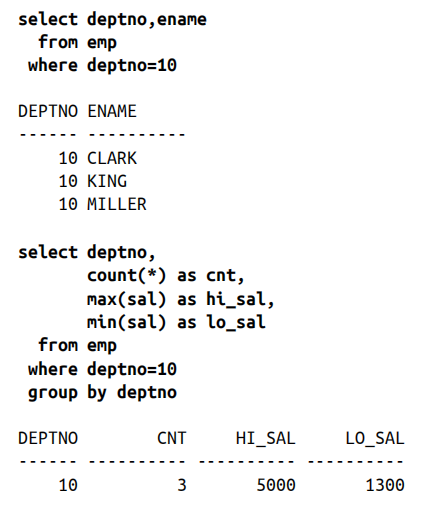
窗口函数的工作原理。总体而言，窗口函数的出现使得许多通常被认为非常棘手的任务变得较为容易了（指的是那些单纯使用标准SQL难以解决的问题）。

## 1、分组

开始讨论窗口函数之前，需要了解SQL的分组是如何工作的。很多人不了解GROUP BY子句的工作原理，也不明白为什么使用了GROUP BY之后查询结果会发生变化。

简单来说，分组就是把相似的行数据聚集在一起。如果一个查询用到了GROUP BY，那么结果集的每一行数据都是一个分组，它们分别代表一行或者几行的记录。并且，它们之所以会被分在同一组，是因为这些记录的某一列或者某几行的值相同。

如果说一个分组就是基于指定的某一列（或者几列）把具有相同值的行数据重新编排而成的一行具有代表性的新数据，那么对于EMP表而言，显而易见的分组示例包括部门编号等于10的全体员工（这些员工之所以被分到同一组，是因为他们的DEPTNO等于10），以及全体文员（这些员工的JOB=’CLERK’）。来看一些查询语句。下列第一个查询给出了部门编号等于10的所有员工；第二个查询把部门编号等于10的员工分组，并返回一些额外的分组信息：一共有多少行（成员的个数），最高工资以及最低工资。



如果SQL不支持把部门编号等于10的员工分为一组，那么我们就必须手动检查原始数据才能获得上述第二个查询返回的那些信息。（如果仅有3行记录，手动做并不困难；但是，如果有300万行记录，又该如何处理呢？）这就引出了另外一个问题：我们为什么希望对数据进行分组？有多种可能的原因：或许我们想知道有多少个不同的分组，也或许我们想计算每个分组包含多少个成员（行）。正如上述查询语句所示，借助SQL的分组功能，即使一个表里有许多行，我们也能快速获取关于它们的信息，而不必逐行检查原始数据。

## 2、SQL分组的定义

一般而言，数学上的“群”（group）定义为（G,·,e），其中G是一个集合，·表示G的二进制运算，而e则是G中的成员。我们以该定义为基础来说明什么是SQL分组。一个SQL分组被定义为（G,e），其中G是一个带有GROUP BY子句的、单一的或自足的（self-contained）查询语句的结果集，e是G的成员。并且，一个SQL分组须满足下满的两个定理。

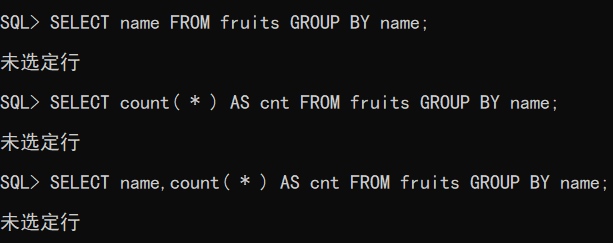
* 对于G的每一个成员e，e具有唯一性，并且存在一个或多个e的实例。
* 对于G的每一个成员e，聚合函数COUNT的返回值大于0。

以上SQL分组定义里的结果集G表明了一个事实，即SQL分组的概念依存于SQL查询，没有SQL查询就不会有SQL分组。因此，对于上述定理的表述内容，我们完全可以把e替换成“行”，因为SQL分组在技术指的就是由行数据构成的结果集。

上述两个定理是下面讨论SQL分组的理论基础，因此你有必要先证明它们的正确性（并且在证明的过程中，我们会像前面一样引入一些具体的SQL查询）。

### 2.1分组不为空

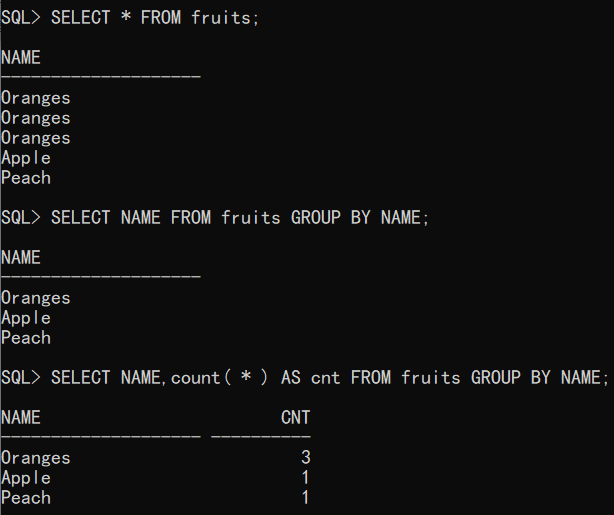
根据上文中的定义，一个分组至少要拥有一个成员（行）。如果我们承认这一点，那么也可以推论出：无法从一个空表生成分组。为证明这个命题的正确性，可以采取反证法，尝试证明它是错误的。下面给出的例子先创建了一个空表，然后尝试通过3个不同的查询语句基于该表生成分组：



上述查询结果说明，我们确实无法从一个空表中生成任何分组。

### 2.2分组具有唯一性

现在来证明GROUP BY子句生成的分组具有唯一性。先向FRUITS表插入5行记录，然后执行一些查询语句生成分组，如下：



上述第一个查询显示Oranges在FRUITS表里出现了3次。然而，上述第2个和第3个查询（它们都使用了GROUP BY）只返回了一行Oranges。这表明GROUP BY结果集里的每一行（即G中包含的e，参见前面的定义）都是唯一的，上述分组的每一个NAME值都对应FRUITS表的一行或多行记录。

分组具有唯一性，这一点很重要。换句话说，如果查询语句使用了GROUP BY子句，那么通常而言SELECT列表里就不再需要使用DISTINCT关键字了。

并不是在说GROUP BY等价于DISTINCT。其实它们是两个完全不同的概念。在

结果集里GROUP BY子句后面指定的那些字段会先删除重复项，因而没有必要

同时使用DISTINCT和GROUP BY。

Frege的抽象公理和罗素：

Frege提出的抽象公理，它以Cantor的素朴集合论为基础，它指出对于任何一个明确指定的性质，都存在这样一个集合，该集合由所有满足该性质的个体构成。正如Robert Stoll曾经支持，该公理的问题在于“对抽象原则的无限制使用”。罗素提醒Frege考虑这样一种状况：如果有一个集合，它的成员也是一些集合，并且规定这些集合的性质为“不属于自身”。

正如罗素指出的，Frege的抽象公理留下了太多自由发挥的空间，如果我们单凭某个条件或性质来定义集合成员关系，那就难免会出现矛盾的状况。为了更形象地解释这个矛盾，罗素提出了“理发师悖论”（Barber Puzzle）。理发师悖论是这样说的：“小镇上有位理发师，他为所有不给自己刮胡子的男人刮胡子，并且值为这些男人刮胡子。可是如此一来，理发师本人的胡子谁来刮呢？”

我们来看一个具体的例子，考虑这样的一个集合：y的每一个成员x，都满足指定条件P，对应的数学符号表示是{xεy | P(x)}。

既然上述集合规定“y的每一个成员x，都满足指定条件P”，那么换一种更直观的说法就是，“当且仅当x满足指定条件P时，x是y的成员。”

现在我们规定，条件P(x)的定义是“x不是x的成员”，即(x~ε x)。这样一来，上面的表述就变成了“当且仅当x不是x的成员时，x是y的成员”，即{x

ε y | ( x ~ε x )}。

此时我们不妨问自己一个问题：上述集合是不是它自身的一个成员？在此我们假设x=y，并再次推演一下上述集合的数学表示。如下所示的集合可以被定义为“当且仅当y不是y的成员时，y是y的成员”，即{y ε y | ( y ~ε y )}。

简而言之，罗素悖论把我们推入了一种两难的境地，有一个集合同时满足两个相互矛盾的条件：它属于其自身，同时又不属于其自身呢？毕竟，由所有书籍构成的集合并不会是一本书。那么，为什么该悖论会被提出来，并作为一个悖论而存在呢？这是因为当我们站在更高的抽象层面考虑集合理论时，该悖论的存在价值就凸显出来了。例如，罗素悖论的一个相对具有实用价值的运用场景是，探讨“所有集合的集合”问题。对于“所有集合的集合”这个概念，根据其定义，它必须是其自身的一个成员（因为它是所有集合的集合）。那么，如果把P(x)加于“所有集合的集合”之上，又会如何呢？简单来说，我们参照罗素悖论会推导出“当且仅当所有集合的集合不是其自身的一个成员时，它是其自身的一个成员”，显然这是一个矛盾。

Ernst Zermelo后来提出了分离公理模式（也被称作受限概括公理模式或者分类公理），完美地规避了素朴集合论的罗素悖论问题。

### 2.3COUNT永远大于0

前文中的查询语句及其结果也证明了最后一个定理的正确性，即针对一个非空的表执行GROUP BY查询，那么聚合函数COUNT的返回值不会等于0。这一点没什么可奇怪的，针对一个分组执行COUNT查询确实不可能返回0。我们已经证明了，无法从一个空表里生成分组，因此一个分组至少会含有一行数据。既然至少有1行数据，那么COUNT查询的结果自然至少等于1。

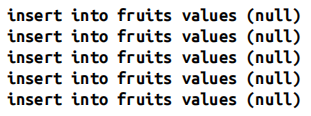
注意：以上我们讨论的是同时使用COUNT和GROUP BY的状况，这和只使用

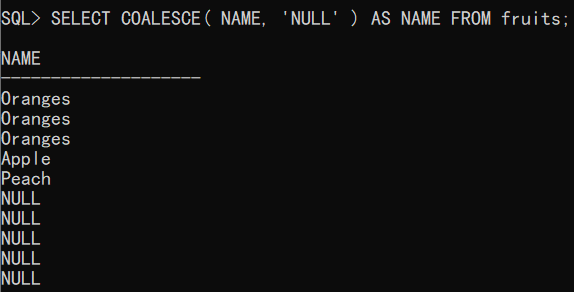
COUNT的状况有所不同。如果不要GROUP BY子句，针对一个空表执行COUNT查询当然会得到0。

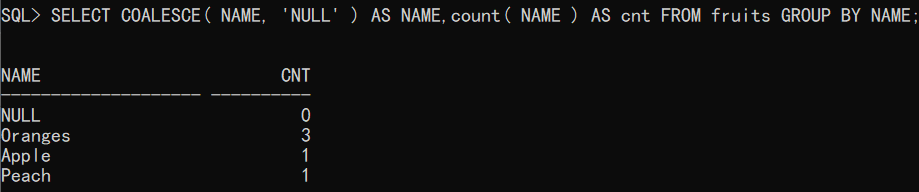
## 3、悖论

悖论通常会针对已有的理论或观点构造出若干矛盾的场景。许多时候，这些矛盾的影响可以被限制在一个较小的范围之内，并且能够通过一些“变通手段”消除其影响。或者，如果它们仅对少数测试案例成立，这样多数时候我们都能放心地忽略掉它们。

前面给出的SQL分组定义确实存在一个悖论，必须要认真讨论。尽管一直在讨论分组问题，但最终仍然要回到SQL查询上。GROUP BY后面可以接多种语法元素：最常出现的是表的某一列或者几列；除此之外，还可能出现常量和表达式。我们由此获得了很好的灵活性，但也被迫要付出代价，因为在SQL中的NULL也是一种合法的“值”。NULL的问题在于它会被聚合函数自动忽略掉。比如，如果一个表只有一行一列，并且它的值是NULL，那么如果我们针对该表执行一个既包含GROUP BY又包含聚合函数COUNT的查询会得到什么结果呢？根据前面的定义，如果既有GROUP BY，又有聚合函数COUNT，则返回值必然会大于等于1。那么，在函数COUNT忽略掉NULL值的情况下，结果会如何呢？这种状况又会对SQL分组定义带来什么样的影响呢？我们来看看下面的例子，它展示了上述NULL分组悖论。（为增强查询结果的可读性，这里用到了COALESCE函数）







如上图所示，正是由于上述NULL值的出现，我们的SQL分组定义现在被迫要面对一个矛盾的场景，即“悖论”。所幸这个矛盾并不是什么大问题，因为它更多地肇始于聚合函数的实现方式，而且和前面给出的定义本身关系不大。对于上述最后面的那个查询语句，如果用自然语言表述问题的话应该是这样的：

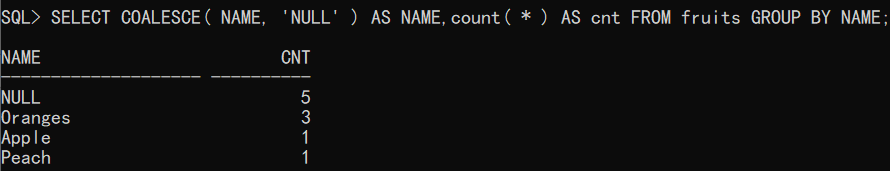
“计算FRUITS表里每一种水果出现的次数，即统计每一个分组对应多少给成员。”

回头再看一下上面的哪些INSERT语句，很明显我们一共插入了5行NULL值，也就是说NULL分组包含5个成员。

不同于通常类型的值，NULL有其特殊的性质，它不会等于任何值，但是可以作

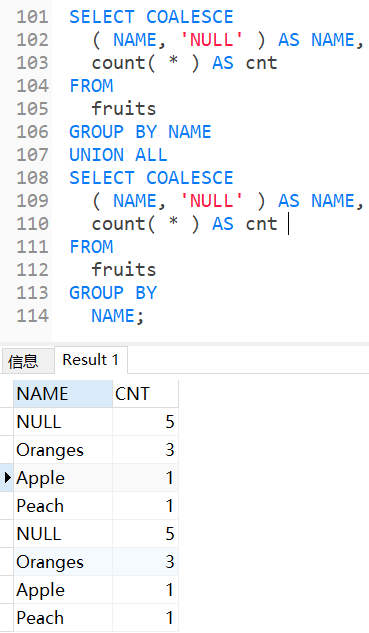
为分组成员而存在。

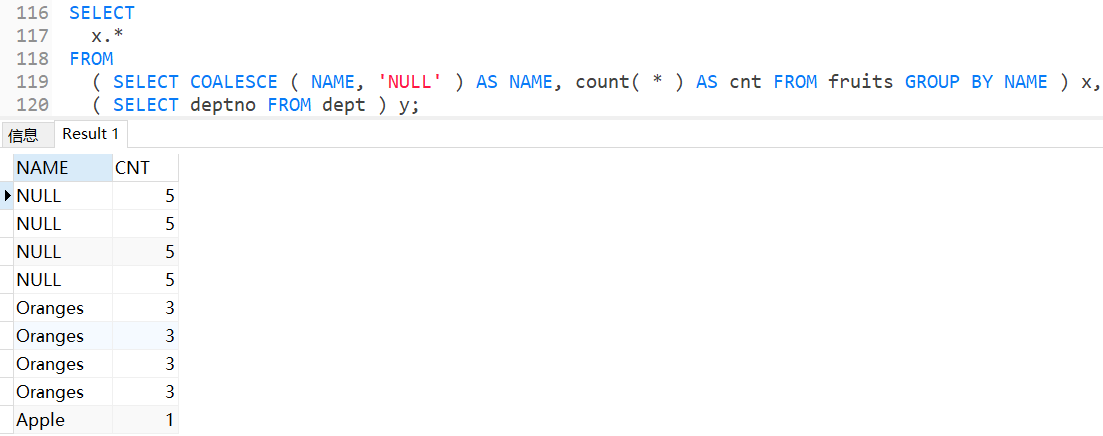
查询语句要怎么写才能不返回0返回5，既能返回我们需要的值，又不违背SQL分组的定义呢？如下所示的例子使用了一种变通的做法，消除NULL分组悖论的影响。



如果不使用COUNT(NAME)而改用COUNT(\*)的话，就能规避掉NULL分组的悖论。如果传递具体的列名做参数，则聚合函数会忽略掉NULL值。但是，传递星号\*而不是列名给COUNT函数，返回值就不会等于0了。星号\*迫使COUNT函数去计算行数，而不是实际的列值，这样一来不论具体的列值是不是NULL，都不会影响最终的返回值。

另一个悖论针对的是定理“结果集中每一个分组（即G中包含的每一个e）具有唯一性”。从本质上来说，SQL结果集和表其实不是集合（因为允许出现重复行），更准确的定义应该是多重集合（multiset或bag），因此我们也可能生成包含重复分组的结果集。考虑如下：





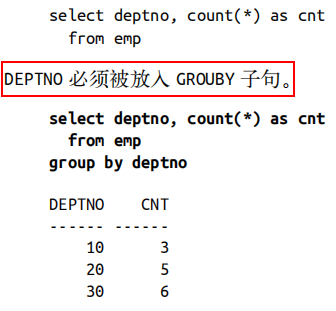
上述所示的最终结果中都出现了重复的分组。分组的第一个性质是，对于(G,e)而言，G是一个带有GROUPBY子句的、单一的或自足的查询语句的结果集。简单地说，任何GROUP BY查询返回的结果集都符合分组的定义。只有把两个GROUP BY查询的结果合并起来的时候，才可能生成含有重复分组的多重集合。上述第一个示例使用了UNION ALL，这不是一个集合运算，而是一个多重集合运算，实际上它是两个查询，分两次执行 GROUP BY 操作。

UNION是一种集合运算，使用UNION就不会出现重复分组了。

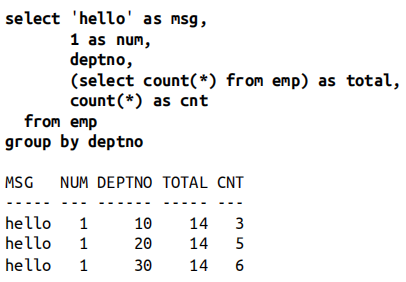
上述第二个查询使用了笛卡尔积，它首先生成分组，然后再执行笛卡尔积。从一个自足的GROUP BY查询的角度来看，它实际上不违背我们的SQL分组定义。因此，上述两个例子并没有丝毫撼动我们给出的SQL分组定义。相反，它们证明了该定义的完备性，因此我们不会再有所怀疑了。

## 4、SELECT和GROUP BY的关系

前文给出了SQL分组定义并给出证明，现在可以讨论更多关于GROUP BY查询的实际问题。对于SQL分组，理解SELECT子句和GROUP BY子句之间的关系很重要。如果我们要使用诸如COUNT这样的聚合函数，就必须记住SELECT列表里的任何字段，只要它不是聚合函数的参数，那么它必须成为分组的一部分。比如，对于如下所示的查询语句。



上述规则并非绝对，也有一些例外情况。例如，常量、用户自定义函数的标量返回值、窗口函数以及非相关标量子查询（non-correlated scalar sybquery）。SELECT子句会在GROUP BY子句之后被执行，因而这些语法元素可以出现在SELECT列表中，而不必（有些状况下也不允许）出现在GROUP BY子句里。例如，



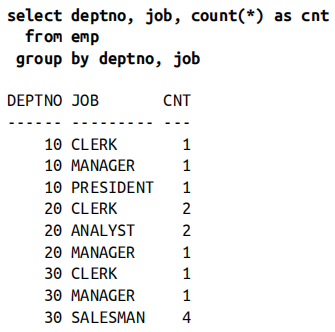
上述的查询可能会让你感到困惑。SELECT列表里那些额外的项目虽然没有被放入GROUP BY子句，但是每个DEPTNO对应的CNT值并不会因此而改变，DEPTNO的值也不会被改变。那么，我们不妨根据查询结果修改一下规则，使其表述更加准确。

如果SELECT列表里的项目可能会改变分组或者改变聚合函数的返回值，则该项

目必须被放入GROUP BY子句。

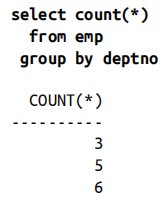
上述查询的SELECT列表虽然额外添加了一些项目，但是它既不会改变任何分组（每个DEPTNO）的CNT值，也不会改变分组本身。

现在，可以问这样的问题了：到底SELECT列表中哪些项目可能会改变分组或者聚合函数的返回值呢？答案很简单：我们想要检索的表的其他列。为上述查询添加一个JOB列。



为了把额外的JOB列也从EMP表中提取出来，分组的方式和查询结果集都要随之发生改变。因而我们必须把DEPTNO和JOB一起放入GROUP BY子句，否则上述查询语句不可能正常执行。在 SELECT 和 GROUP BY 子句里都引入 JOB 列，意味着整个查询的语义从“计算每个部门有多少位员工”变成了“计算每个部门有多少个不同的职位”。在这里我要再次提醒你注意，分组具有唯一性；尽管拆开来看的话，DEPTNO 和 JOB 的值有重复，但是每一个 DEPTNO和 JOB 组合（它们同时出现在 GROUP BY 子句和 SELECT 列表里，因此属于同一个分组）都是独一无二的。（例如，10 和 CLERK 只出现过一次。）

也可以在 SELECT 列表里只放聚合函数，这样 GROUP BY 子句里就可以放任何列了。看一下如下所示的两个查询语句，



并不是一定要为SELECT列表添加一些非聚合函数，但为了提高查询结果的可读性和可用性，我们通常会这么做。

作为一项规定，同时使用GROUP BY和聚合函数的时候，对于SELECT列表里

的任何项目而言，只要它不是聚合函数的参数，那么它就必须被放入GROUP

BY子句。

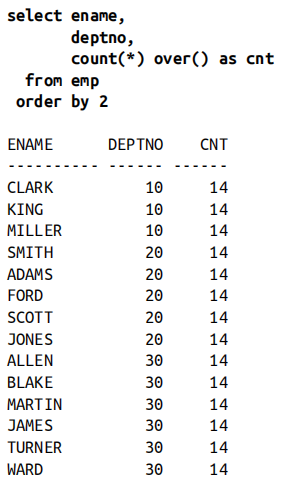
## 5、窗口操作

一旦充分理解了分组的概念并掌握了SQL聚合运算，就比较容易理解窗口函数了。就像聚合函数一样，窗口函数针对指定的行集合（分组）执行聚合运算。不同之处在于，窗口函数能够为每个分组返回多个值，而聚合函数只能返回单一值。聚合运算的对象其实是一组行记录，我们称之为“窗口”。DB2把这一类函数称为“OLAP函数”（在线分析处理函数），Oracle则称之为“分析函数”（Analytic Function）；本书遵循ISO SQL标准的叫法，统一使用术语“窗口函数”。

一个简单的例子：假设我们希望计算整个公司的员工人数，传统的做法是针对EMP表调用COUNT(\*)。

SELECT COUTN(\*) AS cnt FROM emp；

但有时候我们可能需要从非聚合数据行或者从不同纬度的聚合数据行里访问这一类聚合运算结果。窗口函数能帮助我们轻松完成这一类操作。例如：下面的查询语句展示了如何使用窗口函数同时检索出明细行（每个员工一行）和聚合运算结果（员工总人数）。



上述事例调用了窗口函数COUNT(\*) OVER()。关键字OVER表明COUNT函数会作为窗口函数来调用，而不是一次普通的聚合函数调用。基本上，SQL标准中列出的全部聚合函数都能用作窗口函数，关键字OVER的作用是帮助语法解析器区分不同的使用场景。

那么，COUNT(\*) OVER()到底做了什么操作呢？它为上述的查询语句返回的每一行数据提供了额外的一列，该列表示EMP表一个有多少行记录。在这里关键字OVER后面的圆括号是空的，其实我们也可以在里面放上一个额外的子句，以指明窗口函数操作的行记录范围。保持圆括号里什么也没有，这是明确告知窗口函数把全体行记录作为操作对象，因此上述每一行输出的结果都是14。

窗口函数的方便之处在于它可以一行之中同时执行多种不同维度的聚合运算。

## 6、执行时机

在继续深入探讨OVER子句之前，有必要先理清一个重要问题，即窗口函数的执行会被安排在整个SQL处理的最后一步，但会先于ORDER BY子句执行。下面举例说明创建函数的执行时间，这里为前一节的查询语句加上一个WHERE子句，以过滤掉DEPTNO等于20和30的员工。

select ename,

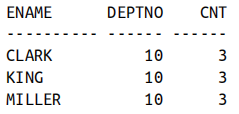
deptno,

count(\*) over() as cnt

from emp

where deptno = 10

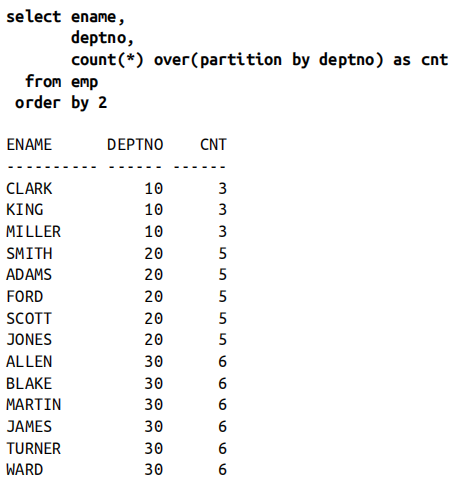
order by 2



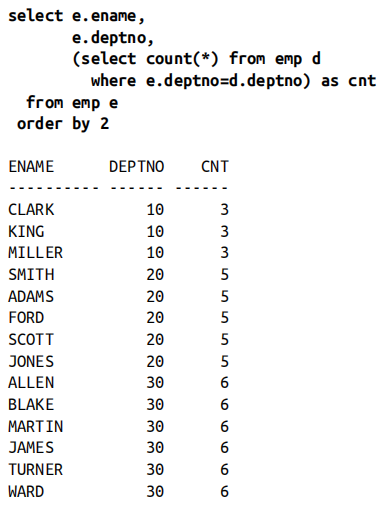
每一行的CNT值不再是14，而是变成3,。上述示例中，正是因为WHERE子句先行把结果集限制为3行，才导致窗口函数的返回值变成了3。（当SQL处理到SELECT子句的时候，就剩下3行数据留给窗口函数了。）该示例表明WHERE和GROUP BY这一类子句执行完之后，才轮到窗口函数执行。

## 7、分区

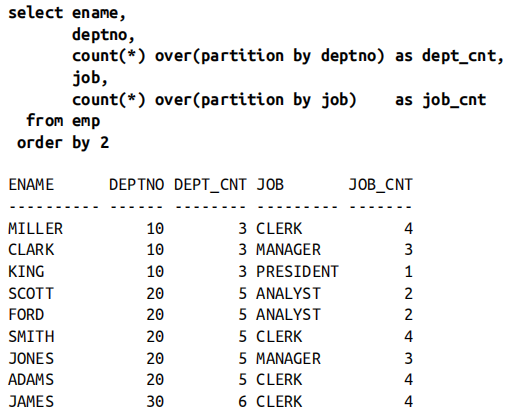
可以使用PARTITION BY子句针对行数据进行分区（patition）或者分组（group），并根据其结果执行聚合运算。如果OVER关键字后面跟着一个空的圆括号，那么窗口函数执行聚合运算时，会把该查询结果集整体作为一个分区来看待。因此，我们不妨把PARTITION BY子句理解成“动态的GROUP BY”，它不同于传统的GROUP BY，因为在最终的结果集中允许出现多种由PARTITION BY生成的分区。借助PARTITION BY针对指定的行数据进行分区和聚合运算（新的分区出现时，聚合运算结果会被重新计算），则所有值（每一个分区的每一个值）都会被返回，而不会只返回一组具有代表性的行记录。看一下如下所示的查询语句。

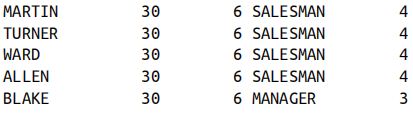


上述查询仍然会返回EMP表的全部14行记录，但是由于使用了PSRTITION BY DEPTNO子句，现在聚合函数会分别计算出每一个部门的员工人数。只有当部门发生变化时，聚合运算的结果才会被重新计算，因此同一个部门（同一个分区）的员工会得到相同的CNT值。同时我们也注意到，每一个分区的信息都会被返回，每一个分区的所有成员也都会被返回。和下面的这个语句相比，上述事宜窗口函数的查询更为高效。



相较于传统的GROUP BY，PSRTITION BY子句的另一个好处是，在同一个SELECT语句里我们可以按照不同的列进行分区，而且不同的窗口函数调用之间互不影响。看一下如下所示的查询，它会逐一列出全体员工并返回每一个人所属的部门，所在部门的员工总数，每一个人的职位，以及公司范围内从事相同工作的员工总数。



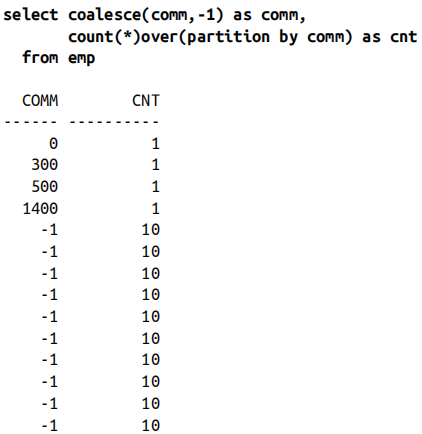


从上述结果集中可以看到，同一个部门的员工会对应相同的DEPT\_CNT值，从事相同工作的员工也都会得到同样的JOB\_CNT值。

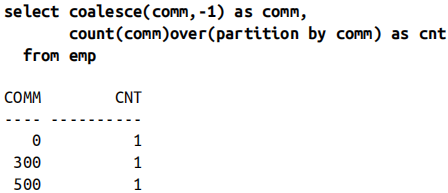
到目前位置，已经讨论完了PARTITION BY子句的工作原理，它与GROUP BY子句类似，但它不会影响SELECT子句的其他项目，也不要求我们写一个GROUP BY子句。

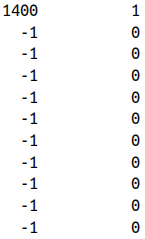
## 8、NULL的影响

类似于GROUP BY子句，PARTITION BY子句会把所有的NULL归入一个分区或者分组。因此，PARTITION BY对NULL值的影响也类似于GROUP BY。下面的查询调用了一个窗口函数来计算每一种业务提成对应的员工人数。（为增强查询结果的可读性，当业务提成为NULL时返回-1。）



上述查询使用了COUNT(\*)，因而返回值是相应的记录行数。我们看到有10个员工的业务提成为NULL。然而，如果不用\*，而改用COMM列的话，查询结果就大相径庭了。





由于上述查询使用了COUNT(COMM)，它只会技术不为NULL的COMM值。我们看到业务提成等于0的员工有1个，等于300的有1个，等等。但是要特别注意业务提成为NULL的人数查询结果是0，为什么呢？因为聚合函数会忽略掉NULL值。更准确地说，聚合函数仅仅计数非NULL值。

当使用COUNT函数时，我们应该思考一下是否要把NULL包括在内。使用COUNT(column)会忽略NULL。如果希望吧NULL值一并计入，则应该使用COUNT(\*)。

## 9、排序

使用窗口函数的时候，数据的排序方式可能会对最终的查询结果产生实质性的影响。因此，窗口函数的OVER子句也支持ORDER BY语法。可以使用ORDER BY子句指定分区内的行数据如何排序。（记住，如果OVER关键字后面没有出现PARTITION BY子句，则此处的“分区”指的就是整个查询结果集。）

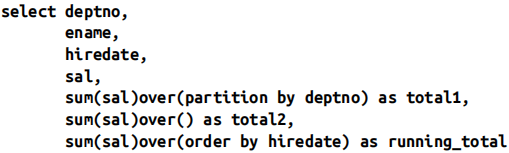
部分窗口函数强制要求对涉及的分区数据做排序。因此，对于这部分窗口函数而

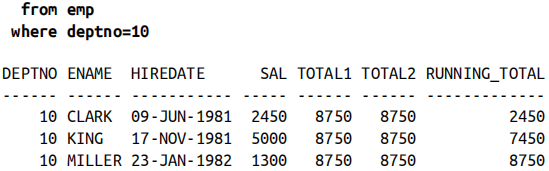
言，ORDER BY不可省略。

当在窗口函数的OVER子句中使用ORDER BY时，我们实际上是在决定两件事。

1. 分区内的行数据如何排序；
2. 计算设计哪些行数据。

如下所示查询，该查询计算出了DEPTNO等于10的员工的工资累计合计值。



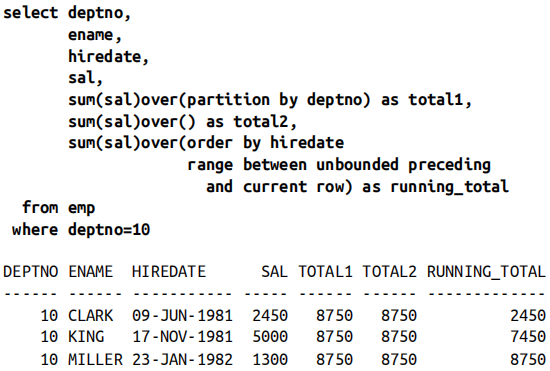


注意，其中一个SUM查询后面的圆括号是空的。那么，为什么TOTAL1和TOTAL2

查询结果相同呢？这里要再次说明一下，是窗口函数执行时机决定了上述结果。经过WHERE子句的过滤，合计的对象就只剩下DEPTNO等于10的工资了。在上述事例中只存在一个分区，即整个结果集只包括DEPTNO等于10的记录。因此，TOTAL1和TOTAL2具有相等的值。

只要看一下SAL列的值，就能理解RUNNING\_TOTAL值是如何计算出来的了。我们还可以逐一累加各个工资值，用实际计算结果验证上述累计合计值的查询结果。但是还有更重要的一点，为什么在计算累计合计值的时候要为OVER子句加上ORDER BY？这是因为在OVER子句加上ORDER BY之后，尽管我们看不到，实际上却是在分区内部指定了一个默认的“滑动窗口”。正是由于ORDER BY HIREDATE子句的存在才使得合计运算能够自动终止当前行。

如下所示的查询与前面那个查询等价，它使用了RANGE BETWEEN子句显式地指定了ORDER BY HIREDATE的默认行为方式。



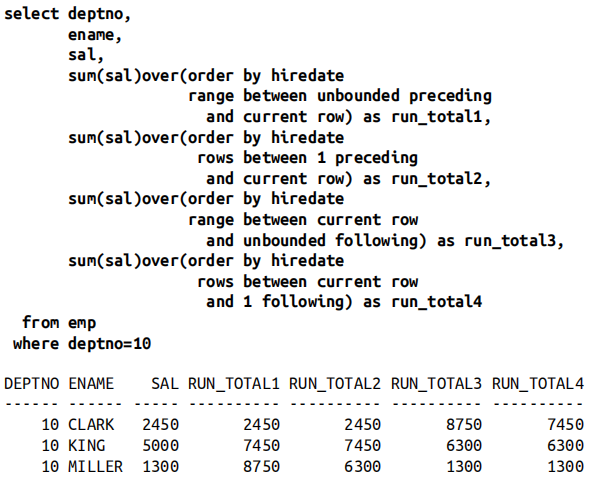
上述查询中出现的RANGE BETWEEN子句在ANSI标准中被称作Framing子句。现在，你应该已经清楚我们为什么要通过在OVER子句中指定ORDER BY来计算累计合计值。我们想要告诉查询语句要针对从第一行开始直至当前行为止的全部行记录执行合计运算。（这也是默认行为，ORDER BY决定“哪些行排在当前行的前面”，就本例而言是按照HIREDATE排序。）

## 10、Framing子句（RANGE BETWEEN）

以前面查询的结果集为例，具体解释一下Framing子句的工作原理，先从最早入职的员工CLARK开始。

1. 首先CLARK的工资是2450，我们应该把入职时间早于CLARK的所有员工的工资一并找出来相加求和。但是，CLARK是DEPTNO等于10的部门中最早入职的员工，因此这里只要针对CLARK的工资求和即可，这就是上述查询结果中第一行RUNNING\_TOTAL值的由来。
2. 按照HIREDATE排序的话，下一个员工是KING，我们看一下Framing子句针对他做了哪些操作。合计运算从当前行的公司5000（KING的工资）开始，在此之前的行数据也要包括在内。入职时间比KING早的员工只有CLARK，因而合计值就是5000+2450=7450。这就是上述查询结果中第二行RUNNING\_TOTAL值的由来。
3. 按照HIREDATE排序的话，分区中最后一个员工是MILLER。SAL列的合计运算从当前行的工资1300开始，在此之前的行数据也要包括在内。CLARK和KING都比MILLER入职时间早，因此MILLER对应的RUNNING\_TOTAL要把他们两个人的工资也计入在内：2450+5000+1300=8750。这就是上述查询结果中第三行RUNNING\_TOTAL值的由来。

如上所述，正是Framing子句能定义动态变化的“数据子窗口”，并将其融入聚合运算。我们可以使用多种形式的语法指定数据子窗口。考虑如下所示的查询语句。



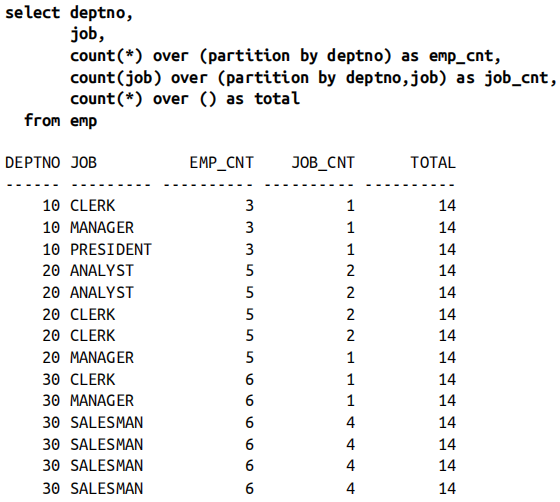
RUN\_TOTAL1就是前面已经解释过的Framing子句UNBOUNDED PRECEDING AND CURRENT ROW。下面我们解释一下其他几个Framing子句。

1. RUN\_TOTAL2：不同于关键字RANGE，此处的Framing子句使用了ROWS，该关键字表明将依据指定数据的行记录产生滑动窗口。1 PRECEDING表明起始行是当前行前面的那一行，因而对应的范围就是当前行以及排在它前面的那一行。因此，RUN\_TOTAL2的值就是当前员工工资加上一张HIREDATE排在他前面的那个员工的工资。
2. RUN\_TOTAL3：与RUN\_TOTAL1恰好相反，它的计算范围包括当前行以及排在它后面的所有行。
3. RUN\_TOTAL4：和RUN\_TOTAL2恰好相反，该累计合计值的计算范围包括当前行和排在它后面一行。

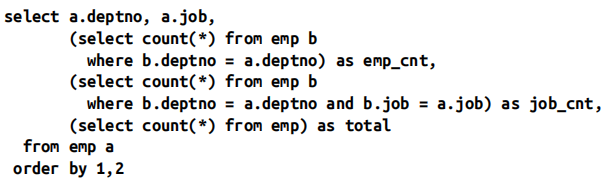
## 11、

## 12、代码可读性+性能=威力

窗口函数非常强大，使用它写出的查询语句能够把明细数据和聚合运算结果融为一体。相比于使用多个自连接和标量子查询，使用窗口函数的代码显得短小精悍。我们来看一下下面的查询语句，它很容易解决如下问题：“每个部门有多少名员工？每个部门有多少个不同职位的员工？（例如，DEPTNO等于10的部门有多少个文员。）EMP表中总共有多少名员工？”



如果不想使用窗口函数，那就要写一个等价而又稍显麻烦的查询语句。

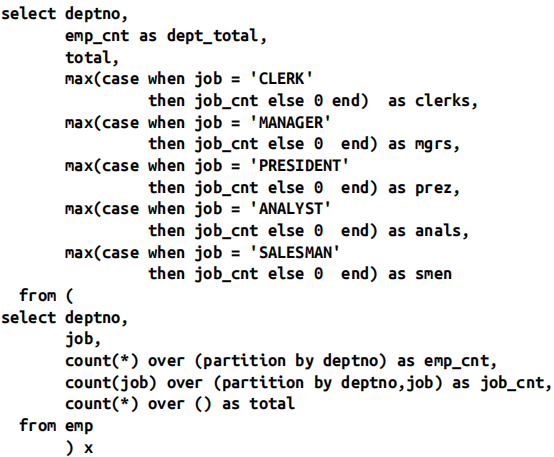


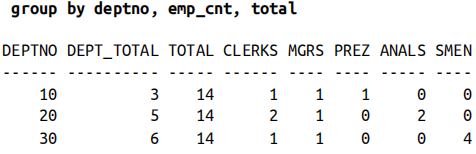
如果把数据规模扩大到1000行或者10000行的话，就应该能看到窗口函数的效率高于多个自连接和标量子查询。

## 13、为报表查询奠定基础

除了在代码可读性和性能方面的优势，窗口函数还为更加复杂的“报表风格”的查询奠定了基础。

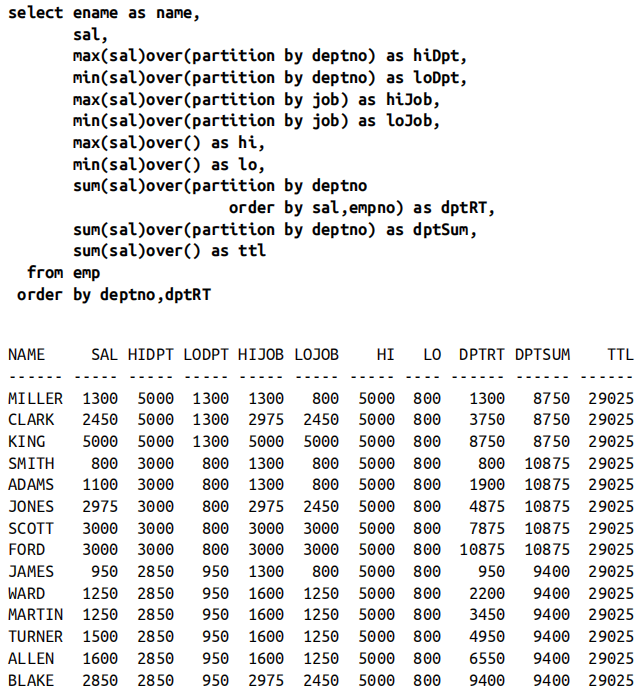
例如，我们来看一下下面这个“报表风格”的查询，它先在一个内嵌视图中使用了窗口函数，然后在外层查询中展开聚合运算。有了窗口函数，我们能同时得到明细数据和聚合运算结果，这一点对于生成报表非常有帮助。下面的这个查询使用窗口函数计算出了多种不同分区的计数值。由于聚合运算的作用对象是多行数据，内嵌视图将返回 EMP 表中所有行，因此外层的 CASE 表达式能够进一步做形式变换，并生成格式良好的报表。





上述查询返回了所有部门，每个部门中员工总人数，EMP表中员工总人数，以及每个部门中每一种职位各有多少个员工。更值得注意的是，只要一个查询就能完成所有这些工作，而且不需要用到任何连接查询或者临时表。

下面给出最后一个实例，它再次展示了使用窗口函数能够多么方便地同时给出多个问题的答案。



上述查询简单、高效地回答了下列几个问题，并且保持了很好的代码可读性（甚至没有使用额外的连接查询）。它只需要把员工机器工资和结果集中的相对应的行数据匹配起来即可。

1. 所有员工中谁的工资最高（HI）；
2. 在所有员工中谁的工资最低（LO）；
3. 在每个部门里谁的工资最高（HIDPT）；
4. 在每个部门里谁的工资最低（LODPT）；
5. 在每个工作种类里谁的工资最高（HIJOB）；
6. 在每个工作种类里谁的工资最低（LOJOB）；
7. 全部工资的合计值（TTL）；
8. 每个部门工资的合计值（DPTSUM）；
9. 每个部门的工资累计合计值（DPTRT）。