学到的知识，这个至少需要 2 个查询。第一个查询按照订单总额降序排列以期拿到第一名，第二个查询按照订单总额升序排列以期拿到最后一名。是不是很烦？因为 Rank 函数只告诉我们排名的结果，却无法自动替我们从中筛选结果。

幸好 Oracle 为我们在排列函数之外提供了两个额外的函数：first、last 函数，专门用来解决这种问题。还是用实例说话：

```
SQL> select min(customer_id)
  2       keep (dense_rank first order by sum(customer_sales) desc) first,
  3       min(customer_id)
  4       keep (dense_rank last order by sum(customer_sales) desc) last
  5   from user_order
  6  group by customer_id;

    FIRST       LAST
--------- ----------
       31          1
```

这里有几个看起来比较疑惑的地方：

①为什么这里要用 min 函数
②Keep 这个东西是干什么的
③fist/last 是干什么的
④dense_rank 和 dense_rank()有什么不同，能换成 rank 吗？

首先解答一下第一个问题：min 函数的作用是用于当存在多个 First/Last 情况下保证返回唯一的记录。假如我们去掉会有什么样的后果呢？

```
SQL> select keep (dense_rank first order by sum(customer_sales) desc) first,
  2           keep (dense_rank last order by sum(customer_sales) desc) last
  3   from user_order
  4  group by customer_id;
select keep (dense_rank first order by sum(customer_sales) desc) first,
                  *
ERROR at line 1:
ORA-00907: missing right parenthesis
```

接下来看看第 2 个问题：keep 是干什么用的？从上面的结果我们已经知道 Oracle 对排名的结果只"保留"2 条数据，这就是 keep 的作用。告诉 Oracle 只保留符合 keep 条件的记录。

那么什么才是符合条件的记录呢？这就是第 3 个问题了。dense_rank 是告诉 Oracle 排列的策略，first/last 则告诉最终筛选的条件。

第 4 个问题：如果我们把 dense_rank 换成 rank 呢？

```
SQL> select min(region_id)
  2       keep(rank first order by sum(customer_sales) desc) first,
  3      min(region_id)
  4       keep(rank last order by sum(customer_sales) desc) last
  5   from user_order
  6  group by region_id;
select min(region_id)
*
ERROR at line 1:
ORA-02000: missing DENSE_RANK
```

四、按层次查询：

现在我们已经见识了如何通过 Oracle 的分析函数来获取 Top/Bottom N，第一个，最后一个记录。有时我们会收到类似下面这样的需求：找出订单总额排名前 1/5 的客户。

很熟悉是不？我们马上会想到第二点中提到的方法，可是 rank 函数只为我们做好了排名，并不知道每个排名在总排名中的相对位置，这时候就引入了另外一个分析函数 NTile，下面我们就以上面的需求为例来讲解一下：

```
SQL> select region_id,
  2         customer_id,
  3         ntile(5) over(order by sum(customer_sales) desc) til
  4    from user_order
  5   group by region_id, customer_id;

 REGION_ID CUSTOMER_ID       TILE
---------- ----------- ----------
        10          31          1
         9          25          1
        10          26          1
         6           6          1
         8          18          2
         5           2          2
         9          23          3
         6           9          3
         7          11          3
         5           3          4
         6           8          4
         8          16          4
         6           7          5
        10          29          5
         5           1          5
```

Ntil 函数为各个记录在记录集中的排名计算比例，我们看到所有的记录被分成 5 个等级，那么假如我们只需要前 1/5 的记录则只需要截取 TILE 的值为 1 的记录就可以了。假如我们需要排名前 25%的记录(也就是 1/4)那么我们只需要设置 ntile(4)就可以了。

Oracle 开发专题之：窗口函数

4.窗口函数进阶-根据时间范围统计
5.窗口函数进阶-first_value/last_value
6.窗口函数进阶-比较相邻记录

一、窗口函数简介：

到目前为止，我们所学习的分析函数在计算/统计一段时间内的数据时特别有用，但是假如计算/统计需要随着遍历记录集的每一条记录而进行呢？举些例子来说：

①列出每月的订单总额以及全年的订单总额
②列出每月的订单总额以及截至到当前月的订单总额
③列出上个月、当月、下一月的订单总额以及全年的订单总额
④列出每天的营业额及一周来的总营业额
⑤列出每天的营业额及一周来每天的平均营业额

仔细回顾一下前面我们介绍到的分析函数，我们会发现这些需求和前面有一些不同：前面我们介绍的分析函数用于计算/统计一个明确的阶段/记录集，而这里有部分需求例如 2，需要随着遍历记录集的每一条记录的同时进行统计。

也即是说：统计不止发生一次，而是发生多次。统计不至发生在记录集形成后，而是发生在记录集形成的过程中。

这就是我们这次要介绍的窗口函数的应用了。它适用于以下几个场合：

①通过指定一批记录：例如从当前记录开始直至某个部分的最后一条记录结束
②通过指定一个时间间隔：例如在交易日之前的前 30 天
③通过指定一个范围值：例如所有占到当前交易量总额 5%的记录

二、窗口函数示例一全统计：

下面我们以需求：列出每月的订单总额以及全年的订单总额为例，来看看窗口函数的应用。

【1】测试环境：

```
SQL> desc orders;
 名称                      是否为空? 类型
 ----------------------- -------- ----------------
 MONTH                             NUMBER(2)
 TOT_SALES                         NUMBER

SQL>
```

【2】测试数据：

```
SQL> select * from orders;

    MONTH  TOT_SALES
--------- ----------
```

```
         1     610697
         2     428676
         3     637031
         4     541146
         5     592935
         6     501485
         7     606914
         8     460520
         9     392898
        10     510117
        11     532889
        12     492458

已选择 12 行。
```

【3】测试语句：

回忆一下前面《Oracle 开发专题之：分析函数(OVER)》一文中，我们使用了 sum(sum(tot_sales)) over (partition by region_id) 来统计每个分区的订单总额。现在我们要统计的不单是每个分区，而是所有分区，partition by region_id 在这里不起作用了。

Oracle 为这种情况提供了一个子句：rows between ... preceding and ... following。从字面上猜测它的意思是：在 XXX 之前和 XXX 之后的所有记录，实际情况如何让我们通过示例来验证：

```
SQL> select month,
  2         sum(tot_sales) month_sales,
  3         sum(sum(tot_sales)) over (order by month
  4           rows between unbounded preceding and unbounded following) total_sales
  5    from orders
  6   group by month;

     MONTH MONTH_SALES TOTAL_SALES
---------- ----------- -----------
         1      610697     6307766
         2      428676     6307766
         3      637031     6307766
         4      541146     6307766
         5      592935     6307766
         6      501485     6307766
         7      606914     6307766
         8      460520     6307766
         9      392898     6307766
        10      510117     6307766
        11      532889     6307766
        12      492458     6307766

已选择 12 行。
```

绿色高亮处的代码在这里发挥了关键作用，它告诉 oracle 统计从第一条记录开始至最后一条记录的每月销售额。这个统计在记录集形成的过程中执行了 12 次，这时相当费时的！但至少我们解决了问题。

unbounded preceding and unbouned following 的意思针对当前所有记录的前一条、后一条记录，也就是表中的所有记录。那么假如我们直接指定从第一条记录开始直至末尾呢？看看下面的结果：

```
SQL> select month,
  2       sum(tot_sales) month_sales,
  3       sum(sum(tot_sales)) over (order by month
  4         rows between 1 preceding and unbounded following) all_sales
  5   from orders
  6  group by month;

     MONTH MONTH_SALES  ALL_SALES
---------- ----------- ----------
         1      610697    6307766
         2      428676    6307766
         3      637031    5697069
         4      541146    5268393
         5      592935    4631362
         6      501485    4090216
         7      606914    3497281
         8      460520    2995796
         9      392898    2388882
        10      510117    1928362
        11      532889    1535464
        12      492458    1025347

已选择 12 行。
```

很明显这个语句错了。实际 1 在这里不是从第 1 条记录开始的意思，而是指当前记录的前一条记录。preceding 前面的修饰符是告诉窗口函数执行时参考的记录数，如同 unbounded 就是告诉 oracle 不管当前记录是第几条，只要前面有多少条记录，都列入统计的范围。

三、窗口函数进阶一滚动统计(累积/均值)：

考虑前面提到的第 2 个需求：列出每月的订单总额以及截至到当前月的订单总额。也就是说 2 月份的记录要显示当月的订单总额和 1,2 月份订单总额的和。3 月份要显示当月的订单总额和 1,2,3 月份订单总额的和，依此类推。

很明显这个需求需要在统计第 N 月的订单总额时，还要再统计这 N 个月来的订单总额之和。想想上面的语句，假如我们能够把 and unbounded following 换成代表当前月份的逻辑多好啊！很幸运的是 Oracle 考虑到了我们这个需求，为此我们只需要将语句稍微改成：curreent row 就可以了。

```
SQL> select month,
  2       sum(tot_sales) month_sales,
  3       sum(sum(tot_sales)) over(order by month
  4         rows between unbounded preceding and current row) current_total_sales
```

```
  5   from orders
  6  group by month;

    MONTH MONTH_SALES CURRENT_TOTAL_SALES
--------- ---------- ------------------
        1     610697          610697
        2     428676         1039373
        3     637031         1676404
        4     541146         2217550
        5     592935         2810485
        6     501485         3311970
        7     606914         3918884
        8     460520         4379404
        9     392898         4772302
       10     510117         5282419
       11     532889         5815308
       12     492458         6307766

已选择 12 行。
```

现在我们能得到滚动的销售总额了！下面这个统计结果看起来更加完美，它展现了所有我们需要的数据：

```
SQL> select month,
  2        sum(tot_sales) month_sales,
  3        sum(sum(tot_sales)) over(order by month
  4        rows between unbounded preceding and current row) current_total_sales,
  5        sum(sum(tot_sales)) over(order by month
  6        rows between unbounded preceding and unbounded following) total_sales
  7   from orders
  8  group by month;

    MONTH MONTH_SALES CURRENT_TOTAL_SALES TOTAL_SALES
--------- ---------- ------------------ -----------
        1     610697          610697    6307766
        2     428676         1039373     6307766
        3     637031         1676404     6307766
        4     541146         2217550     6307766
        5     592935         2810485     6307766
        6     501485         3311970     6307766
        7     606914         3918884     6307766
        8     460520         4379404     6307766
        9     392898         4772302     6307766
       10     510117         5282419     6307766
       11     532889         5815308     6307766
       12     492458         6307766     6307766

已选择 12 行。
```

在一些销售报表中我们会时常看到求平均值的需求，有时可能是针对全年的数据求平均值，有时会是针对截至到当前的所有数据求平均值。很简单，只需要将：
sum(sum(tot_sales))换成 avg(sum(tot_sales))即可。

四、窗口函数进阶－根据时间范围统计：

前面我们说过，窗口函数不单适用于指定记录集进行统计，而且也能适用于指定范围进行统计的情况，例如下面这个 SQL 语句就统计了当天销售额和五天内的评价销售额：

```
select trunc(order_dt) day,
       sum(sale_price) daily_sales,
       avg(sum(sale_price)) over (order by trunc(order_dt)
            range between interval '2' day preceding
                      and interval '2' day following) five_day_avg
  from cust_order
 where sale_price is not null
   and order_dt between to_date('01-jul-2001','dd-mon-yyyy')
   and to_date('31-jul-2001','dd-mon-yyyy')
```

为了对指定范围进行统计，Oracle 使用关键字 range、interval 来指定一个范围。上面的例子告诉 Oracle 查找当前日期的前 2 天，后 2 天范围内的记录，并统计其销售平均值。

五、窗口函数进阶－first_value/last_value：

Oracle 提供了 2 个额外的函数：first_value、last_value，用于在窗口记录集中查找第一条记录和最后一条记录。假设我们的报表需要显示当前月、上一个月、后一个月的销售情况，以及每 3 个月的销售平均值，这两个函数就可以派上用场了。

```
select month,
       first_value(sum(tot_sales)) over (order by month
                         rows between 1 preceding and 1 following) prev_month,

       sum(tot_sales) monthly_sales,

       last_value(sum(tot_sales)) over (order by month
                        rows between 1 preceding and 1 following) next_month,

       avg(sum(tot_sales)) over (order by month
                        rows between 1 preceding and 1 following) rolling_avg
  from orders
 where year = 2001
   and region_id = 6
 group by month
 order by month;
```

首先我们来看:rows between 1 preceding and 1 following 告诉 Oracle 在当前记录的前一条、后一条范围内查找并统计，而 first_value 和 last_value 在这 3 条记录中至分别找出第一条、第三条记录，这样我们就轻松地得到相邻三个月的销售记录及平均值了！

六、窗口函数进阶－比较相邻记录：

通过第五部分的学习，我们知道了如何利用窗口函数来显示相邻的记录，现在假如我们想每次显示当月的销售额和上个月的销售额，应该怎么做呢？

从第五部分的介绍我们可以知道，利用 first_value(sum(tot_sales) over (order by month rows between 1 preceding and 0 following))就可以做到了，其实 Oracle 还有一个更简单的方式让我们来比较 2 条记录，它就是 lag 函数。

leg 函数类似于 preceding 和 following 子句，它能够通过和当前记录的相对位置而被应用，在比较同一个相邻的记录集内两条相邻记录的时候特别有用。

```
select  month,
        sum(tot_sales) monthly_sales,
        lag(sum(tot_sales), 1) over (order by month) prev_month_sales
  from orders
 where year = 2001
    and region_id = 6
 group by month
 order by month;
```

lag(sum(tot_sales),1)中的 1 表示以 1 月为基准。

Oracle 开发专题之：报表函数

一、报表函数简介：

回顾一下前面《Oracle 开发专题之：窗口函数》中关于全统计一节，我们使用了 Oracle 提供的：

```
 sum(sum(tot_sales)) over (order by month rows between unbounded preceding and unbounded followin
g)
```

来统计全年的订单总额，这个函数会在记录集形成的过程中，每检索一条记录就执行一次，它总共执行了 12 次。这是非常费时的。实际上我们还有更简便的方法：

```
SQL> select month,
  2        sum(tot_sales) month_sales,
  3        sum(sum(tot_sales)) over(order by month
  4        rows between unbounded preceding and unbounded following) win_sales,
  5        sum(sum(tot_sales)) over() rpt_sales
  6   from orders
  7  group by month;
```

```
    MONTH MONTH_SALES WINDOW_SALES REPORT_SALES
--------- ---------- ------------ ------------
        1     610697      6307766      6307766
        2     428676      6307766      6307766
        3     637031      6307766      6307766
        4     541146      6307766      6307766
        5     592935      6307766      6307766
        6     501485      6307766      6307766
        7     606914      6307766      6307766
        8     460520      6307766      6307766
        9     392898      6307766      6307766
       10     510117      6307766      6307766
       11     532889      6307766      6307766
       12     492458      6307766      6307766

已选择 12 行。
```

over 函数的空括号表示该记录集的所有记录都应该被列入统计的范围，如果使用了 partition by 则先分区，再依次统计各个分区。

二、RATIO_TO_REPORT 函数：

报表函数特(窗口函数)特别适合于报表中需要同时显示详细数据和统计数据的情况。例如在销售报告中经常会出现这样的需求：列出上一年度每个月的销售总额、年底销售额以及每个月的销售额占全年总销售额的比例：

方法①：

```
select all_sales.*,
       100 * round(cust_sales / region_sales, 2) || '%' Percent
 from (select o.cust_nbr customer,
              o.region_id region,
              sum(o.tot_sales) cust_sales,
              sum(sum(o.tot_sales)) over(partition by o.region_id) region_sales
         from orders_tmp o
        where o.year = 2001
        group by o.region_id, o.cust_nbr) all_sales
 where all_sales.cust_sales > all_sales.region_sales * 0.2;
```

这是一种笨方法也是最易懂的方法。

方法②：

```
select region_id, salesperson_id,
       sum(tot_sales) sp_sales,
       round(sum(tot_sales) / sum(sum(tot_sales))
             over (partition by region_id), 2) percent_of_region
  from orders
```

```
where year = 2001
 group by region_id, salesperson_id
 order by region_id, salesperson_id;
```

方法③

```
select region_id, salesperson_id,
          sum(tot_sales) sp_sales,
          round(ratio_to_report(sum(tot_sales))
                    over (partition by region_id), 2) sp_ratio
   from orders
where year = 2001
group by region_id, salesperson_id
order by region_id, salesperson_id;
```

Oracle 提供的 Ratio_to_report 函数允许我们计算每条记录在其对应记录集或其子集中所占的比例。

Oracle 开发专题之：分析函数总结

这一篇是对前面所有关于分析函数的文章的总结：

一、统计方面：

```
Sum(…) Over ([Partition by …] [Order by …])

Sum(…) Over ([Partition by …] [Order by …]
    Rows Between … Preceding And … Following)

Sum(…) Over ([Partition by …] [Order by …]
    Rows Between … Preceding And Current Row)

Sum(…) Over ([Partition by …] [Order by …]
    Range Between Interval '…' 'Day' Preceding
                And Interval '…' 'Day' Following )
```

具体请参考《Oracle 开发专题之：分析函数(OVER)》和《Oracle 开发专题之：窗口函数》

二、排列方面：

```
Rank() Over ([Partition by …] [Order by …] [Nulls First/Last])

Dense_rank() Over ([Patition by …] [Order by …] [Nulls First/Last])

Row_number() Over ([Partitionby …] [Order by …] [Nulls First/Last])

Ntile(…) Over ([Partition by …] [Order by …])
```

具体请参考《Oracle 开发专题之：分析函数 2》

三、最大值/最小值查找方面：

```
Min(…)/Max(…) Keep (Dense_rank First/Last [Partition by …] [Order by …])
```

具体请参考《Oracle 开发专题之：分析函数 3》

四、首记录/末记录查找方面：

```
First_value / Last_value(Sum(…) Over ([Patition by …] [Order by …]
      Rows Between … Preceding And … Following  ))
```

具体请参考《Oracle 开发专题之：窗口函数》

五、相邻记录之间比较方面：

```
Lag(Sum(…), 1) Over([Patition by …] [Order by …])
```

具体请参考《Oracle 开发专题之：报表函数》

Oracle 开发专题之：26 个分析函数

分析函数	用途
AVG (<distinct|all> expression)	用于计算一个组和窗口内表达式的平均值。Distinct 用于去掉重复的数据后得到该组的平均值
CORR (expression, expression)	返回一对表达式的相关系数，它是如下的缩写： COVAR_POP(expr1, expr2) / STDDEV_POP(expr1) * STDDEV_POP(expr2)). 从统计上讲，相关性是变量之间关联的强度，变量之间的关联意味着在某种程度上一个变量的值可由其他的值进行预测。通过返回一个-1～1 之间的一个数，相关系数给出了关联的强度，0 表示不相关
COUNT (<distinct> <*> <expression>)	它将对一组内发生的事情进行计数。如果指定*或一些非空常数，count 将对所有的行计数。如果指定一个表达式，count 返回表达式非空赋值计数。可以使用 DISTINCT 关键字去掉一组中完全相同的数据后出现的行数
COVAR_POP (expression, expression)	返回一对表达式的总体协方差
COVAR_SAMP (expression, expression)	返回一对表达式的样本协方差

CUME_DIST	计算一行在组中的相对位置。CUME_DIST 总是返回大于 0、小于或等于 1 的数，该数表示该行在 N 行中的位置。例如，在一个 3 行的组中，返回的累计分布值为 1/3、2/3 和 3/3
DENSE_RANK	根据 ORDER BY 子句中表达式的值，从查询返回的每一行，计算它们与其他行的相对位置。组内的数据按 ORDER BY 子句排序，然后给每一行赋一个号，从而形成一个序列，该序列从 1 开始，往后累加。每次 ORDER BY 表达式的值发生变化时，该序列也随之增加。有同样值的行得到同样的数字序号(认为 null 是相等的)。密集的序列返回的是没有间隔的数
FIRST_VALUE	返回组中的第一个值
LAG (expression, <offset>, <default>)	LAG 可以访问结果集中的其他行而不用进行自连接。它允许去处理游标，就好像游标是一个数组一样。在给定组中可参考当前行之前的行，这样就可以从组中与当前行一起选择以前的行。关于如何获取“下一行”可参看 LEAD Offset 是一个正整数，其默认值为 1。若索引超出窗口的范围，就返回默认值(默认返回的是组中第一行)
LAST_VALUE	返回组中的最后一个值
LEAD (expression, <offset>, <default>)	LEAD 与 LAG 相反，LAG 让您可访问组中当前行之前的行，而 LEAD 让您可访问组中当前行之后的行 Offset 是一个正整数，其默认值为 1。若索引超出窗口的范围，就返回默认值(默认返回的是组中最后一行)
MAX(expression)	在一个组的窗口中查找表达式的最大值
MIN(expression)	在一个组的窗口中查找表达式的最小值
NTILE (expression)	将一个组分为“表达式值”的数列表示 例如，如果表达式=4，则给组中的每一行分配一个数(从 1 到 4)。如果组中有 20 行，则给前 5 行分配 1，给下 5 行分配 2 等等。如果组的基数不是由表达式平均分开的，则对这些行进行分配时，组中就没有任何 percentile 的行数比其他 percentile 的行数超过一行，最低的 percentile 是那些拥有额外行的 percentile。例如，若表达式=4，行数=21，则 percentile = 1 的有 6 行，percentile = 2 的有 5 行等等
PERCENT_RANK	它与 CUME_DIST (累积分配)函数类似。对于一个组中给定的行来说，在计算那行的序号时，先减去 1，然后除以 1(1 小于组中所求的行数)。该函数总是返回 0~1(包括 1)之间的数

RANK	根据 ORDER BY 子句中表达式的值，从查询返回的每一行，计算它们与其他行的相对位置。组内的数据按 ORDER BY 子句排序，然后给每一行赋一个数字序号，从而形成一个序列，该序列从 1 开始，往后累加。每次 ORDER BY 表达式的值发生变化时，该序列也随之增加。有同样值的行得到同样的数字序号。然而，如果两行的确得到同样的排序，则序数将随后跳跃。若两行序数为 1，则没有序数 2，序列将给组中的下一行分配值 3，DENSE_RANK 则没有任何跳跃值
RATIO_TO_REPORT (expression)	该函数计算 expression / (sum(expression))的值 它给出相对于总数的百分比，即当前行对 sum(expression)的贡献
REGR_ xxxxxxx (expression, expression)	这些线性回归函数适合最小二乘法回归线，有 9 个不同的回归函数可使用
ROW_NUMBER	返回有序组中一行的偏移量，从而可用于按特定标准排序的行号
STDDEV (expression)	计算当前行关于组的标准偏离
STDDEV_POP (expression)	该函数计算总体标准偏离，并返回总体变量的平方根，其返回值与 VAR_POP 函数的平方根相同
STDDEV_SAMP (expression)	该函数计算累积样本标准偏离，并返回样本变量的平方根，其返回值与 VAR_SAMP 函数的平方根相同
SUM(expression)	该函数计算组中表达式的累积和
VAR_POP (expression)	该函数返回非空集合的总体变量(忽略 null)，VAR_POP 进行如下计算： (SUM(expr*expr)–SUM(expr)*SUM(expr)/COUNT(expr))/ COUNT(expr)
VAR_SAMP (expression)	该函数返回非空集合的样本变量(忽略 null)，它进行如下计算： (SUM(expr*expr)–SUM(expr)*SUM(expr)/ COUNT(expr))/(COUNT(expr)–1)
VARIANCE (expression)	该函数返回表达式的变量，Oracle 计算该变量如下： 如果表达式中行数为 1，则返回 0 如果表达式中行数大于 1，则返回 VAR_SAMP

参考资料：《Mastering Oracle SQL》(By Alan Beaulieu, Sanjay Mishra O'Reilly June 2004 0-596-00632-2)

PLSQL 开发笔记和小结

****

PLSQL 基本结构

****

基本数据类型变量

1. 基本数据类型

Number 数字型

Int 整数型

Pls_integer 整数型，产生溢出时出现错误

Binary_integer 整数型，表示带符号的整数

Char 定长字符型，最大 255 个字符

Varchar2 变长字符型，最大 2000 个字符

Long 变长字符型，最长 2GB

Date 日期型

Boolean 布尔型（TRUE、FALSE、NULL 三者取一）

在 PL/SQL 中使用的数据类型和 Oracle 数据库中使用的数据类型，有的含义是完全一致的，有的是有不同的含义的。

2. 基本数据类型变量的定义方法

变量名 类型标识符 [not null]:=值;

```
declare
   age number(3):=26; --长度为 3，初始值为 26
begin
   commit;
end;
```

其中，定义常量的语法格式：

常量名 constant 类型标识符 [not null]:=值;

```
declare
```

```
    pi constant number(9):=3.1415926;--为 pi 的数字型常量，长度为 9，初始值为 3.1415926
  begin
    commit;
  end;
```

表达式

变量、常量经常需要组成各种表达式来进行运算，下面介绍在 PL/SQL 中常见表达式的运算规则。

1. 数值表达式

PL/SQL 程序中的数值表达式是由数值型常数、变量、函数和算术运算符组成的，可以使用的算术运算符包括+（加法）、-（减法）、*（乘法）、/（除法

）和**（乘方）等。

命令窗口中执行下列 PL/SQL 程序，该程序定义了名为 result 的整数型变量，计算的是 10+3*4-20+5**2 的值，理论结果应该是 27。

```
  ————————————————————————————————————
  set serveroutput on
  Declare
    result integer;
  begin
    result:=10+3*4-20+5**2;
    dbms_output.put_line('运算结果是：'||to_char(result));
  end;
  ————————————————————————————————————
```

dbms_output.put_line 函数输出只能是字符串，因此利用 to_char 函数将数值型结果转换为字符型。

2. 字符表达式

字符表达式由字符型常数、变量、函数和字符运算符组成，唯一可以使用的字符运算符就是连接运算符“||”。

3. 关系表达式

关系表达式由字符表达式或数值表达式与关系运算符组成，可以使用的关系运算符包括以下 9 种。

< 小于

> 大于

= 等于（不是赋值运算符:=）

like 类似于

in 在.....之中

<= 小于等于

>= 大于等于

!= 不等于 或<>

between 在.....之间

关系型表达式运算符两边的表达式的数据类型必须一致。

4. 逻辑表达式

逻辑表达式由逻辑常数、变量、函数和逻辑运算符组成，常见的逻辑运算符包括以下 3 种。

NOT：逻辑非

OR：逻辑或

AND：逻辑与

运算的优先次序为 NOT、AND 和 OR。

PLSQL 函数

PL/SQL 程序中提供了很多函数供扩展功能，除了标准 SQL 语言的函数可以使用外，最常见的数据类型转换函数有以下 3 个。

To_char：将其他类型数据转换为字符型。

To_date：将其他类型数据转换为日期型。

To_number：将其他类型数据转换为数值型。

继续追加中..

系统输出打印

利用 pl/sql 在数据库服务器端打印一句话：

```
set serveroutput on--设置数据库输出，默认为关闭，每次重新打开窗口需要重新设置。
BEGIN
   DBMS_OUTPUT.PUT_LINE('Hello PL/SQL');
END;
```

pl/sql 程序中对大小写不敏感（打印声明的变量）

————————————————————————————————

```
set serveroutput on
DECLARE
 v_char varchar2(20):='a';
 v_char1 varchar2(20):='b';
BEGIN
 DBMS_OUTPUT.PUT_LINE(v_char);
 DBMS_OUTPUT.PUT_LINE(v_char1);
END;
```

pl 语句块是 pl/sql 里最小的编程块，其中可以再嵌套 begin end

```
begin
 dbms_output.put_line('Hello World');
 dbms_output.put_line('2*3='||(2*3));
 dbms_output.put_line('what''s');
end;
```

————————————————————————————————

PL/SQL 中的变量声明

所有变量必须在 declare 中声明，程序中不允许声明。

没有初始化的变量默认值为 null，屏幕上 null 是看不见的，命名习惯：PL/SQL 中变量一般以 v_开头（等同于存储过程中 as 和 begin 区域的变量定义习惯）。

注意 number 也能存小数，最长 38 位，所以以后建议整数都用 binary_integer 存。

long 是字符类型，boolean 类型不能打印。

标准变量类型：数字，字符，时间，布尔。

————————————————————————————————

```
declare
v_number1 number;
v_number2 number(3,2) ;
v_number3 binary_integer :=1;
v_name varchar2(20) :='kettas';
v_date date :=sysdate;
v_long long :='ni hao';
v_b boolean := true;
begin
if (v_number1 is null) then
 dbms_output.put_line( 'hello');
end if;
dbms_output.put_line(v_number1);
dbms_output.put_line(v_number2);
dbms_output.put_line(v_number3);
dbms_output.put_line(v_name);
dbms_output.put_line(v_date);
dbms_output.put_line(v_long);
  --dbms_output.put_line(v_b); --执行该句 ORACLE 提示“调用 'PUT_LINE' 时参数个数或类型错误”
end;
```

————————————————————————————————

备注：

关于声明 number(4,3)中括号中的两个数字的意义，前面的数字叫精度，后面的叫刻度。

刻度：

当刻度为正数的时候，表示四舍五入到小数点后面的位数

当刻度为负数的时候，表示四舍五入到小数点前面的位数

精度：

从数字的最前面不为零开始到刻度精确到的位置

v_Number number(4,3):=123.12312

1、按刻度进行四舍五入得到 123.123

2、确定刻度精确到的位置 123123 处，精度为 6 位（.符号不算）

2、根据精度进行判断 6 位（>4）精度上限值 --报错不能存储

number(3,-3):=44445

1、根据刻度－3 进行四舍五入得到 44000

2、小数点向前移动 3 位 44.此位置为刻度精确到的位置

3、根据精度进行判断 2 位（<3）精度上限值 --不报错可存储结果为 44000

```
DECLARE
 v_Number number(4,3):=123.12312;--实际精度 6 位大于上限精度值 4 位，提示“ORA-06502: PL/SQL: 数字或值错误 : 数值精度太高”
BEGIN
 DBMS_OUTPUT.PUT_LINE(v_Number);
END
;

DECLARE
 v_Number number(7,3):=4555; --实际精度 7 位等于上限精度值，可以存储
BEGIN
 DBMS_OUTPUT.PUT_LINE(v_Number);
END
;
```

****

变量赋值方式

****

oracle 中变量赋值方式是值拷贝而非引用

```
declare
   v_number1 number:=100;
   v_number2 number;
begin
   v_number2:=v_number1;
   v_number1:=200;
   dbms_output.put_line(v_number1); --200
   dbms_output.put_line(v_number2); --100

end;
```

****

PLSQL 复合类型

****

记录类型 record

record 类型最常用，声明的时候可以加 not null，但必须给初始值，如果 record 类型一致可以相互赋值，如果类型不同，里面的字段恰好相同，不能互相赋值。引用记录型变量的方法是“记录变量名.基本类型变量名”。

```
--------------------------------------------
declare
   type t_first is record(
      id number(3),
      name varchar2(20)
   );
   v_first t_first;
```

```
begin
  v_first.id:=1;
  v_first.name:='cheng';
  dbms_output.put_line(v_first.id);
  dbms_output.put_line(v_first.name);
end;
```

record 类型变量间赋值

```
declare
    type t_first is record(
      id number,
      name varchar2(20)
    );
    v_first t_first;
    v_second t_first;
begin
    v_first.id:=1;
    v_first.name:='susu';

    v_second:=v_first;--相互赋值

    v_first.id:=2;
    v_first.name:='kettas';
    dbms_output.put_line(v_first.id);
    dbms_output.put_line(v_first.name);
    dbms_output.put_line(v_second.id);
    dbms_output.put_line(v_second.name);
end;
```

————————————————————————————————

表类型变量 table

语法如下：

type 表类型 is table of 类型 index by binary_integer;

表变量名 表类型;

类型可以是前面的类型定义，index by binary_integer 子句代表以符号整数为索引，这样访问表类型变量中的数据方法就是“表变量名(索引符号整数)”。table 类型，相当于 java 中的 Map 容器，就是一个可变长的数组，key（符号整数索引）必须是整数，可以是负数，value（类型）可以是标量，也可以是 record 类型。可以不按顺序赋值，但必须先赋值后使用。

1. 定义一维表类型变量

```
--------------------------------------------
declare
   type t_tb is table of varchar2(20) index by binary_integer;
   v_tb t_tb;
begin
  v_tb(100):='hello';
  v_tb(98):='world';
  dbms_output.put_line(v_tb(100));
  dbms_output.put_line(v_tb(98));
end;
```

类型为 record 的表类型变量

```
declare
   type t_rd is record(id number,name varchar2(20));
   type t_tb is table of t_rd index by binary_integer;
   v_tb2 t_tb;
begin
   v_tb2(100).id:=1;
   v_tb2(100).name:='hello';
   --dbms_output.put_line(v_tb2(100).id);
   --dbms_output.put_line(v_tb2(100).name);
```

```
    dbms_output.put_line(v_tb2(100).id||'    '||v_tb2(100).name);
  end;
  ----------------------------------------
```

2. 定义多维表类型变量

该程序定义了名为 tabletype1 的多维表类型，相当于多维数组，table1 是多维表类型变量，将数据表 tempuser.testtable 中 recordnumber 为 60 的记录提取出来

存放在 table1 中并显示。

```
  ----------------------------------------
  declare
    type tabletype1 is table of testtable%rowtype index by binary_integer;
    table1 tabletype1;
  begin
     select * into table1(60) from tempuser.testtable where recordnumber=60;
     dbms_output.put_line(table1(60).recordnumber||table1(60).currentdate);
  end;
```

备注：在定义好的表类型变量里，可以使用 count、delete、first、last、next、exists 和 prior 等属性进行操作，使用方法为“表变量名.属性”，返回的是数字。

```
  set serveroutput on
  declare
     type tabletype1 is table of varchar2(9) index by binary_integer;
     table1 tabletype1;
  begin
     table1(1):='成都市';
     table1(2):='北京市';
     table1(3):='青岛市';
     dbms_output.put_line('总记录数：'||to_char(table1.count));
```

```
    dbms_output.put_line('第一条记录：'||table1.first);
    dbms_output.put_line('最后条记录：'||table1.last);
    dbms_output.put_line('第二条的前一条记录：'||table1.prior(2));
    dbms_output.put_line('第二条的后一条记录：'||table1.next(2));
end;
————————————————————————————————
```

****

%type 和%rowtype

****

使用%type 定义变量，为了让 PL/SQL 中变量的类型和数据表中的字段的数据类型一致，Oracle 9i 提供了%type 定义方法。这样当数据表的字段类型修改后，PL/SQL 程序中相应变量的类型也自动修改。

```
————————————————————————————————
create table student(
  id number,
  name varchar2(20),
  age number(3,0)
);

insert into student(id,name,age) values(1,'susu',23);
--查找一个字段的变量

declare
  v_name varchar2(20);
  v_name2 student.name%type;
begin
  select name into v_name2 from student where rownum=1;
  dbms_output.put_line(v_name2);
end;
```

```
--查找多个字段的变量
declare
   v_id student.id%type;
   v_name student.name%type;
   v_age student.age%type;
begin
 select id,name,age into v_id,v_name,v_age from student where rownum=1;
 dbms_output.put_line(v_id||'  '||v_name||'  '||v_age);
end;

--查找一个类型的变量，推荐用*

declare
  v_student student%rowtype;
begin
  select * into v_student from student where rownum=1;
  dbms_output.put_line(v_student.id||'  '||v_student.name||'  '||v_student.age);
end;

--也可以按字段查找，但是字段顺序必须一样，不推荐这样做
declare
  v_student student%rowtype;
begin
 select id,name,age into v_student from student where rownum=1;
 dbms_output.put_line(v_student.id||'  '||v_student.name||'  '||v_student.age);
end;

declare
  v_student student%rowtype;
begin
 select id,name,age into v_student.id,v_student.name,v_student.age from student where id=1;
```

```
--select * into v_student.id,v_student.name,v_student.age from student where id=1;
dbms_output.put_line();
end;
```

备注：insert，update，delete，select 都可以，create table，drop table 不行。DPL，DML，和流程控制语句可以在 pl/sql 里用，但 DDL 语句不行。

```
declare
  v_name student.name%type:='wang';
begin
  insert into student(id,name,age) values(2,v_name,26);
end;

begin
  insert into student(id,name,age) values(5,'hehe',25);
end;

declare
  v_name student.name%type:='hexian';
begin
  update student set name=v_name where id=1;
end;

begin
  update student set name='qinaide' where id=2;
end;
```

****

PLSQL 变量的可见空间

****

变量的作用域和可见性，变量的作用域为变量申明开始到当前语句块结束。当外部过程和内嵌过程定义了相同名字的变量的时候，在内嵌过程中如果直接写这个变量名是没有办法访问外部过程的变量的，可以通过给外部过程定义一个名字<<outername>>,通过 outername 变量名来访问外部过程的变量（待测试..）。

```
————————————————————————————————
declare
    v_i1 binary_integer:=1;
begin
   declare
     v_i2 binary_integer:=2;
   begin
     dbms_output.put_line(v_i1);
     dbms_output.put_line(v_i2);
   end;
  dbms_output.put_line(v_i1);
--dbms_output.put_line(v_i2);   解开后执行 Oracle 会提示“必须说明标识符  'V_I2'”
end;
————————————————————————————————
```

****

PLSQL 流程控制

****

if 判断

```
declare
    v_b boolean:=true;
begin if v_b then
     dbms_output.put_line('ok');
   end if;
end;
```

if else 判断

```
declare
    v_b boolean:=true;
begin
  if v_b then
    dbms_output.put_line('ok');
  else
    dbms_output.put_line('false');
  end if;
end;
```

if elsif else 判断

```
declare
    v_name varchar2(20):='cheng';
begin
  if v_name='0701' then
    dbms_output.put_line('0701');
  elsif v_name='cheng' then
    dbms_output.put_line('cheng');
  else
    dbms_output.put_line('false');
  end if;
end;
```

loop 循环，注意推出 exit 是推出循环，而不是推出整个代码块

```
declare
   v_i binary_integer:=0;
begin
  loop
    if v_i>10 then
```

```
        exit;
      end if;
      v_i:=v_i+1;
      dbms_output.put_line('hehe');
    end loop;
      dbms_output.put_line('over');
end;
```

loop 简化写法

```
declare
     v_i binary_integer :=0;
begin
    loop
      exit when v_i>10;
      v_i :=v_i+1;
      dbms_output.put_line('hehe');
    end loop;
      dbms_output.put_line('over');
end;
```

while 循环

```
declare
     v_i binary_integer:=0;
begin
     while v_i<10 loop
         dbms_output.put_line('hello'||v_i );
         v_i:=v_i+1;
     end loop;
     dbms_output.put_line('over');
end;
```

for 循环，注意不需要声明变量

```
begin
    for v_i in 0..10 loop
       dbms_output.put_line('hello'||v_i);
    end loop;
       dbms_output.put_line('over');
end;
```

****

PLSQL 异常处理

****

1、声明异常

异常名 EXCEPTION;

2、抛出异常

RAISE 异常名

3、处理异常

抛出异常后的逻辑代码不会被继续执行

异常的定义使用

```
————————————————————————————————
begin
      dbms_output.put_line(1/0);
exception
       when others then
          dbms_output.put_line('error');
end;

declare
       e_myException exception;
begin
```

```
        dbms_output.put_line('hello');
        raise e_myException; --raise 抛出异常,用此关键字，抛出后转到自定义的 e_myException ，执行其里面的 putline 函数后，再跳到 end 处，结束 PL/SQL 块，raise 接下面的 2 句不会继续执行。
        dbms_output.put_line('world');
        dbms_output.put_line(1/0);
    exception
        when e_myException then
          dbms_output.put_line(sqlcode); --当前会话执行状态，错误编码
          dbms_output.put_line(sqlerrm); --当前错误信息
          dbms_output.put_line('my error');
        when others then
          dbms_output.put_line('error');
    end;
    ————————————————————————————————
```

****

PLSQL 游标和 goto 语句

****

备注：下面提到的游标为**静态 cursor，包括显示和隐式**。

游标，从 declare、open、fetch、close 是一个完整的生命旅程。当然了一个这样的游标是可以被多次 open 进行使用的，显式 cursor 是静态 cursor，她的作用域是全局的，但也必须明白，静态 cursor 也只有 pl/sql 代码才可以使用它。静态游标变量是在**定义**时就必须**指定 SQL** 语句。

cursor 游标(结果集)用于提取多行数据，定义后不会有数据，使用后才有。一旦游标被打开，就无法再次打开(可以先关闭，再打开)。

```
    declare
        cursor c_student is   select * from book;
    begin
        open c_student;
```

```
    close c_student;
end;
```

第二种游标的定义方式，用变量控制结果集的数量。

```
declare
    v_id binary_integer;
    cursor c_student is select * from book where id>v_id;
begin
    v_id:=10;
    open c_student;
    close c_student;
end;
```

第三种游标的定义方式，带参数的游标，用的最多。

```
declare
    cursor c_student(v_id binary_integer) is select * from book where id>v_id;
begin
    open c_student(10);
    close c_student;
end;
```

游标的使用，一定别忘了关游标。

```
declare
    v_student book%rowtype;
    cursor c_student(v_id binary_integer) is select * from book where id>v_id;
begin
    open c_student(10);
    fetch c_student into v_student;
    close c_student;
```

```
        dbms_output.put_line(v_student.name);
    end;
```

如何遍历游标 fetch

游标的属性 %found，%notfound,%isopen,%rowcount。

%found:若前面的 fetch 语句返回一行数据，则%found 返回 true，如果对未打开的游标使用则报 ORA-1001 异常。

%notfound，与%found 行为相反。

%isopen，判断游标是否打开。

%rowcount:当前游标的指针位移量，到目前位置游标所检索的数据行的个数，若未打开就引用，返回 ORA-1001。

注：

no_data_found 和%notfound 的用法是有区别的，小结如下

1）SELECT . . . INTO 语句触发 no_data_found；

2）当一个显式光标(静态和动态)的 where 子句未找到时触发 %notfound；

3）当 UPDATE 或 DELETE 语句的 where 子句未找到时触发 sql%notfound；

4）在光标的提取(Fetch)循环中要用 %notfound 或%found 来确定循环的退出条件，不要用 no_data_found。

下面是几个实例：

```
create table BOOK
(
  ID       VARCHAR2(10) not null,
  BOOKNAME VARCHAR2(10) not null,
  PRICE    VARCHAR2(10) not null,
  CID      VARCHAR2(10) not null
);

--insert
create or replace procedure say_hello(
i_name in varchar2,
o_result_msg out varchar2
```

```
)
    as
    v_price varchar2(100);
    e_myException exception;

    begin
      insert into book(id,bookname,price) values (1,2,3);
      o_result_msg := 'success';
    exception
      when others then
          rollback;
          o_result_msg := substr(sqlerrm, 1, 200);
    end;

--update or delete
create or replace procedure say_hello(
i_name in varchar2,
o_result_msg out varchar2
)
    as
    v_price varchar2(100);
    e_myException exception;

    begin
      update book set price = '55' where bookname = i_name;
      delete from book where bookname = i_name;
      if sql%notfound then
        raise e_myException;
      end if;
      /*
```

```
 if sql%rowcount = 0 then--写法 2
           raise e_myException;
        end if;
*/
        o_result_msg := 'success';
     exception
        when e_myException then
            rollback;
            o_result_msg := 'update or delete dail';
     end;

--select
create or replace procedure say_hello(
i_name in varchar2,
o_result_msg out varchar2
)
     as
     v_price varchar2(100);
     e_myException exception;

     begin
        select price into v_price from book where bookname = i_name;
        o_result_msg := 'success';
     exception
        when no_data_found then
            rollback;
            o_result_msg := 'select into dail';
     end;
```

loop 方式遍历游标

```
declare
    v_bookname   varchar2(100);
    cursor c_book(i_id number) is select bookname from book where id = i_id;
begin
   Open   c_book(i_id);
   Loop
      Fetch c_book into v_bookname;
      exit when c_student%notfound;
        update book set price = '33' where bookname = v_bookname;
   End Loop;
   Close c_book;
end;
```

或

```
declare
    v_bookname   varchar2(100);
    cursor c_book(i_id number) is select bookname from book where id = i_id;
begin
   Open   c_book(i_id);
    Fetch c_book into v_bookname;
    While c_book%Found
    Loop
       update book set price = '33' where bookname = v_bookname;
    Fetch   c_book into v_bookname;
    End Loop;
   Close c_book;
end;
```

while 循环遍历游标，注意，第一次游标刚打开就 fetch，%found 为 null，进不去循环

解决方法：while nvl(c_student%found,true) loop

```
declare
   v_bookname   varchar2(100);
   cursor c_book(i_id number) is select bookname from book where id = i_id;
begin
   Open   c_book(i_id);
   while nvl(c_book%found,true) --或这种写法：while c_book%found is null or c_book%found loop
      Fetch c_book into v_bookname;
      update book set price = '33' where bookname = v_bookname;
   End Loop;
   Close c_book;
end;
```

for 循环遍历，最简单，用的最多，不需要 声明 v_student,Open 和 Close 游标和 fetch 操作(不用打开游标和关闭游标，实现遍历游标最高效方式)

```
declare
   cursor c_book(i_id number) is select bookname from book where id = i_id;
begin
   for cur in c_book(i_id) --直接将入参 i_id 传入 cursor 即可
   loop
      update book set price = '53' where bookname = cur.bookname;
   end loop;
end;
```

goto 例子，一般不推荐使用 goto，会使程序结构变乱

```
declare
   i number:=0;
begin
   if i=0 then
```

```
        goto hello;
    end if;
    <<hello>>
    begin
        dbms_output.put_line('hello');
        goto over;
    end;
    <<world>>
    begin
        dbms_output.put_line('world');
        goto over;
    end;
    <<over>>
        dbms_output.put_line('over');
end;
```

****

Oracle 存储过程

****

在谈存储过程书写中的一些规则时，先看一下执行它的规则，在命令窗口执行存储过程 sp_get_product_prompt

```
set serveroutput on
var ret1 varchar2(200);
var ret2 varchar2(200);
exec sp_get_product_prompt(83,:ret1,:ret2); --或 execute
print ret1;
print ret2;
```

或

```
set serveroutput on
declare
  ret1 varchar2(200);
```

```
  ret2 varchar2(200);
begin
  sp_get_product_prompt(83,ret1,ret2);
  dbms_output.put_line(ret1);
  dbms_output.put_line(ret2);
end;
```

存储过程入参，不论类型，**缺省**情况下值都为 **null**,入参和出参不能有长度,其中关键字 as 可以替换成 is,存储过程中变量声明在 as 和 begin 之间，同时，存储过程中可以再调用其它的存储过程，如果要保证存储过程之间的事务处理不受影响，可以定义为自治事务。

```
create or replace procedure say_hello(
  v_name in varchar2,
  v_flag number,
  o_ret out number
)
as
begin
  if v_name is null and v_flag is null then --v_name 和 v_flag 都等于 null
     o_ret := 10;
  else
     o_ret := 100;
  end if;
end;
```

对于入参为 null 情况下给予缺省值

```
create or replace procedure say_hello(
  i_name in varchar2,
  i_flag number,
  o_ret out number
)
```

```
as
 v_name   varchar2(100);
begin
 if i_name is null then
   v_name := '0';
 else
   v_name := i_name;
 end if;
 insert into phone(..,wname..,) values(..,v_name,..);

end;
```

或直接在 insert 语句中调用 nvl 函数赋缺省值

insert into phone(..,wname..,) values(..,nvl(v_name,' '),..); ----如果将' '写成",则 insert 进来的 v_name 值还是为"等价于 null 值

带一个参数的存储过程

输入参数 in，输入参数不能进行:=赋值，但可以将它赋给 as 后面定义的变量；

输入参数 in，可以作为变量进行条件判断；

默认不写就是 in；

存储过程没有重载，这个有参的 say_hello 会替代已经存在的无参 say_hello。

```
create or replace procedure say_hello(v_name in varchar2)
as
begin
  --v_name:='a'; --存储过程入参 v_name 不能做为赋值目标
  dbms_output.put_line('hello '||v_name);
end;
```

存储过程输入参数作为变量进行条件判断

```
create or replace procedure say_hello(
```

```
    i_opFlag in number
  )
  as
    v_name varchar2(100);
  begin
    if i_opFlag = 1 then
  v_name :='0';
    else
  v_name :='haha';
    end if;
    dbms_output.put_line('hello '||v_name);
  end;
```

利用存储过程中定义的变量对入参的空值处理：

```
  create or replace procedure say_hello(
    i_name in varchar2
  )
  as
    v_name varchar2(100);
  begin
    if i_name is null then
  v_name :='0';
    else
  v_name :=i_name;--将入赋值给定义变量
    end if;
    dbms_output.put_line('hello '||v_name);
  end;
```

多个参数的存储过程

```
create or replace procedure say_hello(
  v_first_name in varchar2,
  v_last_name in varchar2)
as
begin
  dbms_output.put_line('hello '||v_first_name||'.'||v_last_name);
end;
```

out 输出参数，用于利用存储过程给一个或多个变量赋值，类似于返回值

```
create or replace procedure say_hello(
  v_name in varchar2,
  v_content out varchar2
)
begin
  v_content:='hello'||v_name;
end;
```

调用：

```
declare
  v_con varchar2(200);
  v_in varchar2(20):='wang';
begin
  say_hello(v_in,v_con);
  dbms_output.put_line(v_con);
end;
```

in out 参数，既赋值又取值

```
create or replace procedure say_hello(v_name in out varchar2)
as
```

```
begin
  v_name:='hi '||v_name;
end;
```

调用：

```
declare
  v_inout varchar2(20):='wangsu';
begin
  say_hello(v_inout);
  dbms_output.put_line(v_inout);
end;
```

对存储过程入参赋缺省值

```
create or replace procedure say_hello(
  v_name varchar2 default 'susu',
  v_content varchar2 default 'hello'
)
as
begin
  dbms_output.put_line(v_name||' '||v_content);
end;
```

调用：(用指明形参名的方式调用更好)

```
begin
  say_hello();
end;
```

或

```
begin
  say_hello('cheng');
end;
```

或

```
begin
say_hello(v_name=>'cheng');
end;
```

****

PLSQL 中的 function

****

FUNCTION 和 PROCEDURE 的区别

1、函数有返回值，过程没有

2、函数调用在一个表达式中，过程则是作为 pl/sql 程序的一个语句

过程和函数都以编译后的形式存放在数据库中，函数可以没有参数也可以有多个参数并有一个返回值。过程有零个或多个参数，没有返回值。函数和过程都可以通过参数列表接收或返回零个或多个值，函数和过程的主要区别不在于返回值，而在于他们的调用方式，过程是作为一个独立执行语句调用的，函数以合法的表达式的方式调用

```
create or replace function func(v_name in varchar2)
return varchar2
is
begin
  return(v_name||' hello');
end;
```

调用：

```
declare
  v_name varchar2(20);
begin
  v_name:=func('cheng');
  dbms_output.put_line(v_name);
end;
```

带 out 参数的函数

```
create or replace function func(
  v_name in varchar2,
  v_content out varchar2
)
return varchar2
is
begin
  v_content:=v_name||' hello';
  return v_content;
end;
```

调用：

```
declare
  v_name varchar2(20);
  v_name1 varchar2(20);
begin
  v_name1:=func('susu',v_name);--返回 v_name 值
  dbms_output.put_line(v_name1);--打印 func 结果
  dbms_output.put_line(v_name);--打印 v_name 结果
end;
```

带 in out 参数的函数

```
create or replace function func(
  v_name in out varchar2)
return varchar2
is
begin
  v_name:=v_name||' hello';
```

```
  return 'cheng';
end;
```

调用：

```
declare
  v_inout varchar2(20):='world';
  v_ret varchar2(20);
begin
  v_ret:=func(v_inout);--返回调用 v_inout 值(作为出参)
  dbms_output.put_line(v_ret);--打印 func 结果
  dbms_output.put_line(v_inout);--返回 v_name 结果
end;
```

部分内容参考至 CSDN：http://download.csdn.net/source/405714

文章内容继续更新中..欢迎大家指点。

分析函数简述

注：N 表示数字型，C 表示字符型，D 表示日期型，[]表示内中参数可被忽略，fmt 表示格式。

分析函数计算基于 group by 的列，分组查询出的行被称为"比照(window)"，在根据 over()执行过程中，针对每一行都会重新定义比照。比照为"当前行(current row)"确定执行计算的行的范围。这点一定要理解清楚。它是分析函数生成数据的原理。如果此处模糊，那么你在应用分析函数时恐就不会那么得心应手了。

分析函数与前面章节中讲到的聚合函数非常相似，不同于聚合函数的地方在于它们每个分组序列均返回多行。在本节示例中会同时应用两种函数做对比，以更好体现二者的差异。通过本章节练习相信大家就会注意到，部分聚合函数和分析函数是同一个命令，事实确实如此。如果从**语法格式**上区分的话，没加 over()的即是聚合函数，加了 over()即是分析函数：）

有一点需要**注意**哟，除了 order by 子句的运算外，分析函数在 SQL 语句中将会最后执行。因此，分析函数只能应用于 select 的列或 order by 子句中(记住喽，千万别扔到什么 where、group by、having 之类的地方了)。也正因此，同名

的函数在做为聚合函数和分析函数时得出的结果可能不相同，就是因为此处**运算逻辑**不同造成的。

同时，部分分析函数在选择列时支持 distinct，如果你指定了该参数，则 over 条件中就只能指定 partition 子句，而不能再指定 order by 子句了。

分析函数的语法结构比较复杂，但多数函数都具有相同的语法结构，所以先在之前进行统一介绍，后续单个函数介绍时就不过多说明函数语法结构了。

基本上所有的分析函数均是这种格式：

函数名称 ([参数]) OVER (analytic_clause)

analytic_clause 包含：**[partition 子句][order 子句 [window 子句]]**

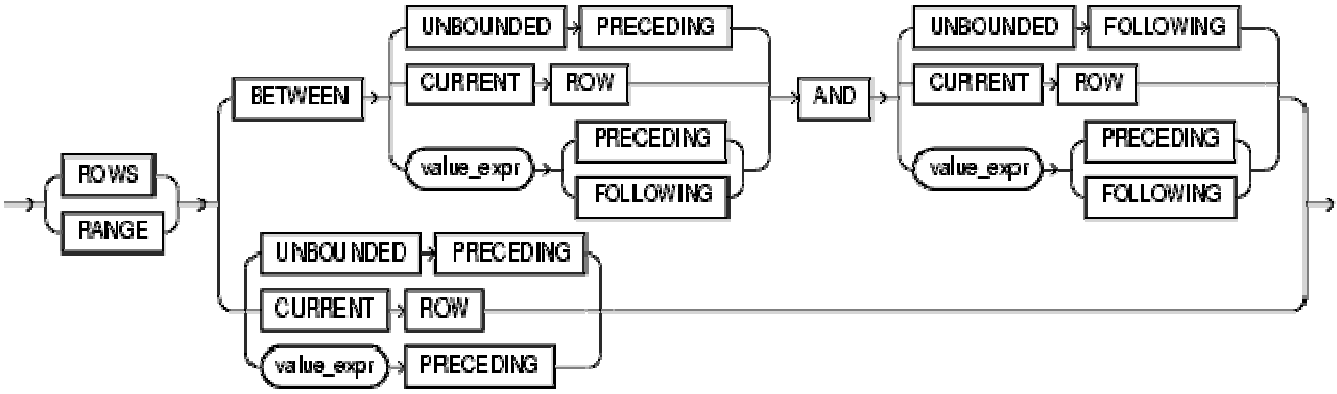
- Partition 子句：Partition by exp1[,exp2]...

  Partition 没啥说的，功能强大参数少，主要用于分组，可以理解成 select 中的 group by。不过它跟 select 语句后跟的 group by 子句并不冲突。

- Order 子句：Order by exp1[asc|desc] [,exp2 [asc|desc]]... [nulls first|last]。部分函数支持 window 子句。

  Order by 的参数基本与 select 中的 order by 相同。大家按那个理解就是了。Nulls first|last 是用来限定 nulls 在分组序列中的所在位置的，我们知道 oracle 中对于 null 的定义是未知，所以默认 ordery 的时候 nulls 总会被排在最前面。如果想控制值为 null 的列的话呢，nulls first|last 参数就能派上用场了。

- Window 子句：En,贴个图吧



  看起来复杂其实简单，而且应用的机率相当的低，不详细介绍了。

- AVG([DISTINCT|ALL] expr) OVER(analytic_clause) 计算平均值。

  例如：

```
--聚合函数
SELECT col, AVG(value) FROM tmp1 GROUP BY col ORDER BY col;
--分析函数
SELECT col, AVG(value) OVER(PARTITION BY col ORDER BY col)
  FROM tmp1
 ORDER BY col;
```

- SUM ([DISTINCT | ALL] expr) OVER (analytic_clause)

  例如：

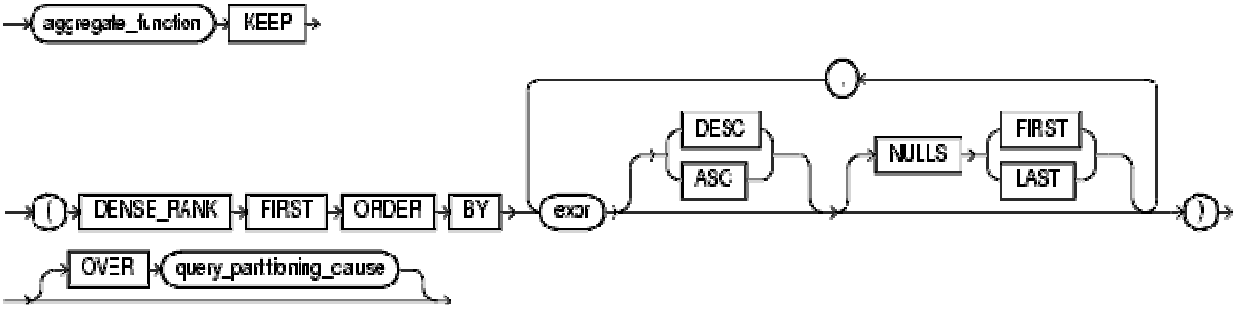
```
--聚合函数
SELECT col, sum(value) FROM tmp1 GROUP BY col ORDER BY col;
```

```
--分析函数
SELECT col, sum(value) OVER(PARTITION BY col ORDER BY col)
  FROM tmp1
 ORDER BY col;
```

- COUNT({* | [DISTINCT | ALL] expr}) OVER (analytic_clause) 查询分组序列中各组行数。
  例如：

```
--分组查询col的数量
SELECT col,count(0) over(partition by col order by col) ct FROM tmp1;
```

- FIRST() 从 DENSE_RANK 返回的集合中取出排在第一的行。



例如：

```
--聚合函数
SELECT col,
       MIN(value) KEEP(DENSE_RANK FIRST ORDER BY col) "Min Value",
       MAX(value) KEEP(DENSE_RANK LAST ORDER BY col) "Max Value"
  FROM tmp1
 GROUP BY col;
--分析函数
SELECT col,
       MIN(value) KEEP(DENSE_RANK FIRST ORDER BY col) OVER(PARTITION BY col),
       MAX(value) KEEP(DENSE_RANK LAST ORDER BY col) OVER(PARTITION BY col)
  FROM tmp1
 ORDER BY col;
```

可以看到二者结果基本相似，但是 ex1 的结果是 group by 后的列，而 ex2 则是每一行都有返回。

- LAST()与上同，不详述。
  例如：见上例。

- FIRST_VALUE (col) OVER (analytic_clause) 返回 over()条件查询出的第一条记录
  例如：

```
insert into tmp1 values ('test6','287');
SELECT col,
       FIRST_VALUE(value) over(partition by col order by value) "First",
       LAST_VALUE(value) over(partition by col order by value) "Last"
  FROM tmp1;
```

- LAST_VALUE (col) OVER (analytic_clause) 返回 over()条件查询出的最后一条记录

例如：见上例。

- LAG(col[,n][,n]) over([partition_clause] order_by_clause) lag 是一个相当有意思的函数，其功能是返回指定列 col 前 n1 行的值(如果前 n1 行已经超出比照范围，则返回 n2，如不指定 n2 则默认返回 null)，如不指定 n1，其默认值为 1。

例如：

```
SELECT col,
       value,
       LAG(value) over(order by value) "Lag",
       LEAD(value) over(order by value) "Lead"
  FROM tmp1;
```

- LEAD(col[,n][,n]) over([partition_clause] order_by_clause) 与上函数正好相反，本函数返回指定列 col 后 n1 行的值。

例如：见上例

- MAX (col) OVER (analytic_clause) 获取分组序列中的最大值。

例如：

```
--聚合函数
SELECT col,
       Max(value) "Max",
       Min(value) "Min"
  FROM tmp1
 GROUP BY col;
--分析函数
SELECT col,
       value,
       Max(value) over(partition by col order by value) "Max",
       Min(value) over(partition by col order by value) "Min"
  FROM tmp1;
```

- MIN (col) OVER (analytic_clause) 获取分组序列中的最小值。

例如：见上例。

- RANK() OVER([partition_clause] order_by_clause) 关于 RANK 和 DENSE_RANK 前面聚合函数处介绍过了，这里不废话不，大概直接看示例吧。

例如：

```
insert into tmp1 values ('test2',120);
SELECT col,
       value,
       RANK() OVER(order by value) "RANK",
       DENSE_RANK() OVER(order by value) "DENSE_RANK",
       ROW_NUMBER() OVER(order by value) "ROW_NUMBER"
  FROM tmp1;
```

- DENSE_RANK () OVER([partition_clause] order_by_clause)

例如：见上例。

- ROW_NUMBER () OVER([partition_clause] order_by_clause) 这个函数需要多说两句，通过上述的对比相信大家应该已经能够看出些端倪。前面讲过，dense_rank 在做排序时如果遇到列有重复值，则重复值所在行的序列值相同，而其后的序列值依旧递增，rank 则是重复值所在行的序列值相同，但其后的序列值从+重复行数开始递增，而 row_number 则不管是否有重复行，(分组内)序列值始终递增

  例如：见上例。

- CUME_DIST() OVER([partition_clause] order_by_clause) 返回该行在分组序列中的相对位置，返回值介于 0 到 1 之间。注意哟，如果 order by 的列是 desc，则该分组内最大的行返回列值 1，如果 order by 为 asc，则该分组内最小的行返回列值 1。

  例如：`SELECT col, value, CUME_DIST() OVER(ORDER BY value DESC) FROM tmp1;`

- NTILE(n) OVER([partition_clause] order_by_clause)

  ntile 是个很有意思的统计函数。它会按照你指定的组数(n)对记录做分组

  例如：`SELECT t.*,ntile(5) over(order by value desc) FROM tmp1 t;`

- PERCENT_RANK() OVER([partition_clause] order_by_clause) 与 CUME_DIST 类似，本函数返回分组序列中各行在分组序列的相对位置。其返回值也是介于 0 到 1 之间，不过其起始值始终为 0 而终结值始终为 1。

  例如：`SELECT col, value, PERCENT_RANK() OVER(ORDER BY value) FROM tmp1;`

- PERCENTILE_CONT(n) WITHIN GROUP (ORDER BY col [DESC|ASC]) OVER(partition_clause)

  本函数功能与前面聚合函数处介绍的完全相同，只是一个是聚合函数，一个是分析函数。

  例如：

```
--聚合函数
SELECT col, max(value), min(value), sum(value),
      PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY value) a,
      PERCENTILE_CONT(0.8) WITHIN GROUP(ORDER BY value) b
  FROM TMP1
 group by col;
--分析函数
SELECT col,
      value,
      sum(value) over(partition by col) "Sum",
      PERCENTILE_CONT(0.5) WITHIN GROUP( ORDER BY value) OVER(PARTITION BY col)
"CONTa",
      PERCENTILE_CONT(0.8) WITHIN GROUP( ORDER BY value) OVER(PARTITION BY col)
"CONTb"
  FROM TMP1;
```

- PERCENTILE_DISC(n) WITHIN GROUP (ORDER BY col [DESC|ASC]) OVER(partition_clause)

  本函数功能与前面聚合函数处介绍的完全相同，只是一个是聚合函数，一个是分析函数。

  例如：

```
--聚合函数
SELECT col, max(value), min(value), sum(value),
      PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY value) a,
```

```
       PERCENTILE_DISC(0.8) WITHIN GROUP(ORDER BY value) b
  FROM TMP1
 group by col;
--分析函数
SELECT col,
       value,
       sum(value) over(partition by col) "Sum",
       PERCENTILE_DISC(0.5) WITHIN GROUP( ORDER BY value) OVER(PARTITION BY col)
"CONTa",
       PERCENTILE_DISC(0.8) WITHIN GROUP( ORDER BY value) OVER(PARTITION BY col)
"CONTb"
  FROM TMP1;
```

- RATIO_TO_REPORT(col) over ([partition_clause]) 本函数计算本行 col 列值在该分组序列 sum(col)中所占比率。如果 col 列为空，则返回空值。

  例如：

```
SELECT col, value,
       RATIO_TO_REPORT(value) OVER(PARTITION BY col) "RATIO_TO_REPORT"
  FROM TMP1
```

- STDDEV ([distinct|all] col) OVER (analytic_clause) 返回列的标准偏差。

  例如：

```
--聚合函数
SELECT col, STDDEV(value) FROM TMP1 GROUP BY col;
--分析函数
SELECT col, value,
       STDDEV(value) OVER(PARTITION BY col ORDER BY value) "STDDEV"
  FROM TMP1;
```

- STDDEV_SAMP(col) OVER (analytic_clause) 功能与上相同，与 STDDEV 不同地方在于如果该分组序列只有一行的话，则 STDDEV_SAMP 函数返回空值，而 STDDEV 则返回 0。

  例如：

```
--聚合函数
SELECT col, STDDEV(value),STDDEV_SAMP(value) FROM TMP1 GROUP BY col;
--分析函数
SELECT col, value,
       STDDEV(value) OVER(PARTITION BY col ORDER BY value) "STDDEV",
       STDDEV_SAMP(value) OVER(PARTITION BY col ORDER BY value) "STDDEV_SAMP"
  FROM TMP1;
```

- STDDEV_POP(col) OVER (analytic_clause) 返回该分组序列总体标准偏差

  例如：

```
--聚合函数
SELECT col, STDDEV_POP(value) FROM TMP1 GROUP BY col;
--分析函数
SELECT col, value,
```

```
       STDDEV_POP(value) OVER(PARTITION BY col ORDER BY value) "STDDEV_POP"
  FROM TMP1;
```

- VAR_POP(col) OVER (analytic_clause) 返回分组序列的总体方差，VAR_POP 进行如下计算：(SUM(expr2) - SUM(expr)2 / COUNT(expr)) / COUNT(expr)

  例如：

```
--聚合函数
SELECT col, VAR_POP(value) FROM TMP1 GROUP BY col;
--分析函数
SELECT col, value,
       VAR_POP(value) OVER(PARTITION BY col ORDER BY value) "VAR_POP"
  FROM TMP1;
```

- VAR_SAMP(col) OVER (analytic_clause) 与上类似，该函数返回分组序列的样本方差，，其计算公式为：(SUM(expr2) - SUM(expr)2 / COUNT(expr)) / (COUNT(expr) - 1)

  例如：

```
--聚合函数
SELECT col, VAR_SAMP(value) FROM TMP1 GROUP BY col;
--分析函数
SELECT col, value,
       VAR_SAMP(value) OVER(PARTITION BY col ORDER BY value) "VAR_SAMP"
  FROM TMP1;
```

- VARIANCE(col) OVER (analytic_clause) 该函数返回分组序列方差，Oracle 计算该变量如下：
  如果表达式中行数为 1，则返回 0，如果表达式中行数大于 1，则返回 VAR_SAMP

  例如：

```
--聚合函数
SELECT col, VAR_SAMP(value),VARIANCE(value) FROM TMP1 GROUP BY col;
--分析函数
SELECT col, value,
       VAR_SAMP(value) OVER(PARTITION BY col ORDER BY value) "VAR_SAMP",
       VARIANCE(value) OVER(PARTITION BY col ORDER BY value) "VARIANCE"
  FROM TMP1;
```