

O'REILLY®

TURING

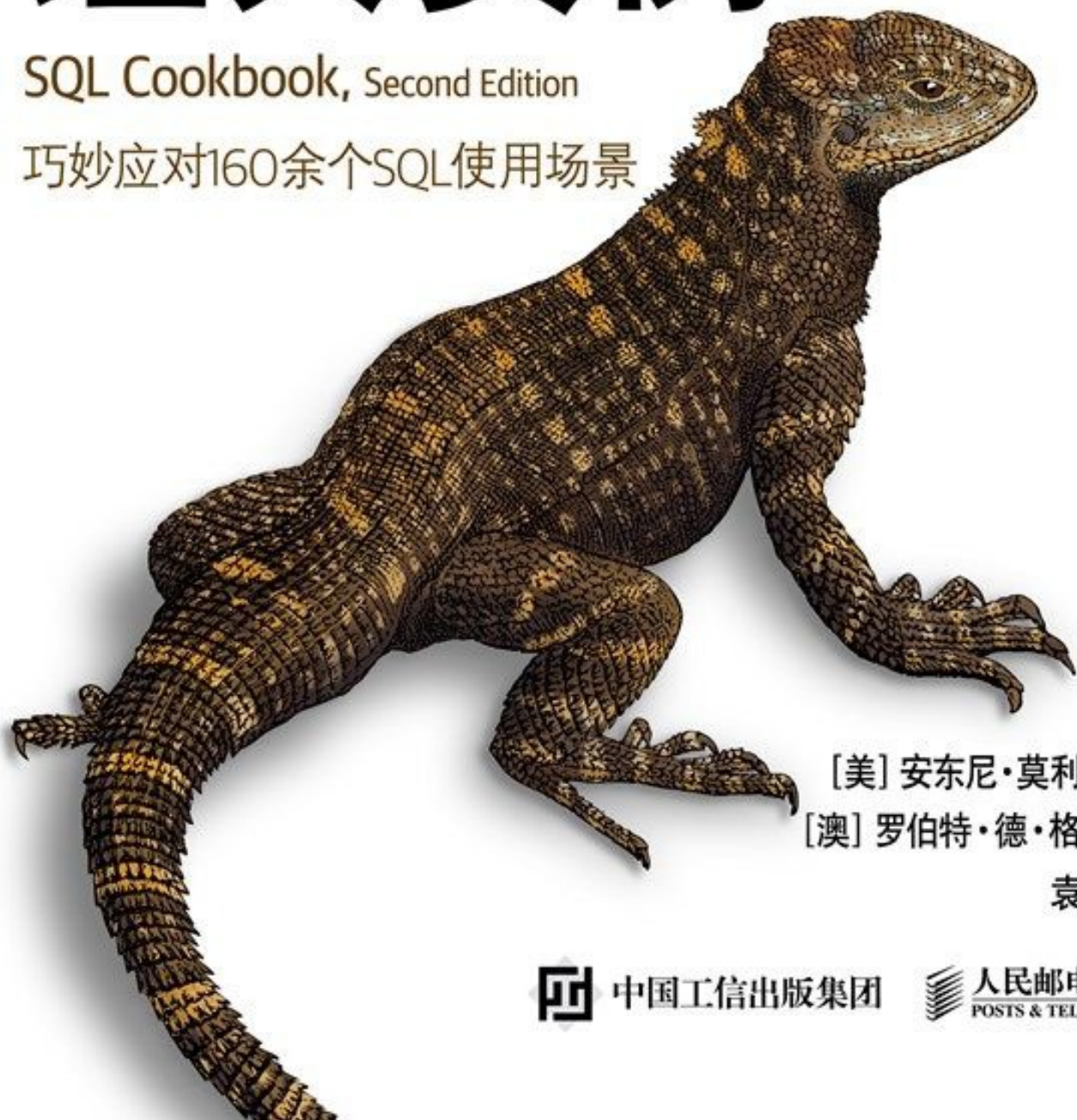
图灵程序设计丛书

第2版

SQL 经典实例

SQL Cookbook, Second Edition

巧妙应对160余个SQL使用场景



[美] 安东尼·莫利纳罗 著
[澳] 罗伯特·德·格拉夫
袁国忠 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：SQL经典实例（第2版）

作者：[美] 安东尼·莫利纳罗 [澳] 罗伯特·德·格拉夫

译者：袁国忠

ISBN：978-7-115-57796-2

版权声明

O'Reilly Media, Inc. 介绍

业界评论

献词

前言

读者对象

未涵盖的内容

平台和版本

使用的表

排版约定

字体约定

编码约定

O'Reilly在线学习平台（O'Reilly Online Learning）

联系我们

第2版致谢

第1版致谢

更多信息

第1章 检索记录

1.1 检索表中所有的行和列

1.2 从表中检索部分行

1.3 查找满足多个条件的行

1.4 从表中检索部分列

1.5 提供有意义的列名

1.6 在WHERE子句中使用别名来引用列

1.7 拼接列值

1.8 在SELECT语句中使用条件逻辑

1.9 限制返回的行数

1.10 从表中随机返回 n 行数据

1.11 查找NULL值

1.12 将NULL转换为实际值

1.13 模式查找

1.14 小结

第2章 查询结果排序

2.1 按指定顺序返回查询结果

2.2 按多字段排序

2.3 按子串排序

2.4 对同时包含字母和数字的数据进行排序

2.5 排序时处理NULL值

2.6 根据依赖于数据的键进行排序

2.7 小结

第3章 使用多张表

3.1 合并多个行集

3.2 合并相关的行

3.3 查找两张表中相同的行

3.4 从一张表中检索没有出现在另一张表中的值

3.5 从一张表中检索在另一张表中没有对应行的行

3.6 在查询中添加连接并确保不影响其他连接

3.7 判断两张表包含的数据是否相同

3.8 识别并避免笛卡儿积

3.9 同时使用连接和聚合

- 3.10 同时使用外连接和聚合
- 3.11 返回多张表中不匹配的行
- 3.12 在运算和比较中使用NULL
- 3.13 小结

第 4 章 插入、更新和删除

- 4.1 插入新记录
- 4.2 插入默认值
- 4.3 用NULL覆盖默认值
- 4.4 将一张表中的行复制到另一张表中
- 4.5 复制表定义
- 4.6 同时插入多张表
- 4.7 禁止在特定列中插入值
- 4.8 修改表中的记录
- 4.9 仅当存在匹配行时才更新
- 4.10 使用来自另一张表中的值进行更新
- 4.11 合并记录
- 4.12 删除表中的所有记录
- 4.13 删除特定记录
- 4.14 删除单条记录
- 4.15 删除违反引用完整性的记录
- 4.16 删除重复记录
- 4.17 删除在另一张表中引用了的记录
- 4.18 小结

第 5 章 元数据查询

- 5.1 列出模式中的所有表

- 5.2 列出表中的列
- 5.3 列出表的索引列
- 5.4 列出表的约束
- 5.5 列出没有相应索引的外键
- 5.6 使用SQL生成SQL
- 5.7 描述Oracle数据库中的数据字典视图
- 5.8 小结

第 6 章 处理字符串

- 6.1 走查字符串
- 6.2 在字符串字面量中嵌入引号
- 6.3 计算字符串中特定字符出现的次数
- 6.4 将不想要的字符从字符串中删除
- 6.5 将数字数据和字符数据分开
- 6.6 判断字符串是否只包含字母和数字
- 6.7 提取姓名中的首字母
- 6.8 根据部分字符串排序
- 6.9 根据字符串中的数字排序
- 6.10 根据表中的行创建分隔列表
- 6.11 将分隔数据转换为多值IN列表
- 6.12 按字母顺序排列字符串中的字符
- 6.13 识别可视为数字的字符串
- 6.14 提取第 n 个子串
- 6.15 拆分IP地址
- 6.16 根据发音比较字符串
- 6.17 查找与模式不匹配的文本

6.18 小结

第 7 章 处理数字

7.1 计算平均值

7.2 找出最大列值和最小列值

7.3 计算列值总和

7.4 计算表中的行数

7.5 计算非NULL列值数

7.6 生成移动总计

7.7 生成移动总积

7.8 平滑值序列

7.9 计算众数

7.10 计算中值

7.11 计算总计占比

7.12 聚合值可为NULL的列

7.13 计算剔除最高值和最低值后的平均值

7.14 将由字母和数字组成的字符串转换为数字

7.15 修改移动总计中的值

7.16 使用绝对中位差找出异常值

7.17 使用本福特法则查找反常数据

7.18 小结

第 8 章 日期算术运算

8.1 加上或减去若干天、若干月或若干年

8.2 确定两个日期相差多少天

8.3 确定两个日期之间有多少个工作日

8.4 确定两个日期相隔多少个月或多少年

- 8.5 确定两个日期相隔多少秒、多少分钟或多少小时
- 8.6 计算一年中有多少个工作日
- 8.7 确定当前记录和下一条记录存储的日期相隔多少天
- 8.8 小结

第 9 章 操作日期

- 9.1 判断特定的年份是否是闰年
- 9.2 确定特定年份有多少天
- 9.3 提取日期的各个组成部分
- 9.4 找出一个月的第一天和最后一天
- 9.5 找出一年中所有的星期 n
- 9.6 找出一个月中第一个和最后一个星期 n
- 9.7 创建日历
- 9.8 列出一年中各个季度的第一天和最后一天
- 9.9 确定给定季度的第一天和最后一天
- 9.10 补全缺失的日期
- 9.11 根据日期的特定部分进行查找
- 9.12 根据日期的特定部分对记录进行比较
- 9.13 找出重叠的日期范围
- 9.14 小结

第 10 章 涉及区间的查询

- 10.1 找出一系列连续的值
- 10.2 找出同一个分组或分区中相邻行的差
- 10.3 找出连续值构成的区间的起点和终点
- 10.4 填补值区间空隙
- 10.5 生成连续的数字值

10.6 小结

第 11 章 高级查找

11.1 在结果集中翻页

11.2 在表中跳过 n 行数据

11.3 在外连接中使用OR逻辑

11.4 确定哪些行是互逆的

11.5 返回前 n 条记录

11.6 找出值最高和最低的记录

11.7 查看后面的行

11.8 平移行值

11.9 结果排名

11.10 消除重复行

11.11 查找马值

11.12 生成简单预测

11.13 小结

第 12 章 报表制作和整形

12.1 将结果集转置为一行

12.2 将结果集转置为多行

12.3 对结果集进行逆转置

12.4 将结果集逆转置为一列

12.5 消除结果集中的重复值

12.6 转置结果集以简化涉及多行的计算

12.7 创建尺寸固定的数据桶

12.8 创建预定数量的桶

12.9 创建水平直方图

- 12.10 创建垂直直方图
- 12.11 返回未被用作分组依据的列
- 12.12 计算简单的小计
- 12.13 计算各种可能的小计
- 12.14 标出非小计行
- 12.15 使用CASE表达式来标识行
- 12.16 创建稀疏矩阵
- 12.17 按时间分组
- 12.18 同时对不同的分组/分区进行聚合
- 12.19 聚合移动值区间
- 12.20 转置包含小计的结果集
- 12.21 小结

第 13 章 分层查询

- 13.1 呈现父子关系
- 13.2 呈现子-父-祖父关系
- 13.3 创建基于表的分层视图
- 13.4 找出给定父行的所有子行
- 13.5 确定叶子节点、分支节点和根节点
- 13.6 小结

第 14 章 杂项

- 14.1 使用SQL Server运算符PIVOT创建交叉报表
- 14.2 使用SQL Server运算符UNPIVOT逆转置交叉报表
- 14.3 使用Oracle子句MODEL转置结果集
- 14.4 从不固定的位置提取子串
- 14.5 确定特定年份有多少天（另一种Oracle解决方案）

- 14.6 找出同时包含字母和数字的字符串
- 14.7 在Oracle中将整数转换为其二进制表示
- 14.8 对经过排名的结果集进行转置
- 14.9 给经过两次转置的结果集添加列标题
- 14.10 在Oracle中将标量子查询转换为复合子
- 14.11 将序列化数据转换为行
- 14.12 计算占总计的百分比
- 14.13 确定编组是否包含指定的值
- 14.14 小结

附录 A 温习窗口函数

A.1 分组

A.1.1 SQL分组的定义

A.1.2 悖论

A.1.3 SELECT和GROUP BY之间的关系

A.2 窗口函数

A.2.1 一个简单示例

A.2.2 执行顺序

A.2.3 分区

A.2.4 NULL的影响

A.2.5 排列顺序很重要时

A.2.6 框架子句

A.2.7 最后一个框架子句示例

A.2.8 可读性 + 性能 = 威力强大

A.2.9 打下基础

附录 B 通用表表达式

B.1 子查询

B.2 通用表表达式

B.3 小结

关于作者

关于封面

版权声明

Copyright © 2021 Robert de Graaf. All rights reserved.

Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2021. Authorized translation of the English edition, 2021 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2021。

简体中文版由人民邮电出版社有限公司出版，2021。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc. 介绍

O'Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O'Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O'Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术人员聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

—*Linux Journal*

献词

献给我的母亲：您是最棒的！谢谢您为我们所做的一切。

——安东尼·莫利纳罗

献给Clare、Maya和Leda。

——罗伯特·德·格拉夫

前言

SQL 是数据从业人员的通用语言，但未能像热门工具那样获得应有的关注。有些人经常使用 SQL 来编写最简单的查询，他们以为这就是 SQL 的全部功能。

本书展示了 SQL 的强大威力，旨在丰富你的工具箱。读完本书后，你将知道如何使用 SQL 来进行统计分析，像使用商业智能工具那样制作报表，进行文本数据匹配，以及执行复杂的日期数据分析，等等。

本书第 1 版一经推出，就成了流行的“第二本 SQL 图书”——学完基本知识后选择阅读的图书。本书有很多优点，比如涉及的主题范围广泛，对读者非常友好。

然而，计算机领域瞬息万变，即便成熟如 SQL 也不例外——它是在 20 世纪 70 年代面世的。本书并未介绍全新的 SQL 特性，但在第 1 版撰写期间，有些特性是新推出的，并非所有的 RDBMS（关系数据库管理系统）都支持，而现在这些特性已趋于稳定并得以标准化。因此，与第 1 版相比，本书第 2 版包含的标准解决方案更多。

这里需要重点说明的特性有两个。第一个是通用表表达式（CTE，包括递归 CTE）。本书第 1 版出版时，只有几款 RDBMS 支持它，但如今的 5 款 RDBMS 都支持它。引入 CTE 旨在克服 SQL 的一些局限性，本书的实例对这些局限性做了阐述。本书新增了一个专门介绍 CTE 的附录，旨在凸显其重要性并阐述其用途。

第二个需要重点说明的特性是窗口函数。在本书第 1 版出版时，窗口函数也非常新，并非所有的 RDBMS 都支持。有鉴于此，第 1 版专辟了一个附录对其进行介绍，第 2 版保留了这个

附录。现在本书涉及的所有 RDBMS 都支持它，我们知的其他 SQL 实现亦如此，但市面上的数据库管理系统如此之多，我们并不确定它们是否都支持窗口函数和 CTE。

除了尽可能将查询标准化，第 6 章和第 7 章还增补了一些内容，其中第 6 章新增了根据文本发音匹配数据的实例，并将正则表达式的内容移到了第 14 章，而第 7 章新增了绝对中位差和本福特法则的实例。

读者对象

本书适合任何想让查询更上一层楼的 SQL 用户。本书要求读者对 SQL 有所了解（比如阅读过 Alan Beaulieu 所著的 *Learning SQL*），要是以前编写过查询来解决实际问题则更佳。

只要符合上面的要求，任何 SQL 用户都适合阅读本书，包括数据工程师、数据科学家、数据可视化人员、BI 人员等。在这些 SQL 用户中，有些可能很少甚至从不直接访问数据库，而只是使用数据可视化工具、BI 或统计工具来查询并取回数据。本书的重点是可以解决实际问题的实用查询，虽然有时会简单地介绍一下理论，但这么做旨在帮助你理解实用查询。

未涵盖的内容

本书是一部实用指南，主要介绍如何使用 SQL 来读懂数据。除非为了阐述具体的实例或技巧，否则本书不会介绍数据库理论、数据库设计或 SQL 背后的理论。

本书也未涉及用于处理 XML 和 JSON 等数据类型的数据库扩展，市面上有其他介绍这些主题的资源。

平台和版本

SQL 发展变化非常快，厂商不断在其产品中新增特性和功能。有鉴于此，有必要指出本书针对的各种平台的版本。

- DB2 11.5
- Oracle Database 19c
- PostgreSQL 12
- SQL Server 2017
- MySQL 8.0

使用的表

本书的大部分示例用到了两张表，即 **EMP** 和 **DEPT**。**EMP** 表很简单，只包含 14 行数据，且字段的数据类型为数字、字符串或日期。**DEPT** 表也很简单，只有 4 行数据，且只包含数字字段和字符串字段。很多经典数据库图书用到了这些表，因此大家对部门和员工之间的多对一关系很熟悉。

除了为数不多的几种解决方案，本书的其他所有解决方案针对的都是这些表。对于你不太可能在实际工作中去实现的解决方案，本书没有为它们专门调整示例数据。

EMP 表和 **DEPT** 表的内容如下。

select * from emp;							
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
-----	-----	-----	-----	-----	-----	-----	-----
7369	SMITH	CLERK	7902	17-DEC-2005	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-2006	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-2006	1250	500	30
7566	JONES	MANAGER	7839	02-APR-2006	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-2006	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-2006	2850		30
7782	CLARK	MANAGER	7839	09-JUN-2006	2450		10
7788	SCOTT	ANALYST	7566	09-DEC-2007	3000		20
7839	KING	PRESIDENT		17-NOV-2006	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-2006	1500	0	30
7876	ADAMS	CLERK	7788	12-JAN-2008	1100		20
7900	JAMES	CLERK	7698	03-DEC-2006	950		30
7902	FORD	ANALYST	7566	03-DEC-2006	3000		20
7934	MILLER	CLERK	7782	23-JAN-2007	1300		10
select * from dept;							
DEPTNO	DNAME	LOC					
-----	-----	-----					

10 ACCOUNTING	NEW YORK
20 RESEARCH	DALLAS
30 SALES	CHICAGO
40 OPERATIONS	BOSTON

本书还使用了 4 张透视表（pivot table），即 **T1**、**T10**、**T100** 和 **T500**。由于这些表只用于简化转置工作，因此本书并未给它们指定意义深远的名称。在每张透视表的名称中，**T** 后面的数字指出了表中包含的行数。例如，下面是 **T1** 和 **T10** 包含的值。

```
select id from t1;

      ID
-----
      1

select id from t10;

      ID
-----
      1
      2
      3
      4
      5
      6
      7
      8
      9
     10
```

在你需要创建一系列行以简化查询编写工作时，透视表提供了极大的便利。

另外，有些厂商支持不完整的 **SELECT** 语句，例如，你可以编写不包含 **FROM** 子句的 **SELECT** 语句。在本书有些地方，为清晰起见，会使用只包含一行数据的支持表 **T1**，而不使用不完整的查询。**T1** 表的用途与 Oracle 的 **DUAL** 表类似，但 **T1** 表可以让解决方案标准化，适用于本书涉及的所有 **RDBMS**。

还有一些表是特定实例或章节专用的，本书将在合适的时候介绍。

排版约定

本书遵循了众多与字体和编码相关的约定，请花时间熟悉它们。遵循这些约定，尤其是编码约定，旨在帮助你理解正文。本书不会在每个实例中都重述这些约定，而是在这里将它们一并列出。

字体约定

本书遵循如下字体约定。

- 大写（UPPERCASE）

在正文中用于标识 SQL 关键字。

- 小写（lowercase）

用于代码示例中的所有查询。诸如 C 和 Java 等语言会将大多数关键字小写，我们发现小写比大写阅读起来更轻松，因此将所有查询都小写。

- 等宽粗体（**Constant width bold**）

在演示交互的示例中，用于标识用户输入。



表示提示、建议或一般性说明。



表示警告或告诫。

编码约定

对于 SQL 语句，我们喜欢使用小写，无论是关键字，还是用户定义的标识符。例如：

```
select empno, ename
  from emp;
```

你的喜好可能不同。例如，很多人喜欢将 SQL 关键字大写。请采用自己喜欢的编码风格或按照项目的要求做。

虽然 SQL 关键字和标识符在代码示例中是小写，但它们在正文中都是大写。我们这样做旨在将这些内容与普通文字区分开，例如：

上面的查询是一条针对 EMP 表的 SELECT 语句。

虽然本书涵盖来自 5 个厂商的 RDBMS，但我们决定对所有的输出都采用统一的格式。

```
EMPNO ENAME
-----
 7369 SMITH
 7499 ALLEN
...
```

很多解决方案在 FROM 子句中使用了内嵌视图或子查询。ANSI SQL 标准要求给这样的视图指定别名（只有 Oracle 不要求指定别名），因此本书在解决方案中使用诸如 X 和 Y 等别名来标识内嵌视图返回的结果集。

```
select job, sal
  from (select job, max(sal) sal
        from emp
       group by job) x
```

注意，字母 **X** 紧跟在最后一个右括号后面，它将作为 **FROM** 子句中子查询返回的“表”的名称。为了让代码的含义不言自明，给列指定别名很有用，但在本书的大部分实例中，给内嵌视图指定别名只是为了遵守规定，因此指定的别名通常很简单，比如 **X**、**Y**、**Z**、**TMP1** 和 **TMP2**。在更好的别名可帮助理解时，我们就使用这样的别名。

在实例的“解决方案”部分，通常在 **SQL** 代码中添加了行号，例如：

```
1 select ename
2      from emp
3  where deptno = 10
```

这些行号并非代码的组成部分，包含它们是为了在“讨论”部分使用数字来引用查询的特定部分。

O'Reilly在线学习平台（O'Reilly Online Learning）



40 多年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独特的由专家和创新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 的在线学习平台让你能够按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本资源和视频资源。有关的更多信息，请访问 <https://www.oreilly.com>。

联系我们

与本书有关的评论和问题，请发给出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室
(100035)

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息¹。本书的网页是 <https://oreil.ly/sql-ckbk-2e>。

¹也可以访问图灵社区本书主页提交中文版勘误。——编者注

对于本书的评论和技术性问题，请发送电子邮件到 bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程和新闻的信息，请访问以下网站：<https://www.oreilly.com>。

我们在 Facebook 的地址如下：<http://facebook.com/oreilly>。

请关注我们的 Twitter 动态：<http://twitter.com/oreillymedia>。

我们的 YouTube 视频地址如下：<http://www.youtube.com/oreillymedia>。

第2版致谢

本书得以出版离不开众人的帮助。感谢 O'Reilly 出版社的 Jess Haberman、Virginia Wilson、Kate Galloway 和 Gary O'Brien。感谢 Nicholas Adams 多次挽狂澜于既倒。万分感谢技术审校 Alan Beaulieu、Scott Haines 和 Thomas Nield。

另外，非常感谢我的家人 Clare、Maya 和 Leda，感谢他们潇洒大度，忍受我在写作期间不能陪伴左右。

——罗伯特·德·格拉夫

第1版致谢

如果没有众人的大力支持，本书就不可能出版。首先要感谢我的母亲 **Connie**，没有您的辛劳和奉献，就没有今天的我。感谢您为我们兄弟所做的一切，做您的孩子真是我们的福气。

感谢我的兄弟 **Joe**。每当我暂停写作，离开巴尔的摩回到家时，你都让我感受到了闲暇时光的美好，促使我结束写作，重新投入到人生中更重要的事情中去。你心地善良，受人尊敬，做你的兄弟让我深感骄傲。

感谢我出色的未婚妻 **Georgia**，没有你的支持，我不可能克服艰难险阻，完成本书。日复一日，你始终与我并肩战斗，这对我来说是考验，对你来说更是如此。我没日没夜地工作、写作，而你的理解和支持始终如一，对此我没齿难忘。谢谢你，我爱你。

感谢未婚妻的家人。感谢岳母 **KiKi** 和岳父 **George** 在我写作期间一如既往的支持。每当我得空拜访，你们都盛情款待，给我宾至如归之感。感谢 **Anna** 和 **Kathy** 两位小妹妹，与你们闲聊有趣至极，让我在写作之余得到了亟需的放松和休息。

感谢编辑 **Jonathan Gennick**，没有你本书就不可能付梓。本书能够出版，你居功至伟。你所做的超出了编辑的职责范畴，从提供实例到重写大量内容，再到在最终交稿期限来临时营造幽默气氛，要是没有你，我不可能坚持到最后。对你所做的这一切，我万分感谢。你能给予我机会并担任我的编辑，我心怀感激。你是经验丰富的 **DBA** 和作家，与你这样技术好、专业知识丰富的人合作真是一件乐事。就算辞掉编辑工作，你也能在任何地方找到数据库管理员（**DBA**）的工作，而像你这样的编辑凤毛麟角。身为 **DBA** 无疑让你在从事编辑工作时具备莫大的优势，因为你总能知道我想说什么，哪怕我表达得不清楚。有你

这样的员工是 O'Reilly 的幸运，有你这样的编辑是我的幸运。

感谢 *Transact-SQL Cookbook* 的作者 Ales Spetic 和 Jonathan Gennick。艾萨克·牛顿有句名言：如果说我比别人看得更远的话，那是因为我站在巨人的肩膀上。在 *Transact-SQL Cookbook* 一书的致谢部分，Ales Spetic 很好地诠释了这句名言。在我看来，这种诠释对所有的 SQL 图书都适用。现将这些诠释摘录如下。

Joe Celko、David Rozenshtein、Anatoly Abramovich、Eugene Berger、Iztik BenGan、Richard Snodgrass 等杰出作者推出了相关作品，但愿本书能起到补充作用。无数个夜晚，我潜心研究这些著作，我拥有的大部分知识来自其中。撰写本书期间，我深深地知道，我花一个晚上学到的秘诀，是这些作者花 10 个晚上才将它们变成铅字的。能够为 SQL 社区尽自己的微薄之力，我深感荣幸。

感谢 Sanjay Mishra 推出名著 *Mastering Oracle SQL*，并给我引荐 Jonathan。要不是 Sanjay，我不可能与 Jonathan 相识，也不可能撰写本书。真是神奇，一封简单的电子邮件就能改变你的人生。特别感谢 David Rozenshtein 推出著作 *Essence of SQL*，让我对如何从集合 /SQL 的角度思考和解决问题有了深刻认识。感谢 David Rozenshtein、Anatoly Abramovich 和 Eugene Birger 推出著作 *Optimizing Transact-SQL*，我当前使用的很多 SQL 高级技巧是从其中学到的。

感谢 Wireless Generation 的整个团队，这是一家杰出的公司，员工也非常出色。万分感谢所有抽出时间审阅书稿、提出意见或建议的人，他们是 Jesse Davis、Joel Patterson、Philip Zee、Kevin Marshall、Doug Daniels、Otis Gospodnetic、Ken Gunn、John Stewart、Jim Abramson、Adam Mayer、Susan Lau、Alexis Le-Quoc 和 Paul Feuer。感谢 Maggie Ho 对本书进行了审阅，并就附录 A 提供宝贵的反馈。感谢 Chuck Van Buren 和 Gillian

Gutenberg 在跑步方面的宝贵建议，晨练助我厘清思路、放松身心。要是不出去活动活动，我都无法坚持将本书写完。感谢 Steve Kang 和 Chad Levinson 在夜晚忍受我喋喋不休地谈论各种 SQL 技术，经过一整天的工作，他们原本只想去联合广场的 Heartland Brewery 喝啤酒、吃汉堡。感谢 Aaron Boyd 的支持和赞誉，最重要的是宝贵建议。Aaron 诚实、勤奋且坦率，大家都很喜欢他，公司因此得以蒸蒸日上。感谢 Olivier Pomel 在本书写作过程中提供帮助和支持，尤其是从行创建分隔列表的 DB2 解决方案，他在没有 DB2 系统进行测试的情况下提供了该解决方案，在我阐述 WITH 子句的工作原理后，他仅用几分钟就设计出了本书中的这种解决方案。

Jonah Harris 和 David Rozenshtein 对初稿做了技术审阅，并提供了有益的反馈。在本书成书阶段，Arun Marathe、Nuno Pinto do Souto 和 Andrew Odewahn 就实例的大纲和挑选展开了讨论。万分感谢你们每一位。

感谢 John Haydu 及 Oracle 公司的 MODEL 子句开发团队，感谢他们抽出时间审阅我为 O'Reilly 撰写的有关 MODEL 子句的文章，让我最终对这个子句的工作原理有了更深入的认识。感谢 Oracle 公司的 Tom Kyte 允许我将其开发的 TO_BASE 函数改编为 SQL 解决方案。感谢微软的 Bruno Denuit 回答我就 SQL Server 2005 引入的窗口函数的功能提出的问题。深深感谢 PostgreSQL 的 Simon Riggs 让我及时了解了 PostgreSQL 的新 SQL 特性。

（知道 PostgreSQL 将引入哪些新特性及何时引入后，我得以将 GENERATE_SERIES 函数等 SQL 新特性纳入本书。在我看来，与使用透视表相比，使用这个函数让解决方案更为优雅。）

最后要感谢 Kay Young。如果一个人才华横溢，对所做的工作充满热情，那么能与同样的人合作将是非常美妙的。本书很多实例源自我在 Wireless Generation 期间与 Kay 合作解决日常问题时设计的 SQL 解决方案。Kay，谢谢你。我还想让你知道，对于你在本书撰写期间提供的各种帮助，我心怀感激。从语法修

改建议到编码，你在本书的撰写过程中扮演着不可或缺的角色。与你一起工作非常美妙，Wireless Generation 因为有你而更为出色。

——安东尼·莫利纳罗

更多信息

扫描下方二维码，即可获取电子书相关信息及读者群通道入口。



第 1 章 检索记录

本章重点介绍基本的 **SELECT** 语句。深入理解这些基础知识很重要，因为在本章介绍的主题中，很多不仅在日常使用 **SQL** 时会涉及，而且在后面更复杂的实例中也会涉及。

1.1 检索表中所有的行和列

1. 问题

你有一张表，你想查看表中的所有数据。

2. 解决方案

使用 **SELECT** 语句来查询表，并使用特殊字符 ***** 指定返回所有的列。

```
1 select *  
2   from emp
```

3. 讨论

在 SQL 中，字符 ***** 有特殊含义，它会返回指定表中所有的列。由于没有指定 **WHERE** 子句，因此将返回所有的行。也可以分别列出每一列。

```
select empno,ename,job,sal,mgr,hiredate,comm,deptno  
from emp
```

在以交互方式执行的临时查询中，使用 **SELECT *** 更容易。不过，在编写程序代码时，分别指定各列更合适。这两种做法性能相同，但显式指定能让你知道查询将返回哪些列。同理，这种查询对其他人（他们可能知道也可能不知道查询的表中包含哪些列）来说更容易理解。对于代码中的查询，使用 **SELECT *** 会带来问题，因为查询返回的列可能不符合预

期。不管怎样，指定所有列时，如果没有返回其中的一列或多列，则可以跟踪异常以确定缺失了哪些列。

1.2 从表中检索部分行

1. 问题

你有一张表，你想查看表中满足特定条件的行。

2. 解决方案

使用 **WHERE** 子句来指定要返回哪些行。例如，要查看部门编号为 10 的所有员工，可以像下面这样做。

```
1 select *  
2   from emp  
3  where deptno = 10
```

3. 讨论

WHERE 子句能够让你只检索感兴趣的行。对于表中的每一行，如果它满足 **WHERE** 子句中表达式指定的条件，就返回它。

大多数数据库支持常用的运算符，比如 **=**、**<**、**>**、**<=**、**>=**、**!** 和 **<>**。另外，你可能想返回满足多个条件的行，为此可以使用 **AND**、**OR** 和圆括号，这将在下一节中演示。

1.3 查找满足多个条件的行

1. 问题

你想返回满足多个条件的行。

2. 解决方案

结合使用 **WHERE** 子句、**OR** 子句和 **AND** 子句。例如，要查找部门编号为 10 的员工、有业务提成的员工以及薪水不超过 2000 美元且部门编号为 20 的员工，可以像下面这样做。

```
1 select *  
2   from emp  
3  where deptno = 10  
4         or comm is not null  
5         or sal <= 2000 and deptno=20
```

3. 讨论

要返回满足多个条件的行，可以结合使用 **AND**、**OR** 和圆括号。在上述解决方案中，**WHERE** 子句查找满足下面任何一个条件的行：

- **DEPTNO** 为 10；
- **COMM** 不为 **NULL**；
- 薪水不超过 2000 美元且 **DEPTNO** 为 20。

圆括号指定将多个条件作为一个整体。

例如，下面是将前 3 个条件放在圆括号内时返回的结果集。

```
select *  
  from emp  
 where (      deptno = 10  
          or comm is not null  
          or sal <= 2000  
        )  
    and deptno=20
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
-----	-----	-----	-----	-----	-----	-----	-----
7369	SMITH	CLERK	7902	17-DEC-1980	800		20
7876	ADAMS	CLERK	7788	12-JAN-1983	1100		20

1.4 从表中检索部分列

1. 问题

你有一张表，你想查看表中特定列（而不是所有列）的值。

2. 解决方案

指定要查看的列。例如，只查看员工的姓名、部门编号和薪水。

```
1 select ename,deptno,sal  
2   from emp
```

3. 讨论

在 **SELECT** 子句中指定列可以避免返回无关的数据。通过网络检索数据时，这样做非常重要，因为可以避免浪费时间而检索不需要的数据。

1.5 提供有意义的列名

1. 问题

你想修改查询返回的列的名称，使其可读性更高且更容易理解。请看下面的查询，它返回的是每位员工的薪水和业务提成。

```
1 select sal,comm
2    from emp
```

SAL 是什么？是销量（sale）的缩写吗？是人名吗？**COMM** 又是什么？是沟通（communication）的意思吗？你希望结果集中标签的含义更明确。

2. 解决方案

要修改查询结果中显示的列名，可以使用关键字 **AS**（*original_name AS new_name*）。有些数据库不强制要求使用 **AS**，但所有数据库都支持这样做。

```
1 select sal as salary, comm as commission
2    from emp
```

SALARY	COMMISSION
-----	-----
800	
1600	300
1250	500
2975	
1250	1400
2850	
2450	
3000	

5000	
1500	0
1100	
950	
3000	
1300	

13. 讨论

使用关键字 **AS** 给查询返回的列指定新名称的行为被称为指定别名，而指定的新名称被称为别名。

通过指定良好的别名，可以让查询及其返回的结果对他人来说更容易理解。

1.6 在 **WHERE** 子句中使用别名来引用列

1. 问题

在结果集中，你已经使用别名提供了含义更明确的列名，但还想使用 **WHERE** 子句将某些行排除在外。为此，你在 **WHERE** 子句中引用了别名，但以失败告终。

```
select sal as salary, comm as commission
  from emp
 where salary < 5000
```

2. 解决方案

将查询作为内嵌视图来引用列的别名。

```
1 select *
2   from (
3 select sal as salary, comm as commission
4   from emp
5       ) x
6  where salary < 5000
```

3. 讨论

在这个简单的实例中，可以在 **WHERE** 子句中直接引用 **COMM** 或 **SAL**，这样就无须使用内嵌视图了。上述解决方案演示了在 **WHERE** 子句中引用如下内容时，需要做什么。

- 聚合函数

- 标量子查询
- 窗口函数
- 别名

将提供别名的查询放在内嵌视图中，就可以在外部查询中引用列的别名。为什么要这样做呢？这是由于 **WHERE** 子句是在 **SELECT** 子句之前评估的，因此，对于前面说明问题时列举的查询，在评估其中的 **WHERE** 子句时，还没有别名 **SALARY** 和 **COMMISSION**。这些别名在 **WHERE** 子句处理完毕后才存在。然而，**FROM** 子句是在 **WHERE** 子句之前评估的。如果将查询放在 **FROM** 子句中，那么将在最外层的 **WHERE** 子句执行前生成该查询的结果，这样一来，最外层的 **WHERE** 子句就能够“看到”别名了。在列名不是太好时，这种技巧很有用。



在本解决方案中，内嵌视图被指定了别名 **X**。并非所有数据库都要求给内嵌视图指定别名，但对于某些数据库而言，确实应该如此。不过，所有数据库都允许这样做。

1.7 拼接列值

1. 问题

你想将多列的值作为一列返回。例如，你希望对 **EMP** 表的查询返回如下结果集。

```
CLARK WORKS AS A MANAGER  
KING WORKS AS A PRESIDENT  
MILLER WORKS AS A CLERK
```

然而，生成这个结果集所需的数据来自 **EMP** 表中两个不同的列，即 **ENAME** 和 **JOB**。

```
select ename, job  
  from emp  
 where deptno = 10
```

ENAME	JOB
CLARK	MANAGER
KING	PRESIDENT
MILLER	CLERK

2. 解决方案

找到并使用 **DBMS**（数据库管理系统）内置的函数来拼接多列的值。

DB2、Oracle 和 PostgreSQL

这些数据库将双竖线用作拼接运算符。

```
1 select ename||' WORKS AS A '||job as msg
2   from emp
3  where deptno=10
```

MySQL

该数据库支持函数 **CONCAT**。

```
1 select concat(ename, ' WORKS AS A ',job) as msg
2   from emp
3  where deptno=10
```

SQL Server

该数据库使用运算符 **+** 来执行拼接操作。

```
1 select ename + ' WORKS AS A ' + job as msg
2   from emp
3  where deptno=10
```

13. 讨论

使用函数 **CONCAT** 可以拼接多列的值。在 DB2、Oracle 和 PostgreSQL 中，函数 **CONCAT** 的简写为 **||**，在 SQL Server 中为 **+**。

1.8 在SELECT语句中使用条件逻辑

1. 问题

在 **SELECT** 语句中，你想执行基于值的 **IF-ELSE** 操作。例如，在生成结果集时，你想在员工的薪水不超过 2000 美元时返回消息 **UNDERPAID**，在员工的薪水不低于 4000 美元时返回消息 **OVERPAID**，在员工的薪水为 2000~4000 美元时返回消息 **OK**。这个结果集如下所示。

ENAME	SAL	STATUS
-----	-----	-----
SMITH	800	UNDERPAID
ALLEN	1600	UNDERPAID
WARD	1250	UNDERPAID
JONES	2975	OK
MARTIN	1250	UNDERPAID
BLAKE	2850	OK
CLARK	2450	OK
SCOTT	3000	OK
KING	5000	OVERPAID
TURNER	1500	UNDERPAID
ADAMS	1100	UNDERPAID
JAMES	950	UNDERPAID
FORD	3000	OK
MILLER	1300	UNDERPAID

2. 解决方案

在 **SELECT** 语句中直接使用 **CASE** 表达式来执行条件逻辑。

```
1 select ename,sal,
2       case when sal <= 2000 then 'UNDERPAID'
3           when sal >= 4000 then 'OVERPAID'
4           else 'OK'
```

```
5         end as status  
6   from emp
```

13. 讨论

CASE 表达式能够根据查询返回的值来执行条件逻辑。为了提高结果集的可读性，可以给 **CASE** 表达式指定别名。上述解决方案给 **CASE** 表达式的结果指定了别名 **STATUS**。**ELSE** 子句是可选的。如果省略了 **ELSE** 子句，那么对于不满足测试条件的行，**CASE** 表达式将返回 **NULL**。

1.9 限制返回的行数

1. 问题

你想限制查询中返回的行数。你不关心顺序，只要返回的行数是指定的 (n)。

2. 解决方案

使用数据库提供的内置函数来控制返回的行数。

DB2

在 DB2 中，使用 **FETCH FIRST** 子句。

```
1 select *  
2   from emp fetch first 5 rows only
```

MySQL 和 PostgreSQL

在 MySQL 和 PostgreSQL 中，使用 **LIMIT** 来限制返回的行数。

```
1 select *  
2   from emp limit 5
```

Oracle

在 Oracle 中，要限制返回的行数，可以在 **WHERE** 子句中对 **ROWNUM** 进行限制。

```
1 select *  
2   from emp  
3  where rownum <= 5
```

SQL Server

在 SQL Server 中，使用关键字 **TOP** 来限制返回的行数。

```
1 select top 5 *  
2   from emp
```

13. 讨论

很多数据库提供了相关的子句（如 **FETCH FIRST** 和 **LIMIT**）来指定查询返回的行数。Oracle 与众不同，它要求你使用函数 **ROWNUM**，对于返回的每一行，该函数都返回一个数字（从 1 开始不断递增）。

当你使用 **ROWNUM <= 5** 来返回前 5 行数据时，将发生如下事情。

- a. Oracle 执行查询。
- b. Oracle 取得第 1 行数据，并将编号设置为 1。
- c. 当前行的编号超过 5 了吗？如果没有，Oracle 就返回它，因为它满足条件“编号不超过 5”；否则，Oracle 就不返回它。
- d. Oracle 取得下一行数据并将编号加 1（编号依次为 2、3、4，以此类推）。
- e. 回到第 3 步。

正如上述过程显示的，取得每一行时，都将其编号设置为 **ROWNUM** 返回的值。这一点很重要。为了让查询只返回特定

行（如第 5 行），很多 Oracle 开发人员会指定条件 **ROWNUM = 5**。

使用 **ROWNUM** 来指定相等条件并不是一个好主意。使用 **ROWNUM = 5** 来试图返回第 5 行时，将发生如下事情。

- a. Oracle 执行查询
- b. Oracle 取得第 1 行数据，并将编号设置为 1。
- c. 编号是 5 吗？如果不是，那么 Oracle 将丢弃当前行，因为它不满足条件；如果是，Oracle 将返回它。然而，编号为 5 的情况永远不会发生！
- d. Oracle 取得下一行数据并将编号设置为 1。这是因为查询返回的第 1 行的编号必须为 1。
- e. 回到第 3 步。

如果仔细研究这个过程，你就会明白使用 **ROWNUM = 5** 无法返回第 5 行的原因。要让编号变为 5，必须先返回 4 行数据。

你可能注意到了，使用 **ROWNUM = 1** 确实能够返回第 1 行数据，这看似与前面的解释相互矛盾。使用 **ROWNUM = 1** 为什么能够返回第 1 行数据呢？这是因为为了确定表是否为空，Oracle 必须至少尝试取得一行数据。如果仔细审视上述过程，并将 5 替换为 1，你就会明白为什么使用条件 **ROWNUM = 1** 可以返回一行数据了。

1.10 从表中随机返回 n 行数据

1. 问题

你想从表中随机返回 n 行数据。为此，你想修改下面的语句，使其返回 5 行数据，且每次执行时返回的行都不同。

```
select ename, job
  from emp
```

2. 解决方案

使用 DBMS 提供的返回随机值的内置函数。在 **ORDER BY** 子句中，使用该内置函数以随机的方式对行进行排序，然后使用上一节介绍的方法来限制返回的行数。

DB2

结合使用内置函数 **RAND**、**ORDER BY** 和 **FETCH**。

```
1 select ename,job
2   from emp
3  order by rand() fetch first 5 rows only
```

MySQL

结合使用内置函数 **RAND**、**LIMIT** 和 **ORDER BY**。

```
1 select ename,job
2   from emp
3  order by rand() limit 5
```


PostgreSQL

结合使用内置函数 **RANDOM**、**LIMIT** 和 **ORDER BY**。

```
1 select ename,job
2   from emp
3  order by random() limit 5
```

Oracle

结合使用（内置包 **DBMS_RANDOM** 中的）内置函数 **VALUE**、**ORDER BY** 子句和内置函数 **ROWNUM**。

```
1 select *
2   from (
3     select ename, job
4       from emp
5     order by dbms_random.value()
6           )
7   where rownum <= 5
```

SQL Server

结合使用内置函数 **NEWID**、**TOP** 和 **ORDER BY** 来返回随机的结果集。

```
1 select top 5 ename,job
2   from emp
3  order by newid()
```

13. 讨论

ORDER BY 子句可以根据函数的返回值来调整结果集的排列顺序。这些解决方案都在 **ORDER BY** 子句中的函数执行后限

制返回的行数。即便你使用的不是 Oracle，仔细研究 Oracle 解决方案也会有所帮助，因为它展示了其他解决方案在幕后发生的情况。

在 **ORDER BY** 子句中，使用函数和使用数值常量导致的排序方式是不同的，明白这一点很重要。在 **ORDER BY** 子句中指定数值常量时，要求根据 **SELECT** 子句中相应位置的列进行排序。而在 **ORDER BY** 子句中指定函数时，要求对每行执行该函数，并根据函数的结果进行排序。

1.11 查找NULL值

1. 问题

你想查找特定列为 NULL 的所有行。

2. 解决方案

要判断一个值是否为 NULL，必须使用 **IS NULL**。

```
1 select *  
2   from emp  
3  where comm is null
```

3. 讨论

由于 NULL 与任何值（包括 NULL 本身）都不相等，也不会相等，因此测试列值是否为 NULL 时，不能使用 = 或 !=。要判断列值是否为 NULL，必须使用 **IS NULL**。也可以使用 **IS NOT NULL** 来查找给定列不为 NULL 的行。

1.12 将NULL转换为实际值

1. 问题

有些列为 NULL，但你不想返回 NULL，而想返回非 NULL 值。

2. 解决方案

使用函数 COALESCE 将 NULL 值替换为实际值。

```
1 select coalesce(comm,0)
2   from emp
```

3. 讨论

函数 COALESCE 可以将一个或多个值作为参数，并返回参数列表中的第一个非 NULL 值。在上述解决方案中，如果 COMM 不为 NULL，就返回它，否则就返回 0。

处理 NULL 值时，最好利用 DBMS 提供的内置功能。在很多情况下，有多个函数可以很好地完成这项任务，但 COALESCE 在所有 DBMS 中都管用。另外，在所有 DBMS 中，都可以使用 CASE 来完成这项任务。

```
select case
      when comm is not null then comm
      else 0
    end
  from emp
```

虽然可以使用 **CASE** 将 **NULL** 值转换为实际值，但使用 **COALESCE** 更容易且更简洁。

1.13 模式查找

1. 问题

你想返回与特定子串或模式匹配的行。请看下面的查询及其返回的结果集。

```
select ename, job
  from emp
 where deptno in (10,20)
```

ENAME	JOB
-----	-----
SMITH	CLERK
JONES	MANAGER
CLARK	MANAGER
SCOTT	ANALYST
KING	PRESIDENT
ADAMS	CLERK
FORD	ANALYST
MILLER	CLERK

在部门编号为 10 和部门编号为 20 的员工中，你只想返回那些姓名包含字母 **I** 或职位名称以 **ER** 结尾的员工。

ENAME	JOB
-----	-----
SMITH	CLERK
JONES	MANAGER
CLARK	MANAGER
KING	PRESIDENT
MILLER	CLERK

2. 解决方案

结合使用 **LIKE** 运算符和 SQL 通配符（%）。

```
1 select ename, job
2   from emp
3  where deptno in (10,20)
4     and (ename like '%I%' or job like '%ER')
```

13. 讨论

用于模式匹配运算 **LIKE** 中时，通配符（%）与任何字符序列都匹配。大多数 SQL 实现还提供了与单个字符匹配的下划线运算符（_）。通过将搜索模式 **I** 放在两个 % 之间，可以与任何包含 **I** 的字符串匹配。在没有将搜索模式放在两个 % 之间的情况下，% 的位置将影响查询结果。如果要查找以 **ER** 结尾的职位名称，那么可以将 % 放在 **ER** 前面；如果要查找以 **ER** 开头的职位名称，则需要将 % 放在 **ER** 后面。

1.14 小结

本章介绍的实例虽然简单，但它们是最基本的。信息检索是数据库查询的核心，这意味着在本书讨论的所有主题中，这些实例都处于核心位置。

第 2 章 查询结果排序

本章重点介绍如何定制查询结果。如果知道如何组织结果集，就可以让数据的可读性更高、含义更明确。

2.1 按指定顺序返回查询结果

1. 问题

你想显示部门编号为 10 的员工的姓名、职位和薪水，并根据薪水按从低到高的顺序排列。换言之，你希望返回如下结果集。

ENAME	JOB	SAL
-----	-----	-----
MILLER	CLERK	1300
CLARK	MANAGER	2450
KING	PRESIDENT	5000

2. 解决方案

使用 **ORDER BY** 子句。

```
1 select ename,job,sal
2   from emp
3  where deptno = 10
4  order by sal asc
```

3. 讨论

ORDER BY 子句能够对结果集中的行进行排序。本解决方案根据 **SAL** 将行按升序排列。由于 **ORDER BY** 默认按升序排列，因此 **ASC** 子句是可选的。另外，还可以按降序排列，为此可以使用 **DESC**。

```
select ename,job,sal
  from emp
 where deptno = 10
 order by sal desc
```

ENAME	JOB	SAL
KING	PRESIDENT	5000
CLARK	MANAGER	2450
MILLER	CLERK	1300

也可以不指定用于排序的列名，而使用数字来指定该列。数字从 1 开始，按从左到右的顺序对应于 **SELECT** 子句中指定的列。

```
select ename,job,sal
  from emp
 where deptno = 10
 order by 3 desc
```

ENAME	JOB	SAL
KING	PRESIDENT	5000
CLARK	MANAGER	2450
MILLER	CLERK	1300

在本实例中，**ORDER BY** 子句中的数字 3 对应于 **SELECT** 中指定的第 3 列，即 **SAL**。

2.2 按多字段排序

1. 问题

你想对 **EMP** 表中的行进行排序，方法是先按 **DEPTNO** 升序排列，然后按薪水降序排列。换言之，你希望返回如下结果集。

EMPNO	DEPTNO	SAL	ENAME	JOB
7839	10	5000	KING	PRESIDENT
7782	10	2450	CLARK	MANAGER
7934	10	1300	MILLER	CLERK
7788	20	3000	SCOTT	ANALYST
7902	20	3000	FORD	ANALYST
7566	20	2975	JONES	MANAGER
7876	20	1100	ADAMS	CLERK
7369	20	800	SMITH	CLERK
7698	30	2850	BLAKE	MANAGER
7499	30	1600	ALLEN	SALESMAN
7844	30	1500	TURNER	SALESMAN
7521	30	1250	WARD	SALESMAN
7654	30	1250	MARTIN	SALESMAN
7900	30	950	JAMES	CLERK

2. 解决方案

在 **ORDER BY** 子句中列出用于排序的列，并用逗号分隔它们。

```
1 select empno,deptno,sal,ename,job
2   from emp
3  order by deptno, sal desc
```

13. 讨论

按从左到右的顺序依次根据 **ORDER BY** 子句中指定的列进行排序。指定用于排序的列时，如果使用的是 **SELECT** 子句中列的数字位置，那么指定的数字不能超过 **SELECT** 子句中指定的列数。通常，可以按 **SELECT** 子句中未指定的列进行排序，但必须指定列名。然而，如果在查询中使用了 **GROUP BY** 或 **DISTINCT** 子句，就不能按 **SELECT** 子句中未指定的列进行排序。

2.3 按子串排序

1. 问题

你想根据字符串的特定部分对查询结果进行排序。例如，你想返回 **EMP** 表中的员工姓名和职位，并按 **JOB** 列的最后两个字符排序。结果集如下所示。

ENAME	JOB
-----	-----
KING	PRESIDENT
SMITH	CLERK
ADAMS	CLERK
JAMES	CLERK
MILLER	CLERK
JONES	MANAGER
CLARK	MANAGER
BLAKE	MANAGER
ALLEN	SALESMAN
MARTIN	SALESMAN
WARD	SALESMAN
TURNER	SALESMAN
SCOTT	ANALYST
FORD	ANALYST

2. 解决方案

DB2、MySQL、Oracle 和 PostgreSQL

在 **ORDER BY** 子句中使用函数 **SUBSTR**。

```
select ename,job
  from emp
 order by substr(job,length(job)-1)
```

SQL Server

在 ORDER BY 子句中使用函数 SUBSTRING。

```
select ename,job  
  from emp  
 order by substring(job,len(job)-1,2)
```

13. 讨论

使用 DBMS 提供的子串函数，可以轻松地按字符串的任何部分排序。要按字符串的最后两个字符排序，可以先找到该字符串的末尾位置（字符串的长度）并减去 1。这样，起始位置就是字符串的倒数第 2 个字符。然后，可以提取从起始位置开始到字符串末尾的所有字符。SQL Server 的函数 **SUBSTRING** 不同于函数 **SUBSTR**，它要求提供第 3 个参数，以用于指定要提取多少个字符。在本实例中，大于或等于 2 的数字都可以。

2.4 对同时包含字母和数字的数据进行排序

1. 问题

数据同时包含了字母和数字，而你想根据数据的字母部分或数字部分进行排序。请看下面这个根据 **EMP** 表创建的视图。

```
create view V
as
select ename||' '||deptno as data
  from emp
```

```
select * from V
```

DATA

SMITH 20
ALLEN 30
WARD 30
JONES 20
MARTIN 30
BLAKE 30
CLARK 10
SCOTT 20
KING 10
TURNER 30
ADAMS 20
JAMES 30
FORD 20
MILLER 10

你想根据 **DEPTNO** 或 **ENAME** 对结果进行排序。根据 **DEPTNO** 排序时生成的结果集如下所示。

DATA

CLARK 10
KING 10
MILLER 10


```
SMITH 20  
ADAMS 20  
FORD 20  
SCOTT 20  
JONES 20  
ALLEN 30  
BLAKE 30  
MARTIN 30  
JAMES 30  
TURNER 30  
WARD 30
```

根据 **ENAME** 排序时生成的结果集如下所示。

```
DATA  
-----  
ADAMS 20  
ALLEN 30  
BLAKE 30  
CLARK 10  
FORD 20  
JAMES 30  
JONES 20  
KING 10  
MARTIN 30  
MILLER 10  
SCOTT 20  
SMITH 20  
TURNER 30  
WARD 30
```

12. 解决方案

Oracle、SQL Server 和 PostgreSQL

使用函数 **REPLACE** 和 **TRANSLATE** 来修改字符串，并根据修改结果进行排序。

```

/* 根据DEPTNO排序 */

1  select data
2    from V
3   order by replace(data,
4                     replace(
5                       translate(data,'0123456789','#####'),'#',''), '')

/* 根据ENAME排序 */

1  select data
2    from V
3   order by replace(
4     translate(data,'0123456789','#####'),'#','')

```

DB2

DB2 对隐式类型转换的要求比 Oracle 和 PostgreSQL 更严格，因此需要显式地将 DEPTNO 转换为 CHAR，以确保视图 V 合法。在本解决方案中，没有重新创建视图 V，而是使用了一个内嵌视图。本解决方案中使用 REPLACE 和 TRANSLATE 的方式与 Oracle 和 PostgreSQL 解决方案相同，但 TRANSLATE 的参数顺序稍有不同。

```

/* 根据DEPTNO排序 */

1  select *
2    from (
3     select ename||' '||cast(deptno as char(2)) as data
4     from emp
5     ) v
6   order by replace(data,
7                     replace(
8                       translate(data,'#####','0123456789'),'#',''), '')

/* 根据ENAME排序 */

1  select *
2    from (
3     select ename||' '||cast(deptno as char(2)) as data

```

```

4    from emp
5        ) v
6    order by replace(
7        translate(data,'#####','0123456789'),'#','')

```

MySQL

当前，MySQL 不支持函数 **TRANSLATE**，因此这里没有提供在 MySQL 中解决这个问题的方案。

13. 讨论

函数 **TRANSLATE** 和 **REPLACE** 删除了每一行的字母或数字，这样就能轻松地按数字或字母进行排序了。下面的查询结果显示了传递给 **ORDER BY** 的值。（这里以 Oracle 解决方案为例，因为该解决方案适用于 3 种 DBMS。在 DB2 中，唯一不同的是 **TRANSLATE** 的参数的排列顺序。）

```

select data,
       replace(data,
       replace(
       translate(data,'0123456789','#####'),'#',''),') nums,
       replace(
       translate(data,'0123456789','#####'),'#','') chars
from V

```

DATA	NUMS	CHARS
-----	-----	-----
SMITH 20	20	SMITH
ALLEN 30	30	ALLEN
WARD 30	30	WARD
JONES 20	20	JONES
MARTIN 30	30	MARTIN
BLAKE 30	30	BLAKE
CLARK 10	10	CLARK
SCOTT 20	20	SCOTT
KING 10	10	KING

TURNER 30	30	TURNER
ADAMS 20	20	ADAMS
JAMES 30	30	JAMES
FORD 20	20	FORD
MILLER 10	10	MILLER

2.5 排序时处理NULL值

1. 问题

你想按照 **COMM** 对来自 **EMP** 表的查询结果进行排序，但这一列的值可能为 **NULL**，因此需要指定是否将 **COMM** 列为 **NULL** 的行排在最后面。

ENAME	SAL	COMM
-----	-----	-----
TURNER	1500	0
ALLEN	1600	300
WARD	1250	500
MARTIN	1250	1400
SMITH	800	
JONES	2975	
JAMES	950	
MILLER	1300	
FORD	3000	
ADAMS	1100	
BLAKE	2850	
CLARK	2450	
SCOTT	3000	
KING	5000	

或者，将 **COMM** 列为 **NULL** 的行排在最前面。

ENAME	SAL	COMM
-----	-----	-----
SMITH	800	
JONES	2975	
CLARK	2450	
BLAKE	2850	
SCOTT	3000	
KING	5000	
JAMES	950	
MILLER	1300	
FORD	3000	
ADAMS	1100	

MARTIN	1250	1400
WARD	1250	500
ALLEN	1600	300
TURNER	1500	0

12. 解决方案

根据想要的数据库排序方式以及 RDBMS 处理 NULL 的方式，可以按升序或降序对可能为 NULL 值的列进行排序。

```
1 select ename,sal,comm
2   from emp
3  order by 3

1 select ename,sal,comm
2   from emp
3  order by 3 desc
```

本解决方案能够对列值不为 NULL 的行按升序或降序排列，你可能想到了这一点，也可能没想到。对于列值为 NULL 和列值不为 NULL 的行，如果要以不同的方式排列它们（例如，你要按升序或降序排列列值不为 NULL 的行，并将所有列值为 NULL 的行都放在最后面），那么可以使用 CASE 表达式根据不同的情况采用不同的排序方式。

DB2、MySQL、PostgreSQL 和 SQL Server

使用 CASE 表达式来创建一个指出列值是否为 NULL 的“标志”。这个标志有两个可能的取值，一个表示列值为 NULL，另一个表示列值不为 NULL。创建这个标志列后，只需在 ORDER BY 子句中指定根据它进行排序。这让你能够轻松地指定将列值为 NULL 的行放在最前面还是最后面，同时不影响列值不为 NULL 的行。

```
/* 按照升序排列COMM列不为NULL的行，并将COMM列为NULL的行放在最后面 */
```

```
1 select ename,sal,comm
2   from (
3 select ename,sal,comm,
4        case when comm is null then 0 else 1 end as is_null
5   from emp
6        ) x
7   order by is_null desc,comm
```

ENAME	SAL	COMM
-----	-----	-----
TURNER	1500	0
ALLEN	1600	300
WARD	1250	500
MARTIN	1250	1400
SMITH	800	
JONES	2975	
JAMES	950	
MILLER	1300	
FORD	3000	
ADAMS	1100	
BLAKE	2850	
CLARK	2450	
SCOTT	3000	
KING	5000	

```
/* 按照降序排列COMM列不为NULL的行，并将COMM列为NULL的行放在最后面 */
```

```
1 select ename,sal,comm
2   from (
3 select ename,sal,comm,
4        case when comm is null then 0 else 1 end as is_null
5   from emp
6        ) x
7   order by is_null desc,comm desc
```

ENAME	SAL	COMM
-----	-----	-----
MARTIN	1250	1400
WARD	1250	500
ALLEN	1600	300
TURNER	1500	0
SMITH	800	

JONES	2975
JAMES	950
MILLER	1300
FORD	3000
ADAMS	1100
BLAKE	2850
CLARK	2450
SCOTT	3000
KING	5000

/* 按照升序排列COMM列不为NULL的行，并将COMM列为NULL的行放在最前面 */

```

1 select ename,sal,comm
2   from (
3 select ename,sal,comm,
4        case when comm is null then 0 else 1 end as is_null
5   from emp
6        ) x
7  order by is_null,comm

```

ENAME	SAL	COMM
SMITH	800	
JONES	2975	
CLARK	2450	
BLAKE	2850	
SCOTT	3000	
KING	5000	
JAMES	950	
MILLER	1300	
FORD	3000	
ADAMS	1100	
TURNER	1500	0
ALLEN	1600	300
WARD	1250	500
MARTIN	1250	1400

/* 按照降序排列COMM列不为NULL的行，并将COMM列为NULL的行放在最前面 */

```

1 select ename,sal,comm
2   from (
3 select ename,sal,comm,
4        case when comm is null then 0 else 1 end as is_null
5   from emp
6        ) x

```



```
7 order by is_null,comm desc
```

ENAME	SAL	COMM
-----	-----	-----
SMITH	800	
JONES	2975	
CLARK	2450	
BLAKE	2850	
SCOTT	3000	
KING	5000	
JAMES	950	
MILLER	1300	
FORD	3000	
ADAMS	1100	
MARTIN	1250	1400
WARD	1250	500
ALLEN	1600	300
TURNER	1500	0

Oracle

Oracle 用户既可以使用适用于其他平台的解决方案，也可以使用 Oracle 特有的解决方案。该解决方案利用 **ORDER BY** 子句扩展 **NULLS FIRST** 和 **NULLS LAST**，来分别确保将列值为 **NULL** 的行放在最前面和最后面，而不管列值不为 **NULL** 的行是如何排序的。

```
/* 按照升序排列COMM列不为NULL的行，并将COMM列为NULL的行放在最后面 */
```

```
1 select ename,sal,comm
2   from emp
3  order by comm nulls last
```

ENAME	SAL	COMM
-----	-----	-----
TURNER	1500	0
ALLEN	1600	300
WARD	1250	500
MARTIN	1250	1400
SMITH	800	
JONES	2975	

JAMES	950
MILLER	1300
FORD	3000
ADAMS	1100
BLAKE	2850
CLARK	2450
SCOTT	3000
KING	5000

/* 按照升序排列COMM列不为NULL的行，并将COMM列为NULL的行放在最前面 */

```
1 select ename,sal,comm
2   from emp
3  order by comm nulls first
```

ENAME	SAL	COMM
-----	-----	-----
SMITH	800	
JONES	2975	
CLARK	2450	
BLAKE	2850	
SCOTT	3000	
KING	5000	
JAMES	950	
MILLER	1300	
FORD	3000	
ADAMS	1100	
TURNER	1500	0
ALLEN	1600	300
WARD	1250	500
MARTIN	1250	1400

/* 按照降序排列COMM列不为NULL的行，并将COMM列为NULL的行放在最前面 */

```
1 select ename,sal,comm
2   from emp
3  order by comm desc nulls first
```

ENAME	SAL	COMM
-----	-----	-----
SMITH	800	
JONES	2975	
CLARK	2450	
BLAKE	2850	
SCOTT	3000	

KING	5000	
JAMES	950	
MILLER	1300	
FORD	3000	
ADAMS	1100	
MARTIN	1250	1400
WARD	1250	500
ALLEN	1600	300
TURNER	1500	0

13. 讨论

除非你使用的 RDBMS 像 Oracle 那样，提供了方便的途径，无须修改不为 **NULL** 的列值，就能轻松地将列值为 **NULL** 的行放在最前面或最后面，否则就需要创建一个辅助列。



截至本书撰写之时，DB2 用户可以在位于 **OVER** 子句的 **ORDER BY** 子句中使用 **NULLS FIRST** 和 **NULLS LAST**，但不能在应用于整个结果集的 **ORDER BY** 子句中使用它们。

这个辅助列（只存在于查询语句中，而不存在于表中）让你能够标识列值为 **NULL** 的行，并将它们放在最前面或最后面。在非 Oracle 解决方案中，下面的查询会返回内嵌视图 **X** 包含的结果集。

<pre>select ename,sal,comm, case when comm is null then 0 else 1 end as is_null from emp</pre>			
ENAME	SAL	COMM	IS_NULL
SMITH	800		0
ALLEN	1600	300	1
WARD	1250	500	1

JONES	2975		0
MARTIN	1250	1400	1
BLAKE	2850		0
CLARK	2450		0
SCOTT	3000		0
KING	5000		0
TURNER	1500	0	1
ADAMS	1100		0
JAMES	950		0
FORD	3000		0
MILLER	1300		0

使用 **IS_NULL** 返回的值，可以轻松地将列值为 **NULL** 的行放在最前面或最后面，而不影响根据 **COMM** 列进行排序。

2.6 根据依赖于数据的键进行排序

1. 问题

你想根据某种条件逻辑选择排序依据。如果 **JOB** 列为 **SALESMAN**，就根据 **COMM** 列排序，否则根据 **SAL** 排序。换言之，你想返回如下结果集。

ENAME	SAL	JOB	COMM
TURNER	1500	SALESMAN	0
ALLEN	1600	SALESMAN	300
WARD	1250	SALESMAN	500
SMITH	800	CLERK	
JAMES	950	CLERK	
ADAMS	1100	CLERK	
MILLER	1300	CLERK	
MARTIN	1250	SALESMAN	1400
CLARK	2450	MANAGER	
BLAKE	2850	MANAGER	
JONES	2975	MANAGER	
SCOTT	3000	ANALYST	
FORD	3000	ANALYST	
KING	5000	PRESIDENT	

2. 解决方案

在 **ORDER BY** 子句中使用 **CASE** 表达式。

```
1 select ename,sal,job,comm
2   from emp
3  order by case when job = 'SALESMAN' then comm else sal end
```

13. 讨论

可以使用 **CASE** 表达式动态地修改结果集排序方式。传递给 **ORDER BY** 子句的值如下所示。

```
select ename,sal,job,comm,  
       case when job = 'SALESMAN' then comm else sal end as  
ordered  
  from emp  
 order by 5
```

ENAME	SAL	JOB	COMM	ORDERED
TURNER	1500	SALESMAN	0	0
ALLEN	1600	SALESMAN	300	300
WARD1	250	SALESMAN	500	500
SMITH	800	CLERK		800
JAMES	950	CLERK		950
ADAMS	1100	CLERK		1100
MILLER	1300	CLERK		1300
MARTIN	1250	SALESMAN	1400	1400
CLARK2	450	MANAGER		2450
BLAKE2	850	MANAGER		2850
JONES2	975	MANAGER		2975
SCOTT	3000	ANALYST		3000
FORD	3000	ANALYST		3000
KING	5000	PRESIDENT		5000

2.7 小结

对任何 SQL 用户来说，对查询结果进行排序都是必须掌握的核心技能。**ORDER BY** 子句的功能非常强大，但正如你在本章中看到的，要卓有成效地使用它，必须对一些玄妙之处有深刻认识。本书后面的很多实例依赖于 **ORDER BY** 子句，因此你必须掌握其用法。

第 3 章 使用多张表

本章介绍如何使用连接和集合运算合并来自多张表的数据。连接是 SQL 的基础，集合运算也很重要。要掌握本书后面介绍的复杂查询，必须熟悉连接和集合运算。

3.1 合并多个行集

1. 问题

你想返回存储在多张表中的数据，即将多个结果集合并。这些表并非必须有相同的键，但它们的列的数据类型必须相同。例如，你想显示 **EMP** 表中部门编号为 10 的员工的姓名和部门编号，以及 **DEPT** 表中每个部门的名称和编号。换言之，你希望返回如下结果集。

ENAME_AND_DNAME	DEPTNO
-----	-----
CLARK	10
KING	10
MILLER	10

ACCOUNTING	10
RESEARCH	20
SALES	30
OPERATIONS	40

2. 解决方案

使用集合运算 **UNION ALL** 合并来自多张表的行。

```
1  select ename as ename_and_dname, deptno
2     from emp
3     where deptno = 10
4     union all
5     select '-----', null
6         from t1
7     union all
8     select dname, deptno
9         from dept
```

13. 讨论

UNION ALL 可以将来自多个数据源的行合并为一个结果集。与所有的集合运算一样，在 **SELECT** 子句中指定的列的数量和类型必须匹配。例如，下面两个查询都将以失败告终。

<pre>select deptno from dept union all select ename from emp</pre>	<pre>select deptno, dname from dept union all select deptno from emp</pre>
--	--

需要指出的是，**UNION ALL** 不会剔除重复的行。要剔除重复的行，可以使用运算符 **UNION**。例如，对 **EMP.DEPTNO** 和 **DEPT.DEPTNO** 执行 **UNION** 操作时，只会返回 4 行数据。

<pre>select deptno from emp union select deptno from dept</pre>
<pre>DEPTNO ----- 10 20 30 40</pre>

使用 **UNION**（而不是 **UNION ALL**）时，很可能引发排序操作以消除重复的行。处理大型结果集时，务必牢记这一点。使用 **UNION** 的效果与下面的查询大致相同，该查询对 **UNION ALL** 的输出执行了 **DISTINCT** 操作。

<pre>select distinct deptno</pre>

```
from (  
  select deptno  
    from emp  
  union all  
  select deptno  
    from dept  
)
```

```
  DEPTNO  
-----  
      10  
      20  
      30  
      40
```

除非必要，否则不要在查询中使用 **DISTINCT**。这条规则也适用于 **UNION**：除非必要，否则不要使用 **UNION**，而应该使用 **UNION ALL**。例如，在本书中，为教学而使用的表不多，但在实际场景中，如果查询单张表，则可能有更合适的方式。

3.2 合并相关的行

1. 问题

你想执行基于相同列或相同列值的连接，以返回多张表中的行。例如，你想显示所有部门编号为 10 的员工的姓名以及每位员工所属部门的位置，但这些数据存储在两张表中。换言之，你想返回如下结果集。

ENAME	LOC
-----	-----
CLARK	NEW YORK
KING	NEW YORK
MILLER	NEW YORK

2. 解决方案

基于 DEPTNO 连接 EMP 表和 DEPT 表。

```
1 select e.ename, d.loc
2    from emp e, dept d
3   where e.deptno = d.deptno
4      and e.deptno = 10
```

3. 讨论

上述解决方案使用了连接，准确地说是相等连接——内连接的一种。连接是一种将两张表中的行合并的操作，而相等连接是基于相等条件（比如一张表的部门编号与另一张表的部门编号相等）的连接。内连接是最基本的连接，它返回的每

一行都包含来自参与连接查询的各张表的数据。

从概念上说，为生成结果集，连接首先会创建 **FROM** 子句中指定的表的笛卡儿积（所有可能的行组合）。

<pre>select e.ename, d.loc, e.deptno as emp_deptno, d.deptno as dept_deptno from emp e, dept d where e.deptno = 10</pre>			
ENAME	LOC	EMP_DEPTNO	DEPT_DEPTNO
CLARK	NEW YORK	10	10
KING	NEW YORK	10	10
MILLER	NEW YORK	10	10
CLARK	DALLAS	10	20
KING	DALLAS	10	20
MILLER	DALLAS	10	20
CLARK	CHICAGO	10	30
KING	CHICAGO	10	30
MILLER	CHICAGO	10	30
CLARK	BOSTON	10	40
KING	BOSTON	10	40
MILLER	BOSTON	10	40

这将返回 **EMP** 表中部门编号为 10 的每位员工与 **DEPT** 表中每个部门的组合。然后 **WHERE** 子句中涉及 **e.deptno** 和 **d.deptno**（连接）的表达式会对结果集进行限制，使其只包含 **EMP.DEPTNO** 和 **DEPT.DEPTNO** 相等的行。

<pre>select e.ename, d.loc, e.deptno as emp_deptno, d.deptno as dept_deptno from emp e, dept d where e.deptno = d.deptno and e.deptno = 10</pre>			
ENAME	LOC	EMP_DEPTNO	DEPT_DEPTNO

CLARK	NEW YORK	10	10
KING	NEW YORK	10	10
MILLER	NEW YORK	10	10

另一种解决方案是显式地指定 **JOIN** 子句（关键字 **INNER** 是可选的）。

```
select e.ename, d.loc
  from emp e inner join dept d
    on (e.deptno = d.deptno)
 where e.deptno = 10
```

如果你喜欢在 **FORM** 子句（而不是 **WHERE** 子句）中指定连接逻辑，那么可以使用 **JOIN** 子句。这里介绍的两种风格都符合 **ANSI** 标准，本书提及的所有 **RDBMS** 的最新版本都支持它们。

3.3 查找两张表中相同的行

1. 问题

你想找出两张表中相同的行，但需要连接多列。例如，请看下面的视图 **V**，它是为了教学而使用 **EMP** 表创建的。

<pre>create view V as select ename,job,sal from emp where job = 'CLERK' select * from V</pre>		
ENAME	JOB	SAL
-----	-----	-----
SMITH	CLERK	800
ADAMS	CLERK	1100
JAMES	CLERK	950
MILLER	CLERK	1300

视图 **V** 只包含普通职员，并没有显示 **EMP** 表中所有可能的列。你想返回 **EMP** 表中与视图 **V** 中行匹配的每位员工的 **EMPNO**、**ENAME**、**JOB**、**SAL** 和 **DEPTNO**。换言之，你希望返回如下结果集。

EMPNO	ENAME	JOB	SAL	DEPTNO
-----	-----	-----	-----	-----
7369	SMITH	CLERK	800	20
7876	ADAMS	CLERK	1100	20
7900	JAMES	CLERK	950	30
7934	MILLER	CLERK	1300	10

2. 解决方案

基于必要的列将表连接起来，以返回正确的结果。也可以使用集合运算 **INTERSECT** 来返回两张表的交集（两张表中相同的行），这样可以避免执行连接操作。

MySQL 和 SQL Server

使用多个连接条件将 **EMP** 表和视图 **V** 连接起来。

```
1 select e.empno,e.ename,e.job,e.sal,e.deptno
2   from emp e, V
3  where e.ename = v.ename
4        and e.job  = v.job
5        and e.sal   = v.sal
```

也可以使用 **JOIN** 子句来执行这个连接。

```
1 select e.empno,e.ename,e.job,e.sal,e.deptno
2   from emp e join V
3     on (    e.ename    = v.ename
4           and e.job     = v.job
5           and e.sal     = v.sal )
```

DB2、Oracle 和 PostgreSQL

MySQL 和 SQL Server 解决方案也适用于 DB2、Oracle 和 PostgreSQL。需要返回视图 **V** 中的值时，应该使用该解决方案。

如果不需要返回视图 **V** 中的列，那么可以结合使用集合运算 **INTERSECT** 和谓词 **IN**。

```
1 select empno,ename,job,sal,deptno
2   from emp
3  where (ename,job,sal) in (
4    select ename,job,sal from emp
5  intersect
6    select ename,job,sal from V
```


13. 讨论

执行连接操作时必须基于合适的列，这样才能返回正确的结果。当两张表中有些列的值可能相同，而其他列的值不同时，这一点尤其重要。

集合运算 **INTERSECT** 会返回两个数据源中相同的行。使用 **INTERSECT** 时，必须对两张表中数据类型相同的列进行比较。别忘了，集合运算默认不会返回重复的行。

3.4 从一张表中检索没有出现在另一张表中的值

1. 问题

你想找出一张表（源表）中没有出现在目标表中的值。例如，你想找出 **DEPT** 表中都有哪些部门没有出现在 **EMP** 表中。在本书使用的示例数据库中，**DEPT** 表中的 **DEPTNO 40** 没有出现在 **EMP** 表中，因此结果集如下所示。

DEPTNO

40

2. 解决方案

解决这个问题时，计算差集的函数很有用。DB2、PostgreSQL、SQL Server 和 Oracle 都支持差集运算。如果你使用的 DBMS 没有提供计算差集的函数，则可以像 MySQL 解决方案那样使用子查询。

DB2、PostgreSQL 和 SQL Server

使用集合运算 **EXCEPT**。

```
1 select deptno from dept
2 except
3 select deptno from emp
```

Oracle

使用集合运算 **MINUS**。

```
1 select deptno from dept
2 minus
3 select deptno from emp
```

MySQL

使用子查询将 **EMP** 表中所有的 **DEPTNO** 都返回给外部查询，而外部查询在 **DEPT** 表中查找 **DEPTNO** 没有出现在子查询返回结果中的行。

```
1 select deptno
2   from dept
3  where deptno not in (select deptno from emp)
```

13. 讨论

DB2、PostgreSQL 和 SQL Server

差集函数让这种操作易如反掌。**EXCEPT** 运算符会将出现在第一个结果集中但属于第二个结果集的行都删除。这种操作很像减法运算。

对于包含 **EXCEPT** 在内的集合运算符，存在一定的限制：在两个 **SELECT** 子句中，指定的列的数量和数据类型必须匹配。另外，**EXCEPT** 会剔除重复的行，同时不同于使用 **NOT IN** 的子查询，**NULL** 不会给它带来麻烦（参见有关 **MySQL** 的讨论）。**EXCEPT** 运算符会返回上查询（位于 **EXCEPT** 前面的查询）中没有出现在下查询（位于 **EXCEPT** 后面的查询）中的行。

Oracle

Oracle 解决方案与使用 **EXCEPT** 运算符的解决方案相同，但 Oracle 差集运算符名为 **MINUS**，而不是 **EXCEPT**。除这一点外，前述说明也适用于 Oracle 解决方案。

MySQL

在 MySQL 解决方案中，子查询会返回 **EMP** 表中所有的 **DEPTNO**，而外部查询会返回 **DEPT** 表中未出现（未包含）在子查询返回的结果集中的所有 **DEPTNO**。

使用 MySQL 解决方案时，必须考虑消除重复行的问题。基于 **EXCEPT** 和 **MINUS** 的解决方案会消除结果集中的重复行，确保每个 **DEPTNO** 都只报告一次。当然，在本书的示例数据库中，**DEPTNO** 是主键，因此在 **DEPT** 表中不会重复。如果 **DEPTNO** 不是主键，则可以像下面这样使用 **DISTINCT**，来确保未出现在 **EMP** 表中的每个 **DEPTNO** 值都只报告一次。

```
select distinct deptno
  from dept
 where deptno not in (select deptno from emp)
```

使用 **NOT IN** 时，务必注意 **NULL** 值。请看下面的 **NEW_DEPT** 表。

```
create table new_dept(deptno integer)
insert into new_dept values (10)
insert into new_dept values (50)
insert into new_dept values (null)
```

如果结合子查询和 **NOT IN** 来查找出现在 **DEPT** 表中而没有出现在 **NEW_DEPT** 表中的 **DEPTNO**，你将发现没有返回任何行。

```
select *
  from dept
```

```
where deptno not in (select deptno from new_dept)
```

DEPTNO 20、DEPTNO 30 和 DEPTNO 40 都未出现在 NEW_DEPT 表中，但上述查询并没有返回它们。这是为什么呢？原因是 NEW_DEPT 表中包含 NULL 值。子查询返回了 3 行，它们的 DEPTNO 值分别是 10、50 和 NULL。从本质上说，IN 和 NOT IN 就是 OR 运算，由于逻辑运算符 OR 处理 NULL 值的方式，导致 IN 和 NOT IN 的结果出乎意料。

为弄明白这一点，请看下面的真值表（T=true、F=false、N=null）。

OR	T	F	N
T	T	T	T
F	T	F	N
N	T	N	N

NOT
T
F
N

AND	T	F	N
T	T	F	N
F	F	F	F
N	N	F	N

现在来看一个使用 IN 的示例以及与之等价但使用 OR 的示例。

```
select deptno
  from dept
 where deptno in ( 10,50,null )
```

```

DEPTNO
-----
      10

select deptno
  from dept
 where (deptno=10 or deptno=50 or deptno=null)

DEPTNO
-----
      10

```

为什么只返回了 DEPTNO 10 呢？DEPT 表中有 4 个 DEPTNO（10、20、30 和 40），对于每个 DEPTNO，都将使用谓词（deptno=10 or deptno=50 or deptno=null）对其进行评估。根据前面的真值表，对于每个 DEPTNO（10、20、30 和 40），这个谓词的评估结果如下所示。

```

DEPTNO=10
(deptno=10 or deptno=50 or deptno=null)
= (10=10 or 10=50 or 10=null)
= (T or F or N)
= (T or N)
= (T)

DEPTNO=20
(deptno=10 or deptno=50 or deptno=null)
= (20=10 or 20=50 or 20=null)
= (F or F or N)
= (F or N)
= (N)

DEPTNO=30
(deptno=10 or deptno=50 or deptno=null)
= (30=10 or 30=50 or 30=null)
= (F or F or N)
= (F or N)
= (N)

```

```
DEPTNO=40
(deptno=10 or deptno=50 or deptno=null)
= (40=10 or 40=50 or 40=null)
= (F or F or N)
= (F or N)
= (N)
```

至此，使用 **IN** 和 **OR** 时只返回 **DEPTNO 10** 的原因就显而易见了。接下来，看看使用 **NOT IN** 和 **NOT OR** 的示例。

```
select deptno
  from dept
 where deptno not in ( 10,50,null )

( no rows )

select deptno
  from dept
 where not (deptno=10 or deptno=50 or deptno=null)

( no rows )
```

为什么没有返回任何行呢？下面来看看真值表。

```
DEPTNO=10
NOT (deptno=10 or deptno=50 or deptno=null)
= NOT (10=10 or 10=50 or 10=null)
= NOT (T or F or N)
= NOT (T or N)
= NOT (T)
= (F)

DEPTNO=20
NOT (deptno=10 or deptno=50 or deptno=null)
= NOT (20=10 or 20=50 or 20=null)
= NOT (F or F or N)
= NOT (F or N)
= NOT (N)
= (N)

DEPTNO=30
```

```

NOT (deptno=10 or deptno=50 or deptno=null)
= NOT (30=10 or 30=50 or 30=null)
= NOT (F or F or N)
= NOT (F or N)
= NOT (N)
= (N)

DEPTNO=40
NOT (deptno=10 or deptno=50 or deptno=null)
= NOT (40=10 or 40=50 or 40=null)
= NOT (F or F or N)
= NOT (F or N)
= NOT (N)
= (N)

```

在 SQL 中，TRUE or NULL 的结果为 TRUE，但 FALSE or NULL 的结果为 NULL！使用谓词 IN 或执行逻辑 OR 运算时，如果涉及 NULL 值，务必牢记这一点。

为了避免 NULL 给 NOT IN 带来的问题，可以结合使用关联子查询和 NOT EXISTS。为什么叫关联子查询呢？这是因为在子查询中引用了外部查询返回的行。下面的示例演示了一种不受 NULL 值影响的解决方案（有关原始查询，请参阅本节“问题”部分）。

```

select d.deptno
  from dept d
 where not exists (
    select 1
      from emp e
     where d.deptno = e.deptno
  )

DEPTNO
-----
40

select d.deptno
  from dept d
 where not exists (

```



```
select 1
  from new_dept nd
 where d.deptno = nd.deptno
)

DEPTNO
-----
30
40
20
```

从概念上讲，该解决方案中的外部查询考虑了 **DEPT** 表中的每一行。对于 **DEPT** 表中的每一行，都将做如下处理。

- a. 执行子查询，看看该部门编号是否出现在了 **EMP** 表中。
请注意，条件 **D.DEPTNO = E.DEPTNO** 会比较两张表中的部门编号。
- b. 如果子查询返回了结果，那么 **EXISTS (...)** 将为 **TRUE**，而 **NOT EXISTS (...)** 将为 **FALSE**，因此丢弃外部查询的话，当前检查的行将被丢弃。
- c. 如果子查询没有返回结果，那么 **NOT EXISTS (...)** 将为 **TRUE**，因此将返回外部查询当前检查的行（因为该行中的部门编号未出现在 **EMP** 表中）。

结合使用关联子查询和 **EXISTS/NOT EXISTS** 时，关联子查询中 **SELECT** 子句列出的内容无关紧要。有鉴于此，我们使用了 **SELECT 1**，旨在让你将注意力放在关联子查询中的连接上，而不是 **SELECT** 子句的内容列表中。

3.5 从一张表中检索在另一张表中没有对应行的行

1. 问题

有两张包含相同键的表，你想从一张表中找出在另一张表中没有与之匹配的行。例如，你想确定哪个部门没有员工，结果集如下所示。

DEPTNO	DNAME	LOC
-----	-----	-----
40	OPERATIONS	BOSTON

如果想确定每个员工所属的部门，就需要在 **EMP** 表和 **DEPT** 表之间建立基于 **DEPTNO** 的相等连接。**DEPTNO** 列是这两张表中都有的值。可惜相等连接无法让你知道哪个部门没有员工，因为在 **EMP** 表和 **DEPT** 表之间建立相等连接时，将返回满足连接条件的所有行，而你想知道的是 **DEPT** 表中不满足连接条件的行。

这个问题与前一个问题之间的差别很细微，因此乍一看它们好像是相同的。差别在于，前一个实例要获得的是未出现在 **EMP** 表中的部门编号列表。然而，本实例可以轻松地返回 **DEPT** 表中的其他列：除了部门编号，还可以返回其他列。

2. 解决方案

返回一张表中的所有行，以及在另一张表中可能有匹配行也可能没有匹配行的行。然后，只留下没有匹配行的行。

DB2、MySQL、PostgreSQL 和 SQL Server

使用外连接并执行基于 NULL 的筛选（关键字 OUTER 是可选的）。

```
1 select d.*
2   from dept d left outer join emp e
3     on (d.deptno = e.deptno)
4  where e.deptno is null
```

13. 讨论

以上解决方案先使用了外连接，然后只保留那些没有匹配行的行。这种操作有时被称为反连接（anti-join）。要想更深入地了解反连接的工作原理，先来看一下执行基于 NULL 筛选前的结果集。

```
select e.ename, e.deptno as emp_deptno, d.*
  from dept d left join emp e
    on (d.deptno = e.deptno)
```

ENAME	EMP_DEPTNO	DEPTNO	DNAME	LOC
SMITH	20	20	RESEARCH	DALLAS
ALLEN	30	30	SALES	CHICAGO
WARD	30	30	SALES	CHICAGO
JONES	20	20	RESEARCH	DALLAS
MARTIN	30	30	SALES	CHICAGO
BLAKE	30	30	SALES	CHICAGO
CLARK	10	10	ACCOUNTING	NEW YORK
SCOTT	20	20	RESEARCH	DALLAS
KING	10	10	ACCOUNTING	NEW YORK
TURNER	30	30	SALES	CHICAGO
ADAMS	20	20	RESEARCH	DALLAS
JAMES	30	30	SALES	CHICAGO
FORD	20	20	RESEARCH	DALLAS
MILLER	10	10	ACCOUNTING	NEW YORK
		40	OPERATIONS	BOSTON

注意，在最后一行中，**EMP.ENAME** 和 **EMP_DEPTNO** 的值都是 **NULL**。这是因为编号为 40 的部门没有员工。为了只保留 **EMP_DEPTNO** 为 **NULL** 的行（只在 **DEPT** 表中保留在 **EMP** 表中没有与之匹配的行），该解决方案使用了 **WHERE** 子句。

3.6 在查询中添加连接并确保不影响其他连接

1. 问题

你有一个查询，它可以返回你想要的结果。你需要获取其他信息，但尝试这样做时，结果集中少了原本该有的数据。例如，你想返回每位员工、他们所属部门的位置以及他们获得奖金的日期。这个问题需要用到包含如下数据的 **EMP_BONUS** 表。

```
select * from emp_bonus
```

EMPNO	RECEIVED	TYPE
7369	14-MAR-2005	1
7900	14-MAR-2005	2
7788	14-MAR-2005	3

当前的查询如下所示。

```
select e.ename, d.loc
  from emp e, dept d
 where e.deptno=d.deptno
```

ENAME	LOC
SMITH	DALLAS
ALLEN	CHICAGO
WARD	CHICAGO
JONES	DALLAS
MARTIN	CHICAGO
BLAKE	CHICAGO
CLARK	NEW YORK
SCOTT	DALLAS
KING	NEW YORK
TURNER	CHICAGO
ADAMS	DALLAS
JAMES	CHICAGO

FORD	DALLAS
MILLER	NEW YORK

你希望查询结果中包含员工获得奖金的日期，为此连接到了 **EMP_BONUS** 表，但返回的行数更少了，因为并非每位员工都获得过奖金。

<pre>select e.ename, d.loc, eb.received from emp e, dept d, emp_bonus eb where e.deptno=d.deptno and e.empno=eb.empno</pre>		
ENAME	LOC	RECEIVED
-----	-----	-----
SCOTT	DALLAS	14-MAR-2005
SMITH	DALLAS	14-MAR-2005
JAMES	CHICAGO	14-MAR-2005

你希望得到如下结果集。

ENAME	LOC	RECEIVED
-----	-----	-----
ALLEN	CHICAGO	
WARD	CHICAGO	
MARTIN	CHICAGO	
JAMES	CHICAGO	14-MAR-2005
TURNER	CHICAGO	
BLAKE	CHICAGO	
SMITH	DALLAS	14-MAR-2005
FORD	DALLAS	
ADAMS	DALLAS	
JONES	DALLAS	
SCOTT	DALLAS	14-MAR-2005
CLARK	NEW YORK	
KING	NEW YORK	
MILLER	NEW YORK	

12. 解决方案

可以使用外连接来获得额外的信息，同时避免返回的数据比原来的查询少。先将 **EMP** 表连接到 **DEPT** 表，以返回所有的员工及其所在的部门，然后外连接到 **EMP_BONUS** 表，以返回员工获得奖金的日期。下面的语法适用于 **DB2**、**MySQL**、**PostgreSQL** 和 **SQL Server**。

```
1 select e.ename, d.loc, eb.received
2   from emp e join dept d
3     on (e.deptno=d.deptno)
4  left join emp_bonus eb
5     on (e.empno=eb.empno)
6  order by 2
```

也可以使用标量子查询（放在 **SELECT** 列表中的子查询）来模拟外连接。

```
1 select e.ename, d.loc,
2       (select eb.received from emp_bonus eb
3        where eb.empno=e.empno) as received
4   from emp e, dept d
5  where e.deptno=d.deptno
6  order by 2
```

使用标量子查询的解决方案适用于所有平台。

13. 讨论

外连接可以返回一张表中的所有行以及另一张表中与之匹配的行。前一个实例也使用了这种连接。为什么使用外连接能够解决这个问题呢？这是因为它不会删除任何原本返回了的行。查询将返回添加外连接前被返回的所有行。它还会返回获得奖金的日期（如果获得过奖金的话）。

对于这种问题，使用标量子查询也是一种便利的解决方案，

因为不需要修改主查询中正确的既有连接。使用标量子查询是一种简易方式，可以在不破坏既有结果集的情况下添加额外的数据。使用标量子查询时，必须确保它们返回标量值（单个值）。如果 **SELECT** 列表中的子查询返回多行，那么将导致错误。

14. 另请参阅

如果想绕开 **SELECT** 列表中的子查询不能返回多行的问题，请参阅 14.10 节。

3.7 判断两张表包含的数据是否相同

1. 问题

你想知道两张表或两个视图中包含的数据（包括基数和值）是否相同。请看下面的视图。

<pre>create view V as select * from emp where deptno != 10 union all select * from emp where ename = 'WARD' select * from V</pre>							
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-2005	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-2006	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-2006	1250	500	30
7566	JONES	MANAGER	7839	02-APR-2006	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-2006	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-2006	2850		30
7788	SCOTT	ANALYST	7566	09-DEC-2007	3000		20
7844	TURNER	SALESMAN	7698	08-SEP-2006	1500	0	30
7876	ADAMS	CLERK	7788	12-JAN-2008	1100		20
7900	JAMES	CLERK	7698	03-DEC-2006	950		30
7902	FORD	ANALYST	7566	03-DEC-2006	3000		20
7521	WARD	SALESMAN	7698	22-FEB-2006	1250	500	30

你想确定这个视图是否与 **EMP** 表包含完全相同的数据。这里复制了表示员工 **WARD** 的行，旨在证明此处提供的解决方案不仅能显示不同的数据，还能显示重复的数据。根据 **EMP** 表中包含的行可知，二者的不同之处包括 3 行表示部门编号为 10 的员工的数据以及两行表示员工 **WARD** 的数据。你要返回的结果集如下所示。

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	
CNT								
---	-----	-----	-----	-----	-----	-----	-----	
1	7521	WARD	SALESMAN	7698	22-FEB-2006	1250	500	30
2	7521	WARD	SALESMAN	7698	22-FEB-2006	1250	500	30
1	7782	CLARK	MANAGER	7839	09-JUN-2006	2450		10
1	7839	KING	PRESIDENT		17-NOV-2006	5000		10
1	7934	MILLER	CLERK	7782	23-JAN-2007	1300		10

12. 解决方案

根据你所使用的 DBMS，可以使用执行差集计算的函数 **MINUS** 或 **EXCEPT** 相对轻松地解决比较表中数据的问题。如果你所使用的 DBMS 中没有提供这样的函数，则可以使用关联子查询。

DB2 和 PostgreSQL

使用集合运算 **EXCEPT** 计算视图 **V** 和 **EMP** 表的差集以及 **EMP** 表和视图 **V** 的差集，然后使用集合运算 **UNION ALL** 合并这两个差集。

```

1 (
2   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
3          count(*) as cnt
4   from V
5   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
6 except
7 select empno,ename,job,mgr,hiredate,sal,comm,deptno,
8          count(*) as cnt
9   from emp

```

```

10  group by empno,ename,job,mgr,hiredate,sal,comm,deptno
11  )
12  union all
13  (
14  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
15         count(*) as cnt
16    from emp
17   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
18  except
19  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
20         count(*) as cnt
21    from v
22   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
23  )

```

Oracle

使用集合运算 **MINUS** 计算视图 **V** 和 **EMP** 表的差集以及 **EMP** 表和视图 **V** 的差集，然后使用集合运算 **UNION ALL** 合并这两个差集。

```

1  (
2  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
3         count(*) as cnt
4    from v
5   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
6  minus
7  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
8         count(*) as cnt
9    from emp
10   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
11  )
12  union all
13  (
14  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
15         count(*) as cnt
16    from emp
17   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
18  minus
19  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
20         count(*) as cnt
21    from v

```

```
22 group by empno,ename,job,mgr,hiredate,sal,comm,deptno
23 )
```

MySQL 和 SQL Server

使用关联子查询找出位于视图 **V** 中但不位于 **EMP** 表中的行，以及位于 **EMP** 表中但不位于视图 **V** 中的行，然后使用 **UNION ALL** 合并这些行。

```
1 select *
2   from (
3 select e.empno,e.ename,e.job,e.mgr,e.hiredate,
4        e.sal,e.comm,e.deptno, count(*) as cnt
5   from emp e
6  group by empno,ename,job,mgr,hiredate,
7          sal,comm,deptno
8        ) e
9  where not exists (
10 select null
11   from (
12 select v.empno,v.ename,v.job,v.mgr,v.hiredate,
13        v.sal,v.comm,v.deptno, count(*) as cnt
14   from v
15  group by empno,ename,job,mgr,hiredate,
16          sal,comm,deptno
17        ) v
18   where v.empno      = e.empno
19        and v.ename    = e.ename
20        and v.job      = e.job
21        and coalesce(v.mgr,0) = coalesce(e.mgr,0)
22        and v.hiredate = e.hiredate
23        and v.sal      = e.sal
24        and v.deptno   = e.deptno
25        and v.cnt      = e.cnt
26        and coalesce(v.comm,0) = coalesce(e.comm,0)
27 )
28 union all
29 select *
30   from (
31 select v.empno,v.ename,v.job,v.mgr,v.hiredate,
32        v.sal,v.comm,v.deptno, count(*) as cnt
33   from v
```

```

34     group by empno,ename,job,mgr,hiredate,
35             sal,comm,deptno
36         ) v
37     where not exists (
38     select null
39         from (
40     select e.empno,e.ename,e.job,e.mgr,e.hiredate,
41            e.sal,e.comm,e.deptno, count(*) as cnt
42         from emp e
43         group by empno,ename,job,mgr,hiredate,
44                sal,comm,deptno
45         ) e
46     where v.empno      = e.empno
47           and v.ename   = e.ename
48           and v.job     = e.job
49           and coalesce(v.mgr,0) = coalesce(e.mgr,0)
50           and v.hiredate = e.hiredate
51           and v.sal      = e.sal
52           and v.deptno   = e.deptno
53           and v.cnt      = e.cnt
54           and coalesce(v.comm,0) = coalesce(e.comm,0)
55 )

```

13. 讨论

虽然使用的方法不同，但所有解决方案都基于相同的理念。

- a. 找出包含在 **EMP** 表中但未包含在视图 **V** 中的行。
- b. 使用 **UNION ALL** 将这些行与包含在视图 **V** 中但未包含在 **EMP** 表中的行合并。

如果两张表中包含的数据完全相同，那么将不会返回任何行；如果两张表中包含的数据不同，则将返回导致它们不同的行。在比较两张表的时候，可以先比较它们的基数，然后再比较其所包含的数据。

下面的查询是比较基数的一个简单示例，适用于所有

DBMS。

```
select count(*)
  from emp
 union
select count(*)
  from dept
```

COUNT(*)

4

14

由于 **UNION** 会剔除重复行，因此如果两张表的基数相同，那么将只返回一行数据。本实例返回了两行数据，这表明这两张表包含的行集不同。

DB2、Oracle 和 PostgreSQL

MINUS 和 **EXCEPT** 的工作原理相同，这里的讨论将以 **EXCEPT** 为例。**UNION ALL** 前后的查询很相似，要理解该解决方案的工作原理，执行 **UNION ALL** 前面的查询即可。执行“解决方案”中的第 1~11 行时，生成的结果集如下所示。

```
(
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from V
   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
 except
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from emp
   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
)
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
CNT							

--							
7521	WARD	SALESMAN	7698	22-FEB-2006	1250	500	30
2							

上述结果集表明，在视图 **V** 中，有一行未包含在 **EMP** 表中，或者虽然这行也包含在 **EMP** 表中，但基数不同。在本例中，查询找到并返回了表示员工 **WARD** 的重复行。如果你还是不明白这个结果集是如何生成的，请分别运行 **EXCEPT** 前后的查询。你将发现在这两个查询返回的结果集之间，唯一的不同是员工 **WARD** 的 **CNT** 值。

UNION ALL 后面的查询所做的工作与 **UNION ALL** 前面的查询相反，它会返回包含在 **EMP** 表中但未包含在视图 **V** 中的行。

<pre>(select empno,ename,job,mgr,hiredate,sal,comm,deptno, count(*) as cnt from emp group by empno,ename,job,mgr,hiredate,sal,comm,deptno minus select empno,ename,job,mgr,hiredate,sal,comm,deptno, count(*) as cnt from v group by empno,ename,job,mgr,hiredate,sal,comm,deptno)</pre>							
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
CNT							

-							
7521	WARD	SALESMAN	7698	22-FEB-2006	1250	500	30
1							
7782	CLARK	MANAGER	7839	09-JUN-2006	2450		10
1							
7839	KING	PRESIDENT		17-NOV-2006	5000		10
1							
7934	MILLER	CLERK	7782	23-JAN-2007	1300		10
1							

UNION ALL 会将这两个查询的结果合并以生成最终的结果集。

MySQL 和 SQL Server

UNION ALL 前后的查询很相似。要弄明白基于子查询的解决方案的工作原理，只需执行 UNION ALL 前面的查询。下面列出了“解决方案”中第 1~27 行的查询。

```
select *
  from (
    select e.empno,e.ename,e.job,e.mgr,e.hiredate,
           e.sal,e.comm,e.deptno, count(*) as cnt
      from emp e
     group by empno,ename,job,mgr,hiredate,
              sal,comm,deptno
        ) e
 where not exists (
select null
  from (
select v.empno,v.ename,v.job,v.mgr,v.hiredate,
       v.sal,v.comm,v.deptno, count(*) as cnt
  from v
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
        ) v
  where v.empno      = e.empno
     and v.ename      = e.ename
     and v.job         = e.job
     and v.mgr         = e.mgr
     and v.hiredate    = e.hiredate
     and v.sal         = e.sal
     and v.deptno      = e.deptno
     and v.cnt         = e.cnt
     and coalesce(v.comm,0) = coalesce(e.comm,0)
)

```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO	CNT

-								

7521	WARD	SALESMAN	7698	22-FEB-2006	1250	500	30
1							
7782	CLARK	MANAGER	7839	09-JUN-2006	2450		10
1							
7839	KING	PRESIDENT		17-NOV-2006	5000		10
1							
7934	MILLER	CLERK	7782	23-JAN-2007	1300		10
1							


注意，这里比较的不是 **EMP** 表和视图 **V**，而是内嵌视图 **E** 和内嵌视图 **V**。首先，计算出每一行的基数，并将其作为该行的一个属性返回。我们要比较的是每一行的数据及其出现的次数。如果你还是不明白这里的比较操作的工作原理，请分别运行各个子查询。然后，找出包含在内嵌视图 **E** 中但未包含在内嵌视图 **V** 中的所有行（包括 **CNT**）。该比较操作是使用关联子查询和 **NOT EXISTS** 完成的。连接查询将用于确定哪些行是相同的，结果为包含在内嵌视图 **E** 中但未被连接查询返回的行。**UNION ALL** 后面的查询执行了相反的操作，它找出了包含在内嵌视图 **V** 中但未包含在内嵌视图 **E** 中的所有行。

```
select *
  from (
select v.empno,v.ename,v.job,v.mgr,v.hiredate,
      v.sal,v.comm,v.deptno, count(*) as cnt
  from v
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
        ) v
 where not exists (
select null
  from (
select e.empno,e.ename,e.job,e.mgr,e.hiredate,
      e.sal,e.comm,e.deptno, count(*) as cnt
  from emp e
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
        ) e
 where v.empno      = e.empno
```

and v.ename = e.ename and v.job = e.job and v.mgr = e.mgr and v.hiredate = e.hiredate and v.sal = e.sal and v.deptno = e.deptno and v.cnt = e.cnt and coalesce(v.comm,0) = coalesce(e.comm,0))							
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
CNT							

-							
7521	WARD	SALESMAN	7698	22-FEB-2006	1250	500	30
2							

最后，UNION ALL 会将这两个结果合并，生成最终的结果集。

 Ales Spetic 和 Jonathan Gennick 在其著作 *Transact-SQL Cookbook* 中，给出了另一种解决方案，详情请参阅该书第 2 章“Comparing Two Sets for Equality”一节。

3.8 识别并避免笛卡儿积

1. 问题

你想返回部门编号为 10 的所有员工的姓名以及这个部门的位置。下面的查询返回的数据是错误的。

```
select e.ename, d.loc
  from emp e, dept d
 where e.deptno = 10
```

ENAME	LOC
-----	-----
CLARK	NEW YORK
CLARK	DALLAS
CLARK	CHICAGO
CLARK	BOSTON
KING	NEW YORK
KING	DALLAS
KING	CHICAGO
KING	BOSTON
MILLER	NEW YORK
MILLER	DALLAS
MILLER	CHICAGO
MILLER	BOSTON

正确的结果集如下所示。

ENAME	LOC
-----	-----
CLARK	NEW YORK
KING	NEW YORK
MILLER	NEW YORK

2. 解决方案

在 **FROM** 子句中的表之间执行连接，以返回正确的结果集。

```
1 select e.ename, d.loc
2   from emp e, dept d
3  where e.deptno = 10
4     and d.deptno = e.deptno
```

13. 讨论

下面来看看 **DEPT** 表中的数据。

```
select * from dept
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

由以上数据可知，编号为 10 的部门位于美国纽约，因此你便知道，在返回的员工记录中，部门位置不是美国纽约的都不正确。错误查询返回的行数为 **FROM** 子句指定的两张表的基数的乘积。在这个查询中，使用部门编号 10 对 **EMP** 表进行筛选的结果为 3 行。由于没有对 **DEPT** 表进行筛选，因此将返回该表的所有行（4 行）。3 乘 4 等于 12，因此这个查询将返回 12 行数据。通常而言，要避免笛卡儿积，需要应用 $n-1$ 条规则，其中 n 为 **FROM** 子句中指定的表的个数。因此，要避免笛卡儿积，至少需要 $n-1$ 次连接。根据表的键和用于连接的列，需要的连接可能远远超过 $n-1$ 次，但编写查询时，先考虑使用 $n-1$ 次连接是不错的选择。



在使用得当的情况下，笛卡儿积很有用。笛卡儿积

常见的用途包括结果集转置（行列转换）、生成值序列和模拟循环，其中后两项也可以使用递归 CTE 来完成。

3.9 同时使用连接和聚合

1. 问题

你想执行聚合操作，但查询涉及多张表，因此需要确保连接不影响聚合。例如，你需要计算部门编号为 10 的所有员工的薪水总额以及奖金总额。有些员工有多笔奖金，但连接 **EMP** 表和 **EMP_BONUS** 表将导致聚合函数 **SUM** 返回的值不正确。这里涉及的 **EMP_BONUS** 表包含如下数据。

```
select * from emp_bonus
```

EMPNO	RECEIVED	TYPE
7934	17-MAR-2005	1
7934	15-FEB-2005	2
7839	15-FEB-2005	3
7782	15-FEB-2005	1

下面的查询将返回部门编号为 10 的所有员工的薪水和奖金。**EMP_BONUS.TYPE** 决定了奖金的金额：1 类奖金为员工薪水的 10%，2 类奖金为员工薪水的 20%，3 类奖金为员工薪水的 30%。

```
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
                  else .3
                end as bonus
from emp e, emp_bonus eb
where e.empno = eb.empno
      and e.deptno = 10
```

EMPNO	ENAME	SAL	DEPTNO	BONUS
-------	-------	-----	--------	-------

7934	MILLER	1300	10	130
7934	MILLER	1300	10	260
7839	KING	5000	10	1500
7782	CLARK	2450	10	245

到目前为止，一切顺利。然而，当你试图连接到 EMP_BONUS 表以计算奖金总额时，问题便出现了。

```

select deptno,
       sum(sal) as total_sal,
       sum(bonus) as total_bonus
  from (
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
                  else .3
                end as bonus
  from emp e, emp_bonus eb
 where e.empno = eb.empno
    and e.deptno = 10
    ) x
 group by deptno

```

DEPTNO	TOTAL_SAL	TOTAL_BONUS
10	10050	2135

TOTAL_BONUS 是正确的，TOTAL_SAL 则不正确。部门编号为 10 的所有员工的薪水总额应为 8750 美元，如下面的查询所示。

```

select sum(sal) from emp where deptno=10

```

SUM(SAL)
8750

为什么 **TOTAL_SAL** 不正确呢？这是因为连接导致有些员工的记录被重复了多次。请看下面的查询，它连接了 **EMP** 表和 **EMP_BONUS** 表。

```
select e.ename,
       e.sal
  from emp e, emp_bonus eb
 where e.empno = eb.empno
       and e.deptno = 10
```

ENAME	SAL
CLARK	2450
KING	5000
MILLER	1300
MILLER	1300

从中可以清楚地看到 **TOTAL_SAL** 不正确的原因：**MILLER** 的薪水被计算了两次。你希望得到的最终结果集如下所示。

DEPTNO	TOTAL_SAL	TOTAL_BONUS
10	8750	2135

12. 解决方案

同时使用连接和聚合时，必须非常小心。当连接导致相同的数据被返回多次时，为了避免聚合函数执行错误的计算，通常有两种方法。一种方法是在调用聚合函数时使用关键字 **DISTINCT**，这样计算时相同的值将只计算一次；另一种方法是在连接前先执行聚合（在内嵌视图中），这样可以避免聚合函数执行错误的计算，因为聚合发生在了连接之前。下面的解决方案使用了关键字 **DISTINCT**。有关如何在连接前使

用内嵌视图执行聚合，将在接下来的“讨论”中介绍。

MySQL 和 PostgreSQL

使用关键字 **DISTINCT** 避免重复计算薪水。

```
1 select deptno,
2       sum(distinct sal) as total_sal,
3       sum(bonus) as total_bonus
4   from (
5   select e.empno,
6          e.ename,
7          e.sal,
8          e.deptno,
9          e.sal*case when eb.type = 1 then .1
10                    when eb.type = 2 then .2
11                    else .3
12                    end as bonus
13   from emp e, emp_bonus eb
14   where e.empno = eb.empno
15         and e.deptno = 10
16         ) x
17  group by deptno
```

DB2、Oracle 和 SQL Server

这些平台支持上面的解决方案，但也支持另一种解决方案，即使用窗口函数 **SUM OVER**。

```
1 select distinct deptno,total_sal,total_bonus
2   from (
3   select e.empno,
4          e.ename,
5          sum(distinct e.sal) over
6          (partition by e.deptno) as total_sal,
7          e.deptno,
8          sum(e.sal*case when eb.type = 1 then .1
9                    when eb.type = 2 then .2
10                    else .3 end) over
11          (partition by deptno) as total_bonus
```

```
12    from emp e, emp_bonus eb
13    where e.empno = eb.empno
14          and e.deptno = 10
15          ) x
```

13. 讨论

MySQL 和 PostgreSQL

本实例“问题”部分的第二个查询连接了 **EMP** 表和 **EMP_BONUS** 表，因此对于员工 **MILLER**，它会返回两行数据，这是导致 **EMP.SAL** 总和不正确的“罪魁祸首”（**MILLER** 的薪水被计算了两次）。解决办法很简单，就是计算查询返回的 **EMP.SAL** 总和时，剔除重复的值。下面的查询提供了另一种解决方案（在要汇总的列可能包含重复值时需要这样做），它先根据 **EMP** 表计算出部门编号为 10 的所有员工的薪水总额，然后再将返回的结果集连接到 **EMP_BONUS** 表。

下面的查询适用于所有 DBMS。

```
select d.deptno,
       d.total_sal,
       sum(e.sal*case when eb.type = 1 then .1
                      when eb.type = 2 then .2
                      else .3 end) as total_bonus
  from emp e,
       emp_bonus eb,
       (
select deptno, sum(sal) as total_sal
  from emp
 where deptno = 10
 group by deptno
       ) d
 where e.deptno = d.deptno
       and e.empno = eb.empno
 group by d.deptno,d.total_sal
```

DEPTNO	TOTAL_SAL	TOTAL_BONUS
-----	-----	-----
10	8750	2135

DB2、Oracle 和 SQL Server

另一种解决方案是利用窗口函数 **SUM OVER**。下面的查询为 DB2、Oracle 和 SQL Server“解决方案”的第 3~14 行。在该查询的后面，列出了它返回的结果集。

<pre> select e.empno, e.ename, sum(distinct e.sal) over (partition by e.deptno) as total_sal, e.deptno, sum(e.sal*case when eb.type = 1 then .1 when eb.type = 2 then .2 else .3 end) over (partition by deptno) as total_bonus from emp e, emp_bonus eb where e.empno = eb.empno and e.deptno = 10 </pre>				
EMPNO	ENAME	TOTAL_SAL	DEPTNO	TOTAL_BONUS
-----	-----	-----	-----	-----
7934	MILLER	8750	10	2135
7934	MILLER	8750	10	2135
7782	CLARK	8750	10	2135
7839	KING	8750	10	2135

窗口函数 **SUM OVER** 被调用了两次，第一次调用 **SUM OVER** 时，计算的是指定分区（分组）的不同薪水总额。在本例中，分区为 **DEPTNO 10**，不同薪水总额为 8750 美元。第二次调用 **SUM OVER** 时，计算的是前述分区的奖金总额。最终的结果集是通过获取不同的 **TOTAL_SAL**、**DEPTNO** 和 **TOTAL_BONUS** 的值得到的。

3.10 同时使用外连接和聚合

1. 问题

本节的问题与 3.9 节相同，但对 **EMP_BONUS** 表做了修改，使得并非部门编号为 10 的每位员工都有奖金。下面列出了修改后的 **EMP_BONUS** 表的内容，以及一个错误的查询，该查询试图计算部门编号为 10 的所有员工的薪水总额以及奖金总额。

select * from emp_bonus		
EMPNO	RECEIVED	TYPE
-----	-----	-----
7934	17-MAR-2005	1
7934	15-FEB-2005	2
select deptno,		
sum(sal) as total_sal,		
sum(bonus) as total_bonus		
from (
select e.empno,		
e.ename,		
e.sal,		
e.deptno,		
e.sal*case when eb.type = 1 then .1		
when eb.type = 2 then .2		
else .3 end as bonus		
from emp e, emp_bonus eb		
where e.empno = eb.empno		
and e.deptno = 10		
)		
group by deptno		
DEPTNO	TOTAL_SAL	TOTAL_BONUS
-----	-----	-----
10	2600	390

TOTAL_BONUS 是正确的，但 **TOTAL_SAL** 并不是部门编号为

10 的所有员工的薪水总额。下面的查询说明了 TOTAL_SAL 不正确的原因。

<pre>select e.empno, e.ename, e.sal, e.deptno, e.sal*case when eb.type = 1 then .1 when eb.type = 2 then .2 else .3 end as bonus from emp e, emp_bonus eb where e.empno = eb.empno and e.deptno = 10</pre>				
EMPNO	ENAME	SAL	DEPTNO	BONUS
-----	-----	-----	-----	-----
7934	MILLER	1300	10	130
7934	MILLER	1300	10	260

以上查询计算的并不是部门编号为 10 的所有员工的薪水总额，而是 MILLER 的薪水（还错误地将其薪水计算了两次）。你原本想返回的结果集如下所示。

DEPTNO	TOTAL_SAL	TOTAL_BONUS
-----	-----	-----
10	8750	390

12. 解决方案

下面的解决方案也与 3.9 节类似，但为涵盖部门编号为 10 的所有员工，这里将外连接到 EMP_BONUS 表。

DB2、MySQL、PostgreSQL 和 SQL Server

外连接到 EMP_BONUS 表，然后以剔除重复项的方式计算部

门编号为 10 的所有员工的薪水总额。

```
1 select deptno,
2         sum(distinct sal) as total_sal,
3         sum(bonus) as total_bonus
4   from (
5 select e.empno,
6        e.ename,
7        e.sal,
8        e.deptno,
9        e.sal*case when eb.type is null then 0
10                  when eb.type = 1 then .1
11                  when eb.type = 2 then .2
12                  else .3 end as bonus
13   from emp e left outer join emp_bonus eb
14     on (e.empno = eb.empno)
15  where e.deptno = 10
16        )
17  group by deptno
```

也可以使用窗口函数 **SUM OVER**。

```
1 select distinct deptno,total_sal,total_bonus
2   from (
3 select e.empno,
4        e.ename,
5        sum(distinct e.sal) over
6        (partition by e.deptno) as total_sal,
7        e.deptno,
8        sum(e.sal*case when eb.type is null then 0
9                  when eb.type = 1 then .1
10                 when eb.type = 2 then .2
11                 else .3
12                end) over
13        (partition by deptno) as total_bonus
14   from emp e left outer join emp_bonus eb
15     on (e.empno = eb.empno)
16  where e.deptno = 10
17        ) x
```

13. 讨论

本实例“问题”部分的第二个查询连接了 **EMP** 表和 **EMP_BONUS** 表，因此只返回了与员工 **MILLER** 相关的行，导致 **EMP.SAL** 总和不正确。（由于部门编号为 10 的其他员工没有奖金，因此薪水总额中不包括这些员工的薪水。）解决办法是将 **EMP** 表外连接到 **EMP_BONUS** 表，这样即便员工没有奖金，与之相关的行也将包含在结果中。如果一个员工没有奖金，那么 **EMP_BONUS.TYPE** 列将为 **NULL**。注意到这一点很重要，因此我们修改了 **CASE** 语句，使其与 3.9 节稍有不同。如果 **EMP_BONUS.TYPE** 为 **NULL**，那么 **CASE** 表达式将返回 0，这对奖金总额没有任何影响。

下面的查询是另一种解决方案。它先根据 **EMP** 表计算部门编号为 10 的所有员工的薪水总额，然后再将返回的结果集连接到 **EMP_BONUS** 表（因此无须使用外连接）。这个查询适用于所有 **DBMS**。

```
select d.deptno,
       d.total_sal,
       sum(e.sal*case when eb.type = 1 then .1
                      when eb.type = 2 then .2
                      else .3 end) as total_bonus
  from emp e,
       emp_bonus eb,
  (
select deptno, sum(sal) as total_sal
  from emp
 where deptno = 10
 group by deptno
  ) d
 where e.deptno = d.deptno
       and e.empno = eb.empno
 group by d.deptno,d.total_sal
```

DEPTNO	TOTAL_SAL	TOTAL_BONUS
-----	-----	-----

10

8750

390

3.11 返回多张表中不匹配的行

1. 问题

你想返回多张表中不匹配的行。要返回 **DEPT** 表中不与 **EMP** 表中任何行匹配的行（没有任何员工的部门），需要使用外连接。请看下面的查询，它返回了 **DEPT** 表中所有的 **DEPTNO** 和 **DNAME**，以及每个部门所有员工的姓名（如果该部门有员工的话）。

```
select d.deptno,d.dname,e.ename
  from dept d left outer join emp e
    on (d.deptno=e.deptno)
```

DEPTNO	DNAME	ENAME
20	RESEARCH	SMITH
30	SALES	ALLEN
30	SALES	WARD
20	RESEARCH	JONES
30	SALES	MARTIN
30	SALES	BLAKE
10	ACCOUNTING	CLARK
20	RESEARCH	SCOTT
10	ACCOUNTING	KING
30	SALES	TURNER
20	RESEARCH	ADAMS
30	SALES	JAMES
20	RESEARCH	FORD
10	ACCOUNTING	MILLER
40	OPERATIONS	

最后一行表明，部门 **OPERATIONS** 也被返回了，虽然这个部门没有任何员工。这是因为将 **EMP** 表外连接到了 **DEPT** 表。现在假设有一位员工没有部门，那么如何返回上述结果集，同时返回另一行，表示这位没有部门的员工呢？换言之，你要在同一个查询中外连接到 **EMP** 表和 **DEPT** 表。下面创建一

位没有部门的员工并尝试返回他。

```
insert into emp (empno,ename,job,mgr,hiredate,sal,comm,deptno)
select 1111,'YODA','JEDI',null,hiredate,sal,comm,null
  from emp
 where ename = 'KING'
```

```
select d.deptno,d.dname,e.ename
  from dept d right outer join emp e
    on (d.deptno=e.deptno)
```

DEPTNO	DNAME	ENAME
10	ACCOUNTING	MILLER
10	ACCOUNTING	KING
10	ACCOUNTING	CLARK
20	RESEARCH	FORD
20	RESEARCH	ADAMS
20	RESEARCH	SCOTT
20	RESEARCH	JONES
20	RESEARCH	SMITH
30	SALES	JAMES
30	SALES	TURNER
30	SALES	BLAKE
30	SALES	MARTIN
30	SALES	WARD
30	SALES	ALLEN
		YODA

以上外连接返回了这位新员工，但未能像前面的查询那样返回部门 **OPERATIONS**。你希望最终的结果集中既包含表示员工 **YODA** 的行，也包含表示部门 **OPERATIONS** 的行。

DEPTNO	DNAME	ENAME
10	ACCOUNTING	CLARK
10	ACCOUNTING	KING
10	ACCOUNTING	MILLER
20	RESEARCH	ADAMS
20	RESEARCH	FORD
20	RESEARCH	JONES
20	RESEARCH	SCOTT

20 RESEARCH	SMITH
30 SALES	ALLEN
30 SALES	BLAKE
30 SALES	JAMES
30 SALES	MARTIN
30 SALES	TURNER
30 SALES	WARD
40 OPERATIONS	YODA

12. 解决方案

使用全外连接返回两张表中的所有数据。

DB2、MySQL、PostgreSQL 和 SQL Server

使用显式命令 **FULL OUTER JOIN** 返回两张表中匹配的行以及不匹配的行。

```
1 select d.deptno,d.dname,e.ename
2   from dept d full outer join emp e
3     on (d.deptno=e.deptno)
```

由于 MySQL 还不支持 **FULL OUTER JOIN**，因此需要使用 **UNION** 合并两个外连接的结果。

```
1 select d.deptno,d.dname,e.ename
2   from dept d right outer join emp e
3     on (d.deptno=e.deptno)
4 union
5 select d.deptno,d.dname,e.ename
6   from dept d left outer join emp e
7     on (d.deptno=e.deptno)
```

Oracle

Oracle 用户既可以使用上述两种解决方案中的任何一种，也可以使用 Oracle 特有的外连接语法。

```
1 select d.deptno,d.dname,e.ename
2   from dept d, emp e
3  where d.deptno = e.deptno(+)
4  union
5 select d.deptno,d.dname,e.ename
6   from dept d, emp e
7  where d.deptno(+) = e.deptno
```

13. 讨论

全外连接不过是将外连接到两张表的操作合而为一了。如果想弄清楚全外连接在幕后是如何工作的，只需运行两个外连接，然后将它们的结果合并。下面的查询返回了 **DEPT** 表中的所有行以及 **EMP** 表中的匹配行（如果存在的话）。

```
select d.deptno,d.dname,e.ename
   from dept d left outer join emp e
     on (d.deptno = e.deptno)
```

DEPTNO	DNAME	ENAME
20	RESEARCH	SMITH
30	SALES	ALLEN
30	SALES	WARD
20	RESEARCH	JONES
30	SALES	MARTIN
30	SALES	BLAKE
10	ACCOUNTING	CLARK
20	RESEARCH	SCOTT
10	ACCOUNTING	KING
30	SALES	TURNER
20	RESEARCH	ADAMS
30	SALES	JAMES
20	RESEARCH	FORD
10	ACCOUNTING	MILLER

下面的查询返回了 **EMP** 表中的所有行以及 **DEPT** 表中的匹配行（如果存在的话）。

```
select d.deptno,d.dname,e.ename
  from dept d right outer join emp e
    on (d.deptno = e.deptno)
```

DEPTNO	DNAME	ENAME
-----	-----	-----
10	ACCOUNTING	MILLER
10	ACCOUNTING	KING
10	ACCOUNTING	CLARK
20	RESEARCH	FORD
20	RESEARCH	ADAMS
20	RESEARCH	SCOTT
20	RESEARCH	JONES
20	RESEARCH	SMITH
30	SALES	JAMES
30	SALES	TURNER
30	SALES	BLAKE
30	SALES	MARTIN
30	SALES	WARD
30	SALES	ALLEN
		YODA

将这两个查询的结果合并，就可以得到所需的最终结果集。

3.12 在运算和比较中使用NULL

1. 问题

NULL 与包含自己在内的任何值都不相等，也不会相等，但你想像评估实际值一样评估可为 NULL 的列返回的值。例如，你想在 EMP 表中找出业务提成（COMM）比 WARD 低的所有员工，包括业务提成为 NULL 的员工。

2. 解决方案

在标准评估中，可以使用诸如 COALESCE 等函数将 NULL 转换为实际值。

```
1 select ename,comm
2   from emp
3  where coalesce(comm,0) < ( select comm
4                             from emp
5                             where ename = 'WARD' )
```

3. 讨论

函数 COALESCE 会返回其参数列表中第一个非 NULL 值。在上面的查询中，遇到业务提成为 NULL 时，就将它转换为 0，然后再与 WARD 的业务提成进行比较。要证明这一点，可以在 SELECT 列表中使用函数 COALESCE。

```
select ename,comm,coalesce(comm,0)
   from emp
  where coalesce(comm,0) < ( select comm
```

```
from emp
where ename = 'WARD' )
```

ENAME	COMM	COALESCE(COMM,0)
SMITH		0
ALLEN	300	300
JONES		0
BLAKE		0
CLARK		0
SCOTT		0
KING		0
TURNER	0	0
ADAMS		0
JAMES		0
FORD		0
MILLER		0

3.13 小结

连接在数据库查询中扮演着至关重要的角色，你经常需要连接多张表以获取所需的信息。掌握本章介绍的连接类型和组合是迈向成功的第一步。

第 4 章 插入、更新和删除

前几章重点介绍了基本的查询技巧，它们的目标都是从数据库中获取数据。本章将把注意力转向如下 3 个主题：

- 在数据库中插入新记录；
- 更新既有记录；
- 删除不再需要的记录。

为了方便查找，本章的实例按主题进行了分组：首先是插入实例，然后是更新实例，最后是删除实例。

插入通常是一项简单的任务。本章从一个简单的问题开始：插入一行数据。然而，在大多数情况下，使用基于集合的方法来创建新行效率会更高，有鉴于此，本章还将介绍同时插入多行数据的技巧。

同样，介绍更新和删除时，也将从简单的任务开始。可以每次只更新和删除一条记录，也可以一次更新整个记录集。另外，有很多便利的记录删除的方式，例如，可以根据一张表中的某些行是否与另一张表中的行相关联来决定是否删除这些行。

SQL 还提供了一种相对较新的方式，让你能够同时执行插入、更新和删除操作。**MERGE** 语句功能非常强大，可用于将数据库表与外部数据源（比如远程系统提供的平面文件）同步，虽然目前而言这不是很有用。有关这种语句的详情，请参阅 4.11 节。

4.1 插入新记录

1. 问题

你想在表中插入一条新记录。例如，你想在 **DEPT** 表中插入一条新记录，并将其 **DEPTNO** 列、**DNAME** 列和 **LOC** 列分别设置为 **50**、**PROGRAMMING** 和 **BALTIMORE**。

2. 解决方案

结合使用 **INSERT** 语句和 **VALUES** 子句一次插入一行。

```
insert into dept (deptno,dname,loc)
values (50,'PROGRAMMING','BALTIMORE')
```

在 **DB2**、**SQL Server**、**PostgreSQL** 和 **MySQL** 中，除了一次插入一行，还可以一次插入多行，为此可以提供多个 **VALUES** 列表。

```
/* 插入多行 */
insert into dept (deptno,dname,loc)
values (1,'A','B'),
       (2,'B','C')
```

3. 讨论

INSERT 语句让你能够在数据库表中创建新行。在所有数据库中，插入单行的语法都相同。

作为一种快捷方式，可以在 **INSERT** 语句中省略列列表。

```
insert into dept  
values (50, 'PROGRAMMING', 'BALTIMORE')
```

然而，如果没有指定目标列，就必须在所有列中插入数据。另外，应注意 **VALUES** 列表中值的排列顺序：必须按数据库响应 **SELECT *** 查询时显示列的顺序提供值。无论使用哪种方式插入，都必须注意列约束，因为如果没有在每列中都插入值，那么在创建的新行中，有些列的值将为 **NULL**。如果这些列不能为 **NULL**，则将导致错误。

4.2 插入默认值

1. 问题

定义表时，可以指定某些列为默认值。你想插入使用默认值的行，这样就不用指定值了。

请看下面的表。

```
create table D (id integer default 0)
```

你想插入 0，而不是在 **INSERT** 语句的值列表中指定 0。你还想显式地插入默认值，而不管默认值是什么。

2. 解决方案

所有数据库都支持使用关键字 **DEFAULT** 显式地将列值指定为默认值。有些数据库还提供了解决这种问题的其他方式。

下面的示例演示了如何使用关键字 **DEFAULT**。

```
insert into D values (default)
```

还可以显式地指定列名。如果没有在所有列中都插入值，则必须这样做。

```
insert into D (id) values (default)
```

由于 Oracle8i 数据库及更早的版本不支持关键字 **DEFAULT**，因此在 Oracle9i 数据库之前，没有显式地插入默认值的方法。

式。

在 MySQL 中，如果所有列都有默认值，则可以让值列表为空。

```
insert into D values ()
```

在这种情况下，所有列都将被设置为默认值。

PostgreSQL 和 SQL Server 支持 **DEFAULT VALUES** 子句。

```
insert into D default values
```

DEFAULT VALUES 子句会导致所有列都为默认值。

13. 讨论

在值列表中，关键字 **DEFAULT** 会在相应列中插入创建表时指定的默认值。所有 **DBMS** 都支持这个关键字。

如果给表中的所有列都指定了默认值（就像前面的 **D** 表那样），那么 MySQL、PostgreSQL 和 SQL Server 用户还有另一种选择，就是使用空的 **VALUES** 列表（用于 MySQL）或 **DEFAULT VALUES** 子句（用于 PostgreSQL 和 SQL Server）来创建一个全为默认值的新行。否则，就需要使用 **DEFAULT** 分别将各列设置为默认值。

如果表中既有默认值列又有非默认值列，那么也可以轻松地将列值设置为默认值。为此，只需在插入列表中不指定该列，这样就无须使用关键字 **DEFAULT** 了。假设前面的 **D** 表还有一个没有指定默认值的列。

```
create table D (id integer default 0, foo varchar(10))
```

要在插入行时将 **ID** 设置为默认值，可以在插入列表中只包含 **FOO**。

```
insert into D (name) values ('Bar')
```

这条语句将插入一个 **ID** 为 0 且 **FOO** 为 **BAR** 的新行。**ID** 为默认值是因为没有给它指定值。

4.3 用NULL覆盖默认值

1. 问题

你想在有默认值的列中插入一行，并想用 **NULL** 覆盖默认值。请看下面的表。

```
create table D (id integer default 0, foo VARCHAR(10))
```

你想在这张表中插入一行数据，并将其 **ID** 设置为 **NULL**。

2. 解决方案

可以在值列表中显式地指定 **NULL**。

```
insert into d (id, foo) values (null, 'Brighten')
```

3. 讨论

并非所有人都知道可以在 **INSERT** 语句的值列表中显式地指定 **NULL**。当不想给列指定值时，通常可以不在列列表和值列表中指定它的值。

```
insert into d (foo) values ('Brighten')
```

这里没有给 **ID** 指定值。很多人以为这个列的值将为 **NULL**，但由于创建表时给这个列指定了默认值，因此上述 **INSERT** 语句将插入一个 **ID** 值为 **0**（默认值）的新行。通过将列值指

定为 **NULL**，可以将相应的列设置为 **NULL**，即便该列有默认值（条件是没有设置禁止将该列设置为 **NULL** 的约束）。

4.4 将一张表中的行复制到另一张表中

1. 问题

你想使用查询将一张表中的行复制到另一张表中。使用的查询可能很复杂，也可能很简单，但你的终极目标是将查询的结果插入另一张表中。例如，你想将 **DEPT** 表中的行复制到 **DEPT_EAST** 表中。假设 **DEPT_EAST** 表已经创建好，其结构与 **DEPT** 表相同（列数和列的数据类型都相同），但当前是空的。

2. 解决方案

在 **INSERT** 语句中将生成所需行的查询用作值列表。

```
1 insert into dept_east (deptno,dname,loc)
2 select deptno,dname,loc
3   from dept
4  where loc in ( 'NEW YORK','BOSTON' )
```

3. 讨论

只需在 **INSERT** 语句后面跟一个返回所需行的查询。如果要复制源表中所有的行，那么可以在查询中不指定 **WHERE** 子句。与常规插入一样，并非必须显式地指定要插入哪些列，但如果没有指定目标列，则必须在所有列中都插入数据，还必须注意 **SELECT** 列表中值的排列顺序，这在 4.1 节介绍过。

4.5 复制表定义

1. 问题

你想创建一张新表，该表包含的列与一张既有表相同。例如，你想创建 **DEPT** 表的一个副本，并将其命名为 **DEPT_2**。但是，你不想复制表中的行，而只想复制表的列结构。

2. 解决方案

DB2

在 **CREATE TABLE** 命令中使用 **LIKE** 子句。

```
create table dept_2 like dept
```

Oracle、MySQL 和 PostgreSQL

在 **CREATE TABLE** 命令中使用一个不返回任何行的子查询。

```
1 create table dept_2
2 as
3 select *
4   from dept
5  where 1 = 0
```

SQL Server

在一个不返回任何行的子查询中使用 **INTO** 子句。

```
1 select *
2   into dept_2
```

```
3  from dept
4  where 1 = 0
```

13. 讨论

DB2

DB2 的 **CREATE TABLE...LIKE** 命令让你能够轻松地以一张表为模板创建另一张表。为此，在关键字 **LIKE** 后面指定模板表的名称即可。

Oracle、MySQL 和 PostgreSQL

使用 **Create Table As Select (CTAS)** 时，查询返回的行都将用来填充新建的表，除非你在 **WHERE** 子句中指定了一个无法满足的条件。在前面的解决方案中，**WHERE** 子句中的表达式 **1 = 0** 导致查询不会返回任何行。因此，这条 **CTAS** 语句的结果是一张空表，其包含的列与 **SELECT** 子句指定的列相同。

SQL Server

使用 **INTO** 复制表时，查询返回的行都将用来填充新建的表，除非你在 **WHERE** 子句中指定了一个无法满足的条件。在前面的解决方案中，谓词中的表达式 **1 = 0** 导致查询不会返回任何行。因此结果是一张空表，其包含的列与 **SELECT** 子句指定的列相同。

4.6 同时插入多张表

1. 问题

你想将查询返回的行插入多张表中。例如，你想将 **DEPT** 表中的行插入 **DEPT_EAST** 表、**DEPT_WEST** 表和 **DEPT_MID** 表中。这 3 张目标表的结构与 **DEPT** 表相同（列数和列的数据类型都相同），并且当前都是空的。

2. 解决方案

解决方案是将查询的结果插入目标表中。与 4.4 节不同，这里有多张目标表。

Oracle

使用 **INSERT ALL** 语句或 **INSERT FIRST** 语句。这两条语句的语法相同，唯一的差别是一条语句使用的是关键字 **ALL**，另一条语句使用的是关键字 **FIRST**。下面的语句使用了 **INSERT ALL**，因此需要考虑所有的目标表。

```
1  insert all
2    when loc in ('NEW YORK','BOSTON') then
3      into dept_east (deptno,dname,loc) values (deptno,dname,loc)
4    when loc = 'CHICAGO' then
5      into dept_mid (deptno,dname,loc) values (deptno,dname,loc)
6    else
7      into dept_west (deptno,dname,loc) values
(deptno,dname,loc)
8    select deptno,dname,loc
9    from dept
```

DB2

使用 **UNION ALL** 合并所有的目标表，以创建一个内嵌视图，再将数据插入这个内嵌视图中。还必须给目标表设置约束条件，确保每行都被插入正确的目标表中。

```
create table dept_east
( deptno integer,
  dname  varchar(10),
  loc    varchar(10) check (loc in ('NEW YORK','BOSTON')))

create table dept_mid
( deptno integer,
  dname  varchar(10),
  loc    varchar(10) check (loc = 'CHICAGO'))

create table dept_west
( deptno integer,
  dname  varchar(10),
  loc    varchar(10) check (loc = 'DALLAS'))

1 insert into (
2   select * from dept_west union all
3   select * from dept_east union all
4   select * from dept_mid
5 ) select * from dept
```

MySQL、PostgreSQL 和 SQL Server

本书撰写之时，这些 DBMS 都不支持多表插入。

13. 讨论

Oracle

Oracle 多表插入使用 **WHEN-THEN-ELSE** 子句来检查嵌套的 **SELECT** 返回的行，以便将它们插入正确的表中。在本实例中，**INSERT ALL** 和 **INSERT FIRST** 的效果相同，但它们之

间存在一个不同之处。**INSERT FIRST** 在遇到满足的条件后，就结束 **WHEN-THEN-ELSE** 检查，而 **INSERT ALL** 会检查所有的条件，即便已经遇到了满足的条件。因此，可以使用 **INSERT ALL** 将相同的行插入多张表中。

DB2

前面的 DB2 解决方案有点儿不合常规。它要求给目标表定义约束条件，以确保从子查询返回的每一行都进入正确的表中。这里的诀窍是使用 **UNION ALL** 合并所有目标表，以生成一个视图，然后将行插入这个视图中。如果给目标表指定的约束条件不是唯一的（比如有多张表的约束条件相同），那么 **INSERT** 语句将不知道该将行插入哪张表中，进而以失败告终。

MySQL、PostgreSQL 和 SQL Server

本书撰写之时，只有 Oracle 和 DB2 提供了相关的机制，可用于在一条语句中将插入查询返回的行插入一张或多张表中。

4.7 禁止在特定列中插入值

1. 问题

你想禁止用户（或软件应用程序）在特定列中插入值。例如，你希望一个程序在 **EMP** 表中插入行，但只能指定 **EMPNO**、**ENAME** 和 **JOB** 列的值。

2. 解决方案

创建一个基于表的视图，只暴露那些你想暴露的列，然后规定只能通过这个视图来插入。

例如，创建一个暴露 **EMP** 表中 3 列数据的视图。

```
create view new_emps as
select empno, ename, job
  from emp
```

授予用户和程序访问这个视图的权限，让它们能够填充该视图中的 3 列。不授予用户在 **EMP** 表中插入行的权限。这样，如果用户要新建 **EMP** 记录，那么可以在 **NEW_EMPS** 视图中插入行，但他们只能给该视图的定义中指定的 3 列设置值，而不能给其他列设置值。

3. 讨论

当在简单视图中插入行时，数据库服务器将把它转换为针对底层表的插入操作。例如，对于下面的插入：

```
insert into new_ems  
  (empno ename, job)  
  values (1, 'Jonathan', 'Editor')
```

将在幕后将其转换为下面的插入。

```
insert into emp  
  (empno ename, job)  
  values (1, 'Jonathan', 'Editor')
```

也可以插入内嵌视图中（这可能不那么有用，而且当前只有 Oracle 支持）。

```
insert into  
  (select empno, ename, job  
   from emp)  
  values (1, 'Jonathan', 'Editor')
```

视图插入是一个复杂的主题，除非视图很简单，否则规则将很快变得非常复杂。如果你要使用视图插入功能，那么务必参阅相关的数据库文档，并确保已完全理解。

4.8 修改表中的记录

1. 问题

你想修改表中部分行或全部行的值。例如，你可能想将部门编号为 20 的所有员工的薪水都提高 10%。下面的结果集显示了这个部门所有员工的 DEPTNO、ENAME 和 SAL。

```
select deptno,ename,sal
  from emp
 where deptno = 20
 order by 1,3
```

DEPTNO	ENAME	SAL
20	SMITH	800
20	ADAMS	1100
20	JONES	2975
20	SCOTT	3000
20	FORD	3000

你想将所有 SAL 值都提高 10%。

2. 解决方案

使用 UPDATE 语句来修改数据库表中既有的行。

```
1 update emp
2   set sal = sal*1.10
3  where deptno = 20
```

3. 讨论

在 **UPDATE** 语句中，使用 **WHERE** 子句来指定要更新哪些行。如果没有指定 **WHERE** 子句，则将更新所有行。在上述解决方案中，表达式 **SAL*1.10** 返回了提高 10% 后的薪水。

进行大规模更新前，你可能想预览结果。为此，可以执行一条 **SELECT** 语句，并在其中使用你打算在 **SET** 子句中使用的表达式。下面的 **SELECT** 语句显示了加薪 10% 后的结果。

```
select deptno,
       ename,
       sal      as orig_sal,
       sal*.10  as amt_to_add,
       sal*1.10 as new_sal
  from emp
 where deptno=20
 order by 1,5
```

DEPTNO	ENAME	ORIG_SAL	AMT_TO_ADD	NEW_SAL
20	SMITH	800	80	880
20	ADAMS	1100	110	1210
20	JONES	2975	298	3273
20	SCOTT	3000	300	3300
20	FORD	3000	300	3300

加薪情况被分成了两列，一列显示增加的薪水，另一列显示加薪后的薪水。

4.9 仅当存在匹配行时才更新

1. 问题

你想更新一张表中的某些行，条件是在另一张表中存在与之对应的行。如果员工出现在了 **EMP_BONUS** 表中，就（在 **EMP** 表中）将其薪水提高 20%。下面的结果集显示了 **EMP_BONUS** 表中当前包含的数据。

```
select empno, ename
  from emp_bonus

  EMPNO ENAME
-----
    7369 SMITH
    7900 JAMES
    7934 MILLER
```

2. 解决方案

在 **UPDATE** 语句的 **WHERE** 子句中，使用子查询在 **EMP** 表中找出那些同时出现在 **EMP_BONUS** 表中的员工。这样，**UPDATE** 语句将只作用于子查询返回的员工，并给他们加薪 20%。

```
1 update emp
2   set sal=sal*1.20
3  where empno in ( select empno from emp_bonus )
```

3. 讨论

子查询返回的结果决定了将更新 **EMP** 表中的哪些行。谓词

IN 会检查 **EMP** 表中 **EMPNO** 的值，看它们是否出现在了子查询返回的 **EMPNO** 值中。如果是的话，就更新相应的 **SAL** 值。

也可以使用 **EXISTS** 来代替 **IN**。

```
update emp
  set sal = sal*1.20
 where exists ( select null
                from emp_bonus
                where emp.empno=emp_bonus.empno )
```

在 **EXISTS** 子查询中，看到 **SELECT** 列表中的 **NULL**，你可能会感到惊讶。不用担心，这个 **NULL** 不会给更新带来任何负面影响。可以说这样做反而提高了可读性，因为它凸显了这样一个事实：与结合使用子查询和运算符 **IN** 的解决方案不同，真正决定将更新哪些行的是子查询中的 **WHERE** 子句，而不是子查询的 **SELECT** 列表返回的值。

4.10 使用来自另一张表中的值进行更新

1. 问题

你想使用一张表中的值更新另一张表中的行。例如，你有一张 **NEW_SAL** 表，其中存储了某些员工调整后的薪水。这张表的内容如下。

```
select *
  from new_sal

DEPTNO      SAL
-----
      10      4000
```

DEPTNO 列是 **NEW_SAL** 表的主键。你想使用 **NEW_SAL** 中的值更新 **EMP** 表中某些员工的薪水和业务提成。如果员工的 **EMP.DEPTNO** 与 **NEW_SAL** 表中的某一行的 **DEPTNO** 相同，就将其 **EMP.SAL** 更新为该行的 **SAL** 值，并将其 **EMP.COMM** 更新为该行的 **SAL** 值的 50%。**EMP** 表的内容如下。

```
select deptno,ename,sal,comm
  from emp
 order by 1

DEPTNO ENAME      SAL      COMM
-----
      10 CLARK      2450
      10 KING      5000
      10 MILLER    1300
      20 SMITH      800
      20 ADAMS     1100
      20 FORD      3000
      20 SCOTT     3000
      20 JONES     2975
      30 ALLEN     1600      300
      30 BLAKE     2850
```

30	MARTIN	1250	1400
30	JAMES	950	
30	TURNER	1500	0
30	WARD	1250	500

12. 解决方案

一种方法是，通过连接 **NEW_SAL** 表和 **EMP** 表，找到新的 **COMM** 值并将其返回给 **UPDATE** 语句。对于这样的更新，经常使用关联子查询或 **CTE** 来执行。另一种方法是，创建一个视图（根据数据库的支持情况，可以是传统视图，也可以是内嵌视图），然后更新这个视图。

DB2

使用关联子查询将 **EMP** 表中的 **SAL** 列和 **COMM** 列设置为新的值。另外，使用关联子查询来确定应该更新 **EMP** 表中的哪些行。

```

1 update emp e set (e.sal,e.comm) = (select ns.sal, ns.sal/2
2                                     from new_sal ns
3                                     where ns.deptno=e.deptno)
4   where exists ( select *
5                  from new_sal ns
6                  where ns.deptno = e.deptno )

```

MySQL

在 **UPDATE** 子句中指定 **EMP** 表和 **NEW_SAL** 表，并在 **WHERE** 子句中连接它们。

```

1 update emp e, new_sal ns
2 set e.sal=ns.sal,
3 e.comm=ns.sal/2

```

```
4 where e.deptno=ns.deptno
```

Oracle

DB2 的解决方案也适用于 Oracle，但在 Oracle 中，还可以使用另一种解决方案，即更新内嵌视图。

```
1 update (  
2   select e.sal as emp_sal, e.comm as emp_comm,  
3         ns.sal as ns_sal, ns.sal/2 as ns_comm  
4   from emp e, new_sal ns  
5   where e.deptno = ns.deptno  
6 ) set emp_sal = ns_sal, emp_comm = ns_comm
```

PostgreSQL

DB2 的解决方案也适用于 PostgreSQL，但还可以在 UPDATE 语句中直接连接（非常方便）。

```
1 update emp  
2   set sal  = ns.sal,  
3       comm = ns.sal/2  
4   from new_sal ns  
5   where ns.deptno = emp.deptno
```

SQL Server

DB2 的解决方案也适用于 SQL Server，但还可以在 UPDATE 语句中直接连接（类似于 PostgreSQL 解决方案）。

```
1 update e  
2   set e.sal  = ns.sal,  
3       e.comm = ns.sal/2  
4   from emp e,  
5       new_sal ns  
6   where ns.deptno = e.deptno
```

13. 讨论

讨论各种解决方案之前，需要说点儿重要的事情，它与使用查询来提供新值的更新相关。与更新相关的子查询中的 **WHERE** 子句的作用与针对目标表的 **WHERE** 子句的作用不同。如果查看本节“解决方案”部分中的 **UPDATE** 语句，你将发现在 **EMP** 表和 **NEW_SAL** 表之间建立了基于 **DEPTNO** 的连接，并将生成的行返回给了 **UPDATE** 语句的 **SET** 子句。对于 **DEPTNO** 为 10 的员工，将返回有效的值，因为 **NEW_SAL** 表中有与之匹配的 **DEPTNO**。但对于其他部门的员工呢？由于 **NEW_SAL** 表中不包含其他部门，因此对于 **DEPTNO** 为 20 或 30 的员工，**SAL** 和 **COMM** 将被设置为 **NULL**。除非使用 DBMS 提供的 **LIMIT**、**TOP** 或其他机制，否则限制返回函数的唯一方法就是使用 **WHERE** 子句。为了正确执行这种更新，我们使用了一个针对目标表的 **WHERE** 子句，同时在关联子查询中使用了一个 **WHERE** 子句。

DB2

为了避免更新 **EMP** 表中的所有行，别忘了在 **UPDATE** 的 **WHERE** 子句中包含一个关联子查询。仅在 **SET** 子句中执行连接（关联子查询）还不够。在 **UPDATE** 中使用 **WHERE** 子句，可以确保只更新 **EMP** 表中这样的行，即其 **DEPTNO** 与 **NEW_SAL** 表中某行的 **DEPTNO** 相匹配。上述讨论适用于所有 RDBMS。

Oracle

在更新内嵌视图的 Oracle 解决方案中，使用了相等连接来确定要更新哪些行。要获悉哪些行将被更新，可以单独执行该解决方案中的查询。要成功地使用这种 **UPDATE**，必须明白

键保留（key-preservation）的概念。在 **NEW_SAL** 表中，**DEPTNO** 列为主键，因为其只在这张表中是独一无二的。然而，在连接 **EMP** 表和 **NEW_SAL** 表生成的结果集中，**NEW_SAL.DEPTNO** 并非独一无二。

<pre>select e.empno, e.deptno e_dept, ns.sal, ns.deptno ns_deptno from emp e, new_sal ns where e.deptno = ns.deptno</pre>			
EMPNO	E_DEPT	SAL	NS_DEPTNO
-----	-----	-----	-----
7782	10	4000	10
7839	10	4000	10
7934	10	4000	10

要让 Oracle 更新这个连接视图，被连接的表之一必须是键保留的，这意味着即便键在结果集中不是独一无二的，但至少它在源表中是独一无二的。在本例中，**DEPTNO** 是 **NEW_SAL** 的主键，因此它在这张表中是独一无二的。由于 **DEPTNO** 在其表中是独一无二的，因此即便它在结果集中出现多次，也被视为是键保留的，这让更新得以成功完成。

PostgreSQL、SQL Server 和 MySQL

在这些平台上，解决方案的语法有点儿差别，但使用的方法是一样的。能够在 **UPDATE** 语句中直接进行连接为我们提供了极大的便利。由于你指定了要更新哪张表（紧跟在关键字 **UPDATE** 后面的表），因此清楚知道要修改哪张表的行。另外，由于是在 **UPDATE** 中使用连接（有显式的 **WHERE** 语句），因此可以绕开编写关联子查询更新时可能遇到的陷阱。具体地说，如果没有使用连接，就会出问题。

4.11 合并记录

1. 问题

你想根据是否存在相应的记录，来决定插入、更新或删除一张表的记录。（如果存在相应的记录，就更新；如果不存在，就插入；如果更新后不满足特定的条件，就删除。）例如，你想按下面的方式修改 **EMP_COMMISSION** 表。

- 对于 **EMP_COMMISSION** 表中的员工，如果他们也包含在 **EMP** 表中，就将其业务提成（**COMM**）更新为 1000。
- 对于 **COMM** 可能被更新为 1000 的任何员工，如果他们的 **SAL** 低于 2000，就将其删除（他们不应该出现在 **EMP_COMMISSION** 表中）。
- 对于包含在 **EMP** 表中但未包含在 **EMP_COMMISSION** 表中的员工，将他们插入 **EMP_COMMISSION** 表中，并使用 **EMP** 表中的 **EMPNO** 值、**ENAME** 值和 **DEPTNO** 值。

从本质上说，你要根据 **EMP** 表中给定行在 **EMP_COMMISSION** 表中是否有匹配的行，来决定在 **EMP_COMMISSION** 中执行 **UPDATE** 操作还是 **INSERT** 操作。然后根据 **UPDATE** 操作是否会导致业务提成过高，来决定是否执行 **DELETE** 操作。

下面显示了 **EMP** 表和 **EMP_COMMISSION** 表当前包含的行。

```
select deptno,empno,ename,comm
  from emp
 order by 1
```

DEPTNO	EMPNO	ENAME	COMM
10	7782	CLARK	
10	7839	KING	
10	7934	MILLER	

20	7369	SMITH	
20	7876	ADAMS	
20	7902	FORD	
20	7788	SCOTT	
20	7566	JONES	
30	7499	ALLEN	300
30	7698	BLAKE	
30	7654	MARTIN	1400
30	7900	JAMES	
30	7844	TURNER	0
30	7521	WARD	500


```

select deptno,empno,ename,comm
  from emp_commission
 order by 1

```

DEPTNO	EMPNO	ENAME	COMM

10	7782	CLARK	
10	7839	KING	
10	7934	MILLER	

12. 解决方案

用于解决这种问题的语句是 **MERGE** 语句，它可以根据需要执行 **UPDATE** 操作或 **INSERT** 操作。

```

1 merge into emp_commission ec
2 using (select * from emp) emp
3   on (ec.empno=emp.empno)
4   when matched then
5       update set ec.comm = 1000
6       delete where (sal < 2000)
7   when not matched then
8       insert (ec.empno,ec.ename,ec.deptno,ec.comm)
9       values (emp.empno,emp.ename,emp.deptno,emp.comm)

```

当前，MySQL 中没有 **MERGE** 语句。这种查询适用于本书提

及其他 RDBMS，以及很多别的 RDBMS。

13. 讨论

在上述解决方案中，第 3 行的连接确定了哪些行已经存在，因此需要对它们进行更新。该连接是在 **EMP_COMMISSION**（别名 **EC**）表和子查询（别名 **EMP**）之间进行的。如果连接成功，这两行就被视为是匹配的，进而执行 **WHEN MATCHED** 子句中指定的 **UPDATE**。否则就说明没有找到匹配的行，进而执行 **WHEN NOT MATCHED** 中的 **INSERT**。因此，对于 **EMP** 表中的行，如果根据 **EMPNO** 确定 **EMP_COMMISSION** 表中没有相应的行，就将它们插入 **EMP_COMMISSION** 表中。在 **EMP** 表的所有员工中，只有那些 **DEPTNO** 为 10 的员工也包含在 **EMP_COMMISSION** 表中，因此应该在 **EMP_COMMISSION** 表中更新他们的 **COMM**。对于其他员工，他们将被插入 **EMP_COMMISSION** 表中。另外，**MILLER** 的 **DEPTNO** 为 10，原本应该更新他的 **COMM**，但由于他的 **SAL** 低于 2000，因此他会被从 **EMP_COMMISSION** 表中删除。

4.12 删除表中的所有记录

1. 问题

你想删除表中的所有记录。

2. 解决方案

使用 **DELETE** 命令删除表中的记录。例如，要删除 **EMP** 表中的所有记录，可以使用如下命令。

```
delete from emp
```

3. 讨论

使用 **DELETE** 命令时，如果没有指定 **WHERE** 子句，则将删除指定表中的所有行。**TRUNCATE** 命令用于表，它不使用 **WHERE** 子句，因此在有些情况下，使用此命令更合适，因为其速度更快。然而，**TRUNCATE** 命令是不能撤销的，至少在 Oracle 中是这样。在特定 RDBMS 中，有关 **TRUNCATE** 和 **DELETE** 在性能和回滚方面的差别，请参阅相关文档。

4.13 删除特定记录

1. 问题

你想将满足特定条件的记录从表中删除。

2. 解决方案

使用 **DELETE** 命令，并在其中使用 **WHERE** 子句指定要删除哪些行。如果要删除编号为 10 的部门的所有员工，那么可以使用如下命令。

```
delete from emp where deptno = 10
```

3. 讨论

通过在 **DELETE** 命令中使用 **WHERE** 子句，可以删除表中的部分（而不是全部）行。别忘了使用 **SELECT** 语句预览你编写的 **WHERE** 子句的影响，确保正确地删除数据，因为即便在非常简单的情况下，也可能错误地删除数据。例如，在前面的实例中，输入错误可能导致删除的是编号为 20 的部门（而不是编号为 10 的部门）的员工！

4.14 删除单条记录

1. 问题

你想从表中删除单条记录。

2. 解决方案

这是 4.13 节的特例。关键是确保选择标准足够严，使得只有你要删除的那条记录符合条件。通常，可以根据主键进行删除，例如，要删除员工 CLARK，可以指定条件 **EMPNO = 7782**。

```
delete from emp where empno = 7782
```

3. 讨论

执行删除操作时，总是需要指定要删除的行，而 **DELETE** 操作带来的影响取决于其 **WHERE** 子句。如果省略了 **WHERE** 子句，那么 **DELETE** 操作的影响范围将是整张表。通过在 **WHERE** 子句中指定条件，可以将影响范围缩小到只有一组或一条记录。删除单条记录时，通常应该使用主键或独一无二的键来指定要删除哪条记录。



如果删除标准是基于主键或独一无二的键的，那么便可确信将只会删除一条记录（因为 **RDBMS** 不允许两条记录有相同的主键或独一无二的键）。否则，可能需要在删除前进行检查，避免无意间删除多条记录。

4.15 删除违反引用完整性的记录

1. 问题

当一张表中的记录引用了另一张表中不存在的记录时，你想将其删除。例如，给有些员工指定的部门不存在，你想将这些员工删除。

2. 解决方案

结合使用谓词 **NOT EXISTS** 和子查询来检查部门编号是否有效。

```
delete from emp
where not exists (
  select * from dept
  where dept.deptno = emp.deptno
)
```

也可以使用谓词 **NOT IN** 来编写查询。

```
delete from emp
where deptno not in (select deptno from dept)
```

3. 讨论

删除其实就是选择，主要工作是在 **WHERE** 子句中指定条件，以正确地指定你要删除哪些记录。

上述 **NOT EXISTS** 解决方案中使用了一个关联子查询，该查

询会检查 **DEPT** 表中是否存在一条其 **DEPTNO** 与给定 **EMP** 记录匹配的记录。如果存在这样的记录，就保留给定的 **EMP** 记录。

否则，就将其删除。对于每条 **EMP** 记录，都以这样的方式进行检查。

上述 **IN** 解决方案中使用了一个子查询，该查询会返回一个有效的部门编号列表。然后，对于每条 **EMP** 记录，都检查其 **DEPTNO** 是否在这个列表中。如果 **EMP** 记录的 **DEPTNO** 不在该列表中，就将其删除。

4.16 删除重复记录

1. 问题

你想删除一张表中的重复记录。请看下面的表。

```
create table dupes (id integer, name varchar(10))

insert into dupes values (1, 'NAPOLEON')
insert into dupes values (2, 'DYNAMITE')
insert into dupes values (3, 'DYNAMITE')
insert into dupes values (4, 'SHE SELLS')
insert into dupes values (5, 'SEA SHELLS')
insert into dupes values (6, 'SEA SHELLS')
insert into dupes values (7, 'SEA SHELLS')

select * from dupes order by 1
```

ID	NAME
1	NAPOLEON
2	DYNAMITE
3	DYNAMITE
4	SHE SELLS
5	SEA SHELLS
6	SEA SHELLS
7	SEA SHELLS

对于每组重复的名称，比如 **SEA SHELLS**，你想随便保留其中一个，然后将其他的都删除。就 **SEA SHELLS** 而言，你不在乎删除的是第 5 行和第 6 行、第 5 行和第 7 行还是第 6 行和第 7 行，只要最后只保留一条有关 **SEA SHELLS** 的记录就行。

2. 解决方案

使用一个子查询，并在其中使用聚合函数（如 **MIN**）来选择要保留的 **ID**（在本例中，只保留 **ID** 值最小的 **NAME**）。

```
1 delete from dupes
2   where id not in ( select min(id)
3                     from dupes
4                     group by name )
```

MySQL 用户需要使用稍微不同的语法，因为在本书撰写之时，在 **DELET** 中不能引用同一张表两次。

```
1 delete from dupes
2   where id not in
3     (select min(id)
4    from (select id,name from dupes) tmp
5     group by name)
```

13. 讨论

删除重复的记录时，首先需要准确地定义两条记录在什么情况下才会被认为是重复的。就本例而言，重复的定义是两条记录的 **NAME** 列值相同。有了定义之后，可以通过其他列来区分重复的记录，进而确定保留其中的哪条记录。如果这个（些）区分列是主键，那就最好不过了。我们使用的区分列是 **ID** 列，这是一个不错的选择，因为任何两条记录的 **ID** 都不同。

本实例解决方案的关键之处在于按重复的值（这里是 **NAME**）将记录分组，然后使用聚合函数从每个分组中选择一个要保留的键值。在该解决方案中，对于每一个 **NAME**，子查询都会返回它对应的最小 **ID**，其指出了要保留哪一行。

```
select min(id)
```

from dupes
group by name
MIN(ID)

2
1
5
4

接下来，**DELETE** 将未被子查询返回的 **ID** 对应的行（本例中是 **ID** 为 3、6 和 7 的行）从表中删除。如果你还不明白其中的原理，则可以在这个子查询的 **SELECT** 列表中添加 **NAME** 并运行它。

select name, min(id)
from dupes
group by name
NAME MIN(ID)

DYNAMITE 2
NAPOLEON 1
SEA SHELLS 5
SHE SELLS 4

这个子查询返回的是要保留的行。**DELECT** 语句中的谓词 **NOT IN** 会导致其他的行都被删除。

4.17 删除在另一张表中引用了的记录

1. 问题

你想删除在另一张表中引用了的记录。请看下面的表（**DEPT_ACCIDENTS**），它记录了一家制造企业发生的每次事故，其中每行都包含发生事故的部门以及事故的类型。

```
create table dept_accidents
( deptno          integer,
  accident_name varchar(20) )

insert into dept_accidents values (10,'BROKEN FOOT')
insert into dept_accidents values (10,'FLESH WOUND')
insert into dept_accidents values (20,'FIRE')
insert into dept_accidents values (20,'FIRE')
insert into dept_accidents values (20,'FLOOD')
insert into dept_accidents values (30,'BRUISED GLUTE')

select * from dept_accidents
```

DEPTNO	ACCIDENT_NAME
10	BROKEN FOOT
10	FLESH WOUND
20	FIRE
20	FIRE
20	FLOOD
30	BRUISED GLUTE

你想将所属部门发生事故的次数不少于 3 次的员工从 **EMP** 表中删除。

2. 解决方案

使用子查询和聚合函数 **COUNT** 找出发生事故不少于 3 次的部门，然后删除在这些部门工作的员工。

```
1 delete from emp
2   where deptno in ( select deptno
3                       from dept_accidents
4                       group by deptno
5                       having count(*) >= 3 )
```

13. 讨论

下面的子查询将找出哪些部门发生事故不少于 3 次。

```
select deptno
   from dept_accidents
  group by deptno
 having count(*) >= 3

      DEPTNO
-----
          20
```

接下来，**DELETE** 会将在这些部门（这里只有 20 号部门）工作的员工都删除。

4.18 小结

插入数据和更新数据的操作频率低于查询数据，因此本书将在余下的篇幅中重点介绍查询。然而，能够维护数据，对于数据库实现其初衷至关重要，而本章实例介绍的方法是数据库维护技能的重要组成部分。在这些命令中，有些（尤其是删除数据的命令）可能会带来深远影响。因此，务必预览要删除的数据，确保这些数据确实是要删除的。另外，务必了解在你使用的 RDBMS 中，哪些操作可以撤销，哪些则不能撤销。

第 5 章 元数据查询

本章的实例可以帮助你找到有关给定模式（**schema**）的信息。例如，你可能想知道自己创建了哪些表或没有基于哪些外键创建索引。本书提及的 **RDBMS** 都提供了可用于获取这些数据的表和视图。本章的实例提供了如何从这些表和视图中获取信息的基本知识。

所有 **RDBMS** 都采用了在表和视图中存储元数据的策略，但策略实现的标准化程度并没有本书介绍的其他 **SQL** 语言特性那么高，因此，在本章中，解决方案随 **RDBMS** 而异的情况比其他章要普遍得多。

下面介绍本书提及的各种 **RDBMS** 中最常用的模式查询。我们可以获取的信息比这些实例展示的要多得多。如果要获取其他信息，请参阅你使用的 **RDBMS** 的文档，详细了解其目录或数据字典表/视图。



为了方便演示，本章的实例都假设有一个名为 **SMEAGOL** 的模式。

5.1 列出模式中的所有表

1. 问题

你想列出自己在给定模式中创建的所有表。

2. 解决方案

下面的所有解决方案都假设你使用的模式为 **SMEAGOL**。在所有 **RDBMS** 中，解决方案采用的基本方法都相同——查询系统表（视图）。对于数据库中的每张表，这个系统表（视图）都包含一行与之相关的信息。

DB2

查询 **SYSCAT.TABLES**。

```
1 select tabname
2   from syscat.tables
3  where tabschema = 'SMEAGOL'
```

Oracle

查询 **SYS.ALL_TABLES**。

```
select table_name
  from all_tables
 where owner = 'SMEAGOL'
```

查询 **INFORMATION_SCHEMA.TABLES**。

```
1 select table_name
```

```
2  from information_schema.tables
3  where table_schema = 'SMEAGOL'
```

13. 讨论

暴露有关自己的信息时，数据库使用的正是你为自己的应用程序创建的机制——表和视图，这有点儿循环的“味道”。例如，Oracle 维护着一个庞大的系统视图（如 **ALL_TABLES**）目录，你可以通过查询这些视图来获取有关表、索引、授权（grant）和其他数据库对象的信息。



Oracle 的目录视图仅仅是视图。它们基于一组底层表，这些底层表包含的信息不便于用户读取。这些视图让用户能够轻松地使用 Oracle 的目录数据。

Oracle 系统视图和 DB2 系统表都是厂商专用的，而 PostgreSQL、MySQL 和 SQL Server 支持信息模式——ISO SQL 标准定义的一组视图，因此有些查询适用于这 3 种 RDBMS。

5.2 列出表中的列

1. 问题

你想列出表中的列，以及这些列的数据类型和在表中的位置。

2. 解决方案

下面的解决方案假设你要列出模式 **SMEAGOL** 中 **EMP** 表的列及其数据类型和用数字表示的位置。

DB2

查询 **SYSCAT.COLUMNS**。

```
1 select colname, typename, colno
2   from syscat.columns
3  where tabname   = 'EMP'
4    and tabschema = 'SMEAGOL'
```

Oracle

查询 **ALL_TAB_COLUMNS**。

```
1 select column_name, data_type, column_id
2   from all_tab_columns
3  where owner       = 'SMEAGOL'
4    and table_name = 'EMP'
```

PostgreSQL、MySQL 和 SQL Server

查询 INFORMATION_SCHEMA.COLUMNS。

```
1 select column_name, data_type, ordinal_position
2   from information_schema.columns
3  where table_schema = 'SMEAGOL'
4    and table_name   = 'EMP'
```

13. 讨论

每种 RDBMS 都提供了相关的途径，让你能够获取有关列数据的详细信息。在前面的解决方案中，只返回了列名、数据类型和位置。其他很有用的信息包括长度、是否可为 NULL 以及默认值。

5.3 列出表的索引列

1. 问题

你想列出给定表的索引、索引基于的列以及这些列在索引中的位置（如果有的话）。

2. 解决方案

下面的解决方案随 RDBMS 而异，但假设你要列出模式 SMEAGOL 中 EMP 表的索引。

DB2

查询 SYSCAT.INDEXES。

```
1 select a.tabname, b.indname, b.colname, b.colseq
2   from syscat.indexes a,
3        syscat.indexcoluse b
4  where a.tabname    = 'EMP'
5        and a.tabschema = 'SMEAGOL'
6        and a.indschema = b.indschema
7        and a.indname   = b.indname
```

Oracle

查询 SYS.ALL_IND_COLUMNS。

```
select table_name, index_name, column_name, column_position
   from sys.all_ind_columns
  where table_name = 'EMP'
        and table_owner = 'SMEAGOL'
```

PostgreSQL

查询 PG_CATALOG.PG_INDEXES 和 INFORMATION_SCHEMA.COLUMNS。

```
1 select a.tablename,a.indexname,b.column_name
2   from pg_catalog.pg_indexes a,
3        information_schema.columns b
4  where a.schemaname = 'SMEAGOL'
5        and a.tablename = b.table_name
```

MySQL

使用命令 SHOW INDEX。

```
show index from emp
```

SQL Server

查询 SYS.TABLES、SYS.INDEXES、SYS.INDEX_COLUMNS 和 SYS.COLUMNS。

```
1 select a.name table_name,
2        b.name index_name,
3        d.name column_name,
4        c.index_column_id
5   from sys.tables a,
6        sys.indexes b,
7        sys.index_columns c,
8        sys.columns d
9  where a.object_id = b.object_id
10     and b.object_id = c.object_id
11     and b.index_id = c.index_id
12     and c.object_id = d.object_id
13     and c.column_id = d.column_id
14     and a.name = 'EMP'
```

13. 讨论

对查询来说，知道在哪些列上创建了或没有创建索引很重要。对于选择性极高，因而常用于筛选的列，在它们上面创建索引可以极大地提高查询的性能。在表之间建立连接时，索引也很有用。知道在哪些列上创建了索引后，就在解决性能问题的路上迈出了第一步。另外，你可能还想获悉有关索引本身的信息：有多深（层级数）、有多少个不同的键、有多少个叶子块（leaf block）等。在本实例的解决方案查询的视图/表中，也提供了这些信息。

5.4 列出表的约束

1. 问题

你想列出给表定义的约束以及这些约束是在哪些列上定义的。例如，你想获悉 **EMP** 表的约束以及这些约束是在哪些列上定义的。

2. 解决方案

DB2

查询 `SYSCAT.TABCONST` 和 `SYSCAT.COLUMNS`。

```
1 select a.tabname, a.constname, b.colname, a.type
2   from syscat.tabconst a,
3        syscat.columns b
4  where a.tabname    = 'EMP'
5        and a.tabschema = 'SMEAGOL'
6        and a.tabname  = b.tabname
7        and a.tabschema = b.tabschema
```

Oracle

查询 `SYS.ALL_CONSTRAINTS` 和 `SYS.ALL_CONS_COLUMNS`。

```
1 select a.table_name,
2        a.constraint_name,
3        b.column_name,
4        a.constraint_type
5   from all_constraints a,
6        all_cons_columns b
7  where a.table_name    = 'EMP'
8        and a.owner      = 'SMEAGOL'
```



```
9    and a.table_name      = b.table_name
10   and a.owner           = b.owner
11   and a.constraint_name = b.constraint_name
```

PostgreSQL、MySQL 和 SQL Server

查询 INFORMATION_SCHEMA.TABLE_CONSTRAINTS 和 INFORMATION_SCHEMA.KEY_COLUMN_USAGE。

```
1  select a.table_name,
2         a.constraint_name,
3         b.column_name,
4         a.constraint_type
5  from information_schema.table_constraints a,
6       information_schema.key_column_usage b
7  where a.table_name      = 'EMP'
8        and a.table_schema = 'SMEAGOL'
9        and a.table_name  = b.table_name
10       and a.table_schema = b.table_schema
11       and a.constraint_name = b.constraint_name
```

13. 讨论

在关系数据库中，约束非常重要，因此你肯定需要知道表有哪些约束。列出表的约束很有用，其中的原因很多：找出缺少主键的表，找出哪些列应为外键但实际上不是（子表包含的数据与父表不同，而你想知道其中的原因），以及理解检查约束。（列是否可为 NULL？列必须满足特定条件吗？）

5.5 列出没有相应索引的外键

1. 问题

你想列出没有相应索引的外键，例如，你想知道是否在 EMP 表的外键上创建了索引。

2. 解决方案

DB2

查询

SYSCAT.TABCONST、SYSCAT.KEYCOLUSE、SYSCAT.INDEXES 和 SYSCAT.INDEXCOLUSE。

```
1  select fkeys.tabname,
2         fkeys.constname,
3         fkeys.colname,
4         ind_cols.indname
5  from (
6  select a.tabschema, a.tabname, a.constname, b.colname
7  from syscat.tabconst a,
8       syscat.keycoluse b
9  where a.tabname      = 'EMP'
10     and a.tabschema   = 'SMEAGOL'
11     and a.type         = 'F'
12     and a.tabname      = b.tabname
13     and a.tabschema    = b.tabschema
14     ) fkeys
15  left join
16  (
17  select a.tabschema,
18         a.tabname,
19         a.indname,
20         b.colname
21  from syscat.indexes a,
```

```

22      syscat.indexcoluse b
23 where a.indschema = b.indschema
24      and a.indname = b.indname
25      ) ind_cols
26      on (fkeys.tabschema = ind_cols.tabschema
27          and fkeys.tabname = ind_cols.tabname
28          and fkeys.colname = ind_cols.colname )
29 where ind_cols.indname is null

```

Oracle

查询 SYS.ALL_CONS_COLUMNS、SYS.ALL_CONSTRAINTS 和 SYS.ALL_IND_COLUMNS。

```

1  select a.table_name,
2         a.constraint_name,
3         a.column_name,
4         c.index_name
5  from all_cons_columns a,
6       all_constraints b,
7       all_ind_columns c
8  where a.table_name = 'EMP'
9        and a.owner = 'SMEAGOL'
10        and b.constraint_type = 'R'
11        and a.owner = b.owner
12        and a.table_name = b.table_name
13        and a.constraint_name = b.constraint_name
14        and a.owner = c.table_owner (+)
15        and a.table_name = c.table_name (+)
16        and a.column_name = c.column_name (+)
17        and c.index_name is null

```

查询

INFORMATION_SCHEMA.KEY_COLUMN_USAGE、INFORMATION_SCHEMA和 PG_CATALOG.PG_INDEXES。

```

1  select fkeys.table_name,
2         fkeys.constraint_name,
3         fkeys.column_name,
4         ind_cols.indexname

```

```

5   from (
6 select a.constraint_schema,
7        a.table_name,
8        a.constraint_name,
9        a.column_name
10  from information_schema.key_column_usage a,
11       information_schema.referential_constraints b
12 where a.constraint_name = b.constraint_name
13       and a.constraint_schema = b.constraint_schema
14       and a.constraint_schema = 'SMEAGOL'
15       and a.table_name = 'EMP'
16       ) fkeys
17  left join
18  (
19 select a.schemaname, a.tablename, a.indexname, b.column_name
20  from pg_catalog.pg_indexes a,
21       information_schema.columns b
22 where a.tablename = b.table_name
23       and a.schemaname = b.table_schema
24       ) ind_cols
25  on ( fkeys.constraint_schema = ind_cols.schemaname
26       and fkeys.table_name = ind_cols.tablename
27       and fkeys.column_name = ind_cols.column_name )
28 where ind_cols.indexname is null

```

MySQL

可以使用命令 **SHOW INDEX** 来检索索引信息，比如索引的名称、索引包含的列，以及这些列在索引中的序数位置。另外，可以通过查询

INFORMATION_SCHEMA.KEY_COLUMN_USAGE 来列出给定表的外键。在 MySQL 5 中，据说会自动在外键上创建索引，但实际上可能被删除。要确定外键列对应的索引是否被删除了，可以执行命令 **SHOW INDEX**，并将其输出同查询

INFORMATION_SCHEMA.KEY_COLUMN_USAGE.COLUMN_NAME 的结果进行比较。如果 **COLUMN_NAME** 包含在 **KEY_COLUMN_USAGE** 中，但命令 **SHOW INDEX** 没有返回它，那么便可确定没有在该列上创建索引。

SQL Server

查询

SYS.TABLES、SYS.FOREIGN_KEYS、SYS.COLUMNS、SYS.IN和 SYS.INDEX_COLUMNS。

```
1  select fkeys.table_name,
2         fkeys.constraint_name,
3         fkeys.column_name,
4         ind_cols.index_name
5  from (
6  select a.object_id,
7         d.column_id,
8         a.name table_name,
9         b.name constraint_name,
10        d.name column_name
11  from sys.tables a
12       join
13       sys.foreign_keys b
14  on ( a.name = 'EMP'
15       and a.object_id = b.parent_object_id
16     )
17       join
18       sys.foreign_key_columns c
19  on ( b.object_id = c.constraint_object_id )
20       join
21       sys.columns d
22  on ( c.constraint_column_id = d.column_id
23       and a.object_id = d.object_id
24     )
25     ) fkeys
26  left join
27  (
28  select a.name index_name,
29         b.object_id,
30         b.column_id
31  from sys.indexes a,
32       sys.index_columns b
33  where a.index_id = b.index_id
34     ) ind_cols
35  on ( fkeys.object_id = ind_cols.object_id
36       and fkeys.column_id = ind_cols.column_id )
```

13. 讨论

在修改行时，每家 **RDBMS** 厂商都使用自己的锁定机制。在通过外键建立了父子关系的情况下，在子列上创建索引可能影响锁定（详情请参阅你使用的 **RDBMS** 的文档）。经常需要通过外键将子表连接到父表，因此在外键上创建索引可能有助于改善性能。

5.6 使用SQL生成SQL

1. 问题

你想创建动态 SQL 语句，目的可能是要自动执行维护任务。具体地说，你要完成 3 项任务：计算表中的行数，禁用给表定义的外键约束，根据表中的数据生成插入脚本。

2. 解决方案

使用字符串来创建 SQL 语句，对于其中需要填入的值（比如命令针对的对象的名称）将由来自表的数据提供。别忘了，这里的查询只生成语句，你必须通过脚本（或其他执行 SQL 语句的方式）手动执行它们。下面的解决方案适用于 Oracle 系统，对于其他 RDBMS，采用的方法完全相同，唯一不同的是数据字典的名称和日期格式等。下面显示的输出是在我的笔记本电脑的 Oracle 实例中执行这些查询时得到的，你执行这些查询时，返回的结果集肯定不一样。

```
/* 生成计算所有表中行数的SQL */

select 'select count(*) from '||table_name||';' cnts
  from user_tables;

CNTS
-----
select count(*) from ANT;
select count(*) from BONUS;
select count(*) from DEMO1;
select count(*) from DEMO2;
select count(*) from DEPT;
select count(*) from DUMMY;
select count(*) from EMP;
select count(*) from EMP_SALES;
```

```

select count(*) from EMP_SCORE;
select count(*) from PROFESSOR;
select count(*) from T;
select count(*) from T1;
select count(*) from T2;
select count(*) from T3;
select count(*) from TEACH;
select count(*) from TEST;
select count(*) from TRX_LOG;
select count(*) from X;

/* 禁用所有表中定义的外键约束 */

select 'alter table '||table_name||
       ' disable constraint '||constraint_name||';' cons
  from user_constraints
 where constraint_type = 'R';

CONS
-----
alter table ANT disable constraint ANT_FK;
alter table BONUS disable constraint BONUS_FK;
alter table DEMO1 disable constraint DEMO1_FK;
alter table DEMO2 disable constraint DEMO2_FK;
alter table DEPT disable constraint DEPT_FK;
alter table DUMMY disable constraint DUMMY_FK;
alter table EMP disable constraint EMP_FK;
alter table EMP_SALES disable constraint EMP_SALES_FK;
alter table EMP_SCORE disable constraint EMP_SCORE_FK;
alter table PROFESSOR disable constraint PROFESSOR_FK;

/* 根据EMP表的一些列生成插入脚本 */

select 'insert into emp(empno,ename,hiredate) '||chr(10)||
       'values( '||empno||','||''''||ename
       ||'',to_date('||''''||hiredate||''') );' inserts
  from emp
 where deptno = 10;

INSERTS
-----
insert into emp(empno,ename,hiredate)
values( 7782,'CLARK',to_date('09-JUN-2006 00:00:00') );

insert into emp(empno,ename,hiredate)

```



```
values( 7839, 'KING', to_date('17-NOV-2006 00:00:00') );  
  
insert into emp(empno,ename,hiredate)  
values( 7934, 'MILLER', to_date('23-JAN-2007 00:00:00') );
```

13. 讨论

需要创建可移植的脚本（以便在多种环境中使用它进行测试）时，使用 SQL 来生成 SQL 很有用。另外，从上述解决方案可知，需要执行批量维护任务以及同时获悉多个对象的信息时，使用 SQL 来生成 SQL 也很有用。使用 SQL 生成 SQL 是一种非常简单的操作，练得越多，完成起来就越容易。该解决方案只是让你对如何创建动态 SQL 脚本有深入认识，坦率地说，这没什么难的，只要去做就能成功。

5.7 描述Oracle数据库中的数据字典视图

1. 问题

你使用的是 Oracle，但不记得有哪些数据字典视图可供你使用，更不记得这些视图的列定义。雪上加霜的是，你还无法访问 Oracle 文档。

2. 解决方案

这是一个专门针对 Oracle 的实例。Oracle 不仅维护着一组健壮的数据字典视图，还提供了包含数据字典视图文档的数据字典视图。这真是完美的循环引用。

查询视图 **DICTIONARY**，列出各个数据字典视图及其用途。

```
select table_name, comments
  from dictionary
 order by table_name;
```

TABLE_NAME	COMMENTS
-----	-----
ALL_ALL_TABLES	Description of all object and
relational	tables accessible to the user
ALL_APPLY	Details about each apply process
that	dequeues from the queue visible to
the	current user
...	

查询视图 **DICTIONARY**，获取有关给定数据字典视图包含的列的描述。

```
select column_name, comments
      from dict_columns
     where table_name = 'ALL_TAB_COLUMNS';
```

COLUMN_NAME	COMMENTS
OWNER	
TABLE_NAME	Table, view or cluster name
COLUMN_NAME	Column name
DATA_TYPE	Datatype of the column
DATA_TYPE_MOD	Datatype modifier of the column
DATA_TYPE_OWNER	Owner of the datatype of the column
DATA_LENGTH	Length of the column in bytes
DATA_PRECISION	Length: decimal digits (NUMBER) or digits (FLOAT)

13. 讨论

在 Oracle 文档不像现在通过互联网触手可及的时代，Oracle 提供的视图 **DICTIONARY** 和 **DICTION_COLUMNS** 带来了极大的便利。只需知道这两个视图，就能深入了解其他数据字典视图，进而深入了解整个数据库。

即便是现在，知道 **DICTIONARY** 和 **DICTION_COLUMNS** 也能提供极大的便利。如果你不确定哪个视图描述了给定的对象类型，那么可以编写并执行一个使用通配符的查询，以找到相应的视图。例如，要获悉哪些视图可能描述了模式中的表，可以像下面这样做。

```
select table_name, comments
```

```
from dictionary
where table_name LIKE '%TABLE%'
order by table_name;
```

这个查询返回了所有包含术语 **TABLE** 的数据字典视图名称。这种方法利用了这样一点，即 Oracle 以相当统一的方式给数据字典视图命名。所有描述表的视图的名称都可能包含术语 **TABLE**。[在有些情况下（比如在 **ALL_TAB_COLUMNS** 中），**TABLE** 被简写为 **TAB**。]

5.8 小结

元数据查询让 SQL 能够做更多的工作，还让你在有些情况下无须熟悉自己的数据库。在使用的数据库及其结构都很复杂时，这很有用。

第 6 章 处理字符串

本章重点介绍如何在 SQL 中操作字符串。别忘了，SQL 并非专门用于执行复杂的字符串操作，在有些情况下，你可能会发现在 SQL 中处理字符串很麻烦，甚至令人沮丧。虽然 SQL 存在这样的局限性，但 DBMS 提供了一些很有用的内置函数，我们会尝试创造性地使用它们。本章淋漓尽致地证明了本书前言中要表达的观点：SQL 有优点，也有缺点。但愿你学过本章之后，对 SQL 在字符串处理方面能做什么以及不能做什么有更深入的认识。在很多情况下，你会意外地发现字符串解析和变换非常容易，而在另一些情况下，你又会惊骇于完成特定任务所需的 SQL。

本章的很多实例使用了函数 **TRANSLATE** 和 **REPLACE**。当前，本书提及的所有 DBMS 都支持它们，唯一的例外是 MySQL，它只支持替换。需要指出的是，在 MySQL 中，可以使用嵌套的 **REPLACE** 函数来实现 **TRANSLATE** 的效果。

本章的第一个实例至关重要，因为后面的多种解决方案均会使用它。在大多数情况下，你希望以每次穿过一个字符的方式来遍历字符串，可惜在 SQL 中，这种任务完成起来并不容易。由于 SQL 的循环功能有限，你需要通过模拟循环来遍历字符串。我们将这种操作称为“走查字符串”，本章的第一个实例对此做了介绍。使用 SQL 来分析字符串时，这种操作非常重要，本章的大部分实例参照并使用了它。强烈建议你弄清楚这种方法的工作原理。

6.1 走查字符串

1. 问题

你想遍历一个字符串，将其中的每个字符都作为一行返回，但 SQL 没有提供循环操作。例如，你想分 4 行显示 EMP 表中的 ENAME "KING"，每一行只包含其中的一个字符。

2. 解决方案

使用笛卡儿积计算以每行一个字符的方式返回字符串中所有的字符时需要多少行，然后使用 DBMS 内置的字符串解析函数提取感兴趣的字符。（如果使用的是 SQL Server，请用 SUBSTRING 和 DATALENGTH 替换 SUBSTR 和 LENGTH。）

```
1 select substr(e.ename,iter.pos,1) as C
2   from (select ename from emp where ename = 'KING') e,
3        (select id as pos from t10) iter
4  where iter.pos <= length(e.ename)
```

```
C
-
K
I
N
G
```

3. 讨论

要遍历字符串的字符，关键是要连接到一张表，该表包含足

够多的行，使得能够执行所需的迭代次数。这里使用的是 **T10** 表，它包含 10 行数据（这张表只有 **ID** 列，其中存储了从 1 到 10 的值）。这个查询最多只能返回 10 行。

下面的例子显示了 **E** 和 **ITER**（特定的姓名和 **T10** 表中 10 行数据）的笛卡儿积（没有对 **ENAME** 进行分析）。

<pre>select ename, iter.pos from (select ename from emp where ename = 'KING') e, (select id as pos from t10) iter</pre>	
ENAME	POS
-----	-----
KING	1
KING	2
KING	3
KING	4
KING	5
KING	6
KING	7
KING	8
KING	9
KING	10

内嵌视图 **E** 的基数为 1，而内嵌视图 **ITER** 的基数为 10，因此笛卡儿积为 10 行。要在 SQL 中模拟循环，首先需要生成这样的积。



通常将 **T10** 表称为“转置”（pivot）表。

为了在返回 4 行数据后退出循环，该解决方案使用了一个 **WHERE** 子句。为了让结果集包含的行数与姓名包含的字符数相同，这个 **WHERE** 子句指定了 **ITER.POS <= LENGTH(E.ENAME)** 作为条件。

<pre>select ename, iter.pos from (select ename from emp where ename = 'KING') e,</pre>	
--	--

<pre>(select id as pos from t10) iter where iter.pos <= length(e.ename)</pre>	
ENAME	POS
-----	-----
KING	1
KING	2
KING	3
KING	4

在 **E.ENAME** 中的每个字符都有对应的一行后，可以将 **ITER.POS** 作为 **SUBSTR** 的参数，以遍历字符串中的字符。由于每一行的 **ITER.POS** 值都比前一行大 1，因此可以使用它来返回 **E.ENAME** 中的下一个字符。这就是该解决方案的工作原理。

根据要完成的任务，你可能需要（也可能不需要）为字符串中的每个字符都生成一行数据。下面的查询演示了如何遍历 **E.ENAME** 并显示这个字符串的不同部分（多个字符）。

<pre>select substr(e.ename,iter.pos) a, substr(e.ename,length(e.ename)-iter.pos+1) b from (select ename from emp where ename = 'KING') e, (select id pos from t10) iter where iter.pos <= length(e.ename)</pre>	
A	B
-----	-----
KING	G
ING	NG
NG	ING
G	KING

在本章的实例中，最常见的场景如下：遍历整个字符串并为每个字符生成一行数据，或者遍历整个字符串并为每个特定的字符或分隔符生成一行数据。

6.2 在字符串字面量中嵌入引号

1. 问题

你想在字符串字面量中嵌入引号。例如，你希望使用 SQL 生成如下结果。

```
QMARKS
-----
g'day mate
beavers' teeth
'
```

2. 解决方案

下面的 3 条 **SELECT** 语句展示了生成引号（位于字符串中的引号和独立的引号）的不同方式。

```
1 select 'g''day mate' qmarks from t1 union all
2 select 'beavers'' teeth'    from t1 union all
3 select ''''                  from t1
```

3. 讨论

处理引号时，把它们想象成像圆括号那样通常将有所帮助。左括号必须有配套的右括号，引号也是如此。别忘了，在任何字符串中，引号的个数都必须是双数。如果想在字符串中嵌入一个引号，则必须使用两个引号。

```
select 'apples core', 'apple's core',
```

case when '' is null then 0 else 1 end from t1	
'APPLESCORE 'APPLE''SCOR CASEWHEN''ISNULLTHEN0ELSE1END	

apples core apple's core	0

下面是最简单的示例，其中外层的两个引号定义了一个字符串字面量，而内层的两个引号定义了该字面量的内容——它是由单个引号组成的。

select '''' as quote from t1	
Q	
-	
,	

处理引号时务必牢记，只包含两个引号，并且这两个引号中间没有任何字符的字符串字面量为 **NULL**。

6.3 计算字符串中特定字符出现的次数

1. 问题

你想计算特定的字符或子串在给定字符串中出现的次数。请看下面的字符串。

```
10,CLARK,MANAGER
```

你想确定这个字符串中有多少个逗号。

2. 解决方案

删除字符串中的逗号，再将原来的字符串长度与删除逗号后的字符串长度相减，就可以确定字符串中包含多少个逗号。所有 DBMS 都提供了获取字符串长度的函数以及从字符串中删除字符的函数。在大多数情况下，这两个函数分别是 **LENGTH** 和 **REPLACE**（SQL Server 用户需要用内置函数 **LEN** 替代 **LENGTH**）。

```
1 select (length('10,CLARK,MANAGER')-  
2         length(replace('10,CLARK,MANAGER',',','')))/length(',')  
3         as cnt  
4   from t1
```

3. 讨论

上述解决方案使用了简单的减法运算。第 1 行的 **LENGTH** 调用返回了字符串原来的长度，而第 2 行的第 1 个 **LENGTH** 调

用返回了（使用 **REPLACE**）将逗号删除后的字符串长度。

通过将上述两个长度相减，得到的差值就是字符串包含的逗号个数。最后一个操作是用长度差值除以查找的字符串的长度，仅当要查找的字符串的长度大于 1 时，这个除法运算才是必不可少的。下面的示例会计算 **LL** 在字符串“**HELLO HELLO**”中出现的次数，如果不执行除法运算，那么返回的结果将是错误的。

```
select
    (length('HELLO HELLO')-
    length(replace('HELLO HELLO','LL','')))/length('LL')
    as correct_cnt,
    (length('HELLO HELLO')-
    length(replace('HELLO HELLO','LL',''))) as incorrect_cnt
from t1
```

CORRECT_CNT	INCORRECT_CNT
2	4

6.4 将不想要的字符从字符串中删除

1. 问题

你想将特定的字符从数据中删除。一种使用场景是，处理格式糟糕的数值数据（尤其是金额数据）。在金额数据中，逗号被用作了千分位分隔符，其中还包含货币符号。另一种使用场景是，你要将数据库中的数据导出为 CSV 文件，但有一个文本字段包含逗号（访问 CSV 文件时，将把逗号视为分隔符）。请看下面的结果集。

ENAME	SAL
-----	-----
SMITH	800
ALLEN	1600
WARD	1250
JONES	2975
MARTIN	1250
BLAKE	2850
CLARK	2450
SCOTT	3000
KING	5000
TURNER	1500
ADAMS	1100
JAMES	950
FORD	3000
MILLER	1300

你要将其中所有的 0 和元音字母都删除，结果如下面的 STRIPPED1 列和 STRIPPED2 列所示。

ENAME	STRIPPED1	SAL	STRIPPED2
-----	-----	-----	-----
SMITH	SMTH	800	8
ALLEN	LLN	1600	16
WARD	WRD	1250	125
JONES	JNS	2975	2975

MARTIN	MRTN	1250	125
BLAKE	BLK	2850	285
CLARK	CLRK	2450	245
SCOTT	SCTT	3000	3
KING	KNG	5000	5
TURNER	TRNR	1500	15
ADAMS	DMS	1100	11
JAMES	JMS	950	95
FORD	FRD	3000	3
MILLER	MLLR	1300	13

12. 解决方案

每个 DBMS 都提供了可以将不想要的字符从字符串中删除的相关函数。解决这个问题时，最有用的函数是 **REPLACE** 和 **TRANSLATE**。

DB2、Oracle、PostgreSQL 和 SQL Server

使用内置函数 **TRANSLATE** 和 **REPLACE** 删除不想要的字符和子串。

```

1 select ename,
2        replace(translate(ename,'aaaaa','AEIOU'),'a','') as
stripped1,
3        sal,
4        replace(cast(sal as char(4)),'0','') as stripped2
5 from emp

```

注意，在 DB2 中给列指定别名时，关键字 **AS** 是可选的，因此可以省略。

MySQL

MySQL 中没有函数 **TRANSLATE**，因此需要多次调用函数

REPLACE。

```
1 select ename,  
2        replace(  
3        replace(  
4        replace(  
5        replace(  
6        replace(ename,'A',''),'E',''),'I',''),'O',''),'U','')  
7        as stripped1,  
8        sal,  
9        replace(sal,0,'') stripped2  
10 from emp
```

13. 讨论

为了删除所有的 0，调用内置函数 **REPLACE**。为了删除元音字母，使用函数 **TRANSLATE** 将所有元音字母都转换为特定字符（这里是 a，也可以使用其他字符），然后使用 **REPLACE** 将这种字符都删除。

6.5 将数字数据和字符数据分开

1. 问题

你在同一列中同时存储了数字数据和字符数据。如果你使用的是同时存储了数量和度量单位（或货币符号）的遗留数据（例如，在列中存储 100 km、AUD\$200 或 40 pounds，而不是将数量和单位存储在不同的列中），那么很可能会遇到这种情况。

你想将字符数据和数字数据分开。请看下面的结果集。

```
DATA
-----
SMITH800
ALLEN1600
WARD1250
JONES2975
MARTIN1250
BLAKE2850
CLARK2450
SCOTT3000
KING5000
TURNER1500
ADAMS1100
JAMES950
FORD3000
MILLER1300
```

你希望结果是下面这样的。

ENAME	SAL
SMITH	800
ALLEN	1600
WARD	1250
JONES	2975
MARTIN	1250

BLAKE	2850
CLARK	2450
SCOTT	3000
KING	5000
TURNER	1500
ADAMS	1100
JAMES	950
FORD	3000
MILLER	1300

12. 解决方案

使用内置函数 **TRANSLATE** 和 **REPLACE** 将字符数据和数字数据分开。与本章的其他实例一样，诀窍是使用函数 **TRANSLATE** 将多种字符转换为特定的字符，这样无须搜索多个数字或字符，而只需搜索表示所有数字或字符的字符。

DB2

使用函数 **TRANSLATE** 和 **REPLACE** 将数字数据和字符数据分开。

```

1 select replace(
2     translate(data,'0000000000','0123456789'),'0','') ename,
3     cast(
4     replace(
5     translate(lower(data),repeat('z',26),
6         'abcdefghijklmnopqrstuvwxyz'),'z','') as integer) sal
7   from (
8   select ename||cast(sal as char(4)) data
9   from emp
10  ) x

```

Oracle

使用函数 **TRANSLATE** 和 **REPLACE** 将数字数据和字符数据分

开。

```
1 select replace(
2     translate(data,'0123456789','0000000000'),'0') ename,
3     to_number(
4         replace(
5             translate(lower(data),
6                 'abcdefghijklmnopqrstuvwxyz',
7                 rpad('z',26,'z')),'z')) sal
8 from (
9 select ename||sal data
10 from emp
11 )
```

PostgreSQL

使用函数 **TRANSLATE** 和 **REPLACE** 将数字数据和字符数据分开。

```
1 select replace(
2     translate(data,'0123456789','0000000000'),'0','') as
ename,
3     cast(
4         replace(
5             translate(lower(data),
6                 'abcdefghijklmnopqrstuvwxyz',
7                 rpad('z',26,'z')),'z','') as integer) as sal
8 from (
9 select ename||sal as data
10 from emp
11 ) x
```

SQL Server

使用函数 **TRANSLATE** 和 **REPLACE** 将数字数据和字符数据分开。

```
1 select replace(
2     translate(data,'0123456789','0000000000'),'0','') as
```

```

ename,
  3          cast(
  4          replace(
  5          translate(lower(data),
  6          'abcdefghijklmnopqrstuvwxyz',
  7          replicate('z',26),'z','') as integer) as sal
  8  from (
  9  select concat(ename,sal) as data
 10  from emp
 11  ) x

```

13. 讨论

在不同的 DBMS 中，语法稍有不同，但使用的方法相同，这里的讨论将以 Oracle 解决方案为例。解决这个问题关键在于将数字数据和字符数据隔离，为此可以使用函数 **TRANSLATE** 和 **REPLACE**。要提取数字数据，首先使用函数 **TRANSLATE** 将所有的字符数据隔离。

```

select data,
       translate(lower(data),
                 'abcdefghijklmnopqrstuvwxyz',
                 rpad('z',26,'z')) sal
  from (select ename||sal data from emp)

```

DATA	SAL
SMITH800	zzzzz800
ALLEN1600	zzzzz1600
WARD1250	zzzz1250
JONES2975	zzzzz2975
MARTIN1250	zzzzzz1250
BLAKE2850	zzzzz2850
CLARK2450	zzzzz2450
SCOTT3000	zzzzz3000
KING5000	zzzz5000
TURNER1500	zzzzzz1500
ADAMS1100	zzzzz1100
JAMES950	zzzzz950

FORD3000	zzzz3000
MILLER1300	zzzzzz1300

使用函数 **TRANSLATE** 将每个非数字字符都转换为小写字母 **z**，然后使用函数 **REPLACE** 将每条记录中的小写字母 **z** 都删除，只留下可以强制转换为数字的数字字符。

```
select data,
       to_number(
         replace(
           translate(lower(data),
                     'abcdefghijklmnopqrstuvwxyz',
                     rpad('z',26,'z')), 'z')) sal
  from (select ename||sal data from emp)
```

DATA	SAL
SMITH800	800
ALLEN1600	1600
WARD1250	1250
JONES2975	2975
MARTIN1250	1250
BLAKE2850	2850
CLARK2450	2450
SCOTT3000	3000
KING5000	5000
TURNER1500	1500
ADAMS1100	1100
JAMES950	950
FORD3000	3000
MILLER1300	1300

为了提取非数字字符，需要使用函数 **TRANSLATE** 将数字字符隔离。

```
select data,
       translate(data,'0123456789','0000000000') ename
  from (select ename||sal data from emp)
```

DATA	ENAME
-----	-----

SMITH800	SMITH000
ALLEN1600	ALLEN0000
WARD1250	WARD0000
JONES2975	JONES0000
MARTIN1250	MARTIN0000
BLAKE2850	BLAKE0000
CLARK2450	CLARK0000
SCOTT3000	SCOTT0000
KING5000	KING0000
TURNER1500	TURNER0000
ADAMS1100	ADAMS0000
JAMES950	JAMES000
FORD3000	FORD0000
MILLER1300	MILLER0000

使用函数 **TRANSLATE** 将每个数字字符都转换为 0，然后使用函数 **REPLACE** 将每条记录中的 0 都删除，只留下非数字字符。

```
select data,
       replace(translate(data,'0123456789','0000000000'),'0')
ename
  from (select ename||sal data from emp)
```

DATA	ENAME
-----	-----
SMITH800	SMITH
ALLEN1600	ALLEN
WARD1250	WARD
JONES2975	JONES
MARTIN1250	MARTIN
BLAKE2850	BLAKE
CLARK2450	CLARK
SCOTT3000	SCOTT
KING5000	KING
TURNER1500	TURNER
ADAMS1100	ADAMS
JAMES950	JAMES
FORD3000	FORD
MILLER1300	MILLER

最后，将这两种方法组合起来，就得到了所需的解决方案。

6.6 判断字符串是否只包含字母和数字

1. 问题

你想从一张表中返回这样的行，即其特定列只包含字母和数字。请看下面的视图 **V**（SQL Server 用户应该使用运算符 **+** 而不是 **||** 来执行拼接操作）。

```
create view V as
select ename as data
  from emp
 where deptno=10
 union all
select ename||', $'|| cast(sal as char(4)) ||'.00' as data
  from emp
 where deptno=20
 union all
select ename|| cast(deptno as char(4)) as data
  from emp
 where deptno=30
```

视图 **V** 代表你的表，它包含的内容如下。

```
DATA
-----
CLARK
KING
MILLER
SMITH, $800.00
JONES, $2975.00
SCOTT, $3000.00
ADAMS, $1100.00
FORD, $3000.00
ALLEN30
WARD30
MARTIN30
BLAKE30
TURNER30
JAMES30
```

然而，你只想从这个视图中返回如下记录。

DATA

CLARK
KING
MILLER
ALLEN30
WARD30
MARTIN30
BLAKE30
TURNER30
JAMES30

简言之，你不想返回除了字母和数字还包含其他字符的行。

12. 解决方案

要解决这个问题，第一感觉是搜索可能出现在字符串中的所有非字母数字字符，但你会发现，采取相反的做法更容易：查找所有的字母数字字符。采取这种方法时，可以将所有的字母数字字符都转换为特定字符，从而将它们视为单个字符进行处理。为什么要这样做呢？这是因为这样可以将字母数字字符作为一个整体进行操作。将所有的字母数字字符都转换为你选择的字符后，将字母数字字符与其他字符隔离就易如反掌了。

DB2

使用函数 **TRANSLATE** 将所有的字母数字字符都转换为特定的字符，然后找出那些除了这个特定字符外还包含其他字符的行。在 **DB2** 中，必须在视图 **V** 中调用函数 **CAST**，否则类型转换错误将导致无法创建这个视图。强制转换为 **CHAR** 时，必须倍加小心，因为其长度是固定的（数据不够的话，

将进行填充）。

```
1 select data
2   from V
3  where translate(lower(data),
4                  repeat('a',36),
5                  '0123456789abcdefghijklmnopqrstuvwxyz') =
6                  repeat('a',length(data))
```

MySQL

在 MySQL 中，创建视图 V 的语法稍有不同。

```
create view V as
select ename as data
  from emp
 where deptno=10
 union all
select concat(ename,', $',sal,'.00') as data
  from emp
 where deptno=20
 union all
select concat(ename,deptno) as data
  from emp
 where deptno=30
```

使用正则表达式轻松地找出包含非字母数字字符的行。

```
1 select data
2   from V
3  where data regexp '^[^0-9a-zA-Z]' = 0
```

Oracle 和 PostgreSQL

使用函数 **TRANSLATE** 将所有的字母数字字符都转换为特定的字符，然后找出那些除了这个特定字符外还包含其他字符的行。在 Oracle 和 PostgreSQL 中，可以不在视图 V 中调用函数 **CAST**。强制转换为 **CHAR** 时，必须倍加小心，因为其长

度是固定的（数据不够的话，将进行填充）。

如果你决定执行强制转换，那么请转换为 **VARCHAR** 或 **VARCHAR2**。

```
1 select data
2   from V
3  where translate(lower(data),
4                  '0123456789abcdefghijklmnopqrstuvwxyz',
5                  rpad('a',36,'a')) = rpad('a',length(data),'a')
```

SQL Server

与其他数据库使用的方法是一样的，但 SQL Server 中没有 **RPAD**。

```
1 select data
2   from V
3  where translate(lower(data),
4                  '0123456789abcdefghijklmnopqrstuvwxyz',
5                  replicate('a',36)) = replicate('a',len(data))
```

13. 讨论

这些解决方案的关键在于能够同时引用多个字符。使用函数 **TRANSLATE**，可以轻松地操作所有数字或所有字符，而无须通过迭代逐个检查每个字符。

DB2、Oracle、PostgreSQL 和 SQL Server

在视图 **V** 的 14 行数据中，只有 9 行只包含字母数字字符。要找出这些行，只需使用函数 **TRANSLATE**。在这个示例中，函数 **TRANSLATE** 将字符 0~9 和 a~z 都转换为了 a。执行转换后，将结果与一个长度与它相同但全为 a 的字符串进行比

较。如果二者相同，就说明包含的都是字母数字字符，没有其他字符。

使用函数 **TRANSLATE**（语法与 Oracle 解决方案相同）。

```
where translate(lower(data),
                '0123456789abcdefghijklmnopqrstuvwxyz',
                rpad('a',36,'a'))
```

将所有的数字和字母都转换为特定的字符（这里为 a）。如果在转换结果中，只包含特定的字符（字符 a），就说明转换前只包含字母数字字符。要查看转换后的结果，可以单独执行函数 **TRANSLATE**。

```
select data, translate(lower(data),
                        '0123456789abcdefghijklmnopqrstuvwxyz',
                        rpad('a',36,'a'))
  from V
```

DATA	TRANSLATE(LOWER(DATA))
CLARK	aaaaa
...	
SMITH, \$800.00	aaaaa, \$aaa.aa
...	
ALLEN30	aaaaaaa
...	

虽然字母数字字符被转换为了特定字符，但字符串的长度没变。由于长度相同，因此那些调用函数 **TRANSLATE** 返回全 a 的行就是要保留的行。通过将转换后的字符串与同长的全 a 字符串进行比较，保留了相等的行，拒绝了其他的行。

```
select data, translate(lower(data),
                        '0123456789abcdefghijklmnopqrstuvwxyz',
                        rpad('a',36,'a')) translated,
       rpad('a',length(data),'a') fixed
  from V
```

DATA	TRANSLATED	FIXED
-----	-----	-----
CLARK	aaaaa	aaaaa
...		
SMITH, \$800.00	aaaaa, \$aaa.aa	aaaaaaaaaaaaaaaa
...		
ALLEN30	aaaaaaa	aaaaaaa
...		

最后，只保留 TRANSLATED 和 FIXED 相等的字符串。

MySQL

WHERE 子句中的表达式（`where data regexp '[^0-9a-zA-Z]' = 0`）导致只包含数字和字母的行被返回。方括号内的取值范围 `0-9a-zA-Z` 表示所有的数字和字母，字符 `^` 表示求反，因此表达式 `^[^0-9a-zA-Z]` 的意思是“不是数字或字母”。如果满足这个条件，就返回 1，否则就返回 0。因此整个表达式的意思是，返回那些不满足条件“不是数字或字母”的行。

6.7 提取姓名中的首字母

1. 问题

你想将全名转换为首字母缩写。对于下面的姓名：

Stewie Griffin

你想返回如下内容。

S.G.

2. 解决方案

务必牢记，SQL 不像 C 或 Python 等语言那么灵活，因此在 SQL 中，创建可以处理任何姓名格式的通用解决方案并非易事。下面的解决方案要求全名要么由名和姓组成，要么由名、中间名/中间名缩写和姓组成。

DB2

使用内置函数 REPLACE、TRANSLATE 和 REPEAT 提取首字母。

```
1 select replace(  
2     replace(  
3         translate(replace('Stewie Griffin', '.', ''),  
4             repeat('#',26),  
5             'abcdefghijklmnopqrstuvwxyz'),  
6         '#','' ), ' ', '.' )  
7     || '.'  
8 from t1
```

MySQL

使用内置函数 **CONCAT**、**CONCAT_WS**、**SUBSTRING** 和 **SUBSTRING_INDEX** 提取首字母。

```
1 select case
2     when cnt = 2 then
3         trim(trailing '.' from
4             concat_ws('.',
5                 substr(substring_index(name, ' ',1),1,1),
6                 substr(name,
7                     length(substring_index(name, '
8                 ',1))+2,1),
9                 substr(substring_index(name, ' ', -1),1,1),
10                '.'))
11     else
12         trim(trailing '.' from
13             concat_ws('.',
14                 substr(substring_index(name, ' ',1),1,1),
15                 substr(substring_index(name, ' ', -1),1,1)
16             ))
17     end as initials
18 from (
19 select name,length(name)-length(replace(name, ' ', '')) as cnt
20 from (
21 select replace('Stewie Griffin','.', '') as name from t1
22 )y
23 )x
```

Oracle 和 PostgreSQL

使用内置函数 **REPLACE**、**TRANSLATE** 和 **RPAD** 提取首字母。

```
1 select replace(
2     replace(
3     translate(replace('Stewie Griffin', '.', ''),
4         'abcdefghijklmnopqrstuvwxyz',
5         rpad('#',26,'#') ), '#', '' ), ' ', '.' ) || '.'
6 from t1
```

SQL Server

```
1 select replace(  
2     replace(  
3     translate(replace('Stewie Griffin', '.', ''),  
4                'abcdefghijklmnopqrstuvwxyz',  
5                replicate('#',26) ), '#', '' ), ' ', '.' ) + '.'  
6 from t1
```

13. 讨论

通过隔离大写字母，可以提取姓名中的首字母。接下来将详细介绍针对各种数据库的解决方案。

DB2

函数 **REPLACE** 会删除姓名中的所有句点（处理中间名缩写），函数 **TRANSLATE** 会将所有小写字母都转换为 # 字符。

```
select translate(replace('Stewie Griffin', '.', ''),  
                repeat('#',26),  
                'abcdefghijklmnopqrstuvwxyz')  
from t1  
  
TRANSLATE('STE  
-----  
S##### G#####
```

至此，不是 # 的字符就是首字母。接下来，使用函数 **REPLACE** 删除所有的 # 字符。

```
select replace(  
    translate(replace('Stewie Griffin', '.', ''),  
              repeat('#',26),
```

```

                                'abcdefghijklmnopqrstuvwxyz'),'#','')
from t1

REP
---
S G

```

再次使用函数 **REPLACE** 将空格替换为句点。

```

select replace(
    replace(
        translate(replace('Stewie Griffin', '.', ''),
            repeat('#',26),
            'abcdefghijklmnopqrstuvwxyz'),'#',''),
    ',.') || '.'
from t1

REPLA
-----
S.G.

```

最后，在首字母缩写末尾添加一个句点。

Oracle 和 PostgreSQL

函数 **REPLACE** 会删除姓名中的所有句点（处理中间名缩写），函数 **TRANSLATE** 会将所有小写字母都转换为 # 字符。

```

select translate(replace('Stewie Griffin','.', ''),
    'abcdefghijklmnopqrstuvwxyz',
    rpad('#',26,'#'))
from t1

TRANSLATE('STE
-----
S##### G#####

```

至此，不是 # 的字符就是首字母。接下来，使用函数

REPLACE 删除所有的 # 字符。

```
select replace(
    translate(replace('Stewie Griffin','.', ''),
        'abcdefghijklmnopqrstuvwxyz',
        rpad('#',26,'#')), '#', '')
    from t1

REP
---
```

再次使用函数 **REPLACE** 将空格替换为句点。

```
select replace(
    replace(
        translate(replace('Stewie Griffin','.', ''),
            'abcdefghijklmnopqrstuvwxyz',
            rpad('#',26,'#') ), '#', ''), ' ', '.') || '.'
    from t1

REPLA
-----
S.G.
```

最后，在首字母缩写末尾添加一个句点。

MySQL

内嵌视图 **Y** 用于删除姓名中所有的句点。内嵌视图 **X** 可以确定姓名中有多少个空格，以便调用函数 **SUBSTR** 相应的次数来提取首字母。3 个 **SUBSTRING_INDEX** 调用会根据空格的位置从字符串中提取姓名的各个组成部分。因为本例中的姓名只有名和姓，所以将执行 **case** 语句的 **ELSE** 部分的代码。

```
select substr(substring_index(name, ' ',1),1,1) as a,
    substr(substring_index(name, ' ',-1),1,1) as b
    from (select 'Stewie Griffin' as name from t1) x
```

```
A B
- -
S G
```

如果问题中的姓名包含中间名或中间名缩写，那么将执行下面的代码来返回中间名缩写。

```
substr(name,length(substring_index(name, ' ',1))+2,1)
```

这些代码会先找到名的末尾，然后向前移动两个字符的位置，到达中间名或中间名缩写的开头，这就给 **SUBSTR** 指定了起始位置。由于只留下了一个字符，因此就成功地返回了中间名或中间名缩写的首字母。接下来，将首字母传递给函数 **CONCAT_WS**，该函数用句点来分隔首字母。

```
select concat_ws('.',
                 substr(substring_index(name, ' ',1),1,1),
                 substr(substring_index(name, ' ',-1),1,1),
                 '.' ) a
  from (select 'Stewie Griffin' as name from t1) x

A
-----
S.G..
```

最后，剔除首字母缩写末尾多余的句点。

6.8 根据部分字符串排序

1. 问题

你想根据子串对结果集进行排序。请看下面的记录。

```
ENAME
-----
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
ADAMS
JAMES
FORD
MILLER
```

你想根据姓名的最后两个字符对记录进行排序。

```
ENAME
-----
ALLEN
TURNER
MILLER
JONES
JAMES
MARTIN
BLAKE
ADAMS
KING
WARD
FORD
CLARK
SMITH
```

12. 解决方案

本解决方案的关键是找到并使用 DBMS 内置函数来提取用作排序依据的子串。这通常是使用函数 SUBSTR 实现的。

DB2、Oracle、MySQL 和 PostgreSQL

结合使用内置函数 LENGTH 和 SUBSTR 提取字符串的特定部分，并将其作为排序依据。

```
1 select ename
2   from emp
3  order by substr(ename,length(ename)-1,2)
```

SQL Server

使用函数 SUBSTRING 和 LEN 提取字符串的特定部分，并将其作为排序依据。

```
1 select ename
2   from emp
3  order by substring(ename,len(ename)-1,2)
```

13. 讨论

通过在 ORDER BY 子句中使用 SUBSTR 表达式，可以提取字符串的任何部分，并将其作为结果集排序依据。不仅仅是 SUBSTR 表达式，其他任何表达式的结果都可以作为排序依据。

6.9 根据字符串中的数字排序

1. 问题

你想根据字符串中的数字对结果集进行排序。请看下面的视图。

```
create view V as
select e.ename ||' '||
       cast(e.empno as char(4))||' '||
       d.dname as data
  from emp e, dept d
 where e.deptno=d.deptno
```

这个视图返回的数据如下。

```
DATA
-----
CLARK    7782 ACCOUNTING
KING     7839 ACCOUNTING
MILLER   7934 ACCOUNTING
SMITH    7369 RESEARCH
JONES    7566 RESEARCH
SCOTT    7788 RESEARCH
ADAMS    7876 RESEARCH
FORD     7902 RESEARCH
ALLEN    7499 SALES
WARD     7521 SALES
MARTIN   7654 SALES
BLAKE    7698 SALES
TURNER   7844 SALES
JAMES    7900 SALES
```

你要根据位于员工姓名和所属部门之间的员工编号对结果进行排序。

```
DATA
-----
```

SMITH	7369	RESEARCH
ALLEN	7499	SALES
WARD	7521	SALES
JONES	7566	RESEARCH
MARTIN	7654	SALES
BLAKE	7698	SALES
CLARK	7782	ACCOUNTING
SCOTT	7788	RESEARCH
KING	7839	ACCOUNTING
TURNER	7844	SALES
ADAMS	7876	RESEARCH
JAMES	7900	SALES
FORD	7902	RESEARCH
MILLER	7934	ACCOUNTING

12. 解决方案

下面的解决方案使用的函数和语法因 DBMS 而异，但所有解决方案采用的方法都相同（使用内置函数 **REPLACE** 和 **TRANSLATE**）。基本思路是使用函数 **REPLACE** 和 **TRANSLATE** 将字符串中的非数字字符删除，只留下用作排序依据的数字值。

DB2

使用内置函数 **REPLACE** 和 **TRANSLATE** 提取字符串中的数字值，并将其用作排序依据。

```

1 select data
2   from V
3  order by
4         cast(
5         replace(
6         translate(data,repeat('#',length(data)),
7         replace(
8         translate(data,'#####','0123456789'),
9         '#','')), '#','') as integer)

```

Oracle

使用内置函数 **REPLACE** 和 **TRANSLATE** 提取字符串中的数字值，并将其用作排序依据。

```
1 select data
2   from V
3  order by
4         to_number(
5         replace(
6         translate(data,
7         replace(
8         translate(data,'0123456789','#####'),
9         '#'),rpad('#',20,'#')),'#'))
```

PostgreSQL

使用内置函数 **REPLACE** 和 **TRANSLATE** 提取字符串中的数字值，并将其用作排序依据。

```
1 select data
2   from V
3  order by
4         cast(
5         replace(
6         translate(data,
7         replace(
8         translate(data,'0123456789','#####'),
9         '#',''),rpad('#',20,'#')),'#','') as integer)
```

MySQL

本书撰写之时，MySQL 中没有提供函数 **TRANSLATE**。

13. 讨论

视图 **V** 只是提供了行，用于证明这里的解决方案可行。这个视图会拼接 **EMP** 表中的多列。上述解决方案演示了如何将这样拼接而成的文本作为输入，并根据嵌入在其中的员工号进行排序。

每种解决方案中的 **ORDER BY** 子句都令人望而生畏，但效果非常好，而且如果你分别对各部分进行解读，则会发现它很容易理解。要根据字符串中的数字进行排序，最简单的做法是删除所有的非数字字符。删除非数字字符后，余下的唯一工作是将由数字字符组成的字符串转换为数字，然后将其作为排序依据。研究各个函数调用前，弄清楚这些函数的调用顺序非常重要。调用是从最内层开始的，顺序如下：

TRANSLATE（第 8 行）、**REPLACE**（第 7 行）、**TRANSLATE**（第 6 行）、**REPLACE**（第 5 行）。最后，使用函数 **CAST** 将结果作为数字返回。

首先，将数字字符转换为字符串的其他部分没有的字符，这里使用函数 **TRANSLATE** 将所有的数字字符都转换为了 **#**。例如，下面的查询显示了原始数据（左边）和第一次转换的结果（右边）。

```
select data,
       translate(data,'0123456789','#####') as tmp
from V
```

DATA			TMP		
-----			-----		
CLARK	7782	ACCOUNTING	CLARK	####	ACCOUNTING
KING	7839	ACCOUNTING	KING	####	ACCOUNTING
MILLER	7934	ACCOUNTING	MILLER	####	ACCOUNTING
SMITH	7369	RESEARCH	SMITH	####	RESEARCH
JONES	7566	RESEARCH	JONES	####	RESEARCH
SCOTT	7788	RESEARCH	SCOTT	####	RESEARCH
ADAMS	7876	RESEARCH	ADAMS	####	RESEARCH
FORD	7902	RESEARCH	FORD	####	RESEARCH
ALLEN	7499	SALES	ALLEN	####	SALES
WARD	7521	SALES	WARD	####	SALES

MARTIN	7654	SALES	MARTIN	####	SALES
BLAKE	7698	SALES	BLAKE	####	SALES
TURNER	7844	SALES	TURNER	####	SALES
JAMES	7900	SALES	JAMES	####	SALES

TRANSLATE 会在字符串中找到每个数字字符，并将其转换为 **#** 字符。修改后的字符串被返回给 **REPLACE**（第 7 行），后者会删除所有的 **#** 字符。

<pre>select data, replace(translate(data,'0123456789','#####'),'#') as tmp from V</pre>					
DATA			TMP		
-----			-----		
CLARK	7782	ACCOUNTING	CLARK	ACCOUNTING	
KING	7839	ACCOUNTING	KING	ACCOUNTING	
MILLER	7934	ACCOUNTING	MILLER	ACCOUNTING	
SMITH	7369	RESEARCH	SMITH	RESEARCH	
JONES	7566	RESEARCH	JONES	RESEARCH	
SCOTT	7788	RESEARCH	SCOTT	RESEARCH	
ADAMS	7876	RESEARCH	ADAMS	RESEARCH	
FORD	7902	RESEARCH	FORD	RESEARCH	
ALLEN	7499	SALES	ALLEN	SALES	
WARD	7521	SALES	WARD	SALES	
MARTIN	7654	SALES	MARTIN	SALES	
BLAKE	7698	SALES	BLAKE	SALES	
TURNER	7844	SALES	TURNER	SALES	
JAMES	7900	SALES	JAMES	SALES	

然后，字符串被返回给解决方案中的第二个（最外层）**TRANSLATE**（第 6 行）。这个 **TRANSLATE** 会在原始字符串中进行查找，当找到出现在返回的字符串中的字符时，就将其转换为 **#** 字符。

这个转换让你能够将所有的非数字字符视为一个字符（因为它们都被转换为了 **#** 字符）。

```
select data, translate(data,
    replace(
        translate(data,'0123456789','#####'),
        '#'),
    rpad('#',length(data),'#')) as tmp
from V
```

DATA			TMP
-----			-----
CLARK	7782	ACCOUNTING	#####7782#####
KING	7839	ACCOUNTING	#####7839#####
MILLER	7934	ACCOUNTING	#####7934#####
SMITH	7369	RESEARCH	#####7369#####
JONES	7566	RESEARCH	#####7566#####
SCOTT	7788	RESEARCH	#####7788#####
ADAMS	7876	RESEARCH	#####7876#####
FORD	7902	RESEARCH	#####7902#####
ALLEN	7499	SALES	#####7499#####
WARD	7521	SALES	#####7521#####
MARTIN	7654	SALES	#####7654#####
BLAKE	7698	SALES	#####7698#####
TURNER	7844	SALES	#####7844#####
JAMES	7900	SALES	#####7900#####

接下来，调用 **REPLACE**（第 5 行）删除所有的 **#** 字符，只留下数字字符。

```
select data, replace(
    translate(data,
        replace(
            translate(data,'0123456789','#####'),
            '#'),
        rpad('#',length(data),'#')),'#') as tmp
from V
```

DATA			TMP
-----			-----
CLARK	7782	ACCOUNTING	7782
KING	7839	ACCOUNTING	7839
MILLER	7934	ACCOUNTING	7934
SMITH	7369	RESEARCH	7369
JONES	7566	RESEARCH	7566
SCOTT	7788	RESEARCH	7788

ADAMS	7876	RESEARCH	7876
FORD	7902	RESEARCH	7902
ALLEN	7499	SALES	7499
WARD	7521	SALES	7521
MARTIN	7654	SALES	7654
BLAKE	7698	SALES	7698
TURNER	7844	SALES	7844
JAMES	7900	SALES	7900

最后，根据使用的 DBMS，调用合适的函数（通常是 **CAST**）将结果转换为数字（第 4 行）。

<pre> select data, to_number(replace(translate(data, replace(translate(data,'0123456789','#####'), '#'), rpad('#',length(data),'#')),'#')) as tmp from V </pre>			
DATA			TMP
-----			-----
CLARK	7782	ACCOUNTING	7782
KING	7839	ACCOUNTING	7839
MILLER	7934	ACCOUNTING	7934
SMITH	7369	RESEARCH	7369
JONES	7566	RESEARCH	7566
SCOTT	7788	RESEARCH	7788
ADAMS	7876	RESEARCH	7876
FORD	7902	RESEARCH	7902
ALLEN	7499	SALES	7499
WARD	7521	SALES	7521
MARTIN	7654	SALES	7654
BLAKE	7698	SALES	7698
TURNER	7844	SALES	7844
JAMES	7900	SALES	7900

编写类似这样的查询时，在 **SELECT** 列表中使用涉及的表达式很有帮助，这让你能够轻松地查看中间结果。然而，本实例的目标是对结果进行排序，因此必须把所有的函数调用都

放在 ORDER BY 子句中。

```
select data
  from V
 order by
      to_number(
        replace(
          translate( data,
            replace(
              translate( data,'0123456789','#####'),
                '#'),rpad('#',length(data),'#')),'#'))
```

DATA

```
-----
SMITH   7369 RESEARCH
ALLEN   7499 SALES
WARD    7521 SALES
JONES   7566 RESEARCH
MARTIN  7654 SALES
BLAKE   7698 SALES
CLARK   7782 ACCOUNTING
SCOTT   7788 RESEARCH
KING    7839 ACCOUNTING
TURNER  7844 SALES
ADAMS   7876 RESEARCH
JAMES   7900 SALES
FORD    7902 RESEARCH
MILLER  7934 ACCOUNTING
```

需要指出的是，视图中的数据包含 3 部分，但只有一部分是数字。如果多个部分都是数字，那么就需要将它们拼接成一个数字，并作为对行进行排序的依据。

6.10 根据表中的行创建分隔列表

1. 问题

你想以分隔列表（分隔符可能是逗号）而不是常见的垂直列的方式返回表中的行。换言之，你要将下面的结果集：

DEPTNO	EMPS
10	CLARK
10	KING
10	MILLER
20	SMITH
20	ADAMS
20	FORD
20	SCOTT
20	JONES
30	ALLEN
30	BLAKE
30	MARTIN
30	JAMES
30	TURNER
30	WARD

转换成如下这样。

DEPTNO	EMPS
10	CLARK, KING, MILLER
20	SMITH, JONES, SCOTT, ADAMS, FORD
30	ALLEN, WARD, MARTIN, BLAKE, TURNER, JAMES

2. 解决方案

这个问题的解决方案随 DBMS 而异，关键是利用 DBMS 提

供的内置函数。弄清楚 DBMS 都提供了哪些函数，才能充分利用 DBMS 的功能，设计出创造性的解决方案，以解决通常使用 SQL 无法解决的问题。

当前，大多数 DBMS 提供了专门为串接字符串而设计的函数，比如 MySQL 的函数 **GROUP_CONCAT**（最早的函数之一）或 SQL Server 的函数 **STRING_ADD**（SQL Server 2017 中新增的）。这些函数的语法类似，让你能够轻松地解决上述问题。

DB2

使用 **LIST_AGG** 创建分隔列表。

```
1 select deptno,  
2         list_agg(ename ',') within GROUP(Order by 0) as emps  
3   from emp  
4  group by deptno
```

MySQL

使用内置函数 **GROUP_CONCAT** 创建分隔列表。

```
1 select deptno,  
2         group_concat(ename order by empno separator, ',') as emps  
3   from emp  
4  group by deptno
```

Oracle

使用内置函数 **SYS_CONNECT_BY_PATH** 创建分隔列表。

```
1 select deptno,  
2         ltrim(sys_connect_by_path(ename, ','), ',') emps  
3   from (  
4 select deptno,
```

```
5      ename,  
6      row_number() over  
7          (partition by deptno order by empno) rn,  
8      count(*) over  
9          (partition by deptno) cnt  
10 from emp  
11 )  
12 where level = cnt  
13 start with rn = 1  
14 connect by prior deptno = deptno and prior rn = rn-1
```

PostgreSQL 和 SQL Server

```
1 select deptno,  
2      string_agg(ename order by empno separator, ',') as emps  
3   from emp  
4  group by deptno
```

13. 讨论

能够在 SQL 中创建分隔列表很有用，因为经常需要这样做。在 SQL:2016 标准中，新增了用于执行这种任务的 **LIST_AGG**，但到目前为止，只有 DB2 实现了这个函数。所幸其他 DBMS 有类似的函数，且语法通常更简单。

MySQL

在 MySQL 中，函数 **GROUP_CONCAT** 会将传递给它的列（本例中是 **ENAME**）的值拼接起来。这是一个聚合函数，查询中需要包含 **GROUP BY**。

PostgreSQL 和 SQL Server

函数 **STRING_AGG** 的语法与函数 **GROUP_CONCAT** 类似，用于 **GROUP_CONCAT** 的查询无须修改就可用于 **STRING_AGG**。

Oracle

要弄明白这里的 Oracle 查询，首先需要将其分解。如果运行内嵌视图本身（第 4~10 行），那么生成的结果集将包含每位员工的如下信息：部门、姓名、在部门的排位（这是按 **EMPNO** 升序排列而派生出来的），以及所属部门的员工总数。

```
select deptno,
       ename,
       row_number() over
         (partition by deptno order by empno) rn,
       count(*) over (partition by deptno) cnt
  from emp
```

DEPTNO	ENAME	RN	CNT
-----	-----	---	---
10	CLARK	1	3
10	KING	2	3
10	MILLER	3	3
20	SMITH	1	5
20	JONES	2	5
20	SCOTT	3	5
20	ADAMS	4	5
20	FORD	5	5
30	ALLEN	1	6
30	WARD	2	6
30	MARTIN	3	6
30	BLAKE	4	6
30	TURNER	5	6
30	JAMES	6	6

生成排位（在查询中给它指定了别名 **RN**）旨在让你能够遍历树。由于函数 **ROW_NUMBER** 会生成一个枚举，此枚举从 1 开始，没有重复，也没有间隙，因此只需将当前值减 1 就可以引用前一行（或父行）。例如，3 前面的数字为 2（3-1）。在这里，2 是 3 的父行，第 12 行说明了这一点。另外，下面的代码行将 **RN** 为 1 的行视为相应 **DEPTNO** 的根，并在遇到新部门（**RN** 为 1 意味着遇到了新部门）时创建一个新的列表。


```
start with rn = 1  
connect by prior deptno = deptno
```

现在来看看函数 **ROW_NUMBER** 的 **ORDER BY** 部分。别忘了，姓名是根据 **EMPNO** 排序的，因此在创建的列表中，姓名也是按这样的顺序排列的。计算每个部门的员工数量，并使用它来确保查询返回的列表只包含相应部门所有员工的姓名。这样做是因为 **SYS_CONNECT_BY_PATH** 会以迭代的方式创建列表，而你不希望列表是不完整的。

在分层查询中，伪列 **LEVEL** 从 1 开始。（在没有使用 **CONNECT BY** 的查询中，**LEVEL** 从 0 开始，但在 10g 及更高的版本中，仅当在查询中使用了 **CONNECT BY** 时，**LEVEL** 才可用。）每评估完部门中的一位员工（层次结构中的一个层级），**LEVEL** 的值都会加 1。因此，当 **LEVEL** 为 **CNT** 时，说明已到达最后一个 **EMPNO**，这时列表是完整的。



函数 **SYS_CONNECT_BY_PATH** 会在列表开头添加你指定的分隔符（本例中是逗号），这可能是你想要的，也可能不是。在本实例的解决方案中，通过调用函数 **LTRIM** 将列表开头的逗号删除了。

6.11 将分隔数据转换为多值IN列表

1. 问题

你想将分隔数据传递给 **WHERE** 子句中的 **IN** 列表迭代器。请看下面的字符串。

```
7654,7698,7782,7788
```

你想在 **WHERE** 子句中使用这个字符串，但下面的 SQL 以失败告终，因为 **EMPNO** 是数值列。

```
select ename,sal,deptno
  from emp
 where empno in ( '7654,7698,7782,7788' )
```

这条 SQL 语句之所以执行失败，是因为 **EMPNO** 为数值列，而 **IN** 列表包含的是单个字符串值。你希望这个字符串被视为用逗号分隔的数值列表。

2. 解决方案

从表面上看，SQL 好像应该能够将分隔字符串视为值列表，但情况并非如此。遇到位于引号内的逗号时，SQL 不可能知道这意味着引号内的内容是一个多值列表。SQL 必须将引号内的所有内容视为单个实体——一个字符串值。你必须将前述字符串拆分成多个 **EMPNO**。本解决方案的关键是拆分字符串，但不是拆分为单个的字符，而是拆分为有效的 **EMPNO** 值。

DB2

通过遍历传递给 **IN** 列表的字符串，可以轻松地将其转换为多行数据。在这里，函数 **ROW_NUMBER**、**LOCATE** 和 **SUBSTR** 很有用。

```
1 select empno,ename,sal,deptno
2   from emp
3  where empno in (
4 select cast(substr(c,2,locate(',',c,2)-2) as integer) empno
5   from (
6 select substr(csv.emps,cast(iter.pos as integer)) as c
7   from (select '','||'7654,7698,7782,7788'||',' emp
8         from t1) csv,
9        (select id as pos
10         from t100 ) iter
11  where iter.pos <= length(csv.emps)
12        ) x
13  where length(c) > 1
14        and substr(c,1,1) = ','
15        )
```

MySQL

通过遍历传递给 **IN** 列表的字符串，可以轻松地将其转换为多行数据。

```
1 select empno, ename, sal, deptno
2   from emp
3  where empno in
4        (
5 select substring_index(
6        substring_index(list.vals,',',iter.pos),',',-1) empno
7   from (select id pos from t10) as iter,
8        (select '7654,7698,7782,7788' as vals
9         from t1) list
10  where iter.pos <=
11        (length(list.vals)-length(replace(list.vals,',','')))+1
12        )
```

Oracle

通过遍历传递给 **IN** 列表的字符串，可以轻松地将其转换为多行数据。在这里，函数 **ROWNUM**、**SUBSTR** 和 **INSTR** 很有用。

```
1 select empno,ename,sal,deptno
2   from emp
3  where empno in (
4      select to_number(
5          rtrim(
6              substr(emps,
7                  instr(emps,',',1,iter.pos)+1,
8                  instr(emps,',',1,iter.pos+1)
9                  instr(emps,',',1,iter.pos)),',')
10         from (select '||'7654,7698,7782,7788'||',' emp
11              from t1) csv,
12              (select rownum pos from emp) iter
13              where iter.pos <= ((length(csv.emps)-
14 length(replace(csv.emps,',')))/length(',')-1
14 )
```

PostgreSQL

通过遍历传递给 **IN** 列表的字符串，可以轻松地将其转换为多行数据。函数 **SPLIT_PART** 能够将字符串轻松地拆分为多个数字。

```
1 select ename,sal,deptno
2   from emp
3  where empno in (
4 select cast(empno as integer) as empno
5   from (
6 select split_part(list.vals,',',iter.pos) as empno
7   from (select id as pos from t10) iter,
8        (select '||'7654,7698,7782,7788'||',' as vals
9         from t1) list
10  where iter.pos <=
11        length(list.vals)-length(replace(list.vals,',',''))
12        ) z
13  where length(empno) > 0
```

SQL Server

通过遍历传递给 **IN** 列表的字符串，可以轻松地将其转换为多行数据。在这里，函数 **ROW_NUMBER**、**CHARINDEX** 和 **SUBSTRING** 很有用。

```
1 select empno,ename,sal,deptno
2   from emp
3  where empno in (select substring(c,2,charindex(',',c,2)-2) as
empno
4   from (
5 select substring(csv.emps,iter.pos,len(csv.emps)) as c
6   from (select ','+'7654,7698,7782,7788'+' ,' as emps
7         from t1) csv,
8        (select id as pos
9         from t100) iter
10  where iter.pos <= len(csv.emps)
11        ) x
12  where len(c) > 1
13    and substring(c,1,1) = ','
14        )
```

13. 讨论

上述解决方案的第一步是遍历字符串，这也是最重要的一步。完成这一步后，使用 **DBMS** 提供的函数将字符串拆分为数字值即可。

DB2 和 SQL Server

内嵌视图 **X**（第 6~11 行）会遍历字符串。该解决方案的思路是，通过遍历字符串，让每一行都比前一行少一个字符。

```
,7654,7698,7782,7788,  
7654,7698,7782,7788,  
654,7698,7782,7788,  
54,7698,7782,7788,  
4,7698,7782,7788,  
,7698,7782,7788,  
7698,7782,7788,  
698,7782,7788,  
98,7782,7788,  
8,7782,7788,  
,7782,7788,  
7782,7788,  
782,7788,  
82,7788,  
2,7788,  
,7788,  
7788,  
788,  
88,  
8,  
,
```

注意，通过将字符串放在两个逗号（分隔符）内，可以避免执行特殊的检查来确定字符串的起始位置或终止位置。

下一步是只保留要在 **IN** 列表中使用的值。需要保留的是那些以逗号开头的行，但仅包含逗号的最后一行除外。使用函数 **SUBSTR** 或 **SUBSTRING** 找出以逗号开头的行，然后保留这些行中从开头的逗号到下一个逗号之间的所有字符。完成这项工作后，将得到的字符串转换为数字，以便与数值列 **EMPNO**（第 4~14 行）进行比较。

```
EMPNO  
-----  
 7654  
 7698  
 7782  
 7788
```

最后一步是在子查询中使用转换得到的数字，以返回所需的行。

MySQL

首先，内嵌视图（第 5~9 行）会遍历字符串。第 10~11 行的表达式会确定字符串中有多少个数字值，这是通过计算逗号（分隔符）数再加 1 得到的。函数 **SUBSTRING_INDEX**（第 6 行）会返回第 n 个逗号前面（左边）的所有字符。

empno
7654
7654,7698
7654,7698,7782
7654,7698,7782,7788

然后，这些行被传递给另一个对函数 **SUBSTRING_INDEX**（第 5 行）的调用，但这次参数为 -1，这将保留最后一个逗号右边的所有值。

empno
7654
7698
7782
7788

最后，将结果插入子查询中。

Oracle

首先，遍历字符串。

```

select emps,pos
  from (select ', '||'7654,7698,7782,7788' ||', ' emps
        from t1) csv,
      (select rownum pos from emp) iter
 where iter.pos <=
        ((length(csv.emps)-length(replace(csv.emps,',')))/length(', '))-1

```

EMPS	POS
,7654,7698,7782,7788,	1
,7654,7698,7782,7788,	2
,7654,7698,7782,7788,	3
,7654,7698,7782,7788,	4

返回的行数决定了列表将包含多少个值。对这个查询来说，**POS** 的值至关重要，因为将字符串拆分为多个值时需要用到它们。然后，使用 **SUBSTR** 和 **INSTR** 对字符串进行分析，并使用 **POS** 来定位字符串中的第 n 个分隔符。通过将字符串放在两个逗号之间，无须执行特殊检查来确定字符串的起始位置或终止位置。传递给 **SUBSTR** 和 **INSTR**（第 7~9 行）的值位于第 n 和第 $n + 1$ 个分隔符之间。通过将下一个逗号在字符串中的位置和当前逗号在字符串中的位置相减，可以提取字符串中的每个值。

```

select substr(emps,
             instr(emps,', ',1,iter.pos)+1,
             instr(emps,', ',1,iter.pos+1)
             -instr(emps,', ',1,iter.pos)) emps
  from (select ', '||'7654,7698,7782,7788' ||', ' emps
        from t1) csv,
      (select rownum pos from emp) iter
 where iter.pos <=
        ((length(csv.emps)-length(replace(csv.emps,',')))/length(', '))-1

```

EMPS
7654,
7698,
7782,

7788,

最后，删除每个值末尾的逗号，将其转换为数字并插入子查询中。

PostgreSQL

首先，内嵌视图 **Z**（第 6~12 行）会遍历字符串。返回的行数取决于字符串中有多少个值。为了确定字符串中包含多少个值，需将原始字符串的长度减去删除分隔符后的长度（第 11 行）。字符串分析工作是由函数 **SPLIT_PART** 完成的，它会获取第 *n* 个分隔符前面的值。

```
select list.vals,
       split_part(list.vals,',',iter.pos) as empno,
       iter.pos
  from (select id as pos from t10) iter,
       (select ',|||'7654,7698,7782,7788'|||' as vals
        from t1) list
 where iter.pos <=
        length(list.vals)-length(replace(list.vals,',',''))
```

vals	empno	pos
,7654,7698,7782,7788,		1
,7654,7698,7782,7788,	7654	2
,7654,7698,7782,7788,	7698	3
,7654,7698,7782,7788,	7782	4
,7654,7698,7782,7788,	7788	5

最后，将值（**EMPNO**）转换为数字，并将其插入子查询中。

6.12 按字母顺序排列字符串中的字符

1. 问题

你想按字母顺序排列字符串中的字符。请看下面的结果集。

ENAME

ADAMS
ALLEN
BLAKE
CLARK
FORD
JAMES
JONES
KING
MARTIN
MILLER
SCOTT
SMITH
TURNER
WARD

你要将以上结果集变成如下这样。

OLD_NAME	NEW_NAME
-----	-----
ADAMS	AADMS
ALLEN	AELLN
BLAKE	ABEKL
CLARK	ACKLR
FORD	DFOR
JAMES	AEJMS
JONES	EJNOS
KING	GIKN
MARTIN	AIMNRT
MILLER	EILLMR
SCOTT	COSTT
SMITH	HIMST
TURNER	ENRRTU

12. 解决方案

随着标准化程度的提高，不同 DBMS 解决方案的相似程度和可移植性也越来越高，这个问题很好地证明了这一点。

DB2

要按字母顺序排列字符，必须遍历每个字符串，然后对其中的字符进行排序。

```
1 select ename,  
2       listagg(c,'') WITHIN GROUP( ORDER BY c)  
3   from (  
4       select a.ename,  
5             substr(a.ename,iter.pos,1  
6             ) as c  
7   from emp a,  
8       (select id as pos from t10) iter  
9       where iter.pos <= length(a.ename)  
10      order by 1,2  
11      ) x  
12      Group By c
```

MySQL

这里的关键是函数 **GROUP_CONCAT**，该函数不仅能拼接每个姓名中的字符，还能对它们进行排序。

```
1 select ename, group_concat(c order by c separator '')  
2   from (  
3 select ename, substr(a.ename,iter.pos,1) c  
4   from emp a,  
5       ( select id pos from t10 ) iter  
6  where iter.pos <= length(a.ename)
```

```
7         ) x
8   group by ename
```

Oracle

函数 `SYS_CONNECT_BY_PATH` 能以迭代的方式创建列表。

```
1 select old_name, new_name
2   from (
3   select old_name, replace(sys_connect_by_path(c, ' '), ' ')
new_name
4   from (
5   select e.ename old_name,
6          row_number() over(partition by e.ename
7                             order by substr(e.ename,iter.pos,1))
rn,
8          substr(e.ename,iter.pos,1) c
9   from emp e,
10        ( select rownum pos from emp ) iter
11  where iter.pos <= length(e.ename)
12  order by 1
13        ) x
14  start with rn = 1
15 connect by prior rn = rn-1 and prior old_name = old_name
16        )
17  where length(old_name) = length(new_name)
```

PostgreSQL

PostgreSQL 如今提供了 `STRING_AGG`，可用于对字符串中的字符进行排序。

```
        select ename, string_agg(c , ''
                                ORDER BY c)
from (
        select a.ename,
               substr(a.ename,iter.pos,1) as c
        from emp a,
             (select id as pos from t10) iter
        where iter.pos <= length(a.ename)
```

```
order by 1,2
) x
Group By c
```

SQL Server

如果你使用的是 SQL Server 2017 或更高的版本，那么可以采用使用 **STRING_AGG** 的 PostgreSQL 解决方案。否则，必须遍历每个字符串并对其中的字符进行排序。

```
1 select ename,
2         max(case when pos=1 then c else '' end)+
3         max(case when pos=2 then c else '' end)+
4         max(case when pos=3 then c else '' end)+
5         max(case when pos=4 then c else '' end)+
6         max(case when pos=5 then c else '' end)+
7         max(case when pos=6 then c else '' end)
8     from (
9         select e.ename,
10              substring(e.ename,iter.pos,1) as c,
11              row_number() over (
12                  partition by e.ename
13                  order by substring(e.ename,iter.pos,1)) as pos
14     from emp e,
15          (select row_number()over(order by ename) as pos
16           from emp) iter
17    where iter.pos <= len(e.ename)
18          ) x
19    group by ename
```

13. 讨论

SQL Server

内嵌视图 **x** 会将每个姓名中的每个字符都作为一行返回。函数 **SUBSTR** 或 **SUBSTRING** 会提取每个姓名中的每个字符，而函数 **ROW_NUMBER** 会按字母顺序对字符进行排位。

ENAME	C	POS
-----	-	---
ADAMS	A	1
ADAMS	A	2
ADAMS	D	3
ADAMS	M	4
ADAMS	S	5
...		

要将字符串中的每个字符都作为一行返回，必须遍历字符串。这是使用内嵌视图 **ITER** 完成的。

将每个姓名中的字符按字母顺序排位后，最后一步是按排位顺序将这些字符组合成一个字符串。第 2~7 行的 **CASE** 语句会检查每个排位，如果在特定排位找到字符，就将其与下一个检查（下一条 **CASE** 语句）的结果拼接起来。由于同时使用了聚合函数 **MAX**，每个 **POS** 位置只返回一个字符，因此每个姓名只返回一行数据。**CASE** 语句检查的排位最大为 6，这是 **EMP** 表中姓名最多可能包含的字符数。

MySQL

内嵌视图 **X**（第 3~6 行）会将每个姓名中的每个字符都作为一行返回。函数 **SUBSTR** 会提取每个姓名中的每个字符。

ENAME	C
-----	-
ADAMS	A
ADAMS	A
ADAMS	D
ADAMS	M
ADAMS	S
...	

内嵌视图 **ITER** 用于遍历字符串，其他工作是由函数 **GROUP_CONCAT** 完成的。通过指定排序方式，这个函数不仅能用来拼接字符，还能按字母顺序对它们进行排序。

Oracle

实际工作是由内嵌视图 **X**（第 5~11 行）完成的，它会提取每个姓名中的字符，并按字母顺序排列它们。这是通过遍历字符串并对其中的字符进行排序实现的。查询的其他部分只是重组姓名。

要查看将姓名拆解后的结果，首先，可以单独执行内嵌视图 **X**。

OLD_NAME	RN	C
-----	----	-
ADAMS	1	A
ADAMS	2	A
ADAMS	3	D
ADAMS	4	M
ADAMS	5	S
...		

然后，将按字母顺序排位的字符重组为姓名。这是使用函数 **SYS_CONNECT_BY_PATH** 实现的，它将每个字符都附加到它前面的字符后面。

OLD_NAME	NEW_NAME
-----	-----
ADAMS	A
ADAMS	AA
ADAMS	AAD
ADAMS	AADM
ADAMS	AADMS
...	

最后，只保留与用来创建它的姓名等长的字符串。

PostgreSQL

为了提高可读性，该解决方案使用了视图 **X** 来遍历字符串。

在这个视图的定义中，函数 **SUBSTR** 会提取每个姓名中的每个字符，让这个视图返回如下结果。

ENAME	C
-----	-
ADAMS	A
ADAMS	A
ADAMS	D
ADAMS	M
ADAMS	S
...	

这个视图还按 **ENAME** 和提取的字符对结果进行了排序。

6.13 识别可视为数字的字符串

1. 问题

你有一个存储字符数据的列，不过其所对应的行既包含数字数据又包含字符数据。请看下面的视图 **V**。

```
create view V as
select replace(mixed, ' ', '') as mixed
  from (
select substr(ename,1,2)||
       cast(deptno as char(4))||
       substr(ename,3,2) as mixed
  from emp
 where deptno = 10
 union all
select cast(empno as char(4)) as mixed
  from emp
 where deptno = 20
 union all
select ename as mixed
  from emp
 where deptno = 30
 ) x
select * from v
```

MIXED

CL10AR

KI10NG

MI10LL

7369

7566

7788

7876

7902

ALLEN

WARD

MARTIN

BLAKE

TURNER

你想返回只有数字或至少包含一个数字的行。对于同时包含数字和字符的行，你想将字符删除，只返回数字。对于前面的示例数据，你希望结果集如下所示。

MIXED

10
10
10
7369
7566
7788
7876
7902

12. 解决方案

在操作字符串和字符方面，函数 **REPLACE** 和 **TRANSLATE** 很有用。关键是将所有的数字都转换为特定的字符，以便通过这个字符轻松地隔离和识别数字。

DB2

使用函数 **TRANSLATE**、**REPLACE** 和 **POSSTR** 来隔离每行的数字字符。视图 **V** 中的 **CAST** 调用必不可少，否则类型转换错误将导致无法创建这个视图。需要使用函数 **REPLACE** 将强制转换为定长 **CHAR** 生成的多余空白删除。

```
1 select mixed old,  
2         cast(  
3           case  
4             when  
5               replace(
```

```

6          translate(mixed,'9999999999','0123456789'),'9','') =
7      then
8          mixed
9      else replace(
10         translate(mixed,
11             repeat('#',length(mixed)),
12             replace(
13                 translate(mixed,'9999999999','0123456789'),'9',''),
14                 '#','')
15         end as integer ) mixed
16 from V
17 where posstr(translate(mixed,'9999999999','0123456789'),'9') >
18 0

```

MySQL

在 MySQL 中，语法稍有不同，需要像下面这样定义视图 V。

```

create view V as
select concat(
    substr(ename,1,2),
    replace(cast(deptno as char(4)),' ',''),
    substr(ename,3,2)
) as mixed
from emp
where deptno = 10
union all
select replace(cast(empno as char(4)), ' ', '')
from emp where deptno = 20
union all
select ename from emp where deptno = 30

```

由于 MySQL 不支持函数 **TRANSLATE**，因此必须遍历每一行数据并逐字符对其进行评估。

```

1 select cast(group_concat(c order by pos separator '') as
2     unsigned)
3     as MIXED1

```

```

3  from (
4 select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
5  from V,
6       ( select id pos from t10 ) iter
7  where iter.pos <= length(v.mixed)
8        and ascii(substr(v.mixed,iter.pos,1)) between 48 and 57
9        ) y
10 group by mixed
11 order by 1

```

Oracle

使用函数 **TRANSLATE**、**REPLACE** 和 **INSTR** 来隔离每行的数字字符。在视图 **V** 中，并非必须调用 **CAST**。使用函数 **REPLACE** 将强制转换为定长 **CHAR** 生成的多余空白删除。如果决定保留视图定义中的显式类型转换调用，建议转换为 **VARCHAR2**。

```

1 select to_number (
2       case
3       when
4
5 replace(translate(mixed,'0123456789','9999999999'),'9')
6       is not null
7       then
8           replace(
9           translate(mixed,
10          replace(
11          translate(mixed,'0123456789','9999999999'),'9'),
12          rpad('#',length(mixed),'#')),'#')
13       else
14           mixed
15       end
16       ) mixed
17  from V
18  where instr(translate(mixed,'0123456789','9999999999'),'9') >
19         0

```

PostgreSQL

使用函数 **TRANSLATE**、**REPLACE** 和 **STRPOS** 来隔离每行的数字字符。在视图 **V** 中，并非必须调用 **CAST**。使用函数 **REPLACE** 将强制转换为定长 **CHAR** 生成的多余空白删除。如果决定保留视图定义中的显式类型转换调用，建议转换为 **VARCHAR**。

```
1 select cast(  
2     case  
3     when  
4  
5 replace(translate(mixed,'0123456789','9999999999'),'9','')  
6     is not null  
7     then  
8         replace(  
9         translate(mixed,  
10            replace(  
11                translate(mixed,'0123456789','9999999999'),'9',''),  
12                rpad('#',length(mixed),'#')),'#','')  
13        else  
14            mixed  
15    end as integer ) as mixed  
16 from V  
17 where strpos(translate(mixed,'0123456789','9999999999'),'9') >  
18 0
```

SQL Server

结合使用内置函数 **ISNUMERIC** 和通配查找，可以轻松地区别包含数字的字符串，但由于 SQL Server 不支持函数 **TRANSLATE**，因此从字符串中提取数字字符的效率不是很高。

13. 讨论

在这里，函数 **TRANSLATE** 很有用，它能够轻松地隔离并识别数字和字符。诀窍是将所有数字都转换为特定的字符，这

样就可以搜索这个特定的字符，而无须搜索每个数字字符。

DB2、Oracle 和 PostgreSQL

这些 DBMS 的语法略有不同，但方法是一样的。下面的讨论将以 PostgreSQL 解决方案为例。

实际工作是由函数 **TRANSLATE** 和 **REPLACE** 完成的。为了获得最终的结果集，需要多个函数调用，下面在一个查询中列出所有这些调用。

```
select mixed as orig,
translate(mixed,'0123456789','9999999999') as mixed1,
replace(translate(mixed,'0123456789','9999999999'),'9','') as
mixed2,
  translate(mixed,
  replace(
  translate(mixed,'0123456789','9999999999'),'9',''),
    rpad('#',length(mixed),'#')) as mixed3,
  replace(
  translate(mixed,
  replace(
  translate(mixed,'0123456789','9999999999'),'9',''),
    rpad('#',length(mixed),'#')),'#','') as mixed4
  from V
where strpos(translate(mixed,'0123456789','9999999999'),'9') > 0
```

ORIG	MIXED1	MIXED2	MIXED3	MIXED4	MIXED5
CL10AR	CL99AR	CLAR	##10##	10	10
KI10NG	KI99NG	KING	##10##	10	10
MI10LL	MI99LL	MILL	##10##	10	10
7369	9999		7369	7369	7369
7566	9999		7566	7566	7566
7788	9999		7788	7788	7788
7876	9999		7876	7876	7876
7902	9999		7902	7902	7902

注意，不包含任何数字字符的行都被删除了。如果查看上述结果集的每一列，就能清楚地知道这是如何完成的。**ORIG**

列显示了被保留的行的值，我们将根据这些行来生成最终的结果集。为了提取数字，第一步是使用函数 **TRANSLATE** 将每个数字字符都转换为 9（这里选择 9 并无特殊考虑，你可以使用任何数字），结果如 **MIXED1** 列所示。将所有的数字字符都转换为 9 后，就可以将它们作为一个整体进行处理了。下一步是使用函数 **REPLACE** 删除所有的数字字符。由于现在所有的数字字符都变成了 9，因此 **REPLACE** 只需查找所有的 9 并将它们删除，结果如 **MIXED2** 列所示。下一步的结果如 **MIXED3** 所示，它使用了 **MIXED2** 列的值。将这些值与 **ORIG** 列的值进行比较。对于 **ORIG** 中的每个字符，如果它出现在了 **MIXED2** 中，**TRANSLATE** 就将其转换为 #。 **MIXED3** 列中的值表明，这次找出了字母（而不是数字），并将它们转换成了字符 #。将所有非数字字符都转换为 # 后，就可以将它们作为一个整体进行处理了。接下来，使用 **REPLACE** 找到并删除每一行中所有的 # 字符，只留下数字字符，结果如 **MIXED4** 列所示。最后一步是将数字字符强制转换为数字。知道详细的步骤后，来看看 **WHERE** 子句是如何工作的。将 **MIXED1** 列所示的结果传递给 **STRPOS**，如果找到 9，那么 **STRPOS** 将返回第一个 9 在字符串中的位置，因此返回的值必然大于 0。当 **STRPOS** 返回的值大于 0 时，意味着相应的行至少包含一个数字字符，因此应该将该行保留。

MySQL

首先，遍历字符串，检查每个字符是否是数字字符。

```
select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
  from V,
      ( select id pos from t10 ) iter
 where iter.pos <= length(v.mixed)
 order by 1,2
```

```
+-----+-----+-----+
| mixed | pos | c   |
+-----+-----+-----+
```

7369	1	7
7369	2	3
7369	3	6
7369	4	9
...		
ALLEN	1	A
ALLEN	2	L
ALLEN	3	L
ALLEN	4	E
ALLEN	5	N
...		
CL10AR	1	C
CL10AR	2	L
CL10AR	3	1
CL10AR	4	0
CL10AR	5	A
CL10AR	6	R
+-----+-----+-----+		

然后，检查每个字符串中的每个字符后，只保留 C 列为数字的行。

```

select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
  from V,
       ( select id pos from t10 ) iter
 where iter.pos <= length(v.mixed)
       and ascii(substr(v.mixed,iter.pos,1)) between 48 and 57
 order by 1,2

```

+-----+-----+-----+		
mixed	pos	c
+-----+-----+-----+		
7369	1	7
7369	2	3
7369	3	6
7369	4	9
...		
CL10AR	3	1
CL10AR	4	0
...		
+-----+-----+-----+		

至此，所有行的 C 列都是数字了。接下来，使用 **GROUP_CONCAT** 将数字拼接起来，得到 **MIXED** 中完整的数字部分，再将结果转换为数字。

```
select cast(group_concat(c order by pos separator '') as unsigned)
        as MIXED1
  from (
select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
  from V,
        ( select id pos from t10 ) iter
 where iter.pos <= length(v.mixed)
       and ascii(substr(x.mixed,iter.pos,1)) between 48 and 57
        ) y
 group by mixed
 order by 1
```

```
+-----+
| MIXED1 |
+-----+
|    10  |
|    10  |
|    10  |
|   7369 |
|   7566 |
|   7788 |
|   7876 |
|   7902 |
+-----+
```

最后，需要指出的是，这会把字符串中所有的数字字符都拼接起来，得到一个数字值。如果输入值为 **99Gennick87**，那么返回的结果将为 **9987**。请务必牢记这一点，尤其是处理序列化数据的时候。

6.14 提取第*n*个子串

1. 问题

你想从字符串中提取特定的子串。请看下面的视图 **V**，它是这个问题的数据源。

```
create view V as
select 'mo,larry,curly' as name
  from t1
 union all
select 'tina,gina,jaunita,regina,leena' as name
  from t1
```

这个视图的输出如下。

```
select * from v

NAME
-----
mo,larry,curly
tina,gina,jaunita,regina,leena
```

你要从每行中提取第二个姓名，因此最终的结果集如下所示。

```
  SUB
-----
larry
gina
```

2. 解决方案

要解决这个问题，关键是将每个姓名作为一行返回，并保留

姓名在列表中的位置。具体如何完成这些任务取决于你使用的是哪个 DBMS。

DB2

遍历视图 **V** 返回的 **NAME** 后，使用函数 **ROW_NUMBER** 留下每个字符串中的第二个姓名。

```
1 select substr(c,2,locate(',',c,2)-2)
2   from (
3 select pos, name, substr(name, pos) c,
4         row_number() over( partition by name
5                             order by length(substr(name,pos)) desc)
6   rn
7   from (
8 select ', ' || csv.name || ', ' as name,
9        cast(iter.pos as integer) as pos
10  from V csv,
11       (select row_number() over() pos from t100 ) iter
12 where iter.pos <= length(csv.name)+2
13       ) x
14 where length(substr(name,pos)) > 1
15       and substr(substr(name,pos),1,1) = ', '
16       ) y
17 where rn = 2
```

MySQL

遍历视图 **V** 返回的 **NAME** 后，根据逗号的位置返回每个字符串中的第二个姓名。

```
1 select name
2   from (
3 select iter.pos,
4        substring_index(
5          substring_index(src.name,',',iter.pos),',',-1) name
6   from V src,
7        (select id pos from t10) iter,
8  where iter.pos <=
9        length(src.name)-length(replace(src.name,',',''))
```

```
10      ) x
11  where pos = 2
```

Oracle

遍历视图 **V** 返回的 **NAME** 后，使用 **SUBSTR** 和 **INSTR** 检索每个列表中的第二个姓名。

```
1 select sub
2   from (
3 select iter.pos,
4         src.name,
5         substr( src.name,
6               instr( src.name,',',1,iter.pos )+1,
7               instr( src.name,',',1,iter.pos+1 ) -
8               instr( src.name,',',1,iter.pos )-1) sub
9   from (select ',||name||',' as name from V) src,
10        (select rownum pos from emp) iter
11  where iter.pos < length(src.name)-
12        length(replace(src.name',''))
13  where pos = 2
```

PostgreSQL

使用函数 **SPLIT_PART** 将每个姓名作为一行返回。

```
1 select name
2   from (
3 select iter.pos, split_part(src.name,',',iter.pos) as name
4   from (select id as pos from t10) iter,
5        (select cast(name as text) as name from v) src
7  where iter.pos <=
8         length(src.name)-length(replace(src.name',''))+1
9   ) x
10  where pos = 2
```

SQL Server

SQL Server 函数 **STRING_SPLIT** 可以完成这项任务，但只能接受单个单元格（single cell），因此我们在一个 CTE 中使用 **STRING_AGG** 按 **STRING_SPLIT** 要求的方式提供数据。

```
1 with agg_tab(name)
2   as
3     (select STRING_AGG(name,',') from V)
4 select value from
5     STRING_SPLIT(
6     (select name from agg_tab),',')
```

13. 讨论

DB2

首先，在内嵌视图 **X** 中遍历字符串。

```
select ', '||csv.name|| ', ' as name,
       iter.pos
  from v csv,
       (select row_number() over() pos from t100 ) iter
 where iter.pos <= length(csv.name)+2

EMPS POS
-----
,tina,gina,jaunita,regina,leena,    1
,tina,gina,jaunita,regina,leena,    2
,tina,gina,jaunita,regina,leena,    3
...
```

然后，遍历每个字符串中的每个字符。

```
select pos, name, substr(name, pos) c,
       row_number() over(partition by name
                          order by length(substr(name, pos)) desc)
  rn
 from (
```

```

select ', ' || csv.name || ', ' as name,
       cast(iter.pos as integer) as pos
  from v csv,
       (select row_number() over() pos from t100 ) iter
 where iter.pos <= length(csv.name)+2
       ) x
 where length(substr(name,pos)) > 1

```

POS	EMPS	C	RN
1	,mo,larry,curly,	,mo,larry,curly,	1
2	,mo,larry,curly,	mo,larry,curly,	2
3	,mo,larry,curly,	o,larry,curly,	3
4	,mo,larry,curly,	,larry,curly,	4
...			

提取字符串的不同部分后，只需确定要保留哪些行。你感兴趣的是那些以逗号开头的行，因此可以将其他行丢弃。

```

select pos, name, substr(name,pos) c,
       row_number() over(partition by name
                          order by length(substr(name, pos)) desc) rn
  from (
select ', ' || csv.name || ', ' as name,
       cast(iter.pos as integer) as pos
  from v csv,
       (select row_number() over() pos from t100 ) iter
 where iter.pos <= length(csv.name)+2
       ) x
 where length(substr(name,pos)) > 1
       and substr(substr(name,pos),1,1) = ','

```

POS	EMPS	C
1	,mo,larry,curly,	,mo,larry,curly,
4	,mo,larry,curly,	,larry,curly,
10	,mo,larry,curly,	,curly,
1	,tina,gina,jaunita,regina,leena,	
6	,tina,gina,jaunita,regina,leena,	

	,gina,jaunita,regina,leena,	2
11	,tina,gina,jaunita,regina,leena,	,jaunita,regina,leena,
3		
19	,tina,gina,jaunita,regina,leena,	,regina,leena,
4		
26	,tina,gina,jaunita,regina,leena,	,leena,
5		

这一步很重要，为你获取第 n 个子串做好了准备。注意，这个查询删除了很多行，因为其 **WHERE** 子句包含如下条件。

```
substr(substr(name,pos),1,1) = ','
```

你可能注意到了，`,mo,larry,curly,` 的排位原来为 4，但现在为 2。别忘了，**WHERE** 子句先于 **SELECT** 执行，因此先保留以逗号开头的行，然后再调用 **ROW_NUMBER** 进行排位。至此，很容易看出这样一点：要获取第 n 个字符串，需要检索 **RN** 为 n 的行。最后，只留下你感兴趣的行（这里是 **RN** 为 2 的行），并使用 **SUBSTR** 从这些行中提取姓名。提取的是行中的第一个姓名：从 `,larry,curly,` 中提取 `larry`；从 `,gina,jaunita,regina,leena,` 中提取 `gina`。

MySQL

使用内嵌视图 **X** 遍历每个字符串。可以通过计算分隔符数量来确定每个字符串包含多少个值。

```
select iter.pos, src.name
  from (select id pos from t10) iter,
       V src
 where iter.pos <=
       length(src.name)-length(replace(src.name,',',''))
```

pos	name
1	mo,larry,curly

2	mo,larry,curly
1	tina,gina,jaunita,regina,leena
2	tina,gina,jaunita,regina,leena
3	tina,gina,jaunita,regina,leena
4	tina,gina,jaunita,regina,leena

在这里，对于每个字符串，返回的行数都比它包含的值的个数少 1，因为这就够了。函数 **SUBSTRING_INDEX** 负责提取所需的值。

<pre> select iter.pos,src.name name1, substring_index(src.name,',',iter.pos) name2, substring_index(substring_index(src.name,',',iter.pos),',',-1) name3 from (select id pos from t10) iter, V src where iter.pos <= length(src.name)-length(replace(src.name,',','')) </pre>		
<pre> +-----+-----+-----+ -+-----+ pos name1 name2 name3 +-----+-----+-----+ -+-----+ 1 mo,larry,curly mo mo 2 mo,larry,curly mo,larry larry 1 tina,gina,jaunita,regina,leena tina tina 2 tina,gina,jaunita,regina,leena tina,gina gina 3 tina,gina,jaunita,regina,leena tina,gina,jaunita jaunita 4 tina,gina,jaunita,regina,leena tina,gina,jaunita,regina regina +-----+-----+-----+ -+-----+ </pre>		

这里显示了 **NAME1** 列、**NAME 2** 列和 **NAME3** 列的值，旨在让你明白嵌套 **SUBSTRING_INDEX** 调用的工作原理。内层的调用返回了第 *n* 个逗号左边的所有字符，而外层的调用返回了（从字符串末尾往左数的）第一个逗号右边的所有字符。最后，只保留 **POS** 为 2 的行，并返回这些行的 **NAME3** 列的值。

SQL Server

这里的“主力”是 **STRING_SPLIT**，但需要以正确的方式将数据提供给它。CTE 只是将两行的 **V.name** 列转换为单个值，这是表值函数 **STRING_SPLIT** 要求的。

Oracle

Oracle 内嵌视图会遍历每个字符串。对于每个字符串，返回它的次数取决于其中有多少个值。该解决方案通过计算字符串中包含的分隔符个数来确定它包含多少个值。由于每个字符串都位于两个逗号内，因此值的个数为逗号个数减 1。然后，将字符串合并（**UNION**）并连接到一张表（这张表的基数不小于最大字符串包含的值的个数）。函数 **SUBSTR** 和 **INSTR** 使用 **POS** 的值来分析每个字符串。

```
select iter.pos, src.name,
       substr( src.name,
              instr( src.name, ',', 1, iter.pos )+1,
              instr( src.name, ',', 1, iter.pos+1 )
              - instr( src.name, ',', 1, iter.pos )-1) sub
  from (select ',|||name|||,' as name from v) src,
       (select rownum pos from emp) iter
 where iter.pos < length(src.name)-length(replace(src.name, ','))
```

POS	NAME	SUB
1	,mo,larry,curly,	mo
1	, tina,gina,jaunita,regina,leena,	tina
2	,mo,larry,curly,	larry

```

2 , tina,gina,jaunita,regina,leena, gina
3 ,mo,larry,curly, curly
3 , tina,gina,jaunita,regina,leena, jaunita
4 , tina,gina,jaunita,regina,leena, regina
5 , tina,gina,jaunita,regina,leena, leena

```

在 **SUBSTR** 中，第一个 **INSTR** 调用确定要提取的子串的起始位置，其他两个 **INSTR** 调用确定第 n 个逗号的位置（与起始位置相同）以及第 $n + 1$ 个逗号的位置。将这两个值相减，就得到了要提取的子串的长度。由于每个值都被放到了不同的行中，因此只需指定 **WHERE POS = n** 就能留下第 n 个字符串（在这里，**POS** = 2，因此要提取的是列表中的第二个子串）。

PostgreSQL

内嵌视图 **X** 会遍历每个字符串。对于每个字符串，返回的行数取决于它包含多少个值。为了确定每个字符串包含多少个值，计算它包含多少个分隔符再加 1。函数 **SPLIT_PART** 使用 **POS** 的值来找到第 n 个分隔符，并将字符串拆分为值。

```

select iter.pos, src.name as name1,
       split_part(src.name,',',iter.pos) as name2
  from (select id as pos from t10) iter,
       (select cast(name as text) as name from v) src
 where iter.pos <=
       length(src.name)-length(replace(src.name,',',''))+1

```

pos	name1	name2
1	mo,larry,curly	mo
2	mo,larry,curly	larry
3	mo,larry,curly	curly
1	tina,gina,jaunita,regina,leena	tina
2	tina,gina,jaunita,regina,leena	gina
3	tina,gina,jaunita,regina,leena	jaunita
4	tina,gina,jaunita,regina,leena	regina
5	tina,gina,jaunita,regina,leena	leena

这里显示了 **NAME1** 列和 **NAME2** 列，旨在让你明白 **SPLIT_PART** 是如何使用 **POS** 来拆分字符串的。拆分字符串后，最后一步是只留下 **POS = n** 的行（其中 n 是你感兴趣的子串的序号，这里为 2）。

6.15 拆分IP地址

1. 问题

你想将 IP 地址的不同部分拆分为列。请看下面的 IP 地址。

111.22.3.4

你希望查询的结果如下所示。

A	B	C	D
-----	-----	-----	---
111	22	3	4

2. 解决方案

具体的解决方案随 DBMS 提供的内置函数而异。不管你使用的是哪种 DBMS，解决方案的关键都在于定位句点及其两边的数字。

DB2

使用递归子句 **WITH** 模拟迭代 IP 地址的过程，同时使用 **SUBSTR** 来轻松地分析它。在 IP 地址前面添加一个句点，这样每组数字前面就都有了一个句点，因此我们能够以相同的方式处理它们。

```
1 with x (pos,ip) as (  
2   values (1, '.92.111.0.222')  
3   union all  
4   select pos+1,ip from x where pos+1 <= 20  
5 )
```

```

6 select max(case when rn=1 then e end) a,
7         max(case when rn=2 then e end) b,
8         max(case when rn=3 then e end) c,
9         max(case when rn=4 then e end) d
10    from (
11 select pos,c,d,
12        case when posstr(d,'.') > 0 then
substr(d,1,posstr(d,'.')-1)
13            else d
14        end as e,
15        row_number() over( order by pos desc) rn
16    from (
17 select pos, ip,right(ip,pos) as c, substr(right(ip,pos),2) as d
18    from x
19   where pos <= length(ip)
20   and substr(right(ip,pos),1,1) = '.'
21        ) x
22        ) y

```

MySQL

函数 **SUBSTR_INDEX** 让你能够轻松地分析 IP 地址。

```

1 select substring_index(substring_index(y.ip,'.',1),'.',-1) a,
2        substring_index(substring_index(y.ip,'.',2),'.',-1) b,
3        substring_index(substring_index(y.ip,'.',3),'.',-1) c,
4        substring_index(substring_index(y.ip,'.',4),'.',-1) d
5    from (select '92.111.0.2' as ip from t1) y

```

Oracle

使用内置函数 **SUBSTR** 和 **INSTR** 来分析 IP 地址并在其中导航。

```

1 select ip,
2        substr(ip, 1, instr(ip,'.')-1 ) a,
3        substr(ip, instr(ip,'.')+1,
4               instr(ip,'.',1,2)-instr(ip,'.')-1 ) b,
5        substr(ip, instr(ip,'.',1,2)+1,
6               instr(ip,'.',1,3)-instr(ip,'.',1,2)-1 ) c,

```

```
7      substr(ip, instr(ip, '.',1,3)+1 ) d
8  from (select '92.111.0.2' as ip from t1)
```

PostgreSQL

使用内置函数 **SPLIT_PART** 来分析 IP 地址。

```
1 select split_part(y.ip, '.',1) as a,
2        split_part(y.ip, '.',2) as b,
3        split_part(y.ip, '.',3) as c,
4        split_part(y.ip, '.',4) as d
5  from (select cast('92.111.0.2' as text) as ip from t1) as y
```

SQL Server

使用递归子句 **WITH** 模拟迭代 IP 地址的过程，同时使用 **SUBSTR** 来轻松地分析它。在 IP 地址前面添加一个句点，这样每组数字前面就都有了一个句点，因此我们能够以相同的方式处理它们。

```
1  with x (pos,ip) as (
2      select 1 as pos, '.92.111.0.222' as ip from t1
3      union all
4      select pos+1, ip from x where pos+1 <= 20
5  )
6  select max(case when rn=1 then e end) a,
7         max(case when rn=2 then e end) b,
8         max(case when rn=3 then e end) c,
9         max(case when rn=4 then e end) d
10  from (
11  select pos,c,d,
12         case when charindex('.',d) > 0
13             then substring(d,1,charindex('.',d)-1)
14             else d
15         end as e,
16         row_number() over(order by pos desc) rn
17  from (
18  select pos, ip, right(ip,pos) as c,
19         substring(right(ip,pos),2,len(ip)) as d
20  from x
```

```
21 where pos <= len(ip)
22     and substring(right(ip,pos),1,1) = '.'
23     ) x
24     ) y
```

13. 讨论

使用 DBMS 提供的内置函数，可以轻松地遍历字符串的不同部分。关键是能够定位地址中的句点，这样就能提取句点之间的数字了。

6.17 节将介绍如何在大部分 RDBMS 中使用正则表达式。正则表达式也非常适合用来分析 IP 地址。

6.16 根据发音比较字符串

1. 问题

在匹配单词方面，有两种极端情况，一是匹配拼写正确和拼写错误的单词，二是匹配拼写方式不同（比如英式拼写和美式拼写）的单词。除了这两种极端情况，很多时候还需要匹配由不同字符串表示的单词。所幸 SQL 提供了一种表示单词发音的方式，让你能够查找拼写不同但发音相同的字符串。

例如，你有一个作者姓名清单，其中一些作者是古代的，拼写不像现在这样固定，还有一些存在拼写和输入错误，如下所示。

```
a_name
----
1 Johnson
2 Jonson
3 Jonsen
4 Jensen
5 Johnsen
6 Shakespeare
7 Shakspear
8 Shaekspir
9 Shakespar
```

在这个清单中，你要找出哪些姓名的发音相同。对于这个问题，解决方案有多种，下面是其中之一（等你阅读完本节后，最后一列的含义将更清晰）。

a_name1	a_name2	soundex_name
----	----	----
Jensen	Johnson	J525
Jensen	Jonson	J525
Jensen	Jonsen	J525
Jensen	Johnsen	J525

Johnsen	Johnson	J525
Johnsen	Jonson	J525
Johnsen	Jonsen	J525
Johnsen	Jensen	J525
...		
Jonson	Jensen	J525
Jonson	Johnsen	J525
Shaekspir	Shakspear	S216
Shakespar	Shakespeare	S221
Shakespeare	Shakespar	S221
Shakspear	Shaekspir	S216

12. 解决方案

使用函数 **SOUNDEX** 将字符串转换为英语发音。使用简单的自连接，可以对同一列中的不同值进行比较。

```

1  select an1.a_name as name1, an2.a_name as name2,
2  SOUNDEX(an1.a_name) as Soundex_Name
3  from author_names an1
4  join author_names an2
5  on (SOUNDEX(an1.a_name)=SOUNDEX(an2.a_name)
6  and an1.a_name not like an2.a_name)

```

13. 讨论

在数据库和计算机面世前，**SOUNDEX** 背后的理念就已存在。这种理念最初由美国人口普查局提出，旨在解决人名和地名存在不同拼写的问题。很多算法有与 **SOUNDEX** 相同的用途，当然还有非英语版本，但这里只介绍 **SOUNDEX**，因为大多数 RDBMS 提供了它。

SOUNDEX 会保留第一个字母，并将其他字母替换为数字。发

音相似的字母将被替换为相同的数字，例如，M 和 N 都被替换为 5。

在前面的示例中，`SOUNDEX_NAME` 列显示了 `SOUNDEX` 的输出。这里显示这些旨在说明发生的情况，解决方案不一定会这样做。有些 RDMS 函数甚至会将 `SOUNDEX` 结果隐藏起来，例如，SQL Server 函数 `DIFFERENCE` 会使用 `SOUNDEX` 对两个字符串进行比较，并返回表示相似程度的数字 0~4（例如，4 表示 `SOUNDEX` 输出完全相同，即两个字符串的 4 字符 `SOUNDEX` 版完全相同）。

在有些情况下，`SOUNDEX` 足以满足需求，有时则满足不了。然而，只要在网上稍微搜索一下 [比如使用 `Data Matching` (Christen, 2012) 进行搜索]，就可以找到其他算法。这些算法通常实现起来更容易，但并非总是如此。实现这些算法时，你既可以根据需要使用用户定义的函数，也可以使用其他编程语言。

6.17 查找与模式不匹配的文本

1. 问题

有一个文本字段，其中包含一些结构化的文本值（比如电话号码），而你想找出那些未正确结构化的值。假设有如下数据：

<pre>select emp_id, text from employee_comment</pre>	
EMP_ID	TEXT
-----	-----

7369	126 Varnum, Edmore MI 48829, 989 313-5351
7499	1105 McConnell Court Cedar Lake MI 48812 Home: 989-387-4321 Cell: (237) 438-3333

你要列出其中电话号码格式不正确的行。例如，你要列出下面这一行，因为其电话号码使用了不同的分隔符。

7369	126 Varnum, Edmore MI 48829, 989 313-5351
------	---

仅当电话号码使用相同的分隔符时，你才认为它们是正确的。

2. 解决方案

这个问题的解决方案由多个部分组成。

- a. 找出一种方式，指出看起来像电话号码的内容是什么样

的。

- b. 将格式正确的电话号码都删除。
- c. 看看是否还有看起来像电话号码的内容，如果有，就说明它们的格式不正确。

```
select emp_id, text
from employee_comment
where regexp_like(text, '[0-9]{3}[-. ] [0-9]{3}[-. ] [0-9]{4}')
and regexp_like(
    regexp_replace(text,
        '[0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}', '***'),
    '[0-9]{3}[-. ] [0-9]{3}[-. ] [0-9]{4}')
```

EMP_ID	TEXT
7369	126 Varnum, Edmore MI 48829, 989 313-5351
7844	989-387.5359
9999	906-387-1698, 313-535.8886

这些行都包含看起来像电话号码但格式不正确的内容。

13. 讨论

本解决方案的关键在于找出“看起来像电话号码”的内容。鉴于电话号码存储在一个说明字段中，因此该字段中的任何文本都可能被解读为无效的电话号码。需要想办法缩小范围，让你关注的内容更合理。例如，你不希望在输出中看到如下内容。

EMP_ID	TEXT
7900	Cares for 100-year-old aunt during the day. Schedule only for evening and night shifts.

这一行显然没有包含任何电话号码，更谈不上无效的电话号码了。我们都能看出这一点，但问题是如何让 RDBMS“看出来”呢？你肯定很想知道答案，请接着往下读。



这个实例摘自 Jonathan Gennick 撰写的文章“Regular Expression Anti-Patterns”，获得了作者的许可。

以下解决方案使用了模式 **A** 来定义“看起来像”电话号码的内容是什么样的。

```
Pattern A: [0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}
```

模式 **A** 会查找这样的内容：两个 3 位数后跟一个 4 位数，这些数之间用连字符（-）、句点（.）或空格分隔。你可能会设计更复杂的模式，例如，你可能会考虑 7 位数的电话号码。但不要脱离主题，当前的重点是定义看起来像电话号码的内容是什么样的。就这个问题而言，这是由模式 **A** 定义的。你可以定义不同的模式 **A**，但整个解决方案依然管用。

以下解决方案在 **WHERE** 子句中使用了模式 **A**，旨在确保只考虑内容看起来像电话号码的行。

```
select emp_id, text
  from employee_comment
 where regexp_like(text, '[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}')
```

接下来，需要定义格式正确的电话号码是什么样的。为此，以下解决方案使用了模式 **B**。

```
Pattern B: [0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}
```

模式 **B** 使用 **\1** 引用了第一个子表达式，因此与 **([-.])** 匹

配的字符必然与 `\1` 也匹配。模式 **B** 描述了格式正确的电话号码是什么样的，对于这些电话号码，不用考虑，因为它们并不是格式不正确的。为了将格式正确的电话号码排除在外，以下解决方案调用了 `REGEXP_REPLACE`。

```
regexp_replace(text,  
                '[0-9]{3}([- .])[0-9]{3}\1[0-9]{4}', '***'),
```

这个 `REGEXP_REPLACE` 调用位于 `WHERE` 子句中。格式正确的电话号码都将被替换为 3 个星号 (*)。同样，模式 **B** 可以是任何你想要的模式，但重点在于它描述了格式正确的电话号码是什么样的。

将格式正确的电话号码都替换为由 3 个星号组成的字符串后，根据定义，任何看起来像电话号码的内容都必然是格式不正确的。以下解决方案对 `REGEXP_REPLACE` 的输出执行 `REGEXP_LIKE`，看看是否还有格式不正确的电话号码。

```
and regexp_like(  
    regexp_replace(text,  
                    '[0-9]{3}([- .])[0-9]{3}\1[0-9]{4}', '***'),  
    '[0-9]{3}([- .])[0-9]{3}[- .][0-9]{4}')
```



正则表达式本身就是一个庞大的主题，必须经过实践才能掌握。掌握正则表达式后，你将发现使用它们可以轻松匹配各种字符串模式。为了获得必要的正则表达式技能，建议你参阅相关的专著，比如 Jeffrey Friedl 编著的 *Mastering Regular Expressions*。

6.18 小结

匹配字符串可能是一项令人痛苦的任务，为了减轻这样的痛苦，SQL 提供了各种工具，掌握这些工具可以避免你陷入困境。尽管可以使用 SQL 原生字符串函数完成很多任务，但可使用的正则表达式函数越来越多，它们可以让你更上一层楼。

第 7 章 处理数字

本章聚焦于涉及数字的常见操作，包括数值计算。对于复杂的计算，SQL 通常不是第一选择，但在处理日常的数值任务方面，它的效率还是很高的。更重要的是，鉴于组织通常将数据放在支持 SQL 的数据库和数据仓库中，要让这些数据发挥作用，必须使用 SQL 来探索和评估它们。本章介绍的技巧可以帮助数据科学家确定哪些数据最有望在未来的分析中发挥作用。



本章有些实例使用了聚合函数和 **GROUP BY** 子句。如果你不熟悉分组，请务必阅读 A.1 节。

7.1 计算平均值

1. 问题

你想计算某一列的平均值，这可能是整张表中所有行的平均值，也可能是部分行的平均值。例如，你可能想计算全部员工的平均薪水以及每个部门的平均薪水。

2. 解决方案

要计算全部员工的平均薪水，只需将 **AVG** 函数应用于包含薪水的列。

在没有指定 **WHERE** 子句的情况下，计算平均值时将只考虑非 **NULL** 值。

```
1 select avg(sal) as avg_sal
2   from emp

      AVG_SAL
-----
2073.21429
```

要计算每个部门的平均薪水，可以使用 **GROUP BY** 子句创建对应于每个部门的分组。

```
1 select deptno, avg(sal) as avg_sal
2   from emp
3  group by deptno

      DEPTNO      AVG_SAL
-----
          10  2916.66667
```

20	2175
30	1566.66667

13. 讨论

要将整张表作为一个分组或窗口，并计算平均值，只需将函数 **AVG** 应用于目标列，无须使用 **GROUP BY** 子句。函数 **AVG** 会忽略 **NULL** 值，明白这一点很重要。下面的示例说明了忽略 **NULL** 值的效果。

```
create table t2(sal integer)
insert into t2 values (10)
insert into t2 values (20)
insert into t2 values (null)
select avg(sal)      select distinct 30/2
  from t2            from t2

  AVG(SAL)                30/2
-----
          15              15

select avg(coalesce(sal,0))  select distinct 30/3
  from t2                    from t2

AVG(COALESCE(SAL,0))                30/3
-----
          10                        10
```

函数 **COALESCE** 会返回你传入的参数列表中的第一个非 **NULL** 值。将 **SAL** 值 **NULL** 转换为 0 后，平均值发生了变化。调用聚合函数时，务必考虑要如何处理 **NULL** 值。

前述解决方案的第二部分使用 **GROUP BY**（第 3 行）将员工记录按部门进行了分组。**GROUP BY** 自动导致 **AVG** 等聚合函数针对每个分组执行并返回一个结果。在这里，**AVG** 将针对

每个基于部门的员工记录分组执行一次。

顺便说一句，**SELECT** 列表无须包含在 **GROUP BY** 子句中指定的列。

```
select avg(sal)
  from emp
 group by deptno
```

```
      AVG(SAL)
-----
2916.66667
      2175
1566.66667
```

虽然 **SELECT** 子句中没有包含 **DEPTNO**，但还是会按 **DEPTNO** 分组。在 **SELECT** 子句中包含作为分组依据的列通常可以提高可读性，但并非必须这样做。然而，没有出现在 **GROUP BY** 子句中的列不能包含在 **SELECT** 列表中。

14. 另请参阅

如果要复习 **GROUP BY** 的功能，请参阅附录 A。

7.2 找出最大列值和最小列值

1. 问题

你想找出给定列中的最大值和最小值。例如，你想找出所有员工的最高薪水和最低薪水，以及每个部门的最高薪水和最低薪水。

2. 解决方案

要找出所有员工的最高薪水和最低薪水，只需分别使用函数 **MAX** 和 **MIN**。

```
1 select min(sal) as min_sal, max(sal) as max_sal
2   from emp
```

MIN_SAL	MAX_SAL
800	5000

要找出每个部门的最高薪水和最低薪水，可以将函数 **MAX** 和 **MIN** 同子句 **GROUP BY** 结合起来使用。

```
1 select deptno, min(sal) as min_sal, max(sal) as max_sal
2   from emp
3  group by deptno
```

DEPTNO	MIN_SAL	MAX_SAL
10	1300	5000
20	800	3000
30	950	2850

13. 讨论

在整张表为分组或窗口的情况下，要找出最高值或最低值，只需将函数 **MAX** 或 **MIN** 应用于目标列，无须使用 **GROUP BY** 子句。

别忘了，函数 **MAX** 和 **MIN** 会忽略 **NULL** 值，在有些分组中，可能所有行的列值都为 **NULL**，也可能只是部分行的列值为 **NULL**。在下面的示例中，使用 **GROUP BY** 的查询返回的两个分组（它们的 **DEPTNO** 分别为 10 和 30）的值都为 **NULL**。

```
select deptno, comm
  from emp
 where deptno in (10,30)
 order by 1
```

DEPTNO	COMM
10	
10	
10	
30	300
30	500
30	
30	0
30	1300
30	

```
select min(comm), max(comm)
  from emp
```

MIN(COMM)	MAX(COMM)
0	1300

```
select deptno, min(comm), max(comm)
  from emp
 group by deptno
```

DEPTNO	MIN(COMM)	MAX(COMM)
10		
20		
30	0	1300

正如附录 A 指出的，即便 **SELECT** 子句中只包含聚合函数，也可以按表中的其他列分组，如下所示。

<pre>select min(comm), max(comm) from emp group by deptno</pre>		
MIN(COMM)	MAX(COMM)	
0	1300	

虽然 **SELECT** 子句中没有包含 **DEPTNO**，但依然可以按照 **DEPTNO** 进行分组。在 **SELECT** 子句中包含作为分组依据的列通常可以提高可读性，但并非必须这样做。然而，在 **GROUP BY** 查询中，**SELECT** 列表中的所有列都必须包含在 **GROUP BY** 子句中。

14. 另请参阅

如果要复习 **GROUP BY** 的功能，请参阅附录 A。

7.3 计算列值总和

1. 问题

你想计算列值的总和，比如计算所有员工的薪水总额。

2. 解决方案

在整张表为分组或窗口时，要计算总和，只需将函数 **SUM** 应用于目标列，无须使用 **GROUP BY** 子句。

```
1 select sum(sal)
2   from emp

      SUM(SAL)
-----
      29025
```

要创建多个数据分组或窗口，并分别计算它们的总和，可以结合使用函数 **SUM** 和 **GROUP BY** 子句。下面的示例会计算各部门员工的薪水总额。

```
1 select deptno, sum(sal) as total_for_dept
2   from emp
3  group by deptno

      DEPTNO      TOTAL_FOR_DEPT
-----
          10          8750
          20         10875
          30          9400
```

13. 讨论

计算每个部门的薪水总额时，需要创建数据分组或窗口。对于每个部门，将该部门所有员工的薪水相加，得到部门薪水总额。这是一个 SQL 聚合示例，因为它关注的不是详情（比如每位员工的薪水），而是每个部门的最终结果。需要指出的是，函数 SUM 会忽略 NULL 值，但你可以有最终结果为 NULL 的分组，如下面的示例所示。在这个示例中，10 号部门的员工都没有业务提成，因此按 DEPTNO 分组，并计算 COMM 列值总和时，SUM 将返回一个值为 NULL 的分组。

```
select deptno, comm
  from emp
 where deptno in (10,30)
 order by 1
```

DEPTNO	COMM
10	
10	
10	
30	300
30	500
30	
30	0
30	1300
30	

```
select sum(comm)
  from emp
```

SUM(COMM)
2100

```
select deptno, sum(comm)
  from emp
 where deptno in (10,30)
 group by deptno
```


DEPTNO	SUM(COMM)
10	
30	2100

14. 另请参阅

如果要复习 **GROUP BY** 的功能，请参阅附录 A。

7.4 计算表中的行数

1. 问题

你想计算特定表包含多少行或特定列包含多少个值。例如，你想确定员工总数以及每个部门的员工数。

2. 解决方案

在整张表为分组或窗口时，要计算行数，只需使用函数 **COUNT** 和字符 *****。

```
1 select count(*)
2   from emp

COUNT(*)
-----
        14
```

要创建多个数据分组或窗口，并分别计算它们包含的行数，可以结合使用函数 **COUNT** 和 **GROUP BY** 子句。

```
1 select deptno, count(*)
2   from emp
3  group by deptno

DEPTNO      COUNT(*)
-----
        10             3
        20             5
        30             6
```

13. 讨论

为了计算各部门的员工数量，可以创建数据分组或窗口。在各个部门中，每找到一位员工，就将计数加 1，最终得到该部门的员工总数。这是一个 SQL 聚合示例，因为它关注的不是详情（比如每位员工的薪水或职位），而是每个部门的最终结果。需要指出的是，如果将列名作为参数传递给函数 **COUNT**，那么它将忽略该列为 **NULL** 值的行，但将字符 ***** 或其他任何常量作为参数时，不会忽略 **NULL** 值。

```
select deptno, comm
from emp
```

DEPTNO	COMM
20	
30	300
30	500
20	
30	1300
30	
10	
20	
10	
30	0
20	
30	
20	
10	

```
select count(*), count(deptno), count(comm), count('hello')
from emp
```

COUNT(*)	COUNT(DEPTNO)	COUNT(COMM)	COUNT('HELLO')
14	14	4	14

```
select deptno, count(*), count(comm), count('hello')
from emp
```

group by deptno			
DEPTNO	COUNT(*)	COUNT(COMM)	COUNT('HELLO')
10	3	0	3
20	5	0	5
30	6	4	6

如果在所有行中作为参数传递给 **COUNT** 的列值都为 **NULL**，那么 **COUNT** 将返回 0。如果表是空的，则 **COUNT** 也将返回 0。还需指出的是，即便 **SELECT** 子句只包含聚合函数，也可以按表中的其他列分组。

<pre>select count(*) from emp group by deptno</pre>	
COUNT(*)	

3	
5	
6	

注意，虽然 **DEPTNO** 没有包含在 **SELECT** 子句中，但依然可以按 **DEPTNO** 分组。在 **SELECT** 子句中包含作为分组依据的列通常可以提高可读性，但并非必须这样做。然而，包含在 **SELECT** 列表中的列也必须包含在 **GROUP BY** 子句中。

14. 另请参阅

如果要复习 **GROUP BY** 的功能，请参阅附录 A。

7.5 计算非NULL列值数

1. 问题

你想计算特定列中的非 NULL 值个数。例如，你想确定有多少位员工有业务提成。

2. 解决方案

计算 EMP 表的 COMM 列中非 NULL 值的个数。

```
select count(comm)
  from emp

COUNT(COMM)
-----
                4
```

3. 讨论

将函数 COUNT 的参数指定为星号（COUNT(*)）时，实际上计算的是行数（而不管实际值如何，即把包含 NULL 值和非 NULL 值的行都计算在内）。然而，将列名作为函数 COUNT 的参数时，计算的是该列值非 NULL 值的个数。上一节的“讨论”部分说明了这种差别。在上述解决方案中，COUNT(COMM) 返回了 COMM 列不为 NULL 值的行数。由于仅当员工有业务提成时，其 COMM 列的值才不为 NULL，因此 COUNT(COMM) 返回的是有业务提成的员工数量。

7.6 生成移动总计

1. 问题

你想计算列值的移动总计。

2. 解决方案

例如，下面的解决方案演示了如何计算所有员工薪水的移动总计。为了提高可读性，对结果按 **SAL** 进行了排序，让你能够轻松地查看移动总计的变化过程。

```
1 select ename, sal,  
2        sum(sal) over (order by sal,empno) as running_total  
3   from emp  
4  order by 2
```

ENAME	SAL	RUNNING_TOTAL
-----	-----	-----
SMITH	800	800
JAMES	950	1750
ADAMS	1100	2850
WARD	1250	4100
MARTIN	1250	5350
MILLER	1300	6650
TURNER	1500	8150
ALLEN	1600	9750
CLARK	2450	12200
BLAKE	2850	15050
JONES	2975	18025
SCOTT	3000	21025
FORD	3000	24025
KING	5000	29025

13. 讨论

使用窗口函数 **SUM OVER** 能够轻松地生成移动总计。在上述解决方案中，**ORDER BY** 子句包含 **SAL** 列和 **EMPNO** 列（主键），以免移动总计中包含重复值。在下面的示例中，**RUNNING_TOTAL2** 列说明了包含重复值时可能出现的问题。

<pre>select empno, sal, sum(sal)over(order by sal,empno) as running_total1, sum(sal)over(order by sal) as running_total2 from emp order by 2</pre>			
ENAME	SAL	RUNNING_TOTAL1	RUNNING_TOTAL2
-----	-----	-----	-----
SMITH	800	800	800
JAMES	950	1750	1750
ADAMS	1100	2850	2850
WARD	1250	4100	5350
MARTIN	1250	5350	5350
MILLER	1300	6650	6650
TURNER	1500	8150	8150
ALLEN	1600	9750	9750
CLARK	2450	12200	12200
BLAKE	2850	15050	15050
JONES	2975	18025	18025
SCOTT	3000	21025	24025
FORD	3000	24025	24025
KING	5000	29025	29025

WARD、MARTIN、SCOTT 和 FORD 的 **RUNNING_TOTAL2** 值不正确。他们的薪水出现了多次，因此应将重复值相加，再将结果加入移动总计。有鉴于此，要生成正确的结果（**RUNNING_TOTAL1** 列所示的值），需要在排序依据中包含 **EMPNO**（其值是独一无二的）。对于 ADAMS，其 **RUNNING_TOTAL1** 列和 **RUNNING_TOTAL2** 列的值都是 2850。将 WARD 的薪水 1250 与 2850 相加后，结果为 4100，但

RUNNING_TOTAL2 列的值为 5350。怎么会这样呢？

这是因为 **WARD** 和 **MARTIN** 的 **SAL** 值相同，都是 1250，将他们的薪水相加后，结果为 2500，再将这个结果与 2850 相加，结果为 5350，因此，**WARD** 和 **MARTIN** 的 **RUNNING_TOTAL2** 列的值都为 5350。按照不会导致重复值的列组合（例如，独一无二的 **SAL** 和 **EMPNO** 的组合）排序，可以确保移动总计是正确的。

7.7 生成移动总积

1. 问题

你想计算数字列的移动总积。此操作与上一节的实例类似，但使用的是乘法运算，而不是加法运算。

2. 解决方案

本解决方案以计算员工薪水的移动总积为例。虽然薪水的移动总积没什么用，但使用这里介绍的方法，可以轻松计算其他更有用的移动总积。

可以将对数相加来模拟乘法运算，然后结合使用窗口函数 **SUM OVER**，就可以计算移动总积了。

```
1 select empno,ename,sal,
2         exp(sum(ln(sal))over(order by sal,empno)) as running_prod
3   from emp
4  where deptno = 10
```

EMPNO	ENAME	SAL	RUNNING_PROD
7934	MILLER	1300	1300
7782	CLARK	2450	3185000
7839	KING	5000	15925000000

在 SQL 中（准确地说是在数学中），计算小于或等于 0 的值的对数是非法的。如果表中有这样的值，则需要避免将这些非法值传递给 SQL 的函数 **LN**。出于可读性考虑，本解决方案没有采取防止传入值为 **NULL** 或非法的措施，但编写生产代码时，必须考虑是否要采取这样的防范措施。如果肯定会

涉及负数或 0，则本解决方案可能不管用。另外，如果涉及 0（但不涉及负数），那么一种常规的规避办法是给所有的值都加 1，因为不管底数是多少，1 的对数都是 0。

如果你使用的是 SQL Server，请将 LN 替换为 LOG。

13. 讨论

上述解决方案利用了这样一点，即可以采取如下方法来计算两个数的乘积。

- a. 分别计算它们的自然对数。
- b. 计算这两个对数的和。
- c. 使用函数 **EXP** 计算数学常数 e 的 n 次幂，其中 n 为前述两个对数的和。

需要注意的是，如果列值可能为 0 或负数，那么这种方法将行不通，因为在 SQL 中计算对数时，传入的值不能小于或等于 0。

有关窗口函数 **SUM OVER** 的工作原理，请参阅 7.6 节。

7.8 平滑值序列

1. 问题

你有一系列随时间变化的值，比如月度销量。通常，相邻数据点之间的波动很大，但你想知道的是总体趋势。为了更好地找出总体趋势，你要实现一个简单的平滑器，比如加权移动平均。

假设有一个报摊，其日销售额（单位为美元）如下。

DATE1	SALES
2020-01-01	647
2020-01-02	561
2020-01-03	741
2020-01-04	978
2020-01-05	1062
2020-01-06	1072
...	...

但你知道，销售数据的波动性导致难以发现总体趋势。例如，一周或一个月的某些天的销量特别高或特别低。或者，由于数据收集方式的影响，有些天的销量并入了下一天，导致波谷后面紧跟着一个波峰，可是你又无法在这两天之间正确地分配销售总量。因此，要弄明白发生的情况，需要平滑一段时间内的数据。

为此，可以计算移动平均，方法是将当前值与前 $n-1$ 个值相加，再除以 n 。如果同时显示以前的值作为参考，结果将如下所示。

DATE1	sales	salesLagOne	SalesLagTwo	MovingAverage
-----	-----	-----	-----	-----
2020-01-01	647	NULL	NULL	NULL
2020-01-02	561	647	NULL	NULL

2020-01-03	741	561	647	649.667
2020-01-04	978	741	561	760
2020-01-05	1062	978	741	927
2020-01-06	1072	1062	978	1037.333
2020-01-07	805	1072	1062	979.667
2020-01-08	662	805	1072	846.333
2020-01-09	1083	662	805	850
2020-01-10	970	1083	662	905

12. 解决方案

计算平均值的公式众所周知。为了更好地解决上述问题，可以使用加权移动平均，对于越接近预测期的值，指定越大的权重。要计算移动平均，可以使用窗口函数 **LAG**。

```
select date1, sales, lag(sales,1) over(order by date1) as
salesLagOne,
lag(sales,2) over(order by date1) as salesLagTwo,
(sales
+ (lag(sales,1) over(order by date1))
+ lag(sales,2) over(order by date1))/3 as MovingAverage
from sales
```

13. 讨论

分析时序数据（出现在特定时段内的数据）时，最简单的一个方法是使用加权移动平均。这里演示的是计算简单移动平均的一个方法，你也可以结合使用分区和移动平均。我们选择的是简单的三点移动平均，但根据要分析的数据的特征，也可以使用不同的公式和不同的数据点数，以充分发挥这种方法的作用。

例如，下面的解决方案演化版本会计算三点加权移动平均，

通过修改系数和分母，它会对越接近预测期的数据，指定越大的权重。

```
select date1, sales, lag(sales,1) over(order by date1),  
lag(sales,2) over(order by date1),  
((3*sales)  
+ (2*(lag(sales,1) over(order by date1)))  
+ (lag(sales,2) over(order by date1)))/6 as SalesMA  
from sales
```

7.9 计算众数

1. 问题

你想找出列值的众数（众数的数学定义是，在给定数据集中，出现频率最高的元素）。例如，你想找出 20 号部门的员工薪水众数。

在下面的薪水中，众数为 3000。

```
select sal
  from emp
 where deptno = 20
 order by sal
```

```
      SAL
-----
      800
     1100
     2975
     3000
     3000
```

2. 解决方案

DB2、MySQL、PostgreSQL 和 SQL Server

为了获取众数，可以使用窗口函数 `DENSE_RANK` 按出现次数对薪水值排名。

```
1 select sal
2   from (
3 select sal,
4        dense_rank()over( order by cnt desc) as rnk
```

```
5  from (
6  select sal, count(*) as cnt
7  from emp
8  where deptno = 20
9  group by sal
10         ) x
11         ) y
12  where rnk = 1
```

Oracle

可以使用聚合函数 **MAX** 的 **KEEP** 扩展来找出薪水众数。需要指出的是，如果出现分不出胜负的情况，即多个薪水值出现的次数都是最多的，那么使用 **KEEP** 的解决方案将只返回最高的薪水。如果要返回所有的众数，则必须修改该解决方案，或使用前面的 **DB2** 解决方案。在本例中，20 号部门的薪水众数为 3000，也是最高的薪水，因此以下解决方案足够用了。

```
1 select max(sal)
2         keep(dense_rank first order by cnt desc) sal
3  from (
4  select sal, count(*) cnt
5  from emp
6  where deptno=20
7  group by sal
8         )
```

13. 讨论

DB2 和 SQL Server

内嵌视图 **X** 会返回每个薪水值及其出现的次数，内嵌视图 **Y** 会使用窗口函数 **DENSE_RANK**（它支持平局）对结果进行排序。

结果是根据每个薪水值出现的次数进行排名的。

```
1 select sal,
2     dense_rank()over(order by cnt desc) as rnk
3   from (
4 select sal,count(*) as cnt
5   from emp
6  where deptno = 20
7  group by sal
8        ) x
```

SAL	RNK
3000	1
800	2
1100	2
2975	2

这个查询的最外层只保留 **RNK** 为 1 的行。

Oracle

首先，内嵌视图会返回每个薪水值及其出现的次数。

```
select sal, count(*) cnt
  from emp
 where deptno=20
  group by sal
```

SAL	CNT
800	1
1100	1
2975	1
3000	2

然后，使用聚合函数 **MAX** 的 **KEEP** 扩展来找出众数。如果对这里的 **KEEP** 子句进行分析，你将发现它包含 3 个子句：**DENSE_RANK**、**FIRST** 和 **ORDER BY CNT DESC**。


```
keep(dense_rank first order by cnt desc)
```

这样做能非常轻松地找出众数。**KEEP** 子句指定了 **MAX** 将返回哪个 **SAL** 值——根据内嵌视图返回的 **CNT** 来决定。**KEEP** 中的子句按从右到左的顺序起作用：先按 **CNT** 值降序排列，再保留按 **DENSE_RANK** 顺序返回的第一个 **CNT** 值。从内嵌视图返回的结果集可知，薪水值 3000 的 **CNT** 值最大（为 2），因此 **MAX(SAL)** 将返回 **CNT** 值最大的 **SAL** 值，本例中为 3000。

14. 另请参阅

有关 Oracle 聚合函数的 **KEEP** 扩展的更深入讨论，请参阅第 11 章，尤其是 11.11 节。

7.10 计算中值

1. 问题

你想计算数字列的中值（中值指的是在一组有序元素中，位于正中央的那个元素的值）。例如，你想找出 20 号部门的薪水中值。在下面的薪水中，中值为 2975。

```
select sal
  from emp
 where deptno = 20
 order by sal
```

```
      SAL
-----
      800
     1100
     2975
     3000
     3000
```

2. 解决方案

在 Oracle 中，可以使用内置函数来计算中值。在其他 DBMS 中，传统的解决方案是使用自连接，但引入窗口函数后，可以使用效率更高的解决方案。

DB2 和 PostgreSQL

使用窗口函数 `PERCENTILE_CONT` 来查找中值。

```
1 select percentile_cont(0.5)
2       within group(order by sal)
3   from emp
```

```
4 where deptno=20
```

SQL Server

使用窗口函数 `PERCENTILE_CONT` 来查找中值。

```
1 select percentile_cont(0.5)
2       within group(order by sal)
3       over()
4   from emp
5  where deptno=20
```

SQL Server 解决方案与 DB2 解决方案相同，但需要包含 `OVER` 子句。

MySQL

MySQL 中没有函数 `PERCENTILE_CONT`，因此需要采取权变措施。可以结合使用函数 `CUME_DIST` 和 CTE 来模拟函数 `PERCENTILE_CONT`。

```
with rank_tab (sal, rank_sal) as
(
  select sal, cume_dist() over (order by sal)
        from emp
        where deptno=20
),
inter as
(
  select sal, rank_sal from rank_tab
  where rank_sal>=0.5
union
  select sal, rank_sal from rank_tab
  where rank_sal<=0.5
)

select avg(sal) as MedianSal
   from inter
```

Oracle

使用函数 **MEDIAN** 或 **PERCENTILE_CONT**。

```
1 select median(sal)
2   from emp
3  where deptno=20

1 select percentile_cont(0.5)
2       within group(order by sal)
3   from emp
4  where deptno=20
```

13. 讨论

Oracle、PostgreSQL、SQL Server 和 DB2

除了使用函数 **MEDIAN** 的 Oracle 解决方案，其他所有解决方案的结构都相同。函数 **PERCENTILE_CONT** 能够直接应用中值的定义，因为中值的定义就是第 50 百分位数。因此，只要使用合适的语法来执行这个函数，并将参数设置为 0.5，就可以找到中值。

当然，也可以给这个函数指定其他百分位数。例如，使用第 5 和/或第 95 百分位数，可以找出异常值。（本章后面讨论绝对中位差时，概述了另一种找出异常值的方法。）

MySQL

MySQL 中没有函数 **PERCENTILE_CONT**，问题解决起来要棘手些。为了找出中值，必须将 **SAL** 值按从低到高的顺序排列。函数 **CUME_DIST** 可以实现这个目标，同时标出每一行的百分位数，因此可以使用它来获得与其他数据库解决方案

使用 `PERCENTILE_CONT` 时一样的结果。

唯一的难点是，在 `WHERE` 子句中不能使用函数 `CUME_DIST`，因此需要先在 CTE 中执行它。

这里存在一个陷阱，如果行数为偶数，那么将没有位于正中央的行。因此，在解决方案中，找出了百分位数不超过 50 的最大值和百分位数不低于 50 的最小值，再计算它们的平均值。无论行数为奇数还是偶数，这种方法都管用，而且在行数为奇数的情况下，找出的将是中值，因为它计算的是两个相同值的平均值。

7.11 计算总计占比

1. 问题

你想计算特定列的各个列值占总计的百分比。例如，你想确定 10 号部门的薪水占薪水总额的百分比。

2. 解决方案

一般而言，在 SQL 中，计算总计占比与在纸上完成这种计算的方法相同，只需先做除法，再做乘法。在本例中，要计算 10 号部门的薪水占 EMP 表中薪水总额的百分比，只需先找出 10 号部门的薪水总额，将其除以整张表的薪水总额，再乘以 100 得到百分比值。

MySQL 和 PostgreSQL

将 10 号部门的薪水总额除以整张表的薪水总额。

```
1 select (sum(  
2         case when deptno = 10 then sal end)/sum(sal)  
3         )*100 as pct  
4   from emp
```

DB2、Oracle 和 SQL Server

在一个内嵌视图中，使用窗口函数 SUM OVER 计算整张表的薪水总额和 10 号部门的薪水总额，然后在外部查询中执行除法运算和乘法运算。

```
1 select distinct (d10/total)*100 as pct  
2   from (
```

```

3 select deptno,
4         sum(sal)over() total,
5         sum(sal)over(partition by deptno) d10
6   from emp
7        ) x
8  where deptno=10

```

13. 讨论

MySQL 和 PostgreSQL

CASE 语句只返回 10 号部门员工的薪水。然后，将这些薪水相加，再除以整张表中的薪水总额。由于聚合函数会忽略 **NULL** 值，因此在 **CASE** 语句中不需要使用 **ELSE** 子句。要获悉相除的是哪两个值，可以执行下面这个不包含除法运算的查询。

```

select sum(case when deptno = 10 then sal end) as d10,
       sum(sal)
  from emp

```

```

D10  SUM(SAL)
---- -
8750      29025

```

根据 **SAL** 列的定义，执行除法运算时可能需要显式地使用 **CAST** 来确保数据类型是正确的。例如，在 **DB2**、**SQL Server** 和 **PostgreSQL** 中，如果 **SAL** 列的数据类型为整型，那么可以使用 **CAST** 来确保返回的是小数值。

```

select (cast(
       sum(case when deptno = 10 then sal end)
       as decimal)/sum(sal)
       )*100 as pct
  from emp

```

DB2、Oracle 和 SQL Server

作为传统解决方案的替代方案，以下解决方案使用窗口函数来计算占总计的百分比。在 DB2 和 SQL Server 中，如果 SAL 列的数据类型为整型，则需要在执行除法运算前使用 CAST 进行强制类型转换。

```
select distinct
    cast(d10 as decimal)/total*100 as pct
  from (
select deptno,
    sum(sal)over() total,
    sum(sal)over(partition by deptno) d10
  from emp
    ) x
 where deptno=10
```

请牢记，窗口函数是在 **WHERE** 子句执行后执行的，因此基于 **DEPTNO** 的过滤器不能放在内嵌视图 **X** 中。下面来看看在包含和不包含基于 **DEPTNO** 的过滤器时，内嵌视图 **X** 返回的结果。先来看不包含该过滤器时的结果。

```
select deptno,
    sum(sal)over() total,
    sum(sal)over(partition by deptno) d10
  from emp
```

DEPTNO	TOTAL	D10
-----	-----	-----
10	29025	8750
10	29025	8750
10	29025	8750
20	29025	10875
20	29025	10875
20	29025	10875
20	29025	10875
20	29025	10875
30	29025	9400
30	29025	9400
30	29025	9400

30	29025	9400
30	29025	9400
30	29025	9400

再来看包含该过滤器时的结果。

<pre>select deptno, sum(sal)over() total, sum(sal)over(partition by deptno) d10 from emp where deptno=10</pre>		
DEPTNO	TOTAL	D10
-----	-----	-----
10	8750	8750
10	8750	8750
10	8750	8750

由于窗口函数是在 **WHERE** 子句执行后执行的，因此 **TOTAL** 值为 10 号部门的薪水总额。然而，你希望 **TOTAL** 为整张表的薪水总额，因此基于 **DEPTNO** 的过滤器必须放在内嵌视图 **X** 外面。

7.12 聚合值可为NULL的列

1. 问题

你想对特定列执行聚合运算，但该列的值可为 **NULL**。你希望聚合结果是精确的，但问题是聚合函数会忽略 **NULL**。例如，你要计算 30 号部门员工的平均业务提成，但有些员工没有业务提成，因此这些员工的 **COMM** 列为 **NULL**。聚合函数会忽略 **NULL**，这将导致结果的准确度大打折扣。你希望执行聚合运算时，将目标列值为 **NULL** 的行也考虑进来。

2. 解决方案

使用函数 **COALESCE** 将 **NULL** 转换为 0，让聚合运算将它们也考虑进来。

```
1 select avg(coalesce(comm,0)) as avg_comm
2   from emp
3  where deptno=30
```

3. 讨论

使用聚合函数时，请牢记它们会忽略 **NULL**。请看下面这个未使用函数 **COALESCE** 的解决方案及其输出：

```
select avg(comm)
  from emp
 where deptno=30

AVG(COMM)
```

这个查询指出，30 号部门的平均业务提成为 550，但从下面显示的行可知，在全部 6 名员工中，只有 4 名员工有业务提成。

```
select ename, comm
  from emp
 where deptno=30
order by comm desc
```

ENAME	COMM
BLAKE	
JAMES	
MARTIN	1400
WARD	500
ALLEN	300
TURNER	0

30 号部门的业务提成总额为 2200，因此平均业务提成应为 2200/6，而不是 2200/4。没有使用函数 **COALESCE** 时，回答的问题是“在 30 号部门中，有业务提成的员工的平均业务提成是多少？”而不是“30 号部门的所有员工的平均业务提成是多少？”使用聚合函数时，务必妥善处理 **NULL**。

7.13 计算剔除最高值和最低值后的平均值

1. 问题

你要计算平均值，但想将最高值和最低值排除在外，以降低扭曲效果。在统计学中，剔除最大值和最小值后的平均值被称为截尾均值（trimmed mean）。例如，你想计算剔除最高薪水和最低薪水后全部员工的平均薪水。

2. 解决方案

MySQL 和 PostgreSQL

使用子查询将最高值和最低值排除在外。

```
1 select avg(sal)
2   from emp
3  where sal not in (
4      (select min(sal) from emp),
5      (select max(sal) from emp)
6  )
```

DB2、Oracle 和 SQL Server

在内嵌视图使用窗口函数 **MAX OVER** 和 **MIN OVER** 来生成一个结果集，并使用这个结果集将最高值和最低值排除在外。

```
1 select avg(sal)
2   from (
3 select sal, min(sal)over() min_sal, max(sal)over() max_sal
4   from emp
5      ) x
```

```
6 where sal not in (min_sal,max_sal)
```

13. 讨论

MySQL 和 PostgreSQL

子查询能够返回表中的最高薪水和最低薪水。结合使用 **NOT IN** 和子查询返回的值，可以将最高薪水和最低薪水排除在外，再计算平均值。别忘了，如果有多位员工的薪水是最高的或最低的，那么计算平均值时他们都将排除在外。如果你的目标是只排除单个最高值或最低值，则可以从 **SUM** 结果中减去它们，再执行除法运算。

```
select (sum(sal)-min(sal)-max(sal))/(count(*)-2)
from emp
```

DB2、Oracle 和 SQL Server

内嵌视图 **X** 会返回每个薪水值以及最高薪水和最低薪水。

```
select sal, min(sal)over() min_sal, max(sal)over() max_sal
from emp
```

SAL	MIN_SAL	MAX_SAL
800	800	5000
1600	800	5000
1250	800	5000
2975	800	5000
1250	800	5000
2850	800	5000
2450	800	5000
3000	800	5000
5000	800	5000
1500	800	5000
1100	800	5000

950	800	5000
3000	800	5000
1300	800	5000

在每一行中，都可以访问最高薪水和最低薪水，因此找出最高薪水和最低薪水易如反掌。外部查询会对内嵌视图 **X** 返回的行进行过滤，将薪水与 **MIN_SAL** 或 **MAX_SAL** 相同的行排除在外，再计算平均值。



计算截尾均值可被视为更稳妥的均值估算，也是一个稳健统计的例子。为何称之为稳健统计呢？这是因为其对偏差等问题不那么敏感。7.16 节将介绍另一个稳健统计工具。对在 **RDBMS** 中分析数据的人来说，这些方法很有价值，因为它们不要求分析人员做出使用有限的 **SQL** 统计工具难以测试的假设。

7.14 将由字母和数字组成的字符串转换为数字

1. 问题

你有一些由字母和数字组成的数据，但只想返回其中的数字。例如，你想返回字符串 `paul123f321` 中的数字 `123321`。

2. 解决方案

DB2

使用函数 `TRANSLATE` 和 `REPLACE` 提取字母数字字符串中的数字。

```
1 select cast(  
2     replace(  
3     translate( 'paul123f321',  
4               repeat('#',26),  
5               'abcdefghijklmnopqrstuvwxyz'),'#','')  
6     as integer ) as num  
7 from t1
```

Oracle、SQL Server 和 PostgreSQL

使用函数 `TRANSLATE` 和 `REPLACE` 提取字母数字字符串中的数字。

```
1 select cast(  
2     replace(  
3     translate( 'paul123f321',  
4               'abcdefghijklmnopqrstuvwxyz',  
5               rpad('#',26,'#')),'#','')  
6     as integer ) as num  
7 from t1
```

MySQL

本书撰写之时，MySQL 不支持函数 `TRANSLATE`，因此这里没有提供 MySQL 解决方案。

13. 讨论

上述两种解决方案之间唯一的不同是语法：DB2 使用函数 `REPEAT` 而不是函数 `RPAD`，并且函数 `TRANSLATE` 的参数列表的顺序不同。这里的讨论以 Oracle/PostgreSQL 解决方案为例，但也适用于 DB2。如果从里往外运行查询（从只包含 `TRANSLATE` 的查询开始），就会发现以下解决方案很简单。首先，`TRANSLATE` 会将所有非数字字符都转换为 `#`。

```
select translate( 'paul123f321',
                  'abcdefghijklmnopqrstuvwxyz',
                  rpad('#',26,'#')) as num
  from t1

NUM
-----
#####123#321
```

将所有非数字字符都转换为 `#` 后，只需使用 `REPLACE` 将 `#` 都删除，然后使用 `CAST` 将结果作为数字返回。这个示例非常简单，因为数据是字母数字字符串。如果数据中还包含其他字符，那么相比于查找所有非数字字符，使用不同的方法来解决这个问题将更容易：找到所有数字字符，并将其他字符都删除，而不是查找所有非数字字符并将它们都删除。下面的示例有助于清楚地说明这种方法。

```
select replace(
```



```

        translate('paul123f321',
            replace(translate( 'paul123f321',
                              '0123456789',
                              rpad('#',10,'#')), '#', ''),
            rpad('#',length('paul123f321'),'#')), '#', '') as num
    from t1

NUM
-----
123321

```

相比于前面的解决方案，以上解决方案更令人费解，但对其进行分解后，就不那么难理解了。先来看最里层的 **TRANSLATE** 调用。

```

select translate( 'paul123f321',
                  '0123456789',
                  rpad('#',10,'#'))

    from t1

TRANSLATE('
-----
paul###f###

```

这里不是将每个非数字字符都替换为 #，而是将每个数字字符都替换为 #。然后，删除所有的 #，只留下非数字字符。

```

select replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')), '#', '')

    from t1

REPLA
-----
paulf

```

接下来，再次调用 **TRANSLATE**，将原始字符串中的每个非数字字符都替换为 #。

```
select translate('paul123f321',
                replace(translate( 'paul123f321',
                                   '0123456789',
                                   rpad('#',10,'#')), '#', '' ),
                rpad('#',length('paul123f321'),'#'))
from t1

TRANSLATE('
-----
####123#321
```

现在仔细研究一下最外层的 **TRANSLATE** 调用。**RPAD** 的第二个参数（在 **DB2** 解决方案中，是 **REPEAT** 的第二个参数）为原始字符串的长度。这个值非常好用，因为在一个字符串中，任何字符出现的次数都不可能超过该字符串的长度。将所有非数字字符都替换为 **#** 后，最后使用 **REPLACE** 将所有的 **#** 都删除，得到一个数字。

7.15 修改移动总计中的值

1. 问题

你想根据另一列的值修改移动总计中的值。请看下面的场景：你要显示一个信用卡账户的交易记录以及每次交易后的余额。本实例会将下面的视图 **V** 作为数据源。

```
create view V (id,amt,trx)
as
select 1, 100, 'PR' from t1 union all
select 2, 100, 'PR' from t1 union all
select 3, 50, 'PY' from t1 union all
select 4, 100, 'PR' from t1 union all
select 5, 200, 'PY' from t1 union all
select 6, 50, 'PY' from t1
```

```
select * from V
```

ID	AMT	TR
--	-----	--
1	100	PR
2	100	PR
3	50	PY
4	100	PR
5	200	PY
6	50	PY

ID 列可以唯一地标识每次交易。**AMT** 列表示交易涉及的金额（要么为收入，要么为支出）。**TR** 列为交易类型：支出用 **PY** 表示，收入用 **PR** 表示。如果 **TR** 列的值为 **PY**，就从移动总计中减去当前 **AMT** 值；如果 **TR** 列的值为 **PR**，就将移动总计加上当前 **AMT** 值。你希望返回的最终结果集如下所示。

TRX_TYPE	AMT	BALANCE
-----	-----	-----
PURCHASE	100	100

PURCHASE	100	200
PAYMENT	50	150
PURCHASE	100	250
PAYMENT	200	50
PAYMENT	50	0

12. 解决方案

使用窗口函数 **SUM OVER** 生成移动总计，并使用 **CASE** 表达式来确定交易类型。

```

1 select case when trx = 'PY'
2           then 'PAYMENT'
3           else 'PURCHASE'
4       end trx_type,
5       amt,
6       sum(
7           case when trx = 'PY'
8               then -amt else amt
9           end
10      ) over (order by id,amt) as balance
11 from V

```

13. 讨论

CASE 表达式用于确定将当前 **AMT** 值加入移动总计还是从移动总计中减去。如果交易类型为支付，就对 **AMT** 值求负，从而将其从移动总计中减去。这个 **CASE** 表达式的结果如下所示。

```

select case when trx = 'PY'
           then 'PAYMENT'
           else 'PURCHASE'
       end trx_type,

```

case when trx = 'PY' then -amt else amt end as amt from V	
TRX_TYPE	AMT
-----	-----
PURCHASE	100
PURCHASE	100
PAYMENT	-50
PURCHASE	100
PAYMENT	-200
PAYMENT	-50

确定交易类型后，再将 **AMT** 值加入移动总计或从移动总计中减去。有关窗口函数 **SUM OVER**（或标量子查询）如何生成移动总计的说明，请参阅 7.6 节。

7.16 使用绝对中位差找出异常值

1. 问题

你想找出数据集中可能存在疑问的值。值存在疑问有多种原因：可能是数据收集方式有问题，比如记录值的仪表存在误差；可能是数据输入错误导致的；还可能是因为数据生成时环境出现异常，这意味着数据点是正确的，但应谨慎根据数据得出任何结论。有鉴于此，你想检测出异常数据。

一种检测异常数据的常用方法是，计算数据的标准偏差，并将超过 3 倍标准偏差（或其他类似距离）的数据点视为异常数据。很多针对非统计学家的统计学课程会教授这种方法。然而，如果数据不符合正态分布，则这种方法可能错误地识别异常数据，而当数据分布不对称，或者如果你远离平均值，数据就不像正态分布那样变得稀疏时更是如此。

2. 解决方案

先使用 7.10 节的解决方案找出中值。你需要将这个查询放在 CTE 中，以便对其结果做进一步查询。偏差是中值与各个值的绝对差，而绝对中位差是偏差的中值，因此需要再次计算中值。

SQL Server

SQL Server 中提供了函数 `PERCENTILE_CONT`，其可以简化查找中值的任务。由于要找出并操作两个不同的中值，因此需要一系列 CTE。

```

with median (median)
as
(select distinct percentile_cont(0.5) within group(order by sal)
      over()
from emp),

Deviation (Deviation)
      as
(Select abs(sal-median)
from emp join median on 1=1),
MAD (MAD) as

(select DISTINCT PERCENTILE_CONT(0.5) within group(order by
deviation) over()
from Deviation )

select abs(sal-median)/MAD, sal, ename, job
from MAD join emp on 1=1

```

PostgreSQL 和 DB2

总体结构与 SQL Server 解决方案相同，但函数 **PERCENTILE_CONT** 的语法不同，因为在 PostgreSQL 和 DB2 中，**PERCENTILE_CONT** 被视为聚合函数，而不是窗口函数。

```

with median (median)
as
(select percentile_cont(0.5) within group(order by sal)
from emp),

devtab (deviation)
      as
(select abs(sal-median)
from emp join median),

MedAbsDeviation (MAD) as
(select percentile_cont (0.5) within group(order by deviation)
from devtab)

select abs(sal-median)/MAD, sal, ename, job

```

```
FROM MedAbsDeviation join emp
```

Oracle

Oracle 中提供了计算中值的函数 **MEDIAN**，因此解决方案更简单，但为了处理标量值偏差，依然需要使用 CTE。

```
with
Deviation (Deviation)
  as
(select abs(sal-median(sal))
from emp),

MAD (MAD) as
(select median(Deviation)
from Deviation )

select abs(sal-median)/MAD, sal, ename, job
FROM MAD join emp
```

MySQL

7.10 节说过，MySQL 中没有提供函数 **MEDIAN** 或 **PERCENTILE_CONT**，这意味着为计算绝对中位差而找出每个中值时，需要在 CTE 中使用两个子查询。这导致 MySQL 解决方案的代码要长些。

```
with rank_tab (sal, rank_sal) as (
select sal, cume_dist() over (order by sal)
from emp),
inter as
(
select sal, rank_sal from rank_tab
where rank_sal>=0.5
union
select sal, rank_sal from rank_tab
where rank_sal<=0.5
)
,
```



```

medianSal (medianSal) as
(
select (max(sal)+min(sal))/2
from inter),
deviationSal (Sal,deviationSal) as
(select Sal,abs(sal-medianSal)
from emp join medianSal
on 1=1
)
,
distDevSal (sal,deviationSal,distDeviationSal) as
(
select sal,deviationSal,cume_dist() over (order by deviationSal)
from deviationSal
),
DevInter (DevInter, sal) as
(
select min(deviationSal), sal
from distDevSal
where distDeviationSal >= 0.5

union

select max(DeviationSal), sal
from distDevSal
where distDeviationSal <= 0.5
),
MAD (MedianAbsoluteDeviance) as
(
select abs(emp.sal-(min(devInter)+max(devInter))/2)
from emp join DevInter on 1=1
)

select emp.sal,MedianAbsoluteDeviance,
(emp.sal-deviationSal)/MedianAbsoluteDeviance
from (emp join MAD on 1=1)
      join deviationSal on emp.sal=deviationSal.sal

```

13. 讨论

每种解决方案都采用了类似的策略。首先，计算中值，然后计算这个中值与各个值的绝对偏差的中值，即绝对中位差。然后，使用查询来找出每个值相对于中值的偏差与绝对中位差的比值。接下来，就可以像使用标准偏差那样使用这些比值了。如果一个值相对于中值的偏差是绝对中位差的 3 倍以上，就可以认为它是异常值。

前面说过，相对于标准偏差，这种方法的优点是，即便数据不呈正态分布，它依然有效。例如，即便数据分布不平衡，绝对中位差给出的答案依然合理。

在我们的薪水数据中，有一个薪水值相对于中值的偏差超过了绝对中位差的 3 倍，它就是 CEO 的薪水。

CEO 的薪水比大多数员工的薪水高这么多，这是否公平呢？对此可谓见仁见智。但考虑到此异常薪水值是 CEO 的，这与我们对薪水数据的理解相符。在其他情况下，如果不能对如此异常的异常值做出明确解释，那么就有理由怀疑这个值是否正确，或者将其与其他值一起处理是否合理。（如果这个值是正确的，那么我们可能认为需要将数据分成多组进行分析。）



很多常见的统计数据（比如平均值和标准偏差）会假设数据分布形状为钟形曲线，即呈正态分布。这样的数据集很多，但不呈正态分布的数据集也不少。

检测数据集是否呈正态分布的方法很多，它们会将数据可视化并执行计算。统计包中通常包含执行这些检测的函数，但 SQL 中没有，而且使用 SQL 也难以实现这样

的函数。然而，通常还有其他的统计工具（非参数统计），它们不假设数据集呈特定分布，因此使用起来更安全。

7.17 使用本福特法则查找反常数据

1. 问题

上一节实例介绍的异常值是一种易于识别的反常数据，但有些存在疑问的数据并不那么容易识别。检测不像异常值那样显而易见的反常数据的一种方式查看数字位的出现频率，这种频率通常符合本福特法则。本福特法则最常用于检测数据造假——在数据集中人为地添加伪造的数字，也可用于检测不符合预期规律的数据。例如，可以使用它来检测诸如重复数据点等错误，这种错误不一定像异常值那样明显。

2. 解决方案

要使用本福特法则，需要先计算数字位的期望分布，然后将其与实际分布进行比较。在最复杂的本福特法则用法中，会考虑第一位、第二位以及数字位组合，但本例中只考虑前几位。

将本福特法则预测的频率与数据的实际频率进行比较。最终的结果集包含 4 列数据，分别是第一位的预测频率、第一位的实际频率、本福特法则预测的前几位的频率，以及前几位的实际频率。

```
with
FirstDigits (FirstDigit)
as
(select left(cast(SAL as CHAR),1) as FirstDigit
    from emp),

TotalCount (Total)
as
```

```

(select count(*)
 from emp),

ExpectedBenford (Digit,Expected)
as
  (select ID,(log10(ID + 1) - log10(ID)) as expected
   from t10
   where ID < 10)

select count(FirstDigit),Digit,
coalesce(count(*)/Total,0) as ActualProportion,Expected
From FirstDigits
  Join TotalCount
  Right Join ExpectedBenford
    on FirstDigits.FirstDigit=ExpectedBenford.Digit
group by Digit
order by Digit

```

13. 讨论

由于要使用两个计数（总行数和第一位为各种不同数字的行数），因此需要使用一个 CTE。严格地说，并不需要将本福特法则的预期结果放在 CTE 的独立查询中，本例之所以这样做旨在能够找出计数为 0 的位组合，并通过右连接将它们显示在表中。

也可以在主查询中生成 FirstDigits 计数，但我们没有为提高可读性而在 **GROUP BY** 子句中重复表达式 **LEFT(CAST(...))**。

本福特法则使用的数学公式很简单。

$$\text{期望频率} = \log_{10} \left(\frac{d+1}{d} \right)$$

可以使用透视表 **T10** 来生成合适的值，再计算实际频率（以便进行比较），这要求我们找出第一位。

在数据集相对较大且其中的值横跨多个数量级（比如 10、100、1000 等）时，本福特法则的效果最好。在这里，这些条件都不满足。然而，考虑到相对于期望频率的偏差，我们依然有理由怀疑这些值很可能是伪造的，所以有必要做进一步调查。

7.18 小结

由于企业的数据经常放在 SQL 支持的数据库中，因此使用 SQL 来尝试弄明白这些数据合情合理。SQL 不像专用包 SAS、统计编程语言 R 和 Python 统计库那样提供了完备的统计工具，但正如你在本章看到的，它确实提供了丰富的计算工具，可用于深入了解数据的统计特性。

第 8 章 日期算术运算

本章介绍执行简单日期算术运算的技巧，涵盖给某个日期加上若干天，找出两个日期之间的工作日数，以及计算两个日期相差多少天等常见任务。

如果能够使用 RDBMS 内置的函数操作日期，那么就能极大地提高效率。本章所有的实例都将竭尽所能地利用 RDBMS 内置的函数。另外，本章所有的实例都使用了相同的日期格式——DD-MON-YYYY。当然还有很多其他常用的日期格式，比如 DD-MM-YYYY 和 ISO 标准格式。

本章坚持使用日期格式 DD-MON-YYYY，旨在让正在使用某种 RDBMS 但想学习其他 RDBMS 的读者受益。明白一种标准格式后，你就无须为默认日期格式操心了，从而能够专注于各种技巧以及 RDBMS 提供的函数。



本章会聚焦于基本的日期算术运算，第 9 章会介绍进阶日期操作技巧。本章所使用的实例都是简单的日期数据类型。如果你使用的是更复杂的日期数据类型，则需要对解决方案做相应调整。

8.1 加上或减去若干天、若干月或若干年

1. 问题

你需要给某个日期加上或减去若干天、若干月或若干年。例如，你想根据员工 **CLARK** 的 **HIREDATE**，返回 6 个日期：**CLARK** 获聘前 5 天和后 5 天，**CLARK** 获聘前 5 个月和后 5 个月，**CLARK** 获聘前 5 年和后 5 年。**CLARK** 获聘于 2006 年 6 月 9 日，因此你要返回如下结果集。

HD_MINUS_5D	HD_PLUS_5D	HD_MINUS_5M	HD_PLUS_5M	HD_MINUS_5Y	HD_PLUS_5Y

04-JUN-2006	14-JUN-2006	09-JAN-2006	09-NOV-2006	09-JUN-2001	09-JUN-2001
12-NOV-2006	22-NOV-2006	17-JUN-2006	17-APR-2007	17-NOV-2001	17-NOV-2001
18-JAN-2007	28-JAN-2007	23-AUG-2006	23-JUN-2007	23-JAN-2002	23-JAN-2002

2. 解决方案

DB2

标准加法运算和减法运算可用于日期值，但在给某个日期加上或减去的值后面，必须指定其单位。

```
1 select hiredate -5 day    as hd_minus_5D,  
2        hiredate +5 day    as hd_plus_5D,  
3        hiredate -5 month as hd_minus_5M,  
4        hiredate +5 month as hd_plus_5M,  
5        hiredate -5 year   as hd_minus_5Y,  
6        hiredate +5 year   as hd_plus_5Y
```

```
7   from emp
8   where deptno = 10
```

Oracle

要加上或减去若干天，可以使用标准加法运算和减法运算；
要加上或减去若干月或若干年，可以使用函数
ADD_MONTHS。

```
1 select hiredate-5           as hd_minus_5D,
2        hiredate+5           as hd_plus_5D,
3        add_months(hiredate,-5) as hd_minus_5M,
4        add_months(hiredate,5)  as hd_plus_5M,
5        add_months(hiredate,-5*12) as hd_minus_5Y,
6        add_months(hiredate,5*12) as hd_plus_5Y
7   from emp
8   where deptno = 10
```

PostgreSQL

使用标准加法运算和减法运算以及关键字 **INTERVAL** 指定时间单位和数量。单位和数量必须放在单引号内。

```
1 select hiredate - interval '5 day'   as hd_minus_5D,
2        hiredate + interval '5 day'   as hd_plus_5D,
3        hiredate - interval '5 month' as hd_minus_5M,
4        hiredate + interval '5 month' as hd_plus_5M,
5        hiredate - interval '5 year'  as hd_minus_5Y,
6        hiredate + interval '5 year'  as hd_plus_5Y
7   from emp
8   where deptno=10
```

MySQL

使用标准加法运算和减法运算以及关键字 **INTERVAL** 指定时间单位和数量。与 PostgreSQL 解决方案不同，单位和数量无须放在单引号内。

```
1 select hiredate - interval 5 day    as hd_minus_5D,  
2        hiredate + interval 5 day    as hd_plus_5D,  
3        hiredate - interval 5 month as hd_minus_5M,  
4        hiredate + interval 5 month as hd_plus_5M,  
5        hiredate - interval 5 year  as hd_minus_5Y,  
6        hiredate + interval 5 year  as hd_plus_5Y  
7    from emp  
8    where deptno=10
```

也可以使用函数 **DATE_ADD**。

```
1 select date_add(hiredate,interval -5 day)    as hd_minus_5D,  
2        date_add(hiredate,interval  5 day)    as hd_plus_5D,  
3        date_add(hiredate,interval -5 month) as hd_minus_5M,  
4        date_add(hiredate,interval  5 month) as hd_plus_5M,  
5        date_add(hiredate,interval -5 year)  as hd_minus_5Y,  
6        date_add(hiredate,interval  5 year)  as hd_plus_5DY  
7    from emp  
8    where deptno=10
```

SQL Server

使用函数 **DATEADD** 给指定日期加上或减去指定单位和数量的时间。

```
1 select dateadd(day,-5,hiredate)    as hd_minus_5D,  
2        dateadd(day,5,hiredate)     as hd_plus_5D,  
3        dateadd(month,-5,hiredate)  as hd_minus_5M,  
4        dateadd(month,5,hiredate)   as hd_plus_5M,  
5        dateadd(year,-5,hiredate)   as hd_minus_5Y,  
6        dateadd(year,5,hiredate)    as hd_plus_5Y  
7    from emp  
8    where deptno = 10
```

13. 讨论

Oracle 解决方案基于这样一个事实：执行日期算术运算时，

整数的默认单位是天。然而，仅在 **DATE** 类型的算术运算中才如此。Oracle 还提供了 **TIMESTAMP** 类型，对这些类型执行算术运算时，应采用 PostgreSQL **INTERVAL** 解决方案。将 **TIMESTAMP** 传递给旧式日期函数（如 **ADD_MONTHS**）时要小心，因为这样做将丢失 **TIMESTAMP** 值中可能包含的短于 1 秒的部分。

ISO SQL 语法使用了 **INTERVAL** 关键字和字符串字面量，这个标准要求将间隔值放在单引号内。PostgreSQL（以及 Oracle9i 及更高版本）遵循了这个标准。MySQL 稍微偏离了这个标准，不要求将间隔值放在引号内。

8.2 确定两个日期相差多少天

1. 问题

你想找出两个日期相隔多长时间，并在表示结果时以天为单位。例如，你想找出员工 ALLEN 和 WARD 的 HIREDATE 之间有多少天。

2. 解决方案

DB2

使用两个内嵌视图找出 ALLEN 和 WARD 的 HIREDATE，再使用函数 **DAYS** 将这两个日期相减。

```
1 select days(ward_hd) - days(allen_hd)
2   from (
3 select hiredate as ward_hd
4   from emp
5  where ename = 'WARD'
6       ) x,
7       (
8 select hiredate as allen_hd
9   from emp
10  where ename = 'ALLEN'
11       ) y
```

Oracle 和 PostgreSQL

使用两个内嵌视图找出 ALLEN 和 WARD 的 HIREDATE，再将这两个日期相减。

```
1 select ward_hd - allen_hd
```

```
2  from (
3 select hiredate as ward_hd
4  from emp
5  where ename = 'WARD'
6       ) x,
7  (
8 select hiredate as allen_hd
9  from emp
10 where ename = 'ALLEN'
11       ) y
```

MySQL 和 SQL Server

使用函数 **DATEDIFF** 找出两个日期相差多少天。在 MySQL 中，函数 **DATEDIFF** 接受两个参数（两个日期，你要找出它们相差多少天），并要求将较早的那个日期作为第一个参数（SQL Server 中与此相反）。在 SQL Server 中，函数 **DATEDIFF** 允许你指定要返回的值的单位（在本实例中，你希望返回相差的天数）。下面是 SQL Server 的解决方案。

```
1 select datediff(day,allen_hd,ward_hd)
2  from (
3 select hiredate as ward_hd
4  from emp
5  where ename = 'WARD'
6       ) x,
7  (
8 select hiredate as allen_hd
9  from emp
10 where ename = 'ALLEN'
11       ) y
```

如果你使用的是 MySQL，那么只需将第一个参数删除，并对调 **ALLEN_HD** 和 **WARD_HD** 的顺序。

13. 讨论

在所有解决方案中，内嵌视图 X 和内嵌视图 Y 会分别返回员工 WARD 和 ALLEN 的 HIREDATE。

```
select ward_hd, allen_hd
  from (
select hiredate as ward_hd
  from emp
 where ename = 'WARD'
    ) y,
  (
select hiredate as allen_hd
  from emp
 where ename = 'ALLEN'
    ) x
```

WARD_HD	ALLEN_HD
22-FEB-2006	20-FEB-2006

由于没有显式地连接内嵌视图 X 和内嵌视图 Y，因此将创建笛卡儿积。在这里，未进行显式连接没有任何害处，因为内嵌视图 X 和内嵌视图 Y 的基数都是 1，所以最终的结果集只有 1 行（这一点显而易见，因为 $1 \times 1 = 1$ ）。

要获取相差的天数，只需根据 DBMS，使用合适的方法将两个返回的值相减。

8.3 确定两个日期之间有多少个工作日

1. 问题

给定两个日期，你想找出它们之间有多少个工作日（包括这两个日期本身）。如果 1 月 10 日是星期一，1 月 11 日是星期二，那么这两个日期之间有两个工作日，因为这两个日期都是工作日。在本实例中，只要不是星期六或星期日，就认为是工作日。

2. 解决方案

本解决方案旨在找出 **BLAKE** 和 **JONES** 的 **HIREDATE** 之间有多少个工作日。为了确定两个日期之间的工作日数量，可以使用一张透视表。在这张表中，两个日期之间的每一天（包含这两个日期本身）都有对应的一行。这样，要计算工作日数，只需计算在返回的日期中，有多少个日期不是星期六或星期日。



如果要剔除节假日，那么可以创建一个 **HOLIDAYS** 表，并在解决方案中使用谓词 **NOT IN** 将 **HOLIDAYS** 表中的日期排除在外。

DB2

使用透视表 **T500** 生成足够的行数，用于表示两个日期之间的每一天。再计算不是周末的天数。使用函数 **DAYNAME** 来确定各个日期是星期几。

```
1 select sum(case when dayname(jones_hd+t500.id day -1 day)
```



```

2             in ( 'Saturday','Sunday' )
3             then 0 else 1
4         end) as days
5     from (
6 select max(case when ename = 'BLAKE'
7             then hiredate
8             end) as blake_hd,
9         max(case when ename = 'JONES'
10            then hiredate
11            end) as jones_hd
12     from emp
13    where ename in ( 'BLAKE','JONES' )
14           ) x,
15         t500
16    where t500.id <= blake_hd-jones_hd+1

```

MySQL

使用透视表 **T500** 生成足够的行数，用于表示两个日期之间的每一天。再计算不是周末的天数。使用函数 **DATE_ADD** 给起始日期加上若干天，以获得各个中间日期。使用函数 **DATE_FORMAT** 确定各个日期是星期几。

```

1 select sum(case when date_format(
2             date_add(jones_hd,
3             interval t500.id-1 DAY), '%a')
4             in ( 'Sat','Sun' )
5             then 0 else 1
6         end) as days
7     from (
8 select max(case when ename = 'BLAKE'
9             then hiredate
10            end) as blake_hd,
11         max(case when ename = 'JONES'
12            then hiredate
13            end) as jones_hd
14     from emp
15    where ename in ( 'BLAKE','JONES' )
16           ) x,
17         t500
18    where t500.id <= datediff(blake_hd,jones_hd)+1

```

Oracle

使用透视表 **T500** 生成足够的行数，用于表示两个日期之间的每一天。再计算不是周末的天数。使用函数 **TO_CHAR** 确定各个日期是星期几。

```
1 select sum(case when to_char(jones_hd+t500.id-1,'DY')
2                   in ( 'SAT','SUN' )
3                   then 0 else 1
4                   end) as days
5   from (
6 select max(case when ename = 'BLAKE'
7               then hiredate
8               end) as blake_hd,
9        max(case when ename = 'JONES'
10              then hiredate
11              end) as jones_hd
12   from emp
13  where ename in ( 'BLAKE','JONES' )
14         ) x,
15        t500
16  where t500.id <= blake_hd-jones_hd+1
```

PostgreSQL

使用透视表 **T500** 生成足够的行数，用于表示两个日期之间的每一天。再计算不是周末的天数。使用函数 **TO_CHAR** 确定各个日期是星期几。

```
1 select sum(case when trim(to_char(jones_hd+t500.id-1,'DAY'))
2                   in ( 'SATURDAY','SUNDAY' )
3                   then 0 else 1
4                   end) as days
5   from (
6 select max(case when ename = 'BLAKE'
7               then hiredate
8               end) as blake_hd,
9        max(case when ename = 'JONES'
10              then hiredate
11              end) as jones_hd
```

```

12  from emp
13  where ename in ( 'BLAKE','JONES' )
14         ) x,
15         t500
16  where t500.id <= blake_hd-jones_hd+1

```

SQL Server

使用透视表 **T500** 生成足够的行数，用于表示两个日期之间的每一天。再计算不是周末的天数。使用函数 **DATENAME** 确定各个日期是星期几。

```

1 select sum(case when datename(dw,jones_hd+t500.id-1)
2                 in ( 'SATURDAY','SUNDAY' )
3                 then 0 else 1
4                 end) as days
5  from (
6 selectmax(case when ename = 'BLAKE'
7               then hiredate
8               end) as blake_hd,
9         max(case when ename = 'JONES'
10              then hiredate
11              end) as jones_hd
12  from emp
13  where ename in ( 'BLAKE','JONES' )
14         ) x,
15         t500
16  where t500.id <= datediff(day,jones_hd-blake_hd)+1

```

13. 讨论

确定特定的日期是星期几时，需要使用的内置函数随 **RDBMS** 而异，但在所有的 **RDBMS** 中，解决方案采用的方法都相同。整个解决方案可分为两步。

- a. 返回两个日期之间的所有日期（包括这两个日期本身）。

b. 计算不是周末的天数（行数）。

内嵌视图 **X** 会执行第 1 步。如果仔细研究内嵌视图 **X**，你将发现它使用了聚合函数 **MAX**，这里使用它旨在将 **NULL** 删除。如果还不明白使用 **MAX** 的作用，下面的输出或许可以帮助你理解。下面显示了未使用 **MAX** 时内嵌视图 **X** 返回的结果。

```
select case when ename = 'BLAKE'
           then hiredate
        end as blake_hd,
       case when ename = 'JONES'
           then hiredate
        end as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )
```

BLAKE_HD	JONES_HD
01-MAY-2006	02-APR-2006

未使用 **MAX** 时，返回了两行。使用 **MAX** 可以将 **NULL** 删除，从而只返回一行（而不是两行）。

```
select max(case when ename = 'BLAKE'
                 then hiredate
            end) as blake_hd,
       max(case when ename = 'JONES'
                 then hiredate
            end) as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )
```

BLAKE_HD	JONES_HD
01-MAY-2006	02-APR-2006

在这里，两个日期之间的天数（包括这两个日期本身）为

30。两个日期位于一行中后，下一步是对于这 30 天中的每一天，都生成一行数据。为返回 30 天（行），使用了透视表 T500。在透视表 T500 中，每个 ID 值都比前一个大 1，因此如果将 T500 返回的每个 ID 值都与 JONES_HD 相加，则可以生成从 JONES_HD 到 BLAKE_HD（包括 BLAKE_HD）的每个日期。结果如下所示（使用的是 Oracle 语法）。

```
select x.*, t500.*, jones_hd+t500.id-1
  from (
select max(case when ename = 'BLAKE'
                then hiredate
              end) as blake_hd,
       max(case when ename = 'JONES'
                then hiredate
              end) as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )
        ) x,
    t500
 where t500.id <= blake_hd-jones_hd+1
```

BLAKE_HD	JONES_HD	ID	JONES_HD+T5
01-MAY-2006	02-APR-2006	1	02-APR-2006
01-MAY-2006	02-APR-2006	2	03-APR-2006
01-MAY-2006	02-APR-2006	3	04-APR-2006
01-MAY-2006	02-APR-2006	4	05-APR-2006
01-MAY-2006	02-APR-2006	5	06-APR-2006
01-MAY-2006	02-APR-2006	6	07-APR-2006
01-MAY-2006	02-APR-2006	7	08-APR-2006
01-MAY-2006	02-APR-2006	8	09-APR-2006
01-MAY-2006	02-APR-2006	9	10-APR-2006
01-MAY-2006	02-APR-2006	10	11-APR-2006
01-MAY-2006	02-APR-2006	11	12-APR-2006
01-MAY-2006	02-APR-2006	12	13-APR-2006
01-MAY-2006	02-APR-2006	13	14-APR-2006
01-MAY-2006	02-APR-2006	14	15-APR-2006
01-MAY-2006	02-APR-2006	15	16-APR-2006
01-MAY-2006	02-APR-2006	16	17-APR-2006
01-MAY-2006	02-APR-2006	17	18-APR-2006
01-MAY-2006	02-APR-2006	18	19-APR-2006
01-MAY-2006	02-APR-2006	19	20-APR-2006

01-MAY-2006	02-APR-2006	20	21-APR-2006
01-MAY-2006	02-APR-2006	21	22-APR-2006
01-MAY-2006	02-APR-2006	22	23-APR-2006
01-MAY-2006	02-APR-2006	23	24-APR-2006
01-MAY-2006	02-APR-2006	24	25-APR-2006
01-MAY-2006	02-APR-2006	25	26-APR-2006
01-MAY-2006	02-APR-2006	26	27-APR-2006
01-MAY-2006	02-APR-2006	27	28-APR-2006
01-MAY-2006	02-APR-2006	28	29-APR-2006
01-MAY-2006	02-APR-2006	29	30-APR-2006
01-MAY-2006	02-APR-2006	30	01-MAY-2006

如果查看 **WHERE** 子句，你将发现为生成所需的 30 行数据，给 **BLAKE_HD** 和 **JONES_HD** 的差加上了 1，因为如果不这样做，将只生成 29 行数据。另外，在外部查询的 **SELECT** 列表中，从 **T500.ID** 中减去了 1。这是因为 **ID** 值从 1 开始，给 **JONES_HD** 加 1 将导致 **JONES_HD** 在计数期间被排除在外。

生成所需的行数后，使用 **CASE** 表达式标出返回的各个日期是工作日还是周末。（如果是工作日，就返回 1；如果是周末，就返回 0。）最后一步是使用聚合函数 **SUM** 将所有的 1 相加，得到最终的答案。

8.4 确定两个日期相隔多少个月或多少年

1. 问题

你想确定两个日期相隔多少个月或多少年。例如，你想找出第一位员工和最后一位员工获聘的日期相隔多少个月，还想确定这段时间相当于多少年。

2. 解决方案

由于一年是 12 个月，因此找出两个日期相隔多少个月，只要再除以 12 就可以得到相隔多少年。熟悉本解决方案后，你可能想根据需要对相差的年数进行舍入处理。例如，在 **EMP** 表中，第一个 **HIREDATE** 为 1980 年 12 月 17 日，最后一个 **HIREDATE** 为 1983 年 1 月 12 日。如果只考虑年份，那么结果将为 3（1983 - 1980）年，但相差的月数大约为 25（两年多一点儿）。应该根据需要调整解决方案。下面的解决方案都返回 25 个月和大约两年。

DB2 和 MySQL

使用函数 **YEAR** 和 **MONTH** 返回指定日期的 4 位年份和 2 位月份。

```
1 select mnth, mnth/12
2   from (
3 select (year(max_hd) - year(min_hd))*12 +
4        (month(max_hd) - month(min_hd)) as mnth
5   from (
6 select min(hiredate) as min_hd, max(hiredate) as max_hd
7   from emp
8        ) x
```

Oracle

使用函数 **MONTHS_BETWEEN** 确定两个日期相隔多少个月（要确定相隔多少年，只需再除以 12）。

```
1 select months_between(max_hd,min_hd),
2         months_between(max_hd,min_hd)/12
3   from (
4 select min(hiredate) min_hd, max(hiredate) max_hd
5   from emp
6        ) x
```

PostgreSQL

使用函数 **EXTRACT** 返回指定日期的 4 位年份和 2 位月份。

```
1 select mnth, mnth/12
2   from (
3 select ( extract(year from max_hd)
4         extract(year from min_hd) ) * 12
5        +
6        ( extract(month from max_hd)
7          extract(month from min_hd) ) as mnth
8   from (
9 select min(hiredate) as min_hd, max(hiredate) as max_hd
10  from emp
11      ) x
12      ) y
```

SQL Server

使用函数 **DATEDIFF** 确定两个日期相隔多长时间，并使用参数 **datepart** 将时间单位指定为月或年。

```
1 select datediff(month,min_hd,max_hd),
2        datediff(year,min_hd,max_hd)
```



```

3  from (
4  select min(hiredate) min_hd, max(hiredate) max_hd
5  from emp
6         ) x

```

13. 讨论

DB2、MySQL 和 PostgreSQL

在 PostgreSQL 解决方案中，提取 MIN_HD 和 MAX_HD 的年份和月份后，计算 MIN_HD 和 MAX_HD 相隔的月数和年数的方法与 DB2 和 MySQL 的解决方案相同。这里的讨论适用于全部 3 种解决方案。

内嵌视图 X 会返回 EMP 表中最早的 HIREDATE 和最晚的 HIREDATE。

```

select min(hiredate) as min_hd,
       max(hiredate) as max_hd
  from emp

```

MIN_HD	MAX_HD
17-DEC-1980	12-JAN-1983

为了确定 MAX_HD 和 MIN_HD 相隔多少个月，将 MIN_HD 和 MAX_HD 的年份差乘以 12，再加上 MAX_HD 和 MIN_HD 的月份差。如果不明白其中的原理，请返回各个日期的组成部分。表示年份和月份的数字值如下。

```

select year(max_hd) as max_yr, year(min_hd) as min_yr,
       month(max_hd) as max_mon, month(min_hd) as min_mon
  from (
select min(hiredate) as min_hd, max(hiredate) as max_hd
  from emp

```

) x			
MAX_YR	MIN_YR	MAX_MON	MIN_MON

1983	1980	1	12

根据上述结果，要确定 **MAX_HD** 和 **MIN_HD** 相隔多少个月，只需使用公式 $(1983 - 1980) \times 12 + (1 - 12)$ 。要确定 **MIN_HD** 和 **MAX_HD** 相隔多少年，只需将前面计算得到的月数除以 12。同样，你可能想对这些值进行舍入处理，这取决于你要获得什么样的结果。

Oracle 和 SQL Server

内嵌视图 **X** 会返回 **EMP** 表中最早的 **HIREDATE** 和最晚的 **HIREDATE**。

select min(hiredate) as min_hd, max(hiredate) as max_hd from emp	
MIN_HD	MAX_HD

17-DEC-1980	12-JAN-1983

在 Oracle 和 SQL Server 中，可以使用内置函数 **MONTHS_BETWEEN** 或 **DATEDIFF** 来计算两个日期相隔多少个月。要确定相隔多少年，只需将相隔的月数除以 12。

8.5 确定两个日期相隔多少秒、多少分钟或多少小时

1. 问题

你想返回两个日期相隔的秒数。例如，你想返回 ALLEN 和 WARD 的 HIREDATE 相隔的秒数、分钟数和小时数。

2. 解决方案

只要能确定两个日期相隔多少天，就能确定它们相隔多少秒、多少分钟和多少小时，因为它们只是不同的时间单位而异。

DB2

使用函数 **DAYS** 确定 ALLEN_HD 和 WARD_HD 相隔多少天，再使用乘法运算计算小时数、分钟数和秒数。

```
1 select dy*24 hr, dy*24*60 min, dy*24*60*60 sec
2   from (
3 select ( days(max(case when ename = 'WARD'
4                  then hiredate
5                  end)) -
6          days(max(case when ename = 'ALLEN'
7                  then hiredate
8                  end))
9         ) as dy
10  from emp
11        ) x
```

MySQL

使用函数 **DATEDIFF** 返回 **ALLEN_HD** 和 **WARD_HD** 相隔多少天，再使用乘法运算计算小时数、分钟数和秒数。

```
1 select datediff(day,allen_hd,ward_hd)*24 hr,  
2        datediff(day,allen_hd,ward_hd)*24*60 min,  
3        datediff(day,allen_hd,ward_hd)*24*60*60 sec  
4   from (  
5 select max(case when ename = 'WARD'  
6           then hiredate  
7           end) as ward_hd,  
8        max(case when ename = 'ALLEN'  
9           then hiredate  
10          end) as allen_hd  
11   from emp  
12  ) x
```

SQL Server

使用函数 **DATEDIFF** 返回 **ALLEN_HD** 和 **WARD_HD** 相隔多少天，再使用参数 **DATEPART** 指定时间单位。

```
1 select datediff(day,allen_hd,ward_hd,hour) as hr,  
2        datediff(day,allen_hd,ward_hd,minute) as min,  
3        datediff(day,allen_hd,ward_hd,second) as sec  
4   from (  
5 select max(case when ename = 'WARD'  
6           then hiredate  
7           end) as ward_hd,  
8        max(case when ename = 'ALLEN'  
9           then hiredate  
10          end) as allen_hd  
11   from emp  
12  ) x
```

Oracle 和 PostgreSQL

使用减法运算返回 **ALLEN_HD** 和 **WARD_HD** 相隔多少天，再使用乘法运算计算小时数、分钟数和秒数。

```

1 select dy*24 as hr, dy*24*60 as min, dy*24*60*60 as sec
2   from (
3 select (max(case when ename = 'WARD'
4             then hiredate
5             end) -
6         max(case when ename = 'ALLEN'
7             then hiredate
8             end)) as dy
9   from emp
10  ) x

```

13. 讨论

在所有的解决方案中，内嵌视图 **X** 都返回 **WARD** 和 **ALLEN** 的 **HIREDATE**。

```

select max(case when ename = 'WARD'
                then hiredate
                end) as ward_hd,
       max(case when ename = 'ALLEN'
                then hiredate
                end) as allen_hd
  from emp

```

WARD_HD	ALLEN_HD
22-FEB-2006	20-FEB-2006

然后，将 **WARD_HD** 和 **ALLEN_HD** 相隔的天数，分别乘以 24（每天的小时数）、1440（每天的分钟数）和 86400（每天的秒数）。

8.6 计算一年中有多少个工作日

1. 问题

你想计算一年中有多少个工作日（同 8.3 节，在本实例中，只要不是星期六或星期日，就认为是工作日）。

2. 解决方案

为了确定一年中有多少个工作日，必须做到以下几点。

- a. 生成这一年内所有的日期。
- b. 设置这些日期的格式，确定它们是星期几。
- c. 计算有多少个工作日。

DB2

使用递归式 **WITH**，以免使用 **SELECT** 来查询一张至少包含 366 行的表。使用函数 **DAYNAME** 确定各个日期是星期几，然后计算每个工作日出现的次数。

```
1 with x (start_date,end_date)
2 as (
3 select start_date,
4        start_date + 1 year end_date
5   from (
6 select (current_date
7        dayofyear(current_date) day)
8        +1 day as start_date
9   from t1
10        ) tmp
11  union all
12 select start_date + 1 day, end_date
13   from x
```

```
14 where start_date + 1 day < end_date
15 )
16 select dayname(start_date),count(*)
17   from x
18  group by dayname(start_date)
```

MySQL

查询 **T500** 表以生成足够的行，从而返回一年中的每一天。使用函数 **DATE_FORMAT** 确定各个日期都是星期几，然后计算工作日出现的次数。

```
1 select date_format(
2         date_add(
3           cast(
4             concat(year(current_date),'-01-01')
5               as date),
6           interval t500.id-1 day),
7         '%W') day,
8       count(*)
9   from t500
10  where t500.id <= datediff(
11         cast(
12           concat(year(current_date)+1,'-01-01')
13             as date),
14         cast(
15           concat(year(current_date),'-01-01')
16             as date))
17 group by date_format(
18         date_add(
19           cast(
20             concat(year(current_date),'-01-01')
21               as date),
22           interval t500.id-1 day),
23         '%W')
```

Oracle

可以使用递归式 **CONNECT BY** 返回一年中的每一天。

```

1 with x as (
2   select level lvl
3   from dual
4   connect by level <= (
5     add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
6   )
7 )
8 select to_char(trunc(sysdate,'y')+lvl-1,'DAY'), count(*)
9   from x
10  group by to_char(trunc(sysdate,'y')+lvl-1,'DAY')

```

PostgreSQL

首先使用内置函数 **GENERATE_SERIES** 为一年中的每一天都生成一行数据，然后使用函数 **TO_CHAR** 确定各个日期都是星期几，最后计算工作日出现的次数。

```

1 select to_char(
2   cast(
3     date_trunc('year',current_date)
4     as date) + gs.id-1,'DAY'),
5   count(*)
6   from generate_series(1,366) gs(id)
7  where gs.id <= (cast
8    ( date_trunc('year',current_date) +
9      interval '12 month' as date) -
10  cast(date_trunc('year',current_date)
11    as date))
12  group by to_char(
13    cast(
14      date_trunc('year',current_date)
15      as date) + gs.id-1,'DAY')

```

SQL Server

使用递归式 **WITH**，以免使用 **SELECT** 来查询一个至少包含 366 行的表。使用函数 **DATENAME** 确定各个日期都是星期几，然后计算工作日出现的次数。


```

1 with x (start_date,end_date)
2 as (
3 select start_date,
4         dateadd(year,1,start_date) end_date
5   from (
6 select cast(
7         cast(year(getdate()) as varchar) + '-01-01'
8         as datetime) start_date
9   from t1
10        ) tmp
11 union all
12 select dateadd(day,1,start_date), end_date
13   from x
14  where dateadd(day,1,start_date) < end_date
15 )
16 select datename(dw,start_date),count(*)
17   from x
18  group by datename(dw,start_date)
19 OPTION (MAXRECURSION 366)

```

13. 讨论

DB2

首先，在递归式 **WITH** 视图 **X** 的内嵌视图 **TMP** 中返回当前年份的第一天。

```

select (current_date
        dayofyear(current_date) day)
        +1 day as start_date
  from t1

START_DATE
-----
01-JAN-2005

```

然后，给 **START_DATE** 加上 1 年，以便同时有起始日期和终止日期。为什么需要同时知道这两个日期呢？这是因为你要

生成一年中的每一天。START_DATE 和 END_DATE 如下所示。

```
select start_date,
       start_date + 1 year end_date
  from (
select (current_date
       dayofyear(current_date) day)
      +1 day as start_date
  from t1
     ) tmp

      START_DATE  END_DATE
      -----
01-JAN-2005 01-JAN-2006
```

接下来，递归地给 START_DATE 加上 1 天，直到它等于 END_DATE。下面显示了递归视图 X 返回的部分行。

```
with x (start_date,end_date)
as (
select start_date,
       start_date + 1 year end_date
  from (
select (current_date -
       dayofyear(current_date) day)
      +1 day as start_date
  from t1
     ) tmp
 union all
select start_date + 1 day, end_date
  from x
 where start_date + 1 day < end_date
 )
select * from x
```

```
START_DATE  END_DATE
-----
01-JAN-2005 01-JAN-2006
02-JAN-2005 01-JAN-2006
03-JAN-2005 01-JAN-2006
...
```

```

29-JAN-2005 01-JAN-2006
30-JAN-2005 01-JAN-2006
31-JAN-2005 01-JAN-2006
...
01-DEC-2005 01-JAN-2006
02-DEC-2005 01-JAN-2006
03-DEC-2005 01-JAN-2006
...
29-DEC-2005 01-JAN-2006
30-DEC-2005 01-JAN-2006
31-DEC-2005 01-JAN-2006

```

最后，在递归视图 **x** 返回的行上使用函数 **DAYNAME**，并计算每个工作日出现的次数。最终结果如下所示。

```

with x (start_date,end_date)
as (
select start_date,
       start_date + 1 year end_date
  from (
select (
          current_date -
          dayofyear(current_date) day)
        +1 day as start_date
  from t1
    ) tmp
 union all
select start_date + 1 day, end_date
  from x
 where start_date + 1 day < end_date
 )
select dayname(start_date),count(*)
  from x
 group by dayname(start_date)

```

START_DATE	COUNT(*)
-----	-----
FRIDAY	52
MONDAY	52
SATURDAY	53
SUNDAY	52
THURSDAY	52
TUESDAY	52

MySQL

该解决方案以查询 **T500** 表的形式，为一年中的每一天都生成一行数据。第 4 行的命令会返回当前年份的第一天，这是通过返回函数 **CURRENT_DATE** 生成的日期的年份部分，再加上月和日（采用 MySQL 的默认日期格式）实现的。结果如下所示。

```
select concat(year(current_date), '-01-01')
  from t1

START_DATE
-----
01-JAN-2005
```

有了当前年份的第一天后，使用函数 **DATEADD** 给第一天加上每一个 **T500.ID** 值，生成该年的每一天，再使用函数 **DATE_FORMAT** 确定各个日期都是星期几。为了根据表 **T500** 生成必要的行数，需要确定当前年份的第一天和下一年的第一天相隔多少天，并返回相应的行数（365 行或 366 行）。下面显示了返回的部分结果。

```
select date_format(
    date_add(
        cast(
            concat(year(current_date), '-01-01')
              as date),
        interval t500.id-1 day),
    '%W') day

  from t500
 where t500.id <= datediff(
    cast(
        concat(year(current_date)+1, '-01-01')
          as date),
    cast(
        concat(year(current_date), '-01-01')
```

```

as date))

DAY
-----
01-JAN-2005
02-JAN-2005
03-JAN-2005
...
29-JAN-2005
30-JAN-2005
31-JAN-2005
...
01-DEC-2005
02-DEC-2005
03-DEC-2005
...
29-DEC-2005
30-DEC-2005
31-DEC-2005

```

返回当前年份的每一天后，使用函数 **DAYNAME** 确定各个日期都是星期几。最终的结果如下所示。

```

select date_format(
    date_add(
        cast(
            concat(year(current_date), '-01-01')
            as date),
        interval t500.id-1 day),
    '%W') day,
    count(*)
from t500
where t500.id <= datediff(
    cast(
        concat(year(current_date)+1, '-01-01')
        as date),
    cast(
        concat(year(current_date), '-01-01')
        as date))
group by date_format(
    date_add(
        cast(
            concat(year(current_date), '-01-01')

```

	as date), interval t500.id-1 day), '%W')
DAY	COUNT(*)
-----	-----
FRIDAY	52
MONDAY	52
SATURDAY	53
SUNDAY	52
THURSDAY	52
TUESDAY	52
WEDNESDAY	52

Oracle

在 Oracle 中，要为当前年份的每一天都生成一行，可以查询透视表 **T500**，也可以使用递归式 **CONNECT BY** 和 **WITH**。对函数 **TRUNC** 的调用可以截断当前日期，得到当前年份的第一天。

如果你使用的是 **CONNECT BY/WITH** 解决方案，则可以使用伪列 **LEVEL** 来生成从 1 开始的序列号。为了生成必要的行数，可以根据当前年份的第一天和下一年第一天相隔的天数（365 天或 366 天）对 **ROWNUM** 或 **LEVEL** 进行过滤。接下来，给当前年份的第一天加上 **ROWNUM** 或 **LEVEL** 来生成每一天。

```
/* Oracle 9i及更高版本 */
with x as (
select level lvl
  from dual
 connect by level <= (
    add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
  )
)
select trunc(sysdate,'y')+lvl-1   from x
```

如果你使用的是透视表解决方案，则可以使用任何表或视图，只要它至少包含 366 行。由于 Oracle 提供了 **ROWNUM**，因此不要求透视表中的值从 1 开始递增。请看下面的示例，它使用透视表 **T500** 来返回当前年份的每一天。

```
/* Oracle 8i及更低版本 */
select trunc(sysdate,'y')+rownum-1 start_date
  from t500
 where rownum <= (add_months(trunc(sysdate,'y'),12)
                  - trunc(sysdate,'y'))
```

```
START_DATE
-----
01-JAN-2005
02-JAN-2005
03-JAN-2005
...
29-JAN-2005
30-JAN-2005
31-JAN-2005
...
01-DEC-2005
02-DEC-2005
03-DEC-2005
...
29-DEC-2005
30-DEC-2005
31-DEC-2005
```

无论采用哪种方法，最终都必须使用函数 **TO_CHAR** 来确定各个日期都是星期几，然后计算每个工作日出现的次数。最终的结果如下所示。

```
/* Oracle 9i及更高版本 */
with x as (
select level lvl
  from dual
 connect by level <= (
    add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
  )
)
```

```

)
select to_char(trunc(sysdate,'y')+1vl-1,'DAY'), count(*)
  from x
 group by to_char(trunc(sysdate,'y')+1vl-1,'DAY')

/* Oracle 8i及更低版本 */
select to_char(trunc(sysdate,'y')+rownum-1,'DAY') start_date,
       count(*)
  from t500
 where rownum <= (add_months(trunc(sysdate,'y'),12)
                  - trunc(sysdate,'y'))
 group by to_char(trunc(sysdate,'y')+rownum-1,'DAY')

```

START_DATE	COUNT(*)
-----	-----
FRIDAY	52
MONDAY	52
SATURDAY	53
SUNDAY	52
THURSDAY	52
TUESDAY	52
WEDNESDAY	52

PostgreSQL

首先，使用函数 **DATE_TRUNC** 返回当前日期的年份（查询 **T1** 表旨在只返回一行）。

```

select cast(
       date_trunc('year',current_date)
       as date) as start_date
  from t1

START_DATE
-----
01-JAN-2005

```

然后，查询行源（可以是任何表表达式，但至少包含 366 行）。该解决方案将函数 **GENERATE_SERIES** 用作行源，当然也可以使用 **T500** 表。接下来，不断在当前年份第一天的

基础上加上 1 天，直到返回该年份的每一天。

```
select cast( date_trunc('year',current_date)
            as date) + gs.id-1 as start_date
  from generate_series (1,366) gs(id)
 where gs.id <= (cast
                  ( date_trunc('year',current_date) +
                    interval '12 month' as date) -
            cast(date_trunc('year',current_date)
                as date))
```

```
START_DATE
-----
01-JAN-2005
02-JAN-2005
03-JAN-2005
...
29-JAN-2005
30-JAN-2005
31-JAN-2005
...
01-DEC-2005
02-DEC-2005
03-DEC-2005
...
29-DEC-2005
30-DEC-2005
31-DEC-2005
```

最后，使用函数 **TO_CHAR** 确定各个日期都是星期几，然后计算每个工作日出现的次数。最终结果如下所示。

```
select to_char(
        cast(
            date_trunc('year',current_date)
            as date) + gs.id-1,'DAY') as start_dates,
       count(*)
  from generate_series(1,366) gs(id)
 where gs.id <= (cast
                  ( date_trunc('year',current_date) +
                    interval '12 month' as date) -
            cast(date_trunc('year',current_date)
                as date))
```

```

                                as date))
group by to_char(
            cast(
                date_trunc('year',current_date)
                as date) + gs.id-1,'DAY')

START_DATE    COUNT(*)
-----
FRIDAY        52
MONDAY        52
SATURDAY      53
SUNDAY        52
THURSDAY      52
TUESDAY       52
WEDNESDAY     52

```

SQL Server

首先，递归式 **WITH** 视图 **X** 中的内嵌视图 **TMP** 会返回当前年份的第一天。

```

select cast(
            cast(year(getdate()) as varchar) + '-01-01'
            as datetime) start_date
from t1

START_DATE
-----
01-JAN-2005

```

返回当前年份的第一天（**START_DATE**）后，给它加上 1 年，以便同时拥有起始日期和终止日期。之所以需要同时知道起始日期和终止日期，是因为你要生成一年中的每一天。

START_DATE 和 **END_DATE** 如下所示。

```

select start_date,
       dateadd(year,1,start_date) end_date
from (
select cast(

```

```

        cast(year(getdate()) as varchar) + '-01-01'
            as datetime) start_date
    from t1
    ) tmp

START_DATE  END_DATE
-----
01-JAN-2005 01-JAN-2006

```

然后，递归地给 **START_DATE** 加上 1 天，直到它等于 **END_DATE**。下面显示了递归视图 **X** 返回的部分行。

```

with x (start_date,end_date)
as (
    select start_date,
           dateadd(year,1,start_date) end_date
    from (
    select cast(
           cast(year(getdate()) as varchar) + '-01-01'
           as datetime) start_date
    from t1
    ) tmp
union all
select dateadd(day,1,start_date), end_date
    from x
    where dateadd(day,1,start_date) < end_date
)
select * from x
OPTION (MAXRECURSION 366)

START_DATE  END_DATE
-----
01-JAN-2005 01-JAN-2006
02-JAN-2005 01-JAN-2006
03-JAN-2005 01-JAN-2006
...
29-JAN-2005 01-JAN-2006
30-JAN-2005 01-JAN-2006
31-JAN-2005 01-JAN-2006
...
01-DEC-2005 01-JAN-2006
02-DEC-2005 01-JAN-2006
03-DEC-2005 01-JAN-2006

```

```
...
29-DEC-2005 01-JAN-2006
30-DEC-2005 01-JAN-2006
31-DEC-2005 01-JAN-2006
```

最后，在递归视图 **X** 返回的行上使用函数 **DAYNAME**，并计算每个工作日出现的次数。最终结果如下所示。

```
with x(start_date,end_date)
as (
  select start_date,
         dateadd(year,1,start_date) end_date
    from (
      select cast(
        cast(year(getdate()) as varchar) + '-01-01'
        as datetime) start_date
        from t1
      ) tmp
  union all
  select dateadd(day,1,start_date), end_date
    from x
   where dateadd(day,1,start_date) < end_date
)
select datename(dw,start_date), count(*)
  from x
 group by datename(dw,start_date)
OPTION (MAXRECURSION 366)
```

START_DATE	COUNT(*)
FRIDAY	52
MONDAY	52
SATURDAY	53
SUNDAY	52
THURSDAY	52
TUESDAY	52
WEDNESDAY	52

8.7 确定当前记录和下一条记录存储的日期相隔多少天

1. 问题

你想确定两个日期（准确地说是存储在两行中的两个日期）相差多少天。例如，对于 10 号部门的每一位员工，你想确定从其获聘起到下一位员工（可能是另一个部门的）获聘过了多少天。

2. 解决方案

要解决这个问题，诀窍是找出当前员工的 **HIREDATE** 后最早的 **HIREDATE**，然后，使用 8.2 节介绍的技巧确定相隔的天数即可。

DB2

使用标量子查询找出当前 **HIREDATE** 后的第一个 **HIREDATE**，然后使用函数 **DAYS** 计算这两个日期相差的天数。

```
1 select x.*,
2       days(x.next_hd) - days(x.hiredate) diff
3   from (
4 select e.deptno, e.ename, e.hiredate,
5       lead(hiredate)over(order by hiredate) next_hd
6   from emp e
7  where e.deptno = 10
8       ) x
```

MySQL 和 SQL Server

使用函数 **LEAD** 访问下一条记录。下面演示了在 SQL Server 解决方案中如何使用函数 **DATEDIFF**。

```
1 select x.ename, x.hiredate, x.next_hd,  
2         datediff(x.hiredate,x.next_hd,day) as diff  
3   from (  
4 select deptno, ename, hiredate,  
5         lead(hiredate)over(order by hiredate) as next_hd  
6   from emp e  
7        ) x  
8  where e.deptno=10
```

如果使用的是 MySQL，则可以删除第一个参数（**DAY**），然后将余下的两个参数交换顺序。

```
2         datediff(x.next_hd, x.hiredate) diff
```

Oracle

使用窗口函数 **LEAD OVER** 访问按照 **HIREDATE** 排序时位于当前行后面一行的 **HIREDATE**，然后执行减法运算。

```
1 select ename, hiredate, next_hd,  
2         next_hd - hiredate diff  
3   from (  
4 select deptno, ename, hiredate,  
5         lead(hiredate)over(order by hiredate) next_hd  
6   from emp  
7        )  
8  where deptno=10
```

PostgreSQL

使用标量子查询找出当前 **HIREDATE** 后的第一个 **HIREDATE**，然后使用减法运算找出相隔的天数。

```
1 select x.*,
```

```
2      x.next_hd - x.hiredate as diff
3  from (
4  select e.deptno, e.ename, e.hiredate,
5         lead(hiredate)over(order by hiredate) as next_hd
6  from emp e
7  where e.deptno = 10
8         ) x
```

13. 讨论

虽然语法不同，但以上所有解决方案采用的方法都相同：使用窗口函数 **LEAD**，然后使用 8.2 节介绍的技巧找出相隔的天数。

如果能够在不使用连接的情况下访问当前行周边的行，就可以提高代码的可读性和效率。使用窗口函数时，别忘了它们是在 **WHERE** 子句后执行的，这就是需要在解决方案中使用内嵌视图的原因。如果将基于 **DEPTNO** 的过滤器移到内嵌视图中，则结果将不同（只考虑 10 号部门员工的 **HIREDATE**）。对于 Oracle 函数 **LEAD** 和 **LAG**，需要特别注意它们在存在重复值时的行为。本书前言中说过，这些解决方案是非“防御性”的，因为有太多无法预见的情况可能导致代码不能正确运行。即便能够预测到所有的问题，在有些情况下，编写出来的 **SQL** 代码也将难以阅读。因此，在大多数情况下，解决方案的目标是介绍一种技巧，你可以将这种技巧用于生产系统中，但必须进行测试和调整，确保它对特定的数据管用。这里之所以讨论重复 **HIREDATE** 的问题，是因为规避方案可能不那么显而易见，对非 Oracle 用户来说尤其如此。在本例中，**EMP** 表中没有重复的 **HIREDATE**，但在你的表中，很可能存在重复的日期值。下面列出了 10 号部门的员工及其 **HIREDATE**。

```
select ename, hiredate
```

```

from emp
where deptno=10
order by 2

ENAME    HIREDATE
-----
CLARK    09-JUN-2006
KING     17-NOV-2006
MILLER   23-JAN-2007

```

下面来插入 4 个重复的日期，使得包括 KING 在内的 5 名员工的 HIREDATE 都是 2006 年 11 月 17 日。

```

insert into emp (empno,ename,deptno,hiredate)
values (1,'ant',10,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,hiredate)
values (2,'joe',10,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,hiredate)
values (3,'jim',10,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,hiredate)
values (4,'choi',10,to_date('17-NOV-2006'))

select ename, hiredate
from emp
where deptno=10
order by 2

ENAME    HIREDATE
-----
CLARK    09-JUN-2006
ant      17-NOV-2006
joe      17-NOV-2006
KING     17-NOV-2006
jim      17-NOV-2006
choi     17-NOV-2007
MILLER   23-JAN-2007

```

现在，10 号部门有多名员工的 HIREDATE 相同。如果你现在对这个数据集执行前面提议的解决方案（将过滤器移到内嵌

视图中，以便只考虑 10 号部门的员工及其 HIREDATE），则会得到如下输出。

```
select ename, hiredate, next_hd,
       next_hd - hiredate diff
  from (
select deptno, ename, hiredate,
       lead(hiredate)over(order by hiredate) next_hd
  from emp
 where deptno=10
  )
```

ENAME	HIREDATE	NEXT_HD	DIFF
CLARK	09-JUN-2006	17-NOV-2006	161
ant	17-NOV-2006	17-NOV-2006	0
joe	17-NOV-2006	17-NOV-2006	0
KING	17-NOV-2006	17-NOV-2006	0
jim	17-NOV-2006	17-NOV-2006	0
choi	17-NOV-2006	23-JAN-2007	67
MILLER	23-JAN-2007	(null)	(null)

在 HIREDATE 相同的 5 位员工中，有 4 位的 DIFF 值为 0。这不正确。对于同一天获聘的所有员工，应将其 HIREDATE 与下一位员工的 HIREDATE 进行比较。换言之，对于 2006 年 11 月 17 日获聘的员工，应将其 HIREDATE 与 MILLER 的 HIREDATE 进行比较。这里的问题是，函数 LEAD 将行按照 HIREDATE 排序，但不跳过重复的 HIREDATE。因此，对于员工 ANT，将其 HIREDATE 与 JOE 的 HIREDATE 进行比较，这两个日期的差为 0，因此 ANT 的 DIFF 值为 0。好在 Oracle 为类似这样的情形提供了规避方案：调用函数 LEAD 时，可以向它传递一个参数，准确地指出后续行（比如下一行、接下来的第 10 行等）的位置。对于员工 ANT，需要将其与接下来的第 5 行（你要跳过其他所有重复的 HIREDATE），而不是下一行进行比较，因为接下来的第 5 行是 MILLER 所在的行。对于员工 JOE，他与 MILLER 的距离是 4 行，KING 与 MILLER 的距离是 3 行，JIM 与 MILLER 的距离是 2 行，

而 CHOI 与 MILLER 的距离是 1 行。要得到正确答案，只需将这些员工与 MILLER 的距离作为参数传递给 LEAD。最终的解决方案如下。

```
select ename, hiredate, next_hd,
       next_hd - hiredate diff
  from (
select deptno, ename, hiredate,
       lead(hiredate,cnt-rn+1)over(order by hiredate) next_hd
  from (
select deptno,ename,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
 where deptno=10
  )
  )
```

ENAME	HIREDATE	NEXT_HD	DIFF
CLARK	09-JUN-2006	17-NOV-2006	161
ant	17-NOV-2006	23-JAN-2007	67
joe	17-NOV-2006	23-JAN-2007	67
jim	17-NOV-2006	23-JAN-2007	67
choi	17-NOV-2006	23-JAN-2007	67
KING	17-NOV-2006	23-JAN-2007	67
MILLER	23-JAN-2007	(null)	(null)

现在，答案是正确的。对于获聘日期相同的所有员工，将其获聘日期与下一个获聘日期（而不是同样的获聘日期）进行比较。如果理解不了这个规避方案，那么可以对查询进行拆解。

先来看内嵌视图。

```
select deptno,ename,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
 where deptno=10
```

DEPTNO	ENAME	HIREDATE	CNT	RN
10	CLARK	09-JUN-2006	1	1
10	ant	17-NOV-2006	5	1
10	joe	17-NOV-2006	5	2
10	jim	17-NOV-2006	5	3
10	choi	17-NOV-2006	5	4
10	KING	17-NOV-2006	5	5
10	MILLER	23-JAN-2007	1	1

窗口函数 **COUNT OVER** 会计算每个 **HIREDATE** 值出现的次数，并在相应行中返回它。对于重复的 **HIREDATE**，在相应行中返回 5。窗口函数 **ROW_NUMBER OVER** 根据 **EMPNO** 为每一位员工排名。排名时按照 **HIREDATE** 分组，因此，除非有重复的 **HIREDATE**，否则每位员工的排名都是 1。计算重复的 **HIREDATE** 数量并进行排名后，可以使用排名来计算到下一个 **HIREDATE**（**MILLER** 的 **HIREDATE**）的距离。调用 **LEAD** 时，距离是通过将 **CNT** 减去 **RN** 再加 1 得到的。

```

select deptno, ename, hiredate,
       cnt-rn+1 distance_to_miller,
       lead(hiredate,cnt-rn+1)over(order by hiredate) next_hd
  from (
select deptno,ename,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
 where deptno=10
    )

```

DEPTNO	ENAME	HIREDATE	DISTANCE_TO_MILLER	NEXT_HD
10	CLARK	09-JUN-2006	1	17-NOV-2006
10	ant	17-NOV-2006	5	23-JAN-2007
10	joe	17-NOV-2006	4	23-JAN-2007
10	jim	17-NOV-2006	3	23-JAN-2007
10	choi	17-NOV-2006	2	23-JAN-2007
10	KING	17-NOV-2006	1	23-JAN-2007
10	MILLER	23-JAN-2007	1 (null)	

如你所见，通过指定要向前跳的距离，可以让函数 **LEAD** 对正确的日期执行减法运算。

8.8 小结

日期是一种常见的数据类型，但有自己的独特之处，因为其结构化程度比简单的数字数据类型高。相比于其他方面，日期处理的标准化程度要低些，但每个 **DBMS** 都提供了一组核心函数，用于执行相同的任务，不过它们在语法上存在细微差别。掌握这些核心函数后，就能成功地处理日期了。

第 9 章 操作日期

本章介绍如何查询和修改日期。涉及日期的查询很常见，因此，你不仅要掌握处理日期的思路，还要对 **RDBMS** 中用于操作日期的函数有深入认识。本章的实例将为你打下坚实的基础，以帮助你编写本书后面会涉及的日期和时间的复杂查询。

介绍实例前，先要重申本书前言中提到的一点，那就是将这些解决方案作为解决具体问题的参考指南，并尽可能从全局的角度考虑。如果实例解决了与当前月份相关的问题，那么或许对解决方案做细微的修改后，就可以解决与任何月份相关的问题。需要再次重申的是，这些解决方案只是参考指南，而非可以照搬的最终答案。任何一本书都不可能提供所有问题的答案，但只要你理解了本书的内容，对这些解决方案进行修改以满足需求将是“小菜一碟”。另外，请考虑这些解决方案的替代版本。如果解决方案使用了 **RDBMS** 提供的特定函数，那么就应该花时间和精力进行研究，看看能否找到效率更高的替代方案。知道有哪些选项可供选择后，你就会成为更优秀的 **SQL** 程序员。



本章的实例会使用简单的日期数据类型，如果你使用的日期数据类型更复杂，则需要对解决方案做相应调整。

9.1 判断特定的年份是否是闰年

1. 问题

你想判断当前年份是否是闰年。

2. 解决方案

如果你使用 SQL 已有一段时间，那么肯定知道解决这个问题有多种方法。我们见过的所有解决方案的效果都很好，但这里介绍的解决方案可能是最简单的。下面的解决方案会检查 2 月份的最后一天，如果它是 2 月 29 日，那么当前年份就是闰年。

DB2

使用递归式 **WITH** 子句返回 2 月份的每一天，然后使用聚合函数 **MAX** 找出 2 月份的最后一天。

```
1  with x (dy,mth)
2    as (
3  select dy, month(dy)
4    from (
5  select (current_date -
6         dayofyear(current_date) days +1 days)
7         +1 months as dy
8    from t1
9       ) tmp1
10 union all
11 select dy+1 days, mth
12    from x
13   where month(dy+1 day) = mth
14 )
15 select max(day(dy))
```

```
16    from x
```

Oracle

使用函数 **LAST_DAY** 找出 2 月份的最后一天。

```
1 select to_char(  
2         last_day(add_months(trunc(sysdate,'y'),1)),  
3         'DD')  
4    from t1
```

PostgreSQL

使用函数 **GENERATE_SERIES** 返回 2 月份的每一天，然后使用聚合函数 **MAX** 找出 2 月份的最后一天。

```
1 select max(to_char(tmp2.dy+x.id,'DD')) as dy  
2    from (  
3 select dy, to_char(dy,'MM') as mth  
4    from (  
5 select cast(cast(  
6             date_trunc('year',current_date) as date)  
7             + interval '1 month' as date) as dy  
8    from t1  
9         ) tmp1  
10         ) tmp2, generate_series (0,29) x(id)  
11  where to_char(tmp2.dy+x.id,'MM') = tmp2.mth
```

MySQL

使用函数 **LAST_DAY** 找出 2 月份的最后一天。

```
1 select day(  
2         last_day(  
3         date_add(  
4         date_add(  
5         date_add(current_date,  
6                   interval -dayofyear(current_date) day),  
7                   interval 1 day),
```



```
8          interval 1 month))) dy
9  from t1
```

SQL Server

使用函数 **DAY** 检查当前年份是否有 2 月 29 日，如果没有，那么这个函数将返回 **NULL**。如果函数 **DAY** 的返回值不是 **NULL**，那么函数 **COALESCE** 将返回这个值，否则返回 28。

```
select coalesce
      (day
      (cast(concat
            (year(getdate()), '-02-29')
            as date))
      ,28);
```

13. 讨论

DB2

递归视图 **X** 中的内嵌视图 **TMP1** 会返回 2 月份的第一天，这是通过如下步骤完成的。

- a. 从当前日期开始。
- b. 使用 **DAYOFYEAR** 确定当前日期是当前年份的第几天。
- c. 从当前日期中减去第 2 步得到的天数，得到前一年的 12 月 31 日，再加上 1 得到当前年份的第一天，即 1 月 1 日。
- d. 加上一个月得到 2 月 1 日。上述操作的结果如下所示。

```
select (current_date
      dayofyear(current_date) days +1 days) +1 months as dy
from t1
```

DY

01-FEB-2005

然后，使用函数 **MONTH** 获取内嵌视图 **TEMP1** 返回的日期中的月份部分。

```
select dy, month(dy) as mth
  from (
select (current_date
        dayofyear(current_date) days +1 days) +1 months as dy
  from t1
    ) tmp1
```

DY	MTH
-----	---
01-FEB-2005	2

至此得到的结果为生成 2 月份的每一天的递归操作提供了起点。为返回 2 月份的每一天，不断地给 **DY** 加上 1 天，直到日期不再属于 2 月份。下面显示了这个 **WITH** 操作的部分结果。

```
with x (dy,mth)
  as (
select dy, month(dy)
  from (
select (current_date -
        dayofyear(current_date) days +1 days) +1 months as dy
  from t1
    ) tmp1
union all
select dy+1 days, mth
  from x
 where month(dy+1 day) = mth
 )
select dy,mth
  from x
```

DY	MTH
----	-----

```

-----
01-FEB-2005    2
...
10-FEB-2005    2
...
28-FEB-2005    2

```

最后，将函数 **MAX** 应用于 **DY** 列，以返回 2 月份的最后一天。如果这一天是 29 日，就说明当前年份为闰年。

Oracle

首先，使用函数 **TRUNC** 找到当前年份的第一天。

```

select trunc(sysdate,'y')
  from t1

DY
-----
01-JAN-2005

```

由于一年的第一天是 1 月 1 日，因此下一步是加上 1 个月，得到 2 月 1 日。

```

select add_months(trunc(sysdate,'y'),1) dy
  from t1

DY
-----
01-FEB-2005

```

接下来，使用函数 **LAST_DAY** 找到 2 月份的最后一天。

```

select last_day(add_months(trunc(sysdate,'y'),1)) dy
  from t1

DY
-----
28-FEB-2005

```

最后，使用 `TO_CHAR` 返回 28 或 29（这一步是可选的）。

PostgreSQL

首先，查看内嵌视图 **TMP1** 返回的结果。这个视图会使用函数 `DATE_TRUNC` 找到当前年份的第一天，并将结果转换为 `DATE`。

```
select cast(date_trunc('year',current_date) as date) as dy
from t1

DY
-----
01-JAN-2005
```

然后，给当前年份的第一天加上 1 个月，得到 2 月份的第一天，并将结果转换为日期。

```
select cast(cast(
            date_trunc('year',current_date) as date)
            + interval '1 month' as date) as dy
from t1

DY
-----
01-FEB-2005
```

接下来，内嵌视图 **TMP1** 会返回 `DY` 及其月份部分的数字表示。为获取月份的数字表示，使用了函数 `TO_CHAR`。

```
select dy, to_char(dy,'MM') as mth
from (
select cast(cast(
            date_trunc('year',current_date) as date)
            + interval '1 month' as date) as dy
from t1
) tmp1
```

DY	MTH
-----	---
01-FEB-2005	2

这就是内嵌视图 **TMP2** 的结果集。下一步是使用很有用的函数 **GENERATE_SERIES** 返回 29 行数据（值 1~29）。对于 **GENERATE_SERIES**（别名 **X**）返回的每一行，都将其与内嵌视图 **TMP2** 返回的 **DY** 相加，下面显示了部分结果。

```
select tmp2.dy+x.id as dy, tmp2.mth
  from (
select dy, to_char(dy,'MM') as mth
  from (
select cast(cast(
                date_trunc('year',current_date) as date)
                + interval '1 month' as date) as dy
  from t1
    ) tmp1
    ) tmp2, generate_series (0,29) x(id)
 where to_char(tmp2.dy+x.id,'MM') = tmp2.mth
```

DY	MTH
-----	---
01-FEB-2005	02
...	
10-FEB-2005	02
...	
28-FEB-2005	02

最后，使用函数 **MAX** 返回 2 月份的最后一天。通过调用函数 **TO_CHAR**，让返回的值为 28 或 29。

MySQL

首先，找出当前年份的第一天，方法是计算当前日期是当年的第几天。然后，从当前日期中减去这个天数，再加 1。所有这些操作都是使用函数 **DATE_ADD** 完成的。

```
select date_add(
    date_add(current_date,
        interval -dayofyear(current_date) day),
        interval 1 day) dy
    from t1

DY
-----
01-JAN-2005
```

接下来，使用函数 **DATE_ADD** 再加上 1 个月。

```
select date_add(
    date_add(
        date_add(current_date,
            interval -dayofyear(current_date) day),
            interval 1 day),
            interval 1 month) dy
    from t1

DY
-----
01-FEB-2005
```

得到 2 月份的第 1 天后，使用函数 **LAST_DAY** 获得 2 月份的最后一天。

```
select last_day(
    date_add(
        date_add(
            date_add(current_date,
                interval -dayofyear(current_date) day),
                interval 1 day),
                interval 1 month)) dy
    from t1

DY
-----
28-FEB-2005
```

最后，使用函数 **DAY** 返回 28 或 29（这一步是可选的）。

SQL Server

在大多数 RDBMS 中，可以这样创建日期：创建一个符合日期格式的字符串，然后使用 **CAST** 将其转换为日期。因此，可以通过检索当前日期中的年份来获取当前年份。在 SQL Server 中，这是通过对 **GET_DATE** 返回的结果调用 **YEAR** 得到的。

```
select YEAR(GETDATE());
```

这将以整数的方式返回年份。然后，可以使用 **CONCAT** 和 **CAST** 创建表示当年 2 月 29 日的日期。

```
select cast(concat  
                                                    (year(getdate()), '-02-29');
```

不过，如果当年不是闰年，那么这样的日期将不存在。例如，根本就没有 2019 年 2 月 29 日，如果你试图使用 **DAY** 等函数获取其组成部分，则返回的值将为 **NULL**。因此 SQL Server 解决方案使用了 **COALESCE** 和 **DAY** 来判断 2 月是否有 29 日。

9.2 确定特定年份有多少天

1. 问题

你想计算当前年份有多少天。

2. 解决方案

当前年份的天数，就是下一年第一天与当前年份第一天之间相差的天数。所有解决方案都采取如下步骤。

- a. 找出当前年份的第一天。
- b. 给上面的日期加上 1 年，得到下一年的第一天。
- c. 将前两步得到的日期相减。

在不同的 RDBMS 中，解决方案之间的唯一差别在于，用来执行这些步骤的内置函数不同。

DB2

使用函数 **DAYOFYEAR** 找出当前年份的第一天，并使用 **DAYS** 确定两个日期之间相隔的天数。

```
1 select days((curr_year + 1 year)) - days(curr_year)
2   from (
3 select (current_date -
4         dayofyear(current_date) day +
5         1 day) curr_year
6   from t1
7        ) x
```

Oracle

使用函数 **TRUNC** 找出当前年份的第一天，然后使用 **ADD_MONTHS** 找出下一年的第一天。

```
1 select add_months(trunc(sysdate,'y'),12) - trunc(sysdate,'y')
2   from dual
```

PostgreSQL

使用函数 **DATE_TRUNC** 找出当前年份的第一天，然后使用间隔算术运算确定下一年的第一天。

```
1 select cast((curr_year + interval '1 year') as date) - curr_year
2   from (
3 select cast(date_trunc('year',current_date) as date) as
curr_year
4   from t1
5        ) x
```

MySQL

使用 **ADDDATE** 找出当前年份的第一天，然后使用 **DATEDIFF** 和间隔算术运算确定当前年份的天数。

```
1 select datediff((curr_year + interval 1 year),curr_year)
2   from (
3 select adddate(current_date,-dayofyear(current_date)+1)
curr_year
4   from t1
5        ) x
```

SQL Server

使用函数 **DATEADD** 找出当前年份的第一天，然后使用 **DATEDIFF** 返回当前年份的天数。

```
1 select datediff(d,curr_year,dateadd(yy,1,curr_year))
2   from (
```

```

3 select dateadd(d,-datepart(dy,getdate()+1,getdate())) curr_year
4   from t1
5       ) x

```

13. 讨论

DB2

首先，找出当前年份的第一天，方法是使用 **DAYOFYEAR** 确定当前日期是当前年份的第几天，将当前日期减去这个天数，得到上一年的最后一天，然后加上 1 天。

```

select (current_date
        dayofyear(current_date) day +
        1 day) curr_year
  from t1

CURR_YEAR
-----
01-JAN-2005

```

有了当前年份的第一天后，只需加上 1 年，就可得到下一年的第一天。最后，将下一年的第一天与当前年份的第一天相减。

Oracle

首先，找出当前年份的第一天（这个任务完成起来很容易），方法是调用内置函数 **TRUNC**，并将第二个参数设置为 **Y**（从而将日期截断为当年的第一天）。

```

select select trunc(sysdate,'y') curr_year
  from dual

CURR_YEAR

```

```
-----  
01-JAN-2005
```

然后，加上 1 年，得到下一年的第一天。最后，将这两个日期相减，得到当前年份的天数。

PostgreSQL

首先，找出当前年份的第一天，方法是调用函数 `DATE_TRUNC`。

```
select cast(date_trunc('year',current_date) as date) as curr_year  
from t1  
  
CURR_YEAR  
-----  
01-JAN-2005
```

然后，加上 1 年得到下一年的第一天，再将两个日期相减。务必用较晚的日期减去较早的日期。结果就是当前年份的天数。

MySQL

首先，找出当前年份的第一天。为此，使用 `DAYOFYEAR` 确定当前日期是当前年份的第几天，将当前日期减去这个天数，再加 1。

```
select adddate(current_date,-dayofyear(current_date)+1) curr_year  
from t1  
  
CURR_YEAR  
-----  
01-JAN-2005
```

有了当前年份的第一天后，加上 1 年得到下一年的第一天。

然后，将下一年的第一天与当前年份的第一天相减，得到当前年份的天数。

SQL Server

首先，找出当前年份的第一天。为此，使用 **DATEPART** 确定当前日期是当前年份的第几天，再使用 **DATEADD** 将当前日期减去这个天数并加 1。

```
select dateadd(d,-datepart(dy,getdate()+1,getdate()) curr_year
      from t1

CURR_YEAR
-----
01-JAN-2005
```

有了当前年份的第一天后，下一步是给它加上 1 年，得到下一年的第一天。然后，将下一年的第一天和当前年份的第一天相减，得到当前年份的天数。

9.3 提取日期的各个组成部分

1. 问题

你想将当前日期拆分成 6 部分：日、月、年、秒、分钟和小时，并将结果作为数字返回。

2. 解决方案

这里处理的是当前日期，但这些解决方案也适用于其他日期。当前，大多数 RDBMS 实现了 ANSI 标准定义的日期部分提取函数 **EXTRACT**，但 SQL Server 是个例外。这些 RDBMS 同时保留了提取日期部分的遗留方法。

DB2

DB2 实现了一系列内置函数，让你能够轻松地提取日期的不同部分。函数 **HOUR**、**MINUTE**、**SECOND**、**DAY**、**MONTH** 和 **YEAR** 会分别返回日期的相应部分。（如果要提取日，可以使用 **DAY**；如果要提取小时，可以使用 **HOUR**。以此类推。）

```
1 select      hour( current_timestamp ) hr,
2             minute( current_timestamp ) min,
3             second( current_timestamp ) sec,
4             day( current_timestamp ) dy,
5             month( current_timestamp ) mth,
6             year( current_timestamp ) yr
7   from t1

select
    extract(hour from current_timestamp)
, extract(minute from current_timestamp)
, extract(second from current_timestamp)
```

```

, extract(day from current_timestamp)
, extract(month from current_timestamp)
, extract(year from current_timestamp)

```

HR	MIN	SEC	DY	MTH	YR
20	28	36	15	6	2005

Oracle

使用函数 `TO_CHAR` 和 `TO_NUMBER` 提取日期的特定部分。

```

1 select to_number(to_char(sysdate,'hh24')) hour,
2        to_number(to_char(sysdate,'mi')) min,
3        to_number(to_char(sysdate,'ss')) sec,
4        to_number(to_char(sysdate,'dd')) day,
5        to_number(to_char(sysdate,'mm')) mth,
6        to_number(to_char(sysdate,'yyyy')) year
7 from dual

```

HOUR	MIN	SEC	DAY	MTH	YEAR
20	28	36	15	6	2005

PostgreSQL

使用函数 `TO_CHAR` 和 `TO_NUMBER` 提取日期的特定部分。

```

1 select to_number(to_char(current_timestamp,'hh24'),'99') as hr,
2        to_number(to_char(current_timestamp,'mi'),'99') as min,
3        to_number(to_char(current_timestamp,'ss'),'99') as sec,
4        to_number(to_char(current_timestamp,'dd'),'99') as day,
5        to_number(to_char(current_timestamp,'mm'),'99') as mth,
6        to_number(to_char(current_timestamp,'yyyy'),'9999') as yr
7 from t1

```

HR	MIN	SEC	DAY	MTH	YR
20	28	36	15	6	2005

MySQL

使用函数 **DATE_FORMAT** 提取日期的特定部分。

```
1 select date_format(current_timestamp,'%k') hr,  
2         date_format(current_timestamp,'%i') min,  
3         date_format(current_timestamp,'%s') sec,  
4         date_format(current_timestamp,'%d') dy,  
5         date_format(current_timestamp,'%m') mon,  
6         date_format(current_timestamp,'%Y') yr  
7   from t1
```

HR	MIN	SEC	DAY	MTH	YR
20	28	36	15	6	2005

SQL Server

使用函数 **DATEPART** 提取日期的特定部分。

```
1 select datepart( hour, getdate()) hr,  
2         datepart( minute,getdate()) min,  
3         datepart( second,getdate()) sec,  
4         datepart( day, getdate()) dy,  
5         datepart( month, getdate()) mon,  
6         datepart( year, getdate()) yr  
7   from t1
```

HR	MIN	SEC	DAY	MTH	YR
20	28	36	15	6	2005

13. 讨论

以上解决方案并无神奇之处，只是利用了 **RDBMS** 内置的函数。请花时间研究可以使用的日期函数。这些解决方案只展示了它们使用的日期函数的“冰山一角”。你将发现，与上面

的展示相比，这些函数可以接受的参数要多得多，可以返回的信息也多得多。

9.4 找出一个月的第一天和最后一天

1. 问题

你想找出当前月份的第一天和最后一天。

2. 解决方案

下面介绍的解决方案找出了当前月份的第一天和最后一天，但只要稍做调整，就可用于找出任何月份的第一天和最后一天。

DB2

使用函数 **DAY** 确定当前日期是当前月份的第几天，然后将当前日期减去这个天数并加 1，得到当前月份的第一天。为了得到当前月份的最后一天，给当前日期加上 1 个月，再减去将函数 **DAY** 应用于当前日期加 1 个月时返回的天数。

```
1 select (date(current_date) - day(date(current_date)) day + 1
day) firstday,
2         (date(current_date)+1 month
3         - day(date(current_date)+1 month) day) lastday
4   from t1
```

Oracle

使用函数 **TRUNC** 找出月份的第一天，并使用函数 **LAST_DAY** 找出月份的最后一天。

```
1 select trunc(sysdate,'mm') firstday,
2        last_day(sysdate) lastday
```



像上面那样使用 **TRUNC** 时，将导致时间部分丢失，但 **LAST_DAY** 会保留时间部分。

PostgreSQL

使用函数 **DATE_TRUNC** 将当前日期截断为当前月份的第一天。有了月份的第一天后，只需加上 1 个月再减去 1 天，就可找出该月份的最后一天。

```
1 select firstday,
2         cast(firstday + interval '1 month'
3              - interval '1 day' as date) as lastday
4   from (
5 select cast(date_trunc('month',current_date) as date) as
firstday
6   from t1
7        ) x
```

MySQL

使用函数 **DAY** 找出当前日期是其所属月份的第几天，然后使用函数 **DATE_ADD** 将当前日期减去这个天数并加 1，得到该月份的第一天。要找出当前月份的最后一天，可以使用函数 **LAST_DAY**。

```
1 select date_add(current_date,
2                 interval -day(current_date)+1 day) firstday,
3         last_day(current_date) lastday
4   from t1
```

SQL Server

使用函数 **DAY** 找出当前日期是其所属月份的第几天，然后使

用函数 **DATEADD** 将当前日期减去这个天数并加 1，得到该月份的第一天。要找出该月份的最后一天，也可以使用函数 **DAY** 和 **DATEADD**，方法是给当前日期加 1 个月得到一个新日期，并计算该新日期是其所属月份的第几天，再将新日期减去这个天数。

```
1 select dateadd(day,-day(getdate()+1,getdate())) firstday,  
2         dateadd(day,  
3             -day(dateadd(month,1,getdate())),  
4             dateadd(month,1,getdate())) lastday  
5 from t1
```

13. 讨论

DB2

要找出月份的第一天，只需确定当前日期是其所属月份的第几天，然后将当前日期减去这个天数。如果日期是 3 月 14 日，那么它就是 3 月的第 14 天。将 3 月 14 日减去 14 天，就能得到 2 月的最后一天。然后，只需再加上 1 天，就可得到当前月份的第一天。找出最后一天的方法与此类似：确定当前日期是其所属月份的第几天，然后将当前日期减去这个天数，就可得到上个月的最后一天。由于要找出的是本月（而不是上个月）的最后一天，因此需要给当前日期加上 1 个月。

Oracle

要找出当月的第一天，使用函数 **TRUNC** 并将第二个参数设置为“mm”，然后将当前日期截断为其所属月份的第一天。要找出当月的最后一天，只需使用函数 **LAST_DAY**。

PostgreSQL

要找出当月的第一天，使用函数 **DATE_TRUNC** 并将其第二个参数设置为“month”，然后将当前日期截断为其所属月份的第一天。要找出当月的最后一天，给当月第一天加 1 个月，再减去 1 天。

MySQL

要找出当月的第一天，使用函数 **DAY**。函数 **DAY** 会指出传递给它的日期是其所属月份的第几天。如果将当前日期减去 **DAY(CURRENT_DATE)** 返回的值，那么将得到上个月的最后一天，再加上 1 天将得到当月的第一天。要找出当月的最后一天，只需使用函数 **LAST_DAY**。

SQL Server

要找出当月的第一天，可以使用函数 **DAY**。函数 **DAY** 会指出传递给它的日期是其所属月份的第几天。如果将当前日期减去 **DAY(GETDATE())** 返回的值，那么将得到上个月的最后一天，再加上 1 天可以得到当月的第一天。要找出当月的最后一天，可以使用函数 **DATEADD**。将当前日期加上 1 个月，再减去 **DAY(DATEADD(MONTH,1,GETDATE()))** 返回的值，可以得到当月的最后一天。

9.5 找出一年中所有的星期 n

1. 问题

你想找出一年中所有为星期 n 的日期。例如，你可能想列出当前年份中所有的星期五。

2. 解决方案

无论使用的是哪个 RDBMS，解决方案的关键都是返回当前年份的每一天，然后只保留那些为星期 n 的日期。下面的解决方案会保留所有的星期五。

DB2

使用递归式 **WITH** 子句返回当年的每一天，然后使用函数 **DAYNAME** 将星期五对应的那些日期留下。

```
1  with x (dy,yr)
2    as (
3  select dy, year(dy) yr
4    from (
5  select (current_date -
6         dayofyear(current_date) days +1 days) as dy
7    from t1
8       ) tmp1
9  union all
10 select dy+1 days, yr
11    from x
12   where year(dy +1 day) = yr
13 )
14 select dy
15    from x
16   where dayname(dy) = 'Friday'
```

Oracle

使用递归式 **CONNECT BY** 子句返回当年的每一天，然后使用函数 **TO_CHAR** 将星期五对应的那些日期留下。

```
1  with x
2    as (
3  select trunc(sysdate,'y')+level-1 dy
4    from t1
5   connect by level <=
6             add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
7 )
8 select *
9   from x
10  where to_char( dy, 'dy') = 'fri'
```

PostgreSQL

使用递归式 CTE 生成相应年份的每一天，并剔除那些不是星期五的日期。该解决方案使用的是 ANSI 标准函数 **EXTRACT**，因此适用于众多不同的 RDBMS。

```
1  with recursive cal (dy)
2    as (
3  select current_date
4    -(cast
5      (extract(doy from current_date) as integer)
6      -1)
7    union all
8    select dy+1
9      from cal
10     where extract(year from dy)=extract(year from (dy+1))
11    )
12
13  select dy,extract(dow from dy) from cal
14  where cast(extract(dow from dy) as integer) = 5
```

MySQL

使用递归式 CTE 找出指定年份的每一天，然后剔除不是星期五的日期。

```
1      with recursive cal (dy,yr)
2  as
3  (
4      select dy, extract(year from dy) as yr
5  from
6      (select adddate
7          (adddate(current_date, interval -
dayofyear(current_date)
8      day), interval 1 day) as dy) as tmp1
9  union all
10     select date_add(dy, interval 1 day), yr
11  from cal
12  where extract(year from date_add(dy, interval 1 day)) = yr
13  )
14     select dy from cal
15     where dayofweek(dy) = 6
```

SQL Server

使用递归式 WITH 子句返回当年的每一天，然后使用函数 DAYNAME 将星期五对应的那些日期留下。

```
1  with x (dy,yr)
2  as (
3  select dy, year(dy) yr
4  from (
5  select getdate()-datepart(dy,getdate())+1 dy
6  from t1
7      ) tmp1
8  union all
9  select dateadd(dd,1,dy), yr
10  from x
11  where year(dateadd(dd,1,dy)) = yr
12  )
13  select x.dy
14  from x
15  where datename(dw,x.dy) = 'Friday'
16  option (maxrecursion 400)
```

13. 讨论

DB2

要找出当前年份所有为星期五的日期，必须能够返回当前年份的每一天。首先，使用函数 **DAYOFYEAR** 找出当前年份的第一天，为此将当前日期减去 **DAYOFYEAR(CURRENT_DATE)** 返回的值，得到上一年的最后一天，再加上 1 天得到当年的第一天。

```
select (current_date
        dayofyear(current_date) days +1 days) as dy
  from t1

DY
-----
01-JAN-2005
```

然后，有了当年的第一天后，使用 **WITH** 子句不断地加 1 天，直到进入下一年，这样得到的结果集将为当年的每一天。下面显示了递归视图 **X** 返回的部分行。

```
with x (dy,yr)
  as (
select dy, year(dy) yr
  from (
select (current_date
        dayofyear(current_date) days +1 days) as dy
  from t1
    ) tmp1
union all
select dy+1 days, yr
  from x
 where year(dy +1 day) = yr
)
select dy
```



```

from x

DY
-----
01-JAN-2020
...
15-FEB-2020
...
22-NOV-2020
...
31-DEC-2020

```

最后，使用函数 **DAYNAME** 将星期五对应的那些行留下。

Oracle

要找出当年的所有星期五，必须能够返回当年的每一天。为此，首先使用函数 **TRUNC** 找出当年的第一天。

```

select trunc(sysdate,'y') dy
      from t1

      DY
      -----
01-JAN-2020

```

然后，使用 **CONNECT BY** 子句返回当年的每一天。（如果想弄明白如何使用 **CONNECT BY** 来生成行，请参阅 10.5 节。）



虽然本实例解决方案使用的是 **WITH** 子句，但你也可以使用内嵌视图。

下面显示了视图 **X** 返回的结果集的一部分。

```

with x
  as (
select trunc(sysdate,'y')+level-1 dy
from t1

```

```

connect by level <=
      add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
)
select *
from x

DY
-----
01-JAN-2020
...
15-FEB-2020
...
22-NOV-2020
...
31-DEC-2020

```

最后，使用函数 **TO_CHAR** 将星期五对应的那些行留下。

PostgreSQL

要找出所有的星期五，首先需要找出所有的日期。为此，需要找出年份的第一天，然后使用递归式 CTE 填充其他日期。别忘了，PostgreSQL 是要求使用关键字 **RECURSIVE** 来标识递归式 CTE 的 RDBMS 之一。

MySQL

要找出当年所有的星期五，必须能够返回当年的每一天。首先，找出当年的第一天，方法是将当前日期减去 **DAYOFYEAR(CURRENT_DATE)** 返回的值，再加上 1。

```

select adddate(
      adddate(current_date,
              interval -dayofyear(current_date) day),
              interval 1 day ) dy
from t1

DY
-----

```

```
01-JAN-2020
```

有了当年的第一天后，只需使用递归式 CTE 来返回当年的每一天。

```
with cal (dy) as  
(select current
```

```
union all  
select dy+1
```

```
    DY
```

```
    -----
```

```
    01-JAN-2020
```

```
    ...
```

```
    15-FEB-2020
```

```
    ...
```

```
    22-NOV-2020
```

```
    ...
```

```
    31-DEC-2020
```

最后，使用函数 **DAYNAME** 将是星期五的日期留下。

SQL Server

要找出当年的所有星期五，必须能够返回当年的每一天。首先，使用函数 **DATEPART** 找出当年的第一天，为此将当前日期减去 **DATEPART(DY,GETDATE())** 返回的值，再加上 1。

```
select getdate()-datepart(dy,getdate())+1 dy  
from t1
```

```
DY
```

```
-----
```

```
01-JAN-2005
```

然后，有了当年的第一天后，使用 **WITH** 子句和函数

DATEADD 不断地加上 1 天，直到进入下一年，这样得到的结果集将为当年的每一天。下面显示了递归视图 **X** 返回的部分行。

```
with x (dy,yr)
  as (
select dy, year(dy) yr
  from (
select getdate()-datepart(dy,getdate())+1 dy
  from t1
    ) tmp1
  union all
select dateadd(dd,1,dy), yr
  from x
  where year(dateadd(dd,1,dy)) = yr
)
select x.dy
  from x
option (maxrecursion 400)

DY
-----
01-JAN-2020
...
15-FEB-2020
...
22-NOV-2020
...
31-DEC-2020
```

最后，使用函数 **DATENAME** 将星期五对应的那些行留下。要让该解决方案可行，必须将 **MAXRECURSION** 至少设置为 366。（在递归视图 **X** 中，基于当前日期的年份部分的过滤器确保生成的行数不会超过 366。）

9.6 找出一个月中第一个和最后一个星期 n

1. 问题

假如你想找出当月的第一个和最后一个星期一。

2. 解决方案

下面的解决方案找出了当月的第一个和最后一个星期一，它们也可用于找出任何月份的第一个和最后一个星期 n 。由于两个相邻的星期 n 之间相隔 7 天，因此找出第一个星期 n 后，加上 7 天可以找出第二个星期 n ，加上 14 天可以找出第三个星期 n 。同样，有了特定月份的最后一个星期 n 后，减去 7 天可以找出第三个星期 n ，减去 14 天可以找出第二个星期 n 。

DB2

使用递归式 **WITH** 子句生成当月的每一天，并使用 **CASE** 表达式标出所有的星期一。在被标记的日期中，最早和最晚的将分别是第一个和最后一个星期一。

```
1  with x (dy,mth,is_monday)
2      as (
3  select dy,month(dy),
4          case when dayname(dy)='Monday'
5                then 1 else 0
6          end
7  from (
8  select (current_date-day(current_date) day +1 day) dy
9  from t1
10         ) tmp1
11  union all
```

```

12 select (dy +1 day), mth,
13         case when dayname(dy +1 day)='Monday'
14             then 1 else 0
15         end
16   from x
17  where month(dy +1 day) = mth
18 )
19 select min(dy) first_monday, max(dy) last_monday
20   from x
21  where is_monday = 1

```

Oracle

结合使用函数 **NEXT_DAY** 和 **LAST_DAY** 以及巧妙的日期算术运算，找出当月的第一个和最后一个星期一。

```

select next_day(trunc(sysdate,'mm')-1,'MONDAY') first_monday,
       next_day(last_day(trunc(sysdate,'mm'))-7,'MONDAY')
last_monday
  from dual

```

PostgreSQL

使用函数 **DATE_TRUNC** 找出当月的第一天。有了当月的第一天后，要找出当月的第一个和最后一个星期一，可以使用简单的算术运算，这些运算涉及一周各天对应的数字（星期日到星期六分别对应数字 1~7）。

```

1 select first_monday,
2        case to_char(first_monday+28,'mm')
3            when mth then first_monday+28
4                else first_monday+21
5        end as last_monday
6   from (
7 select case sign(cast(to_char(dy,'d') as integer)-2)
8        when 0
9        then dy
10       when -1
11       then dy+abs(cast(to_char(dy,'d') as integer)-2)

```

```

12             when 1
13             then (7-(cast(to_char(dy,'d') as integer)-2))+dy
14         end as first_monday,
15         mth
16     from (
17 select cast(date_trunc('month',current_date) as date) as dy,
18         to_char(current_date,'mm') as mth
19     from t1
20     ) x
21     ) y

```

MySQL

使用函数 **ADDDATE** 找出当月的第一天。有了当月的第一天后，要找出当月的第一个和最后一个星期一，可以使用简单的算术运算，这些运算涉及一周各天对应的数字（星期日到星期六分别对应数字 1~7）。

```

1 select first_monday,
2         case month(adddate(first_monday,28))
3             when mth then adddate(first_monday,28)
4             else adddate(first_monday,21)
5         end last_monday
6     from (
7 select case sign(dayofweek(dy)-2)
8         when 0 then dy
9         when -1 then adddate(dy,abs(dayofweek(dy)-2))
10        when 1 then adddate(dy,(7-(dayofweek(dy)-2)))
11     end first_monday,
12     mth
13     from (
14 select adddate(adddate(current_date,-day(current_date)),1) dy,
15         month(current_date) mth
16     from t1
17     ) x
18     ) y

```

SQL Server

使用递归式 **WITH** 子句生成当月的每一天，然后使用 **CASE** 表

达式标出所有的星期一。在标出的日期中，最早和最晚的分别是第一个和最后一个星期一。

```
1  with x (dy,mth,is_monday)
2    as (
3  select dy,mth,
4         case when datepart(dw,dy) = 2
5             then 1 else 0
6         end
7  from (
8  select dateadd(day,1,dateadd(day,-day(getdate()),getdate()))
dy,
9         month(getdate()) mth
10 from t1
11      ) tmp1
12 union all
13 select dateadd(day,1,dy),
14         mth,
15         case when datepart(dw,dateadd(day,1,dy)) = 2
16             then 1 else 0
17         end
18 from x
19 where month(dateadd(day,1,dy)) = mth
20 )
21 select min(dy) first_monday,
22        max(dy) last_monday
23 from x
24 where is_monday = 1
```

13. 讨论

DB2 和 SQL Server

DB2 和 SQL Server 使用不同的函数来解决这个问题，但方法完全相同。如果仔细研究这两种解决方案，将发现它们唯一的不同是将日期相加的方式。这里的讨论适用于这两种解决方案，但使用 DB2 解决方案中的代码来展示中间步骤的结

果。



如果你使用的 SQL Server 或 DB2 版本不支持递归式 **WITH** 子句，那么可以转而使用 PostgreSQL 解决方案演示的方法。

为了找出当月的第一个和最后一个星期一，首先返回该月的第一天。递归视图 **X** 中的内嵌视图 **TMP1** 可以找出当月的第一天，为此它会先找出当前日期是其所属月份的第几天（例如，4 月 10 日是 4 月的第 10 天）。如果将当前日期减去这个天数，那么将得到上个月最后一天。（例如，将 4 月 10 日减去 10 天，将得到 3 月的最后一天。）执行这个减法运算后，只需再加 1 天，就可得到当月的第一天。

```
select (current_date-day(current_date) day +1 day) dy
       from t1
```

DY

01-JUN-2005

然后，使用函数 **MONTH** 找出当前日期所属的月份，并使用简单的 **CASE** 表达式判断该月的第一天是否为星期一。

```
select dy, month(dy) mth,
       case when dayname(dy)='Monday'
            then 1 else 0
       end is_monday
  from (
select (current_date-day(current_date) day +1 day) dy
  from t1
    ) tmp1
```

DY MTH IS_MONDAY

01-JUN-2005 6 0

接下来，使用 **WITH** 子句的递归功能不断地加 1 天，直到进入下个月。在此过程中，使用一个 **CASE** 表达式来确定该月的哪些天为星期一（并用 1 将星期一标出）。下面显示了递归视图 **X** 的部分输出。

```
with x (dy,mth,is_monday)
  as (
    select dy,month(dy) mth,
           case when dayname(dy)='Monday'
                then 1 else 0
           end is_monday
    from (
      select (current_date-day(current_date) day +1 day) dy
      from t1
      ) tmp1
    union all
    select (dy +1 day), mth,
           case when dayname(dy +1 day)='Monday'
                then 1 else 0
           end
    from x
    where month(dy +1 day) = mth
  )
select *
from x
```

DY	MTH	IS_MONDAY
01-JUN-2005	6	0
02-JUN-2005	6	0
03-JUN-2005	6	0
04-JUN-2005	6	0
05-JUN-2005	6	0
06-JUN-2005	6	1
07-JUN-2005	6	0
08-JUN-2005	6	0
...		

只有星期一的 **IS_MONDAY** 列才为 1，因此最后一步是将聚合函数 **MIN** 和 **MAX** 用于 **IS_MONDAY** 列为 1 的行，找出该月的第一个和最后一个星期一。

Oracle

函数 **NEXT_DAY** 能够轻松地解决这个问题。为了找出当月的第一个星期一，首先返回上个月的最后一天，这是使用函数 **TRUNC** 并执行一些日期算术运算实现的。

```
select trunc(sysdate,'mm')-1 dy
      from dual
```

```
DY
-----
31-MAY-2005
```

然后，使用函数 **NEXT_DAY** 找出上个月最后一天后面的第一个星期一（当月的第一个星期一）。

```
select next_day(trunc(sysdate,'mm')-1,'MONDAY') first_monday
      from dual
```

```
FIRST_MONDAY
-----
06-JUN-2005
```

为找出当月的最后一个星期一，应先使用函数 **TRUNC** 返回当月的第一天。

```
select trunc(sysdate,'mm') dy
      from dual
```

```
DY
-----
01-JUN-2005
```

接下来，找出当月最后一周（最后 7 天）的第一天。为此，使用函数 **LAST_DAY** 找出当月的最后一天，然后减去 7 天。

```
select last_day(trunc(sysdate,'mm'))-7 dy
```

```
from dual

DY
-----
23-JUN-2005
```

如果以上描述不好理解，那么可以这样想：通过从当月的最后一天开始回溯 7 天，可以确保在该月余下的天数中，至少有一个星期一、星期二、星期三、星期四、星期五、星期六和星期日。最后，使用函数 **NEXT_DAY** 找出当月的下一个（也是最后一个）星期一。

```
select next_day(last_day(trunc(sysdate,'mm'))-7,'MONDAY')
last_monday
from dual

LAST_MONDAY
-----
27-JUN-2005
```

PostgreSQL 和 MySQL

PostgreSQL 和 MySQL 解决方案使用的方法相同，但调用的函数不同。虽然这两种解决方案的代码很长，但其中的查询非常简单：查找当月的第一个和最后一个星期一时，涉及的额外开销很少。

首先找出当月的第一天，然后找出该月的第一个星期一。由于没有用于查找下一个星期 n 的函数，因此需要执行一些算术运算。在这两种解决方案中，第 7 行的 **CASE** 表达式确定当月的第一天是星期几，并获取其数字表示，再将这个数字与星期一的数字表示相减。调用 PostgreSQL 函数 **TO_CHAR**，并将格式指定为 *D* 或 *d*（在 MySQL 中，调用函数 **DAYOFWEEK**）时，将返回一个 1~7（分别表示星期日到星期六）的数字，而星期一的数字表示为 2。**CASE** 语句中的第一个测试确定当月第一天是星期几，并获取其数字表示，再

将这个数字减去星期一的数字表示（2），并获取结果的符号（**SIGN**）。如果结果为 0，那么说明当月第一天为星期一，因此它就是当月的第一个星期一。如果结果为-1，则说明当月第一天为星期日，因此要找到第一个星期一，只需加上 1（2-1）天，其中 2 和 1 分别是星期一和星期日的数字表示。



如果无法理解上述原理，那么请忘记一周中各天的名称，只管做数学运算。假设你要查找星期二后面的第一个星期五。调用函数 **TO_CHAR** 并将格式设置为 *d*（或调用函数 **DAYOFWEEK**）时，如果传入的日期为星期五，那么将返回 6，而传入的日期为星期二时将返回 3。要从 3 得到 6，只需计算它们的差（ $6 - 3 = 3$ ），再将这个差值与较小的值相加（ $(6 - 3) + 3 = 6$ ）。因此，不管具体的日期是什么，如果起始日的数字表示小于你要查找的星期 *n* 的数字表示，那么只要在起始日的基础上加上这两个数字的差，就可以得到要查找的星期 *n*。

如果 **SIGN** 的结果为 1，那么说明当月的第一天为星期二、星期三、星期四、星期五或星期六。当月第一天的数字表示大于星期一的数字表示 2 时，可以将 7 减去当月第一天的数字表示和星期一的数字表示 2 的差，然后将当月第一天加上这里计算得到的结果，就可以得到当月的第一个星期一。



同样，如果无法理解上述原理，那么请忘记一周中各天的名称，只管做数学运算。假设当月第一天为星期五，而你要查找下一个星期二。星期二的数字表示为 3，比星期五的数字表示 6 小。要从 6 得到 3，可将 7 减去这两个数字的差（ $7 - (|3 - 6|) = 4$ ），再将当月第一天加上得到的结果（4）。（注意， $|3 - 6|$ 中竖线表示计算差的绝对值。）在这里，不是给 4 加上 6（结果为

10)，而是在星期五的基础上加上 4 天，得到下一个星期二。

这个 **CASE** 表达式旨在在 PostgreSQL 和 MySQL 中模拟函数 **NEXT_DAY** 的效果。如果不从当月的第一天开始，那么 **DY** 将为 **CURRENT_DATE** 返回的值，因此这个 **CASE** 表达式将返回当前日期后面的第一个星期一。（除非 **CURRENT_DATE** 返回的日期是星期一，否则在这种情况下，返回的将是当前日期。）

有了当月的第一个星期一后，要找出最后一个星期一，可以加上 21 天或 28 天。第 2~5 行的 **CASE** 表达式检查加上 28 天后是否会进入下个月，进而决定加上 21 天还是 28 天。这个 **CASE** 表达式完成这项任务的过程如下。

- a. 给 **FIRST_MONDAY** 的值加上 28。
- b. 使用 PostgreSQL 中的 **TO_CHAR** 或 MySQL 中的 **MONTH**，**CASE** 表达式从 **FIRST_MONDAY + 28** 的结果中提取当月的名称。
- c. 将第 2 步的结果同内嵌视图返回的 **MTH** 进行比较，其中 **MTH** 是 **CURRENT_DATE** 所在月份的名称。如果这两个月份名相同，就说明需要加上 28 天，因此这个 **CASE** 表达式返回 **FIRST_MONDAY + 28**；如果这两个月份名不同，就说明不能加上 28 天，因此这个 **CASE** 表达式返回 **FIRST_MONDAY + 21**。鉴于各月包含的天数，只需考虑加上 28 天还是 21 天这两种情况。



可以扩展本解决方案，通过加上 7 天和 14 天来分别找出第二个和第三个星期一。

9.7 创建日历

1. 问题

你想创建当月的日历。该日历的格式应该与你桌上摆着的日历相同：包含沿水平方向排列的 7 列和沿垂直方向排列的 5 行。

2. 解决方案

下面的每种解决方案都稍有不同，但解决问题的方法相同：返回当月的每一天，然后将每周的各天合并成一行以创建日历。

日历有多种格式。例如，Unix 命令 **CAL** 生成的日历以从星期日到星期六的顺序排列一周的各天。这里的解决方案基于 ISO 周，生成的日历将一周各天以从星期一到星期日的顺序排列。熟悉这些解决方案后，你将发现，要调整日历的格式，只需在转置前调整 ISO 周指定的值。



使用 SQL 的格式设置功能来创建易于阅读的输出时，查询将更长。不要被冗长的查询吓到。如果将这里的查询分解并分别运行各个部分，你就会发现它们其实非常简单。

DB2

使用递归式 **WITH** 子句返回当月的每一天，然后使用 **CASE** 和 **MAX** 合并一周的各天。

```

1  with x(dy, dm, mth, dw, wk)
2  as (
3  select (current_date -day(current_date) day +1 day) dy,
4         day((current_date -day(current_date) day +1 day)) dm,
5         month(current_date) mth,
6         dayofweek(current_date -day(current_date) day +1 day)
dw,
7         week_iso(current_date -day(current_date) day +1 day) wk
8  from t1
9  union all
10 select dy+1 day, day(dy+1 day), mth,
11        dayofweek(dy+1 day), week_iso(dy+1 day)
12  from x
13  where month(dy+1 day) = mth
14  )
15  select max(case dw when 2 then dm end) as Mo,
16         max(case dw when 3 then dm end) as Tu,
17         max(case dw when 4 then dm end) as We,
18         max(case dw when 5 then dm end) as Th,
19         max(case dw when 6 then dm end) as Fr,
20         max(case dw when 7 then dm end) as Sa,
21         max(case dw when 1 then dm end) as Su
22  from x
23  group by wk
24  order by wk

```

Oracle

使用递归式 **CONNECT BY** 子句返回当月的每一天，然后使用 **CASE** 和 **MAX** 合并一周的各天。

```

1  with x
2  as (
3  select *
4  from (
5  select to_char(trunc(sysdate, 'mm')+level-1, 'iw') wk,
6         to_char(trunc(sysdate, 'mm')+level-1, 'dd') dm,
7         to_number(to_char(trunc(sysdate, 'mm')+level-1, 'd')) dw,
8         to_char(trunc(sysdate, 'mm')+level-1, 'mm') curr_mth,
9         to_char(sysdate, 'mm') mth
10  from dual
11  connect by level <= 31

```



```

12      )
13  where curr_mth = mth
14 )
15 select max(case dw when 2 then dm end) Mo,
16        max(case dw when 3 then dm end) Tu,
17        max(case dw when 4 then dm end) We,
18        max(case dw when 5 then dm end) Th,
19        max(case dw when 6 then dm end) Fr,
20        max(case dw when 7 then dm end) Sa,
21        max(case dw when 1 then dm end) Su
22  from x
23 group by wk
24 order by wk

```

PostgreSQL

使用函数 **GENERATE_SERIES** 返回当月的每一天，然后使用 **CASE** 和 **MAX** 合并一周的各天。

```

1 select max(case dw when 2 then dm end) as Mo,
2        max(case dw when 3 then dm end) as Tu,
3        max(case dw when 4 then dm end) as We,
4        max(case dw when 5 then dm end) as Th,
5        max(case dw when 6 then dm end) as Fr,
6        max(case dw when 7 then dm end) as Sa,
7        max(case dw when 1 then dm end) as Su
8  from (
9 select *
10  from (
11 select cast(date_trunc('month',current_date) as date)+x.id,
12        to_char(
13          cast(
14            date_trunc('month',current_date)
15              as date)+x.id,'iw') as wk,
16        to_char(
17          cast(
18            date_trunc('month',current_date)
19              as date)+x.id,'dd') as dm,
20        cast(
21          to_char(
22            cast(
23              date_trunc('month',current_date)

```

```

24             as date)+x.id,'d') as integer) as dw,
25         to_char(
26             cast(
27                 date_trunc('month',current_date)
28                 as date)+x.id,'mm') as curr_mth,
29         to_char(current_date,'mm') as mth
30     from generate_series (0,31) x(id)
31         ) x
32     where mth = curr_mth
33         ) y
34     group by wk
35     order by wk

```

MySQL

使用递归 CTE 返回当月的每一天，然后使用 **CASE** 和 **MAX** 合并一周的各天。

```

with recursive x(dy, dm, mth, dw, wk)
as (
    select dy,
           day(dy) dm,
           datepart(m, dy) mth,
           datepart(dw, dy) dw,
           case when datepart(dw, dy) = 1
                then datepart(ww, dy) - 1
                else datepart(ww, dy)
           end wk
    from (
        select date_add(day, -day(getdate()) + 1, getdate()) dy
        from t1
        ) x
    union all
    select dateadd(d, 1, dy), day(date_add(d, 1, dy)), mth,
           datepart(dw, dateadd(d, 1, dy)),
           case when datepart(dw, date_add(d, 1, dy)) = 1
                then datepart(wk, date_add(d, 1, dy)) - 1
                else datepart(wk, date_add(d, 1, dy))
           end
    from x
    where datepart(m, date_add(d, 1, dy)) = mth
)

```

```

select max(case dw when 2 then dm end) as Mo,
       max(case dw when 3 then dm end) as Tu,
       max(case dw when 4 then dm end) as We,
       max(case dw when 5 then dm end) as Th,
       max(case dw when 6 then dm end) as Fr,
       max(case dw when 7 then dm end) as Sa,
       max(case dw when 1 then dm end) as Su
  from x
 group by wk
 order by wk;

```

SQL Server

使用递归式 **WITH** 子句返回当月的每一天，然后使用 **CASE** 和 **MAX** 合并一周的各天。

```

1  with x(dy, dm, mth, dw, wk)
2    as (
3  select dy,
4         day(dy) dm,
5         datepart(m, dy) mth,
6         datepart(dw, dy) dw,
7         case when datepart(dw, dy) = 1
8              then datepart(dw, dy) - 1
9              else datepart(dw, dy)
10        end wk
11  from (
12  select dateadd(day, -day(getdate()) + 1, getdate()) dy
13  from t1
14        ) x
15  union all
16  select dateadd(d, 1, dy), day(dateadd(d, 1, dy)), mth,
17         datepart(dw, dateadd(d, 1, dy)),
18         case when datepart(dw, dateadd(d, 1, dy)) = 1
19              then datepart(dw, dateadd(d, 1, dy)) - 1
20              else datepart(dw, dateadd(d, 1, dy))
21        end
22  from x
23  where datepart(m, dateadd(d, 1, dy)) = mth
24 )
25 select max(case dw when 2 then dm end) as Mo,
26        max(case dw when 3 then dm end) as Tu,

```

```

27         max(case dw when 4 then dm end) as We,
28         max(case dw when 5 then dm end) as Th,
29         max(case dw when 6 then dm end) as Fr,
30         max(case dw when 7 then dm end) as Sa,
31         max(case dw when 1 then dm end) as Su
32     from x
33     group by wk
34     order by wk

```

13. 讨论

DB2

首先，返回要为其创建日历的月份中的每一天，为此使用了递归式 **WITH** 子句。除了该月份中的每一天（**DM**），还需要返回其他内容：星期几（**DW**）、当前月份（**MTH**）以及所属的 ISO 周（**WK**）。在递归发生前，递归视图 **X**（**UNION ALL** 前面的部分）返回的结果如下所示。

```

select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
from t1

```

DY	DM	MTH	DW	WK
01-JUN-2005	01	06	4	22

然后，不断增大 **DM** 的值，直到进入下个月（遍历当月的每一天）。在遍历月份中每一天的同时，还将返回当前日期是星期几以及它所属的 ISO 周。下面显示了部分结果。

```

with x(dy, dm, mth, dw, wk)
as (

```

```

select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
  from t1
union all
select dy+1 day, day(dy+1 day), mth,
       dayofweek(dy+1 day), week_iso(dy+1 day)
  from x
 where month(dy+1 day) = mth
)
select *
  from x

```

DY	DM	MTH	DW	WK
01-JUN-2020	01	06	4	22
02-JUN-2020	02	06	5	22
...				
21-JUN-2020	21	06	3	25
22-JUN-2020	22	06	4	25
...				
30-JUN-2020	30	06	5	26

至此，返回的结果如下：当前月份各天的日期和两位数表示，当前月份的两位数表示，当前月份各天为星期几的 1 位数表示（用 1~7 分别表示星期日到星期六）以及当前月份各天所属的 ISO 周的两位数表示。有了这些信息后，可以使用 **CASE** 表达式确定各天都是星期几。下面显示了返回的部分结果。

```

with x(dy, dm, mth, dw, wk)
  as (
select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
  from t1
union all

```

```

select dy+1 day, day(dy+1 day), mth,
       dayofweek(dy+1 day), week_iso(dy+1 day)
  from x
 where month(dy+1 day) = mth
)
select wk,
       case dw when 2 then dm end as Mo,
       case dw when 3 then dm end as Tu,
       case dw when 4 then dm end as We,
       case dw when 5 then dm end as Th,
       case dw when 6 then dm end as Fr,
       case dw when 7 then dm end as Sa,
       case dw when 1 then dm end as Su
  from x

```

WK	MO	TU	WE	TH	FR	SA	SU
--	--	--	--	--	--	--	--
22			01				
22				02			
22					03		
22						04	
22							05
23	06						
23		07					
23			08				
23				09			
23					10		
23						11	
23							12

从上述部分输出结果可知，每一天都作为一行返回。现在需要做的是按周分组，然后将属于同一周的所有日期合并成一行。为了将属于同一周的所有日期作为一行返回，可以使用函数 **MAX** 并按 **WK**（ISO 周）分组。为了确保日历的格式正确并按正确的顺序排列日期，按 **WK** 对结果进行了排序。最终的输出如下所示。

```

with x(dy, dm, mth, dw, wk)
  as (
select (current_date - day(current_date) day +1 day) dy,
       day((current_date - day(current_date) day +1 day)) dm,

```

```

        month(current_date) mth,
        dayofweek(current_date -day(current_date) day +1 day) dw,
        week_iso(current_date -day(current_date) day +1 day) wk
    from t1
union all
select dy+1 day, day(dy+1 day), mth,
        dayofweek(dy+1 day), week_iso(dy+1 day)
    from x
    where month(dy+1 day) = mth
)
select max(case dw when 2 then dm end) as Mo,
        max(case dw when 3 then dm end) as Tu,
        max(case dw when 4 then dm end) as We,
        max(case dw when 5 then dm end) as Th,
        max(case dw when 6 then dm end) as Fr,
        max(case dw when 7 then dm end) as Sa,
        max(case dw when 1 then dm end) as Su
    from x
    group by wk
    order by wk

MO TU WE TH FR SA SU
-- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

```

Oracle

首先，使用递归子句 **CONNECT BY** 为要为其创建日历的月份中的每一天都生成一行数据。如果你使用的版本低于 Oracle9i，那么将不能以这样的方式使用 **CONNECT BY**。在这种情况下，可以像其他平台中的解决方案那样使用递归 CET。

除月份中的每一天，还需返回有关各天的其他信息：它是当月的第几天（**DM**），是星期几（**DW**），所属的月份（**MTH**）以及所属的 ISO 周（**WK**）。对于当月的第一天，**WITH** 视图

X 返回的结果如下所示。

```
select trunc(sysdate,'mm') dy,
       to_char(trunc(sysdate,'mm'),'dd') dm,
       to_char(sysdate,'mm') mth,
       to_number(to_char(trunc(sysdate,'mm'),'d')) dw,
       to_char(trunc(sysdate,'mm'),'iw') wk
from dual
```

DY	DM	MT	DW	WK
01-JUN-2020	01	06	4	22

然后，不断增大 **DM** 的值，直到进入下个月（遍历当月的每一天）。遍历当月每一天的同时，还将返回当前日期是星期几以及它所属的 **ISO** 周。下面显示了部分输出结果（为提高可读性，添加了每天的完整日期信息）。

```
with x
as (
select *
from (
select trunc(sysdate,'mm')+level-1 dy,
       to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
from dual
connect by level <= 31
)
where curr_mth = mth
)
select *
from x
```

DY	WK	DM	DW	CU	MT
01-JUN-2020	22	01	4	06	06
02-JUN-2020	22	02	5	06	06
...					

21-JUN-2020 25 21	3 06 06
22-JUN-2020 25 22	4 06 06
...	
30-JUN-2020 26 30	5 06 06

至此，对于当月的每一天，你都返回了一行数据。这些行包含如下信息：指出当前日期为月份中第几天的两位数，月份的两位数表示，指出当前日期为星期几的 1 位数（用 1~7 分别表示星期日到星期六）以及当前日期所属 ISO 周的两位数表示。有了这些信息后，可以使用 **CASE** 表达式确定各天都是星期几。下面显示了返回的部分结果。

```
with x
  as (
select *
from (
select trunc(sysdate,'mm')+level-1 dy,
       to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
  from dual
 connect by level <= 31
        )
 where curr_mth = mth
)
select wk,
       case dw when 2 then dm end as Mo,
       case dw when 3 then dm end as Tu,
       case dw when 4 then dm end as We,
       case dw when 5 then dm end as Th,
       case dw when 6 then dm end as Fr,
       case dw when 7 then dm end as Sa,
       case dw when 1 then dm end as Su
  from x

WK MO TU WE TH FR SA SU
-- -- -- -- -- -- --
22      01
22      02
```

22			03			
22				04		
22					05	
23	06					
23		07				
23			08			
23				09		
23					10	
23						11
23						
23						12

从上述输出可知，每一天都作为一行被返回，而每一行都有7个表示星期几的列，但表示日期的数字只出现在其中一列中。现在的任务是将属于同一周的所有日期合并成一行。为将属于同一周的所有日期作为一行返回，可以使用聚合函数 **MAX** 并按 **WK**（ISO 周）分组。为了确保日历的格式正确并按正确的顺序排列日期，按 **WK** 对结果进行了排序。最终的输出如下所示。

```
with x
  as (
select *
  from (
select to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
  from dual
 connect by level <= 31
        )
 where curr_mth = mth
 )
select max(case dw when 2 then dm end) Mo,
       max(case dw when 3 then dm end) Tu,
       max(case dw when 4 then dm end) We,
       max(case dw when 5 then dm end) Th,
       max(case dw when 6 then dm end) Fr,
       max(case dw when 7 then dm end) Sa,
       max(case dw when 1 then dm end) Su
  from x
```

group by wk order by wk						
MO	TU	WE	TH	FR	SA	SU
--	--	--	--	--	--	--
		01	02	03	04	05
06	07	08	09	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

MySQL、PostgreSQL 和 SQL Server

这 3 种数据库的解决方案相同，唯一的差别是处理日期时使用的函数。这里的讨论以 SQL Server 解决方案为例。首先，对于月份中的每一天，都返回一行，为此可以使用递归式 **WITH** 子句。在返回的每一行中，都包含如下内容：月份中的第几天（**DM**），星期几（**DW**），当前月份（**MTH**）以及所属的 ISO 周（**WK**）。在递归发生前，递归视图 **X**（**UNION ALL** 前面的部分）返回的结果如下所示。

<pre> select dy, day(dy) dm, datepart(m,dy) mth, datepart(dw,dy) dw, case when datepart(dw,dy) = 1 then datepart(ww,dy)-1 else datepart(ww,dy) end wk from (select dateadd(day,-day(getdate())+1,getdate()) dy from t1) x </pre>						
DY		DM	MTH		DW	WK
-----		--	---		-----	--
01-JUN-2005		1	6		4	23

然后，不断增大 **DM** 的值，直到进入下个月（遍历当月的每

一天)。在遍历过程中，还将返回当前日期是星期几及其所属的 ISO 周。下面显示了部分结果。

```

with x(dy, dm, mth, dw, wk)
  as (
select dy,
      day(dy) dm,
      datepart(m, dy) mth,
      datepart(dw, dy) dw,
      case when datepart(dw, dy) = 1
            then datepart(dw, dy) - 1
            else datepart(dw, dy)
      end wk
  from (
select dateadd(day, -day(getdate()) + 1, getdate()) dy
  from t1
  ) x
 union all
select dateadd(d, 1, dy), day(dateadd(d, 1, dy)), mth,
      datepart(dw, dateadd(d, 1, dy)),
      case when datepart(dw, dateadd(d, 1, dy)) = 1
            then datepart(dw, dateadd(d, 1, dy)) - 1
            else datepart(dw, dateadd(d, 1, dy))
      end
  from x
 where datepart(m, dateadd(d, 1, dy)) = mth
 )
select *
  from x

```

DY	DM	MTH	DW	WK
01-JUN-2005	01	06	4	23
02-JUN-2005	02	06	5	23
...				
21-JUN-2005	21	06	3	26
22-JUN-2005	22	06	4	26
...				
30-JUN-2005	30	06	5	27

现在，对于当月的每一天，都有如下内容：指出它为月份中第几天的两位数，所属月份的两位数表示，指出它为星期几

的 1 位数（用 1~7 分别表示星期日到星期六）以及它所属 ISO 周的两位数表示。

接下来，使用 **CASE** 表达式确定各天都是星期几。下面显示了部分结果。

```
with x(dy, dm, mth, dw, wk)
  as (
select dy,
      day(dy) dm,
      datepart(m, dy) mth,
      datepart(dw, dy) dw,
      case when datepart(dw, dy) = 1
            then datepart(dw, dy) - 1
            else datepart(dw, dy)
      end wk
  from (
select dateadd(day, -day(getdate()) + 1, getdate()) dy
  from t1
    ) x
 union all
select dateadd(d, 1, dy), day(dateadd(d, 1, dy)), mth,
      datepart(dw, dateadd(d, 1, dy)),
      case when datepart(dw, dateadd(d, 1, dy)) = 1
            then datepart(dw, dateadd(d, 1, dy)) - 1
            else datepart(dw, dateadd(d, 1, dy))
      end
  from x
  where datepart(m, dateadd(d, 1, dy)) = mth
)
select case dw when 2 then dm end as Mo,
      case dw when 3 then dm end as Tu,
      case dw when 4 then dm end as We,
      case dw when 5 then dm end as Th,
      case dw when 6 then dm end as Fr,
      case dw when 7 then dm end as Sa,
      case dw when 1 then dm end as Su
  from x
```

WK	MO	TU	WE	TH	FR	SA	SU
--	--	--	--	--	--	--	--
22			01				

22		02		
22			03	
22				04
22				05
23	06			
23		07		
23			08	
23				09
23				10
23				11
23				12

从上述输出可知，每一天都作为一行被返回。在每一行中，表示日期的数字位于指出它是星期几的相应列中。现在需要将属于同一周的日子合并成一行，为此按 **WK**（ISO 周）分组，并将 **MAX** 函数应用于各列，结果为下面所示的日历格式。

```

with x(dy, dm, mth, dw, wk)
    as (
select dy,
       day(dy) dm,
       datepart(m, dy) mth,
       datepart(dw, dy) dw,
       case when datepart(dw, dy) = 1
            then datepart(wk, dy) - 1
            else datepart(wk, dy)
       end wk
    from (
select dateadd(day, -day(getdate()) + 1, getdate()) dy
    from t1
    ) x
union all
select dateadd(d, 1, dy), day(dateadd(d, 1, dy)), mth,
       datepart(dw, dateadd(d, 1, dy)),
       case when datepart(dw, dateadd(d, 1, dy)) = 1
            then datepart(wk, dateadd(d, 1, dy)) - 1
            else datepart(wk, dateadd(d, 1, dy))
       end
    from x
where datepart(m, dateadd(d, 1, dy)) = mth
)

```

```

select max(case dw when 2 then dm end) as Mo,
       max(case dw when 3 then dm end) as Tu,
       max(case dw when 4 then dm end) as We,
       max(case dw when 5 then dm end) as Th,
       max(case dw when 6 then dm end) as Fr,
       max(case dw when 7 then dm end) as Sa,
       max(case dw when 1 then dm end) as Su
  from x
 group by wk
 order by wk

```

```

MO TU WE TH FR SA SU
-- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

```

9.8 列出一年中各个季度的第一天和最后一天

1. 问题

你想返回给定年份中各个季度的第一天和最后一天。

2. 解决方案

一年有 4 个季度，因此需要生成 4 行数据。生成所需的行数后，只需使用 RDBMS 提供的日期函数来返回各个季度的第一天和最后一天。你的目标是生成如下结果集（这里使用的是当年，也可以使用任何年份）。

QTR	Q_START	Q_END
1	01-JAN-2020	31-MAR-2020
2	01-APR-2020	30-JUN-2020
3	01-JUL-2020	30-SEP-2020
4	01-OCT-2020	31-DEC-2020

DB2

使用 EMP 表和窗口函数 ROW_NUMBER OVER 来生成 4 行数据。也可以像很多实例那样使用 WITH 子句来生成行，或者查询任何至少包含 4 行的表。下面的解决方案使用了窗口函数 ROW_NUMBER OVER。

```
1 select quarter(dy-1 day) QTR,  
2          dy-3 month Q_start,  
3          dy-1 day Q_end  
4   from (  
5 select (current_date -  
6        (dayofyear(current_date)-1) day
```



```

7          + (rn*3) month) dy
8  from (
9  select row_number()over() rn
10  from emp
11  fetch first 4 rows only
12      ) x
13      ) y

```

Oracle

使用函数 **ADD_MONTHS** 找出各个季度的第一天和最后一天。使用 **ROWNUM** 表示季度编号。下面的解决方案使用 **EMP** 表来生成 4 行数据。

```

1 select rownum qtr,
2        add_months(trunc(sysdate,'y'),(rownum-1)*3) q_start,
3        add_months(trunc(sysdate,'y'),rownum*3)-1 q_end
4  from emp
5  where rownum <= 4

```

PostgreSQL

找出当前日期所属年份的第一天，并使用递归 CTE 填充其他 3 个季度的第一天，再找出每个季度的最后一天。

```

with recursive x (dy,cnt)
  as (
select
    current_date -cast(extract(day from current_date)as integer)
+1 dy
    , id
  from t1
 union all
select cast(dy + interval '3 months' as date) , cnt+1
  from x
 where cnt+1 <= 4
 )
select  cast(dy - interval '3 months' as date) as Q_start
        , dy-1 as Q_end
        from x

```

MySQL

找出当前日期所属年份的第一天，并使用 CTE 生成 4 行数据，每个季度 1 行。使用 **ADDDATE** 找出每个季度的最后一天（前一个季度的最后一天加上 3 个月或后一个季度的第一天减去 1 天）。

```
1      with recursive x (dy,cnt)
2  as (
3      select
4      adddate(current_date,(-dayofyear(current_date))+1) dy
5      ,id
6      from t1
7      union all
8      select adddate(dy, interval 3 month ), cnt+1
9      from x
10     where cnt+1 <= 4
11 )
12
13     select quarter(adddate(dy,-1)) QTR
14     , date_add(dy, interval -3 month) Q_start
15     , adddate(dy,-1) Q_end
16     from x
17     order by 1
```

SQL Server

使用递归式 **WITH** 子句生成 4 行数据。使用函数 **DATEADD** 找出各个季度的第一天和最后一天。使用函数 **DATEPART** 确定第一天和最后一天所属的季度。

```
1  with x (dy,cnt)
2  as (
3  select dateadd(d,-(datepart(dy,getdate())-1),getdate()),
4         1
5  from t1
6  union all
7  select dateadd(m,3,dy), cnt+1
```

```

8   from x
9   where cnt+1 <= 4
10 )
11 select datepart(q,dateadd(d,-1,dy)) QTR,
12         dateadd(m,-3,dy) Q_start,
13         dateadd(d,-1,dy) Q_end
14   from x
15 order by 1

```

13. 讨论

DB2

首先，生成 4 行数据，这些行包含值 1~4，表示一年的 4 个季度。内嵌视图 X 会使用窗口函数 **ROW_NUMBER OVER** 和 **FETCH FIRST** 子句返回 **EMP** 表的前 4 行。

```

select row_number()over() rn
   from emp
  fetch first 4 rows only

RN
--
1
2
3
4

```

然后，找出当年的第一天，再加上 n 个月，其中 n 为 **RN** 的 3 倍（在年份第一天的基础上分别加上 3 个月、6 个月、9 个月和 12 个月）。

```

select (current_date
       (dayofyear(current_date)-1) day
       + (rn*3) month) dy
   from (
select row_number()over() rn

```

```

    from emp
  fetch first 4 rows only
    ) x

DY
-----
01-APR-2005
01-JUL-2005
01-OCT-2005
01-JAN-2005

```

此时，DY 的值为各季度最后一天的后面一天。接下来，找出各个季度的第一天和最后一天。为此，将 DY 减去 1 天，得到各个季度的最后一天，将 DY 减去 3 个月，得到各个季度的第一天。为了确定所属的季度，对 DY-1（各个季度的最后一天）应用了函数 QUARTER。

Oracle

结合使用 ROWNUM、TRUNC 和 ADD_MONTHS 可以轻松地解决这个问题。为了找出每个季度的第一天，只需将相应年份的第一天加上 n 个月，其中 n 为 $(\text{ROWNUM}-1)*3$ （0、3、6、9）。为了找出每个季度的最后一天，只需将相应年份的第一天加上 n 个月，其中 n 为 $\text{ROWNUM}*3$ 再减去 1 天。在处理季度时，使用 TO_CHAR 和/或 TRUNC 并将格式选项设置为 Q 可能很有用。

PostgreSQL、MySQL 和 SQL Server

与前面的某些实例一样，在本实例中，这 3 种 RDBMS 解决方案的结构也相同，但日期操作语法不同。首先，找出给定年份的第一天，然后使用 DATEADD 或其等价函数递归地加上 n 个月，其中 n 为当前迭代的 3 倍（有 4 次迭代，因此将加上 $3*1$ 个月、 $3*2$ 个月等）。

```

with x (dy,cnt)

```

```

    as (
select dateadd(d,-(datepart(dy,getdate())-1),getdate()),
       1
  from t1
 union all
select dateadd(m,3,dy), cnt+1
  from x
 where cnt+1 <= 4
 )
select dy
  from x

DY
-----
01-APR-2020
01-JUL-2020
01-OCT-2020
01-JAN-2020

```

DY 的值为各个季度最后一天的后面一天。要找出各个季度的最后一天，只需使用函数 **DATEADD** 将 DY 减去 1 天。要找出各个季度的第一天，可以使用函数 **DATEADD** 将 DY 减去 3 个月。为了确定所属的季度，可以将函数 **DATEPART** 应用于各个季度的最后一天。如果你使用的是 PostgreSQL，那么在给起始日期加上 3 个月后，需要使用 **CAST** 确保数据类型一致，否则递归 CET 中的 **UNION ALL** 会因数据类型不一致而失败。

9.9 确定给定季度的第一天和最后一天

1. 问题

在以 YYYYQ 格式（4 位的年份和 1 位的季度）给定年份和季度的情况下，你想返回该季度的第一天和最后一天。

2. 解决方案

本解决方案的关键在于使用求模函数根据 YYYYQ 值找出季度。（也可以不使用求模函数，而直接提取最后一位，因为年份是 4 位的。）有了表示季度的数字后，只需将其乘以 3，就可得到该季度的最后一个月。在下面的解决方案中，内嵌视图 X 返回了年份和季度的 4 种组合。换言之，内嵌视图 X 返回的结果集如下所示。

```
select 20051 as yrq from t1 union all
select 20052 as yrq from t1 union all
select 20053 as yrq from t1 union all
select 20054 as yrq from t1
  YRQ
-----
  20051
  20052
  20053
  20054
```

DB2

使用函数 SUBSTR 从内嵌视图 X 返回的结果中提取年份，并使用函数 MOD 确定季度。

```
1 select (q_end-2 month) q_start,
```

```

2      (q_end+1 month)-1 day q_end
3  from (
4  select date(substr(cast(yrq as char(4)),1,4) ||'-'||
5      rtrim(cast(mod(yrq,10)*3 as char(2))) ||'-1') q_end
6  from (
7  select 20051 yrq from t1 union all
8  select 20052 yrq from t1 union all
9  select 20053 yrq from t1 union all
10 select 20054 yrq from t1
11      ) x
12      ) y

```

Oracle

使用函数 **SUBSTR** 从内嵌视图 **X** 返回的结果中提取年份，并使用函数 **MOD** 确定季度。

```

1  select add_months(q_end,-2) q_start,
2      last_day(q_end) q_end
3  from (
4  select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') q_end
5  from (
6  select 20051 yrq from dual union all
7  select 20052 yrq from dual union all
8  select 20053 yrq from dual union all
9  select 20054 yrq from dual
10      ) x
11      ) y

```

PostgreSQL

使用函数 **SUBSTR** 从内嵌视图 **X** 返回的结果中提取年份，并使用函数 **MOD** 确定季度。

```

1  select date(q_end-(2*interval '1 month')) as q_start,
2      date(q_end+interval '1 month'-interval '1 day') as q_end
3  from (
4  select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') as
q_end
5  from (

```

```

6 select 20051 as yrq from t1 union all
7 select 20052 as yrq from t1 union all
8 select 20053 as yrq from t1 union all
9 select 20054 as yrq from t1
10      ) x
11      ) y

```

MySQL

使用函数 **SUBSTR** 从内嵌视图 **X** 返回的结果中提取年份，并使用函数 **MOD** 确定季度。

```

1 select date_add(
2       adddate(q_end, -day(q_end)+1),
3       interval -2 month) q_start,
4       q_end
5   from (
6 select last_day(
7       str_to_date(
8       concat(
9       substr(yrq,1,4),mod(yrq,10)*3), '%Y%m')) q_end
10  from (
11 select 20051 as yrq from t1 union all
12 select 20052 as yrq from t1 union all
13 select 20053 as yrq from t1 union all
14 select 20054 as yrq from t1
15      ) x
16      ) y

```

SQL Server

使用函数 **SUBSTRING** 从内嵌视图 **X** 返回的结果中提取年份，并使用求模运算符 **%** 确定季度。

```

1 select dateadd(m,-2,q_end) q_start,
2       dateadd(d,-1,dateadd(m,1,q_end)) q_end
3   from (
4 select cast(substring(cast(yrq as varchar),1,4)+'-' +
5       cast(yrq%10*3 as varchar)+'-1' as datetime) q_end
6   from (

```



```

7 select 20051 as yrq from t1 union all
8 select 20052 as yrq from t1 union all
9 select 20053 as yrq from t1 union all
10 select 20054 as yrq from t1
11      ) x
12      ) y

```

13. 讨论

DB2

首先，找出年份和季度。使用函数 **SUBSTR** 从内嵌视图 **X** 返回的 **YRQ** 中提取年份。为了获取季度，计算 **YRQ** 除以 10 的余数。有了表示季度的数字后，将其乘以 3 得到该季度的最后一个月份。

```

select substr(cast(yrq as char(4)),1,4) yr,
       mod(yrq,10)*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
      ) x

```

YR	MTH
----	-----
2005	3
2005	6
2005	9
2005	12

有了年份以及各个季度的最后一个月份后，根据这些值来创建日期，具体地说是各季度最后一个月份的第一天的日期。为此，使用拼接运算符 **||** 将年份和月份拼接起来，然后使用函数 **DATE** 将结果转换为日期。

```

select date(substr(cast(yrq as char(4)),1,4) || '-' ||
            rtrim(cast(mod(yrq,10)*3 as char(2))) || '-1') q_end
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
      ) x

```

```

Q_END
-----
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005

```

Q_END 的值为各个季度最后一个月的第一天。为了获得这些月份的最后一天，将 **Q_END** 加上 1 个月再减去 1 天。为了找出各个季度的第一天，将 **Q_END** 减去 2 个月。

Oracle

首先，找出年份和季度。使用函数 **SUBSTR** 从内嵌视图 **X** 返回的 **YRQ** 中提取年份。为了获取季度，计算 **YRQ** 除以 10 的余数。有了表示季度的数字后，将其乘以 3 得到该季度的最后一个月份。

```

select substr(yrq,1,4) yr, mod(yrq,10)*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
      ) x

```

```

YR      MTH
-----
2005      3
2005      6
2005      9

```

有了年份以及各个季度的最后一个月份后，根据这些值来创建日期，具体地说是各季度最后一个月份的第一天的日期。为此，使用拼接运算符 || 将年份和月份拼接起来，然后使用函数 **TO_DATE** 将结果转换为日期。

```
select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') q_end
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
      ) x

Q_END
-----
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

Q_END 的值为各个季度最后一个月份的第一天。为了获得这些月份的最后一天，将函数 **LAST_DAY** 应用于 **Q_END**。为了找出各个季度的第一天，使用函数 **ADD_MONTHS** 将 **Q_END** 减去 2 个月。

PostgreSQL

首先，找出年份和季度。使用函数 **SUBSTR** 从内嵌视图 **X** 返回的 **YRQ** 中提取年份。为了获取季度，计算 **YRQ** 除以 10 的余数。有了表示季度的数字后，将其乘以 3 得到该季度的最后一个月份。

```
select substr(cast(yrq as varchar),1,4) yr, mod(yrq,10)*3 mth
  from (
select 20051 yrq from t1 union all
```

```
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
) x
```

YR	MTH
2005	3
2005	6
2005	9
2005	12

有了年份以及各个季度的最后一个月份后，根据这些值来创建日期，具体地说是各季度最后一个月份的第一天的日期。为此，使用拼接运算符 || 将年份和月份拼接起来，然后使用函数 **TO_DATE** 将结果转换为日期。

```
select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') q_end
from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
) x
```

Q_END
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005

Q_END 的值为各个季度最后一个月份的第一天。为了获得这些月份的最后一天，将 **Q_END** 加上 1 个月再减去 1 天。为了找出各个季度的第一天，将 **Q_END** 减去 2 个月。最后将最终结果转换为日期。

MySQL

首先，找出年份和季度。使用函数 **SUBSTR** 从内嵌视图 **X** 返回的 **YRQ** 中提取年份。为了获取季度，计算 **YRQ** 除以 10 的余数。有了表示季度的数字后，将其乘以 3 得到该季度的最后一个月份。

```
select substr(cast(yrq as varchar),1,4) yr, mod(yrq,10)*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
  ) x
```

YR	MTH
----	-----
2005	3
2005	6
2005	9
2005	12

有了年份以及各个季度的最后一个月份后，根据这些值来创建日期，具体地说是各季度的最后一天。为此，使用函数 **CONCAT** 将年份和月份拼接起来，然后使用函数 **STR_TO_DATE** 将结果转换为日期，再使用函数 **LAST_DAY** 找出各个季度的最后一天。

```
select last_day(
  str_to_date(
    concat(
      substr(yrq,1,4),mod(yrq,10)*3),'%Y%m')) q_end
  from (
select 20051 as yrq from t1 union all
select 20052 as yrq from t1 union all
select 20053 as yrq from t1 union all
select 20054 as yrq from t1
  ) x
```

Q_END

31-MAR-2005

30-JUN-2005
30-SEP-2005
31-DEC-2005

有了各个季度的最后一天后，余下的唯一工作是找出各个季度的第一天。为此，使用函数 **DAY** 确定各个季度的最后一天是其所属月份的第几天，再使用函数 **ADDDATE** 将 **Q_END** 减去这个天数，得到上个月的第一天。然后，加上 1 天，得到各个季度最后一个月份的第一天。最后，使用函数 **DATE_ADD** 将各个季度最后一个月份的第一天减去 2 个月，得到各个季度的第一天。

SQL Server

首先，找出年份和季度。使用函数 **SUBSTR** 从内嵌视图 **X** 返回的 **YRQ** 中提取年份。为了获取季度，计算 **YRQ** 除以 10 的余数。有了表示季度的数字后，将其乘以 3 得到该季度的最后一个月份。

```
select substring(yrq,1,4) yr, yrq%10*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
  ) x
```

YR	MTH
----	-----
2005	3
2005	6
2005	9
2005	12

有了年份以及各个季度的最后一个月份后，根据这些值来创建日期，具体地说是各季度最后一个月份的第一天的日期。为此，使用拼接运算符 **+** 将年份和月份拼接起来，然后使用

函数 **CAST** 将结果转换为日期。

```
select cast(substring(cast(yrq as varchar),1,4)+'-' +
              cast(yrq%10*3 as varchar)+'-1' as datetime) q_end
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
  ) x

Q_END
-----
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

Q_END 的值为各个季度最后一个月的第一天。为了获得这些月份的最后一天，使用函数 **DATEADD** 将 **Q_END** 加上 1 个月再减去 1 天。为了找出各个季度的第一天，使用函数 **DATEADD** 将 **Q_END** 减去 2 个月。

9.10 补全缺失的日期

1. 问题

对于给定时间范围内的每个日期（每个月、每周或每年），你都要生成一行数据。例如，对于所有聘请了员工的年份，你要计算每个月聘请的员工数量。从所有员工的获聘日期可知，在 2000 年到 2003 年期间，每一年都聘请了员工。

```
select distinct
    extract(year from hiredate) as year
from emp
```

```
YEAR
-----
2000
2001
2002
2003
```

你想计算从 2000 年到 2003 年，每个月聘请的员工数量。下面显示了要返回的结果集的一部分。

MTH	NUM_HIRED
-----	-----
01-JAN-2001	0
01-FEB-2001	2
01-MAR-2001	0
01-APR-2001	1
01-MAY-2001	1
01-JUN-2001	1
01-JUL-2001	0
01-AUG-2001	0
01-SEP-2001	2
01-OCT-2001	0
01-NOV-2001	1
01-DEC-2001	2

12. 解决方案

这里比较棘手的地方是，对于那些没有聘请员工的月份，也需要返回相应的行，但其中的员工聘请数量为 0。由于在 2000 年到 2003 年期间，并非每个月都聘请了员工，因此你必须生成所有的月份，然后基于 **HIREDATE** 外连接到 **EMP** 表（将 **HIREDATE** 截断到月份，使其能够与生成的月份匹配）。

DB2

使用递归式 **WITH** 子句生成每个月份（从 2000 年 1 月 1 日到 2003 年 12 月 31 日的每个月份的第一天）。有了指定时间范围内的每个月份后，外连接到 **EMP** 表，并使用聚合函数 **COUNT** 计算每个月聘请的员工数量。

```
1  with x (start_date,end_date)
2    as (
3  select (min(hiredate)
4         dayofyear(min(hiredate)) day +1 day) start_date,
5         (max(hiredate)
6         dayofyear(max(hiredate)) day +1 day) +1 year end_date
7  from emp
8  union all
9  select start_date +1 month, end_date
10 from x
11 where (start_date +1 month) < end_date
12 )
13 select x.start_date mth, count(e.hiredate) num_hired
14 from x left join emp e
15 on (x.start_date = (e.hiredate-(day(hiredate)-1) day))
16 group by x.start_date
17 order by 1
```

Oracle

使用 **CONNECT BY** 子句生成 2000 年到 2003 年期间的每个月份，然后外连接到 **EMP** 表，并使用聚合函数 **COUNT** 计算每个月聘请的员工数量。

```
1  with x
2    as (
3  select add_months(start_date,level-1) start_date
4    from (
5  select min(trunc(hiredate,'y')) start_date,
6         add_months(max(trunc(hiredate,'y')),12) end_date
7    from emp
8         )
9  connect by level <= months_between(end_date,start_date)
10 )
11 select x.start_date MTH, count(e.hiredate) num_hired
12    from x left join emp e
13         on (x.start_date = trunc(e.hiredate,'mm'))
14  group by x.start_date
15  order by 1
```

PostgreSQL

使用 **CTE** 生成从最早聘请日期开始的月份，然后根据年份和月份左外连接到 **EMP** 表，以计算每个月聘请的员工数量。

```
        with recursive x (start_date, end_date)
as
(
    select
        cast(min(hiredate) - (cast(extract(day from min(hiredate))
        as integer) - 1) as date)
        , max(hiredate)
    from emp
    union all
        select cast(start_date + interval '1 month' as date)
        , end_date
    from x
    where start_date < end_date
)

select x.start_date,count(hiredate)
```

```

from x left join emp
on (extract(month from start_date) =
    extract(month from emp.hiredate)
    and extract(year from start_date)
    = extract(year from emp.hiredate))
group by x.start_date
order by 1

```

MySQL

使用递归式 CTE 生成起始日期和终止日期之间的所有月份，然后连接到 **EMP** 表，以计算每个月聘请的员工数量。

```

with recursive x (start_date,end_date)
as
(
select
    adddate(min(hiredate),
    -dayofyear(min(hiredate))+1) start_date
    ,adddate(max(hiredate),
    -dayofyear(max(hiredate))+1) end_date
from emp
union all
select date_add(start_date,interval 1 month)
, end_date
from x
where date_add(start_date, interval 1 month) < end_date
)

select x.start_date mth, count(e.hiredate) num_hired
from x left join emp e
on (extract(year_month from start_date)
    =
    extract(year_month from e.hiredate))
group by x.start_date
order by 1;

```

SQL Server

使用递归式 **WITH** 子句生成每个月份（从 2000 年 1 月 1 日到

2003 年 12 月 31 日的每个月份的第一天)。有了指定时间范围内的所有月份后，外连接到 **EMP** 表，并使用聚合函数 **COUNT** 计算每个月聘请的员工数量。

```
1  with x (start_date,end_date)
2    as (
3  select (min(hiredate) -
4         datepart(dy,min(hiredate))+1) start_date,
5         dateadd(yy,1,
6         (max(hiredate) -
7         datepart(dy,max(hiredate))+1)) end_date
8  from emp
9  union all
10 select dateadd(mm,1,start_date), end_date
11  from x
12  where dateadd(mm,1,start_date) < end_date
13 )
14 select x.start_date mth, count(e.hiredate) num_hired
15  from x left join emp e
16    on (x.start_date =
17        dateadd(dd,-day(e.hiredate)+1,e.hiredate))
18  group by x.start_date
19  order by 1
```

13. 讨论

DB2

首先，生成 2000 年到 2003 年期间的每个月份（实际上是每个月的的第一天）。为此，先找出边界月份，方法是对 **HIREDATE** 应用函数 **MIN** 和 **MAX**，再对结果应用函数 **DAYOFYEAR**。

```
select (min(hiredate)
        dayofyear(min(hiredate)) day +1 day) start_date,
       (max(hiredate)
        dayofyear(max(hiredate)) day +1 day) +1 year end_date
```

```

from emp

START_DATE  END_DATE
-----
01-JAN-2000 01-JAN-2004

```

然后，不断地给 **START_DATE** 加 1 个月，以返回所需的所有月份。**END_DATE** 的值多了 1 天，这没问题，因为在递归地给 **START_DATE** 加 1 个月时，可以在到达 **END_DATE** 前停止。下面显示了创建的部分月份。

```

with x (start_date,end_date)
  as (
select (min(hiredate)
        dayofyear(min(hiredate)) day +1 day) start_date,
       (max(hiredate)
        dayofyear(max(hiredate)) day +1 day) +1 year end_date
  from emp
 union all
select start_date +1 month, end_date
  from x
 where (start_date +1 month) < end_date
 )
select *
  from x

START_DATE  END_DATE
-----
01-JAN-2000 01-JAN-2004
01-FEB-2000 01-JAN-2004
01-MAR-2000 01-JAN-2004
...
01-OCT-2003 01-JAN-2004
01-NOV-2003 01-JAN-2004
01-DEC-2003 01-JAN-2004

```

有了需要的所有月份后，就可以外连接到 **EMP.HIREDATE** 了。由于每一个 **START_DATE** 值都是所需月份的第一天，因此将 **EMP.HIREDATE** 截断为其所属月份的第一天。最后，将聚合函数 **COUNT** 应用于 **EMP.HIREDATE**。

Oracle

首先，生成 2000 年到 2003 年期间每个月份的第一天。为此，先对 **HIREDATE** 结合使用 **MIN**、**MAX**、**TRUNC** 和 **ADD_MONTHS** 来找出边界月份。

```
select min(trunc(hiredate,'y')) start_date,  
       add_months(max(trunc(hiredate,'y')),12) end_date  
from emp
```

```
START_DATE  END_DATE  
-----  
01-JAN-2000 01-JAN-2004
```

然后，不断地给 **START_DATE** 加 1 个月来生成必要的月份。**END_DATE** 的值多了 1 天，这没问题，因为在递归地给 **START_DATE** 加 1 个月时，可以在到达 **END_DATE** 前停止。下面显示了创建的部分月份。

```
with x as (  
  select add_months(start_date,level-1) start_date  
    from (  
  select min(trunc(hiredate,'y')) start_date,  
         add_months(max(trunc(hiredate,'y')),12) end_date  
    from emp  
    )  
  connect by level <= months_between(end_date,start_date)  
)  
select *  
from x
```

```
START_DATE  
-----  
01-JAN-2000  
01-FEB-2000  
01-MAR-2000  
...  
01-OCT-2003  
01-NOV-2003  
01-DEC-2003
```

有了需要的所有月份后，就可以外连接到 **EMP.HIREDATE** 了。由于每一个 **START_DATE** 值都是所需月份的第一天，因此将 **EMP.HIREDATE** 截断为其所属月份的第一天。最后，将聚合函数 **COUNT** 应用于 **EMP.HIREDATE**。

PostgreSQL

该解决方案使用 CTE 来生成所需的月份，与接下来的 MySQL 和 SQL Server 解决方案类似。先使用聚合函数生成边界日期，为此可以使用函数 **MIN()** 和 **MAX()** 找出最早和最晚的聘请日期，但如果找出最早聘请日期所属月份的第一天，那么输出将更容易理解。

MySQL

首先，结合使用聚合函数 **MIN** 和 **MAX** 以及函数 **DAYOFYEAR** 和 **ADDDATE** 找出边界日期。内嵌视图 **X** 返回的结果集如下所示。

```
with recursive x (start_date,end_date)
as (
  select
    adddate(min(hiredate),
      -dayofyear(min(hiredate))+1) start_date
    ,adddate(max(hiredate),
      -dayofyear(max(hiredate))+1) end_date
  from emp
  union all
  select date_add(start_date,interval 1 month)
    , end_date
  from x
  where date_add(start_date, interval 1 month) < end_date
)
select * from x

select adddate(min(hiredate),-dayofyear(min(hiredate))+1)
min_hd,
      adddate(max(hiredate),-dayofyear(max(hiredate))+1)
```

```

max_hd
      from emp

      MIN_HD      MAX_HD
      -----
01-JAN-2000 01-JAN-2003

```

然后，CTE 不断地加上 1 个月，直到到达 **MAX_HD** 所属年份的最后一个月。

```

MTH
-----
01-JAN-2000
01-FEB-2000
01-MAR-2000
...
01-OCT-2003
01-NOV-2003
01-DEC-2003

```

有了所需的所有月份后，外连接到 **EMP.HIREDATE**（务必将 **EMP.HIREDATE** 截断为其所属月份的第一天），并将聚合函数 **COUNT** 应用于 **EMP.HIREDATE**，以计算每个月聘请的员工数量。

SQL Server

首先，生成 2000 年到 2003 年期间的每个月份（实际上是每个月的第一天）。为此，先找出边界月份，方法是对 **HIREDATE** 应用函数 **MIN** 和 **MAX**，再对结果应用函数 **DAYOFYEAR**。

```

select (min(hiredate) -
       datepart(dy,min(hiredate))+1) start_date,
       dateadd(yy,1,
       (max(hiredate) -
       datepart(dy,max(hiredate))+1)) end_date
from emp

```


START_DATE	END_DATE

01-JAN-2000	01-JAN-2004

然后，不断地给 **START_DATE** 加上 1 个月，以返回所需的所有月份。**END_DATE** 的值多了 1 天，这没问题，因为在递归地给 **START_DATE** 加 1 个月时，可以在到达 **END_DATE** 前停止。下面显示了创建的部分月份。

```
with x (start_date,end_date)
  as (
select (min(hiredate) -
        datepart(dy,min(hiredate))+1) start_date,
        dateadd(yy,1,
        (max(hiredate) -
        datepart(dy,max(hiredate))+1)) end_date
  from emp
 union all
select dateadd(mm,1,start_date), end_date
  from x
 where dateadd(mm,1,start_date) < end_date
 )
select *
  from x
```

START_DATE	END_DATE

01-JAN-2000	01-JAN-2004
01-FEB-2000	01-JAN-2004
01-MAR-2000	01-JAN-2004
...	
01-OCT-2003	01-JAN-2004
01-NOV-2003	01-JAN-2004
01-DEC-2003	01-JAN-2004

有了需要的所有月份后，就可以外连接到 **EMP.HIREDATE** 了。由于每一个 **START_DATE** 值都是所需月份的第一天，因此将 **EMP.HIREDATE** 截断为其所属月份的第一天。最后，将聚合函数 **COUNT** 应用于 **EMP.HIREDATE**。

9.11 根据日期的特定部分进行查找

1. 问题

你想查找与给定月份、星期 n 或其他时间单位匹配的日期。例如，你想找出获聘日期为 2 月份或 12 月份以及为星期二的员工。

2. 解决方案

使用 RDBMS 提供的函数确定日期所属的月份或星期 n 。在很多场景下，这里介绍的解决方案很有用。假设你要根据 **HIREDATE** 进行检索，但想忽略年份（只提取月份或 **HIREDATE** 的其他部分），就可以使用这里的解决方案。这些解决方案根据日期所属的月份或为星期 n 进行检索。如果你熟悉 RDBMS 提供的日期函数，则可以轻松地修改这些解决方案，以根据年份、季度、年份和季度组合、月份和年份组合等进行检索。

DB2 和 MySQL

使用函数 **MONTHNAME** 和 **DAYNAME** 分别确定员工获聘日期所属月份以及为星期几。

```
1 select ename
2   from emp
3  where monthname(hiredate) in ('February','December')
4     or dayname(hiredate) = 'Tuesday'
```

Oracle 和 PostgreSQL

使用函数 **TO_CHAR** 确定员工获聘日期所属月份以及为星期几，使用函数 **RTRIM** 删除末尾的空白。

```
1 select ename
2   from emp
3  where rtrim(to_char(hiredate,'month')) in
4         ('february','december')
5         or rtrim(to_char(hiredate,'day')) = 'tuesday'
```

SQL Server

使用函数 **DATENAME** 确定员工获聘日期所属月份以及为星期几。

```
1 select ename
2   from emp
3  where datename(m,hiredate) in ('February','December')
4         or datename(dw,hiredate) = 'Tuesday'
```

13. 讨论

对于以上所有解决方案，关键在于知道该使用哪些函数以及如何使用它们。要查看返回的值，可以将函数调用放在 **SELECT** 子句，并查看输出。下面是 10 号部门员工的结果集（这里使用的是 SQL Server 语法）。

```
select ename,datename(m,hiredate) mth,datename(dw,hiredate) dw
   from emp
  where deptno = 10
```

ENAME	MTH	DW
CLARK	June	Tuesday
KING	November	Tuesday
MILLER	January	Saturday

知道函数的返回值后，就可以轻松获悉上述解决方案返回的结果了。

9.12 根据日期的特定部分对记录进行比较

1. 问题

你想查找哪些员工的获聘日期属于同一个月份且所属的星期 n 相同。如果一位员工的获聘日期为 2008 年 3 月 10 日（星期一），而另一位员工的获聘日期为 2001 年 3 月 2 日（星期一），那么就认为他们匹配，因为所属的月份和星期 n 都相同。在 **EMP** 表中，只有 3 位员工满足这些条件，而你想返回下面的结果集。

```
MSG
-----
JAMES was hired on the same month and weekday as FORD
SCOTT was hired on the same month and weekday as JAMES
SCOTT was hired on the same month and weekday as FORD
```

2. 解决方案

由于要对任意两位员工的 **HIREDATE** 进行比较，因此需要自连接 **EMP** 表。这将提供各种可能的 **HIREDATE** 组合以进行比较。自连接后，只需从每个 **HIREDATE** 中提取月份和星期 n ，并进行比较。

DB2

自连接 **EMP** 表后，使用函数 **DAYOFWEEK** 返回日期所属星期 n 的数字表示，使用函数 **MONTHNAME** 返回日期所属月份的名称。

```
1 select a.ename ||
2           ' was hired on the same month and weekday as ' ||
```

```
3      b.ename msg
4  from emp a, emp b
5 where (dayofweek(a.hiredate),monthname(a.hiredate)) =
6      (dayofweek(b.hiredate),monthname(b.hiredate))
7  and a.empno < b.empno
8 order by a.ename
```

Oracle 和 PostgreSQL

自连接 **EMP** 表后，使用函数 **TO_CHAR** 将 **HIREDATE** 格式化为星期 *n* 和月份，以便进行比较。

```
1 select a.ename ||
2      ' was hired on the same month and weekday as ' ||
3      b.ename as msg
4  from emp a, emp b
5 where to_char(a.hiredate,'DMON') =
6      to_char(b.hiredate,'DMON')
7  and a.empno < b.empno
8 order by a.ename
```

MySQL

自连接 **EMP** 表后，使用函数 **DATE_FORMAT** 将 **HIREDATE** 格式化为星期 *n* 和月份，以便进行比较。

```
1 select concat(a.ename,
2      ' was hired on the same month and weekday as ',
3      b.ename) msg
4  from emp a, emp b
5  where date_format(a.hiredate,'%w%M') =
6      date_format(b.hiredate,'%w%M')
7  and a.empno < b.empno
8 order by a.ename
```

SQL Server

自连接 **EMP** 表后，使用函数 **DATENAME** 将 **HIREDATE** 格式化

为星期 n 和月份，以便进行比较。

```
1 select a.ename +  
2         ' was hired on the same month and weekday as '+  
3         b.ename msg  
4   from emp a, emp b  
5  where datename(dw,a.hiredate) = datename(dw,b.hiredate)  
6        and datename(m,a.hiredate) = datename(m,b.hiredate)  
7        and a.empno < b.empno  
8  order by a.ename
```

13. 讨论

上述几种解决方案之间唯一的差别在于设置 **HIREDATE** 时使用的日期函数不同。下面将以 Oracle/PostgreSQL 解决方案为例（因为它们包含的代码最少），但这里的讨论适用于所有解决方案。

首先，自连接 **EMP**，以便对任意两位员工的 **HIREDATE** 进行比较。请看下面的查询结果（通过筛选只留下了与 **SCOTT** 相关的结果）。

```
select a.ename as scott, a.hiredate as scott_hd,  
       b.ename as other_emps, b.hiredate as other_hds  
  from emp a, emp b  
 where a.ename = 'SCOTT'  
       and a.empno != b.empno
```

SCOTT	SCOTT_HD	OTHER_EMPS	OTHER_HDS
SCOTT	09-DEC-2002	SMITH	17-DEC-2000
SCOTT	09-DEC-2002	ALLEN	20-FEB-2001
SCOTT	09-DEC-2002	WARD	22-FEB-2001
SCOTT	09-DEC-2002	JONES	02-APR-2001
SCOTT	09-DEC-2002	MARTIN	28-SEP-2001
SCOTT	09-DEC-2002	BLAKE	01-MAY-2001
SCOTT	09-DEC-2002	CLARK	09-JUN-2001

SCOTT	09-DEC-2002	KING	17-NOV-2001
SCOTT	09-DEC-2002	TURNER	08-SEP-2001
SCOTT	09-DEC-2002	ADAMS	12-JAN-2003
SCOTT	09-DEC-2002	JAMES	03-DEC-2001
SCOTT	09-DEC-2002	FORD	03-DEC-2001
SCOTT	09-DEC-2002	MILLER	23-JAN-2002

通过自连接 EMP，可以将 SCOTT 的 HIREDATE 同其他所有员工的 HIREDATE 进行比较。使用基于 EMPNO 的筛选器，可以避免将 OTHER_HDS 列为 SCOTT 的 HIREDATE 的行返回。然后，使用 RDBMS 提供的日期格式设置函数来比较两个 HIREDATE 所属的星期 *n* 和月份，并只保留匹配的行。

<pre> select a.ename as emp1, a.hiredate as emp1_hd, b.ename as emp2, b.hiredate as emp2_hd from emp a, emp b where to_char(a.hiredate,'DMON') = to_char(b.hiredate,'DMON') and a.empno != b.empno order by 1 </pre>			
EMP1	EMP1_HD	EMP2	EMP2_HD
-----	-----	-----	-----
FORD	03-DEC-2001	SCOTT	09-DEC-2002
FORD	03-DEC-2001	JAMES	03-DEC-2001
JAMES	03-DEC-2001	SCOTT	09-DEC-2002
JAMES	03-DEC-2001	FORD	03-DEC-2001
SCOTT	09-DEC-2002	JAMES	03-DEC-2001
SCOTT	09-DEC-2002	FORD	03-DEC-2001

至此，我们正确地找出了 HIREDATE 匹配的员工，但结果集包含 6 行数据，而不是本节“问题”部分要求的 3 行。导致出现多余行的“罪魁祸首”是基于 EMPNO 的筛选器。使用“不等于”作为筛选条件不能将互逆的行剔除，例如，上述第一行显示 FORD 与 SCOTT 匹配，而最后一行显示 SCOTT 与 FORD 匹配。从技术上说，上述结果集中的 6 行数据是准确的，但显得多余。要删除重复的行，可以使用“小于”作为筛

选条件。（为了让中间查询结果与最终结果集更接近，此处没有列出 **HIREDATE**。）

```
select a.ename as emp1, b.ename as emp2
  from emp a, emp b
 where to_char(a.hiredate,'DMON') =
        to_char(b.hiredate,'DMON')
        and a.empno < b.empno
 order by 1
```

EMP1	EMP2
-----	-----
JAMES	FORD
SCOTT	JAMES
SCOTT	FORD

最后，拼接结果集以得到所需的消息。

9.13 找出重叠的日期范围

1. 问题

你想找出所有这样的情况，即员工的当前项目还未结束就开始了下一个项目。请看下面的 **EMP_PROJECT** 表。

<pre>select * from emp_project</pre>				
EMPNO	ENAME	PROJ_ID	PROJ_START	PROJ_END
-----	-----	-----	-----	-----
7782	CLARK	1	16-JUN-2005	18-JUN-2005
7782	CLARK	4	19-JUN-2005	24-JUN-2005
7782	CLARK	7	22-JUN-2005	25-JUN-2005
7782	CLARK	10	25-JUN-2005	28-JUN-2005
7782	CLARK	13	28-JUN-2005	02-JUL-2005
7839	KING	2	17-JUN-2005	21-JUN-2005
7839	KING	8	23-JUN-2005	25-JUN-2005
7839	KING	14	29-JUN-2005	30-JUN-2005
7839	KING	11	26-JUN-2005	27-JUN-2005
7839	KING	5	20-JUN-2005	24-JUN-2005
7934	MILLER	3	18-JUN-2005	22-JUN-2005
7934	MILLER	12	27-JUN-2005	28-JUN-2005
7934	MILLER	15	30-JUN-2005	03-JUL-2005
7934	MILLER	9	24-JUN-2005	27-JUN-2005
7934	MILLER	6	21-JUN-2005	23-JUN-2005

从与 KING 相关的结果可知，他在 **PROJ_ID 5** 结束前就开始了 **PROJ_ID 8**，并且在开始 **PROJ_ID 5** 的时候，**PROJ_ID 2** 还没有结束。你想返回如下结果集。

EMPNO	ENAME	MSG
-----	-----	-----
7782	CLARK	project 7 overlaps project 4
7782	CLARK	project 10 overlaps project 7
7782	CLARK	project 13 overlaps project 10
7839	KING	project 8 overlaps project 5

7839	KING	project 5 overlaps project 2
7934	MILLER	project 12 overlaps project 9
7934	MILLER	project 6 overlaps project 3

12. 解决方案

这里的关键是找出这样的行，即其 **PROJ_START**（新项目开始日期）不早于另一个项目的 **PROJ_START**，同时不晚于该项目的 **PROJ_END**。为此，需要能够对同一位员工参与的任意两个项目进行比较。通过以员工号相同的方式自连接 **EMP_PROJECT** 表，可以生成每位员工参与的任意两个项目的组合。要找出时间重叠的项目，只需找出这样的项目：其 **PROJ_START** 落在相应员工参与的另一个项目的 **PROJ_START** 和 **PROJ_END** 之间。

DB2、PostgreSQL 和 Oracle

自连接 **EMP_PROJECT** 表，然后使用拼接运算符 **||** 生成消息，指出两个项目的时间是重叠的。

```
1 select a.empno,a.ename,
2       'project '||b.proj_id||
3       ' overlaps project '||a.proj_id as msg
4   from emp_project a,
5        emp_project b
6  where a.empno = b.empno
7        and b.proj_start >= a.proj_start
8        and b.proj_start <= a.proj_end
9        and a.proj_id != b.proj_id
```

MySQL

自连接 **EMP_PROJECT** 表，然后使用函数 **CONCAT** 生成消息，指出两个项目的时间是重叠的。

```

1 select a.empno,a.ename,
2         concat('project ',b.proj_id,
3         ' overlaps project ',a.proj_id) as msg
4   from emp_project a,
5        emp_project b
6  where a.empno = b.empno
7        and b.proj_start >= a.proj_start
8        and b.proj_start <= a.proj_end
9        and a.proj_id != b.proj_id

```

SQL Server

自连接 **EMP_PROJECT** 表，然后使用拼接运算符 **+** 生成消息，指出两个项目的时间是重叠的。

```

1 select a.empno,a.ename,
2         'project '+b.proj_id+
3         ' overlaps project '+a.proj_id as msg
4   from emp_project a,
5        emp_project b
6  where a.empno = b.empno
7        and b.proj_start >= a.proj_start
8        and b.proj_start <= a.proj_end
9        and a.proj_id != b.proj_id

```

13. 讨论

上述几种解决方案之间唯一的差别是拼接字符串的方式，因此这里以 **DB2** 解决方案为例，但所做的讨论适用于所有解决方案。首先，自连接 **EMP_PROJECT** 表，以便对不同项目的 **PROJ_START** 进行比较。下面显示了自连接输出中与员工 **KING** 相关的部分，从中可知每个项目都能够“看到”其他所有的项目。

```

select a.ename,
       a.proj_id as a_id,

```

```

        a.proj_start as a_start,
        a.proj_end as a_end,
        b.proj_id as b_id,
        b.proj_start as b_start
    from emp_project a,
         emp_project b
    where a.ename = 'KING'
          and a.empno = b.empno
          and a.proj_id != b.proj_id
    order by 2

```

ENAME	A_ID	A_START	A_END	B_ID	B_START
KING	2	17-JUN-2005	21-JUN-2005	8	23-JUN-2005
KING	2	17-JUN-2005	21-JUN-2005	14	29-JUN-2005
KING	2	17-JUN-2005	21-JUN-2005	11	26-JUN-2005
KING	2	17-JUN-2005	21-JUN-2005	5	20-JUN-2005
KING	5	20-JUN-2005	24-JUN-2005	2	17-JUN-2005
KING	5	20-JUN-2005	24-JUN-2005	8	23-JUN-2005
KING	5	20-JUN-2005	24-JUN-2005	11	26-JUN-2005
KING	5	20-JUN-2005	24-JUN-2005	14	29-JUN-2005
KING	8	23-JUN-2005	25-JUN-2005	2	17-JUN-2005
KING	8	23-JUN-2005	25-JUN-2005	14	29-JUN-2005
KING	8	23-JUN-2005	25-JUN-2005	5	20-JUN-2005
KING	8	23-JUN-2005	25-JUN-2005	11	26-JUN-2005
KING	11	26-JUN-2005	27-JUN-2005	2	17-JUN-2005
KING	11	26-JUN-2005	27-JUN-2005	8	23-JUN-2005
KING	11	26-JUN-2005	27-JUN-2005	14	29-JUN-2005
KING	11	26-JUN-2005	27-JUN-2005	5	20-JUN-2005
KING	14	29-JUN-2005	30-JUN-2005	2	17-JUN-2005
KING	14	29-JUN-2005	30-JUN-2005	8	23-JUN-2005
KING	14	29-JUN-2005	30-JUN-2005	5	20-JUN-2005
KING	14	29-JUN-2005	30-JUN-2005	11	26-JUN-2005

从上述结果集可知，自连接让你能够轻松地找出时间重叠的项目：只需返回所有这样的行，即其 **B_START** 在 **A_START** 和 **A_END** 之间。这项工作是由解决方案的第 7~8 行完成的。

```

and b.proj_start >= a.proj_start
and b.proj_start <= a.proj_end

```

有了所需的行后，创建消息将易如反掌，只需将返回的值拼

接起来。

如果员工参与的项目数的最大值是固定的，那么 Oracle 用户可以使用窗口函数 **LEAD OVER**，以免进行自连接。在自连接的代价很高（如果自连接消耗的资源比 **LEAD OVER** 多）时，这提供了极大的方便。例如，下面的解决方案使用了 **LEAD OVER**，并只返回与 **KING** 相关的结果。

```
select empno,
       ename,
       proj_id,
       proj_start,
       proj_end,
       case
         when lead(proj_start,1)over(order by proj_start)
              between proj_start and proj_end
         then lead(proj_id)over(order by proj_start)
         when lead(proj_start,2)over(order by proj_start)
              between proj_start and proj_end
         then lead(proj_id)over(order by proj_start)
         when lead(proj_start,3)over(order by proj_start)
              between proj_start and proj_end
         then lead(proj_id)over(order by proj_start)
         when lead(proj_start,4)over(order by proj_start)
              between proj_start and proj_end
         then lead(proj_id)over(order by proj_start)
         end is_overlap
  from emp_project
 where ename = 'KING'
```

EMPNO	ENAME	PROJ_ID	PROJ_START	PROJ_END	IS_OVERLAP
7839	KING	2	17-JUN-2005	21-JUN-2005	5
7839	KING	5	20-JUN-2005	24-JUN-2005	8
7839	KING	8	23-JUN-2005	25-JUN-2005	
7839	KING	11	26-JUN-2005	27-JUN-2005	
7839	KING	14	29-JUN-2005	30-JUN-2005	

由于员工 **KING** 参与的项目是固定的（5 个），因此可以使用 **LEAD OVER** 查看所有项目的日期值，而无须使用自连

接。此时，生成最终的结果将易如反掌，只需保留 IS_OVERLAP 不为 NULL 的行。

```
select empno,ename,
       'project '||is_overlap||
       ' overlaps project '||proj_id msg
  from (
select empno,
       ename,
       proj_id,
       proj_start,
       proj_end,
       case
       when lead(proj_start,1)over(order by proj_start)
         between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,2)over(order by proj_start)
         between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,3)over(order by proj_start)
         between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,4)over(order by proj_start)
         between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       end is_overlap
  from emp_project
 where ename = 'KING'
  )
 where is_overlap is not null
```

EMPNO	ENAME	MSG
7839	KING	project 5 overlaps project 2
7839	KING	project 8 overlaps project 5

要让解决方案适用于所有员工（而不仅仅是 KING），需要在函数 LEAD OVER 中根据 ENAME 进行分区。

```
select empno,ename,
       'project '||is_overlap||
       ' overlaps project '||proj_id msg
```

```

from (
select empno,
       ename,
       proj_id,
       proj_start,
       proj_end,
       case
       when lead(proj_start,1)over(partition by ename
                                   order by proj_start)
        between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       when lead(proj_start,2)over(partition by ename
                                   order by proj_start)
        between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       when lead(proj_start,3)over(partition by ename
                                   order by proj_start)
        between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       when lead(proj_start,4)over(partition by ename
                                   order by proj_start)
        between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       end is_overlap
from emp_project
)
where is_overlap is not null

```

EMPNO	ENAME	MSG
7782	CLARK	project 7 overlaps project 4
7782	CLARK	project 10 overlaps project 7
7782	CLARK	project 13 overlaps project 10
7839	KING	project 5 overlaps project 2
7839	KING	project 8 overlaps project 5
7934	MILLER	project 6 overlaps project 3
7934	MILLER	project 12 overlaps project 9

9.14 小结

在数据库查询中，经常需要操作日期（同时存储了日期的事件常常会让人提出一些与日期相关的怪异问题），而日期又是标准化程度相对较低的 SQL 领域之一。但愿阅读本章后你能够明白，虽然在不同的 RDBMS 中日期查询的语法不同，但有些逻辑是通用的。

第 10 章 涉及区间的查询

本章介绍涉及区间的日常查询。在日常生活中，区间很常见，例如，项目的持续时间就是一个区间。在 SQL 中，经常需要执行涉及区间的查询、生成区间或操作基于区间的数据。相比于本书前面的查询，本章的查询要稍微复杂些，但这些查询也很常见。通过阅读本章，你将逐渐认识到，如果能够得到充分利用，那么 SQL 威力将非常强大。

10.1 找出一系列连续的值

1. 问题

你想判断哪些行表示一系列连续的项目。请看下面从视图 **V** 返回的结果集，它包含一系列项目及其开始日期和结束日期。

```
select *
  from V
```

PROJ_ID	PROJ_START	PROJ_END
1	01-JAN-2020	02-JAN-2020
2	02-JAN-2020	03-JAN-2020
3	03-JAN-2020	04-JAN-2020
4	04-JAN-2020	05-JAN-2020
5	06-JAN-2020	07-JAN-2020
6	16-JAN-2020	17-JAN-2020
7	17-JAN-2020	18-JAN-2020
8	18-JAN-2020	19-JAN-2020
9	19-JAN-2020	20-JAN-2020
10	21-JAN-2020	22-JAN-2020
11	26-JAN-2020	27-JAN-2020
12	27-JAN-2020	28-JAN-2020
13	28-JAN-2020	29-JAN-2020
14	29-JAN-2020	30-JAN-2020

所谓表示一系列连续项目的行，指的是除第一行外，其他各行的 **PROJ_START** 都与其前一行的 **PROJ_END** 相等（这里的前一行，指的是 **PROJ_ID** 比当前行小 1 的行）。在视图 **V** 返回的前 5 行中，**PROJ_ID** 为 1~3 的行属于同一“组”，因为其中每行的 **PROJ_END** 都与下一行的 **PROJ_START** 相等。由于你要找出一系列连续项目所在的日期范围，因此要返回所有其 **PROJ_END** 与下一行的 **PROJ_START** 相等的行。如果整个结果集只包含前 5 行，那么你要返回的是前 3 行。最终的结

果集如下所示（以视图 V 中的全部 14 行为例）。

PROJ_ID	PROJ_START	PROJ_END
1	01-JAN-2020	02-JAN-2020
2	02-JAN-2020	03-JAN-2020
3	03-JAN-2020	04-JAN-2020
6	16-JAN-2020	17-JAN-2020
7	17-JAN-2020	18-JAN-2020
8	18-JAN-2020	19-JAN-2020
11	26-JAN-2020	27-JAN-2020
12	27-JAN-2020	28-JAN-2020
13	28-JAN-2020	29-JAN-2020

上述结果集中没有包含 PROJ_ID 为 4、5、9、10 和 14 的行，因为这些行的 PROJ_END 与下一行的 PROJ_START 都不相等。

12. 解决方案

本解决方案利用窗口函数 **LEAD OVER** 来查找下一行的 **BEGIN_DATE**，从而避免使用自连接。而在窗口函数被引入前，必须使用自连接。

```
1 select proj_id, proj_start, proj_end
2   from (
3 select proj_id, proj_start, proj_end,
4        lead(proj_start)over(order by proj_id) next_proj_start
5   from V
6        ) alias
7  where next_proj_start = proj_end
```

13. 讨论

DB2、MySQL、PostgreSQL、SQL Server 和 Oracle

虽然完全可以编写使用自连接的解决方案，但对于这类问题，使用窗口函数 **LEAD OVER** 更合适，也更直观。函数 **LEAD OVER** 能够在不进行自连接的情况下查看其他行。（如果要这样做，则这个函数必须对结果集进行排序。）下面显示了内嵌视图（第 3~5 行）返回的结果集中 **ID** 为 1 和 4 的行。

```
select *
  from (
select proj_id, proj_start, proj_end,
       lead(proj_start)over(order by proj_id) next_proj_start
  from v
  )
 where proj_id in ( 1, 4 )
```

PROJ_ID	PROJ_START	PROJ_END	NEXT_PROJ_START
1	01-JAN-2020	02-JAN-2020	02-JAN-2020
4	04-JAN-2020	05-JAN-2020	06-JAN-2020

通过查看以上代码片段及其结果集，很容易知道为什么 **PROJ_ID 4** 被排除在完成解决方案的最终结果集之外。这是因为其 **PROJ_END** 为 **05-JAN-2020**，与“下一个”项目的起始日期 **06-JAN-2020** 不匹配。

当需要解决此类问题（具体地说是需要查看部分结果）时，函数 **LEAD OVER** 很有用。使用窗口函数时，别忘了它们是在 **FROM** 和 **WHERE** 子句之后执行的，因此在前一个查询中，函数 **LEAD OVER** 必须放在一个内嵌视图中。如果不这样做，那么函数 **LEAD OVER** 将应用于执行 **WHERE** 子句（将除 **PROJ_ID** 为 1 和 4 的行之外的行都排除在外）后的结果集。

现在，根据你看待数据的方式，你可能想在最终的结果集中包含 **PROJ_ID 4**。来看视图 **V** 的前 5 行。

```
select *
  from V
 where proj_id <= 5
```

PROJ_ID	PROJ_START	PROJ_END
1	01-JAN-2020	02-JAN-2020
2	02-JAN-2020	03-JAN-2020
3	03-JAN-2020	04-JAN-2020
4	04-JAN-2020	05-JAN-2020
5	06-JAN-2020	07-JAN-2020

如果根据你的需求，将 **PROJ_ID 4** 也视为连续的（因为其 **PROJ_START** 与 **PROJ_ID 3** 的 **PROJ_END** 相同），只将 **PROJ_ID 5** 排除在外，那么前面的解决方案就是错误的，至少是不完整的。

```
select proj_id, proj_start, proj_end
  from (
select proj_id, proj_start, proj_end,
       lead(proj_start)over(order by proj_id) next_start
  from V
 where proj_id <= 5
    )
 where proj_end = next_start
```

PROJ_ID	PROJ_START	PROJ_END
1	01-JAN-2020	02-JAN-2020
2	02-JAN-2020	03-JAN-2020
3	03-JAN-2020	04-JAN-2020

如果你认为应该将 **PROJ_ID 4** 也包含进来，那么只需在查询中添加 **LAG OVER**，并在 **WHERE** 子句中再添加一个筛选器。

```
select proj_id, proj_start, proj_end
  from (
select proj_id, proj_start, proj_end,
       lead(proj_start)over(order by proj_id) next_start,
```

```

        lag(proj_end)over(order by proj_id) last_end
    from V
where proj_id <= 5
    )
where proj_end = next_start
    or proj_start = last_end

PROJ_ID PROJ_START  PROJ_END
-----
1 01-JAN-2020 02-JAN-2020
2 02-JAN-2020 03-JAN-2020
3 03-JAN-2020 04-JAN-2020
4 04-JAN-2020 05-JAN-2020

```

这样，最终的结果集中将包含 **PROJ_ID 4**，只有 **PROJ_ID 5** 被排除在外。在代码中使用这些解决方案时，务必考虑需求究竟是什么。

10.2 找出同一个分组或分区中相邻行的差

1. 问题

你想返回每位员工的 **DEPTNO**、**ENAME**、**SAL** 及其与当前部门中（**DEPTNO** 相同）下一位员工的 **SAL** 差（以判断在同一部门中，资历和薪水之间是否存在相关性）。这里的下一位员工是根据获聘时间确定的。对于每个部门最后获聘的员工，将 **SAL** 差设置为 **N/A**。结果集如下所示。

DEPTNO	ENAME	SAL	HIREDATE	DIFF
10	CLARK	2450	09-JUN-2006	-2550
10	KING	5000	17-NOV-2006	3700
10	MILLER	1300	23-JAN-2007	N/A
20	SMITH	800	17-DEC-2005	-2175
20	JONES	2975	02-APR-2006	-25
20	FORD	3000	03-DEC-2006	0
20	SCOTT	3000	09-DEC-2007	1900
20	ADAMS	1100	12-JAN-2008	N/A
30	ALLEN	1600	20-FEB-2006	350
30	WARD	1250	22-FEB-2006	-1600
30	BLAKE	2850	01-MAY-2006	1350
30	TURNER	1500	08-SEP-2006	250
30	MARTIN	1250	28-SEP-2006	300
30	JAMES	950	03-DEC-2006	N/A

2. 解决方案

在本实例中，窗口函数 **LEAD OVER** 和 **LAG OVER** 又一次提供了极大的方便。无须使用连接，就可以轻松地访问下一行数据和前一行数据。对于这种问题，也可以使用其他方法（比如子查询或自连接）来解决，但显得比较笨拙。


```

1 with next_sal_tab (deptno,ename,sal,hiredate,next_sal)
2 as
3 (select deptno, ename, sal, hiredate,
4         lead(sal)over(partition by deptno
5                       order by hiredate) as next_sal
6  from emp )
7
8  select deptno, ename, sal, hiredate
9  ,      coalesce(cast(sal-next_sal as char), 'N/A') as diff
10  from next_sal_tab

```

这里为展示解决方案的多样性，使用的是 CTE 而不是子查询。当前，在大多数 RDBMS 中，这两种方法都管用，具体选择哪种方法，通常取决于可读性。

13. 讨论

首先，使用窗口函数 **LEAD OVER** 找出当前部门中“下一位”员工的薪水。在每个部门中，最后获聘的员工的 **NEXT_SAL** 值都为 **NULL**。

```

select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno order by hiredate) as
next_sal
  from emp

```

DEPTNO	ENAME	SAL	HIREDATE	NEXT_SAL
10	CLARK	2450	09-JUN-2006	5000
10	KING	5000	17-NOV-2006	1300
10	MILLER	1300	23-JAN-2007	
20	SMITH	800	17-DEC-2005	2975
20	JONES	2975	02-APR-2006	3000
20	FORD	3000	03-DEC-2006	3000
20	SCOTT	3000	09-DEC-2007	1100
20	ADAMS	1100	12-JAN-2008	
30	ALLEN	1600	20-FEB-2006	1250
30	WARD	1250	22-FEB-2006	2850

30	BLAKE	2850	01-MAY-2006	1500
30	TURNER	1500	08-SEP-2006	1250
30	MARTIN	1250	28-SEP-2006	950
30	JAMES	950	03-DEC-2006	

然后，计算每位员工的薪水与其部门中接下来聘请的员工的薪水之差。

```
select deptno,ename,sal,hiredate, sal-next_sal diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno order by hiredate)
next_sal
  from emp
  )
```

DEPTNO	ENAME	SAL	HIREDATE	DIFF
-----	-----	-----	-----	-----
10	CLARK	2450	09-JUN-2006	-2550
10	KING	5000	17-NOV-2006	3700
10	MILLER	1300	23-JAN-2007	
20	SMITH	800	17-DEC-2005	-2175
20	JONES	2975	02-APR-2006	-25
20	FORD	3000	03-DEC-2006	0
20	SCOTT	3000	09-DEC-2007	1900
20	ADAMS	1100	12-JAN-2008	
30	ALLEN	1600	20-FEB-2006	350
30	WARD	1250	22-FEB-2006	-1600
30	BLAKE	2850	01-MAY-2006	1350
30	TURNER	1500	08-SEP-2006	250
30	MARTIN	1250	28-SEP-2006	300
30	JAMES	950	03-DEC-2006	

接下来，使用函数 **COALESCE** 将后面没有其他员工的员工的薪水差设置为 **N/A**。为了返回 **N/A**，必须将 **DIFF** 的值转换为字符串。

```
select deptno,ename,sal,hiredate,
       nvl(to_char(sal-next_sal),'N/A') diff
  from (
select deptno,ename,sal,hiredate,
```

```

        lead(sal)over(partition by deptno order by hiredate)
next_sal
  from emp
  )

```

DEPTNO	ENAME	SAL	HIREDATE	DIFF
10	CLARK	2450	09-JUN-2006	-2550
10	KING	5000	17-NOV-2006	3700
10	MILLER	1300	23-JAN-2007	N/A
20	SMITH	800	17-DEC-2005	-2175
20	JONES	2975	02-APR-2006	-25
20	FORD	3000	03-DEC-2006	0
20	SCOTT	3000	09-DEC-2007	1900
20	ADAMS	1100	12-JAN-2008	N/A
30	ALLEN	1600	20-FEB-2006	350
30	WARD	1250	22-FEB-2006	-1600
30	BLAKE	2850	01-MAY-2006	1350
30	TURNER	1500	08-SEP-2006	250
30	MARTIN	1250	28-SEP-2006	300
30	JAMES	950	03-DEC-2006	N/A

出于可读性考虑，本书提供的解决方案大多数没有处理“假使.....将会怎么样？”的问题，但以这里这样的方式使用函数 **LEAD OVER** 时，必须讨论存在重复值的情形。在 **EMP** 表包含的示例数据中，没有任何两位员工的 **HIREDATE** 相同，但满足这种条件的可能性不大。本书通常不讨论“假使.....将会怎么样？”的问题（比如存在重复值的问题，因为 **EMP** 表中没有重复值），但使用 **LEAD** 时，处理这种问题的方案可能不那么显而易见。请看下面的查询，它返回了 10 号部门中相邻员工的 **SAL** 差（员工是根据获聘日期排列的）。

```

select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno
                     order by hiredate) next_sal
  from emp
 where deptno=10 and empno > 10

```

)				
DEPTNO	ENAME	SAL	HIREDATE	DIFF

10	CLARK	2450	09-JUN-2006	-2550
10	KING	5000	17-NOV-2006	3700
10	MILLER	1300	23-JAN-2007	N/A

就 **EMP** 表当前包含的数据而言，上述解决方案是正确的，但如果这张表包含获聘日期相同的员工，那么该解决方案就不管用了。请看下面的例子，其中有 4 位员工的获聘日期与 KING 相同。

```

insert into emp (empno,ename,deptno,sal,hiredate)
values (1,'ant',10,1000,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (2,'joe',10,1500,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (3,'jim',10,1600,to_date('17-NOV-2006'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (4,'jon',10,1700,to_date('17-NOV-2006'))

select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno
                     order by hiredate) next_sal
  from emp
 where deptno=10
 )

```

DEPTNO	ENAME	SAL	HIREDATE	DIFF

10	CLARK	2450	09-JUN-2006	1450
10	ant	1000	17-NOV-2006	-500
10	joe	1500	17-NOV-2006	-
10	KING	5000	17-NOV-2006	3400
10	jim	1600	17-NOV-2006	-100

10 jon	1700	17-NOV-2006	400
10 MILLER	1300	23-JAN-2007	N/A

注意，对于在 2006 年 11 月 17 日获聘的所有员工，都是将其薪水与同一天获聘的另一位员工的薪水进行比较，只有 JON 例外。这是不正确的。对于所有在 2006 年 11 月 17 日获聘的员工，都应该计算其薪水与 MILLER（而不是在 2006 年 11 月 17 日获聘的另一位员工）的薪水之差。以员工 ANT 为例，当前其 DIFF 值为-500。这是因为将其 SAL 同 JOE 的 SAL 进行了比较，而前者比后者少 500，因此 ANT 的 DIFF 的值为-500。对于员工 ANT，正确的 DIFF 值为-300，因为 ANT 的薪水比 MILLER 少 300（根据 HIREDATE，MILLER 才是下一位获聘的员工）。前面的解决方案之所以不管用，是因为 Oracle 函数 LEAD OVER 的默认行为。默认情况下，LEAD OVER 只往前跳一行。对于员工 ANT，根据 HIREDATE 确定的下一个 SAL 为 JOE 的 SAL，LEAD OVER 往前跳一行，且不忽略重复值。所幸 Oracle 考虑到了这样的情形，允许你向 LEAD OVER 传递另一个参数，指定应往前跳多远。在前面的示例中，解决方案很简单，只需进行简单的计数，确定在 2006 年 11 月 17 日获聘的每一位员工与 2007 年 1 月 23 日获聘的员工 MILLER 相隔多远。下面演示了如何完成这项任务。

```
select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal,cnt-rn+1)over(partition by deptno
                             order by hiredate) next_sal
  from (
select deptno,ename,sal,hiredate,
       count(*)over(partition by deptno,hiredate) cnt,
       row_number()over(partition by deptno,hiredate order by sal)
  rn
  from emp
 where deptno=10
```

)				
)				
DEPTNO	ENAME	SAL	HIREDATE	DIFF
10	CLARK	2450	09-JUN-2006	1450
10	ant	1000	17-NOV-2006	-300
10	joe	1500	17-NOV-2006	200
10	jim	1600	17-NOV-2006	300
10	jon	1700	17-NOV-2006	400
10	KING	5000	17-NOV-2006	3700
10	MILLER	1300	23-JAN-2007	N/A

现在，解决方案是正确的。如你所见，对于在 2006 年 11 月 17 日获聘的每位员工，都将其薪水与 MILLER 的薪水进行比较。从结果可知，员工 ANT 的 DIFF 值现在为-300，与期望的结果一致。如果这一点不那么好理解，请查看传递给 LEAD OVER 的表达式——CNT- RN+1，这是在 2006 年 11 月 17 日获聘的每一位员工与 MILLER 相隔的距离。请看下面的内嵌视图，它显示了 CNT 和 RN 的值。

<pre> select deptno,ename,sal,hiredate, count(*)over(partition by deptno,hiredate) cnt, row_number()over(partition by deptno,hiredate order by sal) rn from emp where deptno=10 </pre>					
DEPTNO	ENAME	SAL	HIREDATE	CNT	RN
10	CLARK	2450	09-JUN-2006	1	1
10	ant	1000	17-NOV-2006	5	1
10	joe	1500	17-NOV-2006	5	2
10	jim	1600	17-NOV-2006	5	3
10	jon	1700	17-NOV-2006	5	4
10	KING	5000	17-NOV-2006	5	5
10	MILLER	1300	23-JAN-2007	1	1

对于 HIREDATE 值相同的每一位员工，CNT 的值指出了一共有多少位。RN 的值为 10 号部门的员工排名。排名时根据

DEPTNO 和 HIREDATE 进行了分区，因此仅当某位员工的 HIREDATE 与另一位员工相同时，其 RN 值才可能大于 1。排名时根据 SAL 进行了排序。（排序方式是随意选择的，虽然根据 SAL 排序很方便，但也可以根据 EMPNO 进行排序。）知道 HIREDATE 相同的员工数量以及这些员工的排名后，要计算这些员工与 MILLER 相隔的距离，只需将 HIREDATE 相同的员工数量减去当前员工的排名再加 1（CNT-RN+1）。下面显示了计算得到的距离及其对 LEAD OVER 的影响。

<pre> select deptno,ename,sal,hiredate, lead(sal)over(partition by deptno order by hiredate) incorrect, cnt-rn+1 distance, lead(sal,cnt-rn+1)over(partition by deptno order by hiredate) correct from (select deptno,ename,sal,hiredate, count(*)over(partition by deptno,hiredate) cnt, row_number()over(partition by deptno,hiredate order by sal) rn from emp where deptno=10) </pre>						
DEPTNO	ENAME	SAL	HIREDATE	INCORRECT	DISTANCE	CORRECT
10	CLARK	2450	09-JUN-2006	1000	1	1000
10	ant	1000	17-NOV-2006	1500	5	1300
10	joe	1500	17-NOV-2006	1600	4	1300
10	jim	1600	17-NOV-2006	1700	3	1300
10	jon	1700	17-NOV-2006	5000	2	1300
10	KING	5000	17-NOV-2006	1300	1	1300
10	MILLER	1300	23-JAN-2007		1	

至此，可以清楚地看到将正确的距离传递给 LEAD OVER 带来的影响。INCORRECT 列显示了使用默认距离 1 时 LEAD OVER 返回的值，而 CORRECT 列显示了对于 HIREDATE 值相同的每位员工，使用到 MILLER 的正确距离时，LEAD OVER 返回的值。现在，剩下的唯一工作是，计算每一行中

CORRECT 和 SAL 的差，完成这项任务的方法前面已经介绍过了。

10.3 找出连续值构成的区间的起点和终点

1. 问题

本实例是 10.1 节的扩展，也使用 10.1 节中的视图 **V**。在 10.1 节中，你找出了连续值构成的区间，而这里只想找出区间的起点和终点。与 10.1 节不同，在这个实例中，如果某行并非一组连续值的一部分，则你依然要返回它。为什么呢？这是因为这样的行是其自身构成的区间的起点和终点。这里将下面的视图 **V** 作为数据源。

```
select *  
  from V
```

PROJ_ID	PROJ_START	PROJ_END
1	01-JAN-2020	02-JAN-2020
2	02-JAN-2020	03-JAN-2020
3	03-JAN-2020	04-JAN-2020
4	04-JAN-2020	05-JAN-2020
5	06-JAN-2020	07-JAN-2020
6	16-JAN-2020	17-JAN-2020
7	17-JAN-2020	18-JAN-2020
8	18-JAN-2020	19-JAN-2020
9	19-JAN-2020	20-JAN-2020
10	21-JAN-2020	22-JAN-2020
11	26-JAN-2020	27-JAN-2020
12	27-JAN-2020	28-JAN-2020
13	28-JAN-2020	29-JAN-2020
14	29-JAN-2020	30-JAN-2020

而你希望最终的结果集如下所示。

PROJ_GRP	PROJ_START	PROJ_END
1	01-JAN-2020	05-JAN-2020
2	06-JAN-2020	07-JAN-2020

3	16-JAN-2020	20-JAN-2020
4	21-JAN-2020	22-JAN-2020
5	26-JAN-2020	30-JAN-2020

12. 解决方案

这个问题比 10.1 节的问题要复杂些。首先，必须找出所有的区间。一系列行是否构成区间取决于它们的 **PROJ_START** 值和 **PROJ_END** 值。对于特定的行，仅当其 **PROJ_START** 值与前一行的 **PROJ_END** 值相等时，才被视为“连续”的，即属于分组的一部分。如果一行的 **PROJ_START** 值与前一行的 **PROJ_END** 值不相等，且其 **PROJ_END** 值与后一行的 **PROJ_START** 值不相等，则它自己构成一个分组。找出所有的区间后，必须将各个区间中的行编组，进而只返回区间的起点和终点。

请看前面要返回的最终结果集的第 1 行，其 **PROJ_START** 为视图 **V** 中 **PROJ_ID** 1 的 **PROJ_START**，而 **PROJ_END** 为视图 **V** 中 **PROJ_ID** 4 的 **PROJ_END**。虽然 **PROJ_ID** 4 后面的值不是连续的，但它是连续值区间的终点，因此包含在第 1 个分组中。

对于这个问题，最简单的解决方法是使用窗口函数 **LAG OVER**。使用 **LAG OVER** 来判断当前行的 **PROJ_START** 是否与前一行的 **PROJ_END** 相等，可以将所有的行分成多组。将行分组后，可以使用聚合函数 **MIN** 和 **MAX** 找出各个分组的起点和终点。

```
1 select proj_grp, min(proj_start), max(proj_end)
2   from (
3 select proj_id,proj_start,proj_end,
4         sum(flag)over(order by proj_id) proj_grp
5   from (
```

```

6 select proj_id,proj_start,proj_end,
7         case when
8             lag(proj_end)over(order by proj_id) = proj_start
9             then 0 else 1
10        end flag
11 from V
12      ) alias1
13      ) alias2
14 group by proj_grp

```

13. 讨论

在这里，窗口函数 **LAG OVER** 很有用。无须使用自连接、标量子查询或视图，你就能够查看前一行的 **PROJ_END** 值。在没有 **CASE** 表达式的情况下，**LAG OVER** 函数返回的结果如下所示。

```

select proj_id,proj_start,proj_end,
       lag(proj_end)over(order by proj_id) prior_proj_end
from V

```

PROJ_ID	PROJ_START	PROJ_END	PRIOR_PROJ_END
-----	-----	-----	-----
1	01-JAN-2020	02-JAN-2020	
2	02-JAN-2020	03-JAN-2020	02-JAN-2020
3	03-JAN-2020	04-JAN-2020	03-JAN-2020
4	04-JAN-2020	05-JAN-2020	04-JAN-2020
5	06-JAN-2020	07-JAN-2020	05-JAN-2020
6	16-JAN-2020	17-JAN-2020	07-JAN-2020
7	17-JAN-2020	18-JAN-2020	17-JAN-2020
8	18-JAN-2020	19-JAN-2020	18-JAN-2020
9	19-JAN-2020	20-JAN-2020	19-JAN-2020
10	21-JAN-2020	22-JAN-2020	20-JAN-2020
11	26-JAN-2020	27-JAN-2020	22-JAN-2020
12	27-JAN-2020	28-JAN-2020	27-JAN-2020
13	28-JAN-2020	29-JAN-2020	28-JAN-2020
14	29-JAN-2020	30-JAN-2020	29-JAN-2020

在前面的解决方案中，**CASE** 表达式会将 **LAG OVER** 返回的值同当前行的 **PROJ_START** 值进行比较。如果它们相等，就返回 0，否则就返回 1。然后根据 **CASE** 表达式返回的 0 和 1 生成移动总计，以便将行编组。生成的移动总计如下所示。

```
select proj_id,proj_start,proj_end,
       sum(flag)over(order by proj_id) proj_grp
  from (
select proj_id,proj_start,proj_end,
       case when
           lag(proj_end)over(order by proj_id) = proj_start
           then 0 else 1
       end flag
  from V
  )
```

PROJ_ID	PROJ_START	PROJ_END	PROJ_GRP
1	01-JAN-2020	02-JAN-2020	1
2	02-JAN-2020	03-JAN-2020	1
3	03-JAN-2020	04-JAN-2020	1
4	04-JAN-2020	05-JAN-2020	1
5	06-JAN-2020	07-JAN-2020	2
6	16-JAN-2020	17-JAN-2020	3
7	17-JAN-2020	18-JAN-2020	3
8	18-JAN-2020	19-JAN-2020	3
9	19-JAN-2020	20-JAN-2020	3
10	21-JAN-2020	22-JAN-2020	4
11	26-JAN-2020	27-JAN-2020	5
12	27-JAN-2020	28-JAN-2020	5
13	28-JAN-2020	29-JAN-2020	5
14	29-JAN-2020	30-JAN-2020	5

将行编组后，只需将聚合函数 **MIN** 和 **MAX** 分别应用于 **PROJ_START** 和 **PROJ_END**，并根据移动总计列 **PROJ_GRP** 进行分组。

10.4 填补值区间空隙

1. 问题

你想返回从 2005 年起的 10 年间，每一年聘请的员工数量，但其中有些年份并没有聘请任何员工。你希望返回如下结果集。

YR	CNT
-----	-----
2005	1
2006	10
2007	2
2008	1
2009	0
2010	0
2011	0
2012	0
2013	0
2014	0

2. 解决方案

本解决方案的诀窍是，对于没有聘请任何员工的年份，返回 0。如果某年没有聘请任何员工，那么在 **EMP** 表中，将没有获聘日期属于该年份的行。既然表中没有获聘日期属于该年份的行，那么怎么返回该年份聘请的员工数量呢？为了解决这个问题，必须使用外连接。你必须提供一个包含所有年份的结果集，然后对 **EMP** 表执行计数操作，看看各个年份是否聘请了员工。

DB2

将 **EMP** 表用作透视表（因为它包含 14 行数据），并使用内置函数 **YEAR** 生成 10 行数据，从 2005 年开始的 10 年间的每一年对应一行。外连接到 **EMP** 表，并计算每年聘请的员工数量。

```
1 select x.yr, coalesce(y.cnt,0) cnt
2   from (
3 select year(min(hiredate)over()) -
4         mod(year(min(hiredate)over()),10) +
5         row_number()over()-1 yr
6   from emp fetch first 10 rows only
7   ) x
8  left join
9   (
10 select year(hiredate) yr1, count(*) cnt
11   from emp
12  group by year(hiredate)
13   ) y
14   on ( x.yr = y.yr1 )
```

Oracle

Oracle 解决方案的结构与 DB2 解决方案相同，唯一不同的是语法。

```
1 select x.yr, coalesce(cnt,0) cnt
2   from (
3 select extract(year from min(hiredate)over()) -
4         mod(extract(year from min(hiredate)over()),10) +
5         rownum-1 yr
6   from emp
7  where rownum <= 10
8   ) x
9  left join
10   (
11 select to_number(to_char(hiredate,'YYYY')) yr, count(*) cnt
12   from emp
13  group by to_number(to_char(hiredate,'YYYY'))
14   ) y
15   on ( x.yr = y.yr )
```

PostgreSQL 和 MySQL

将 **T10** 表作为透视表（因为它包含 10 行数据），并使用内置函数 **EXTRACT** 生成 10 行数据，从 2005 年开始的 10 年间的每一年对应一行。外连接到 **EMP** 表，并计算每年聘请的员工数量。

```
1 select y.yr, coalesce(x.cnt,0) as cnt
2   from (
3 select min_year-mod(cast(min_year as int),10)+rn as yr
4   from (
5 select (select min(extract(year from hiredate))
6         from emp) as min_year,
7         id-1 as rn
8   from t10
9        ) a
10       ) y
11  left join
12       (
13 select extract(year from hiredate) as yr, count(*) as cnt
14   from emp
15  group by extract(year from hiredate)
16         ) x
17   on ( y.yr = x.yr )
```

SQL Server

将 **EMP** 表用作透视表（因为它包含 14 行数据），并使用内置函数 **YEAR** 生成 10 行数据，从 2005 年开始的 10 年间的每一年对应一行。外连接到 **EMP** 表，并计算每年聘请的员工数量。

```
1 select x.yr, coalesce(y.cnt,0) cnt
2   from (
3 select top (10)
4         (year(min(hiredate)over()) -
5          year(min(hiredate)over())%10)+
6          row_number()over(order by hiredate)-1 yr
7   from emp
```

```

8      ) x
9  left join
10     (
11 select year(hiredate) yr, count(*) cnt
12   from emp
13  group by year(hiredate)
14        ) y
15    on ( x.yr = y.yr )

```

13. 讨论

虽然语法不同，但所有解决方案采用的方法都相同。内嵌视图 **X** 会返回 10 年间的每一年，为此首先找出最早的 **HIREDATE** 所属的年份，然后计算最早年份与其除以 10 的余数的差，并将这个差值加上 **RN-1**。要明白其中的工作原理，只需执行内嵌视图 **X** 并查看返回的值。下面列出了使用窗口函数 **MIN OVER**（适用于 DB2、Oracle 和 SQL Server）的内嵌视图 **X** 的结果集，以及使用标量子查询（适用于 MySQL 和 PostgreSQL）的内嵌视图 **X** 的结果集。

```

select year(min(hiredate)over()) -
       mod(year(min(hiredate)over()),10) +
       row_number()over()-1 yr,
       year(min(hiredate)over()) min_year,
       mod(year(min(hiredate)over()),10) mod_yr,
       row_number()over()-1 rn
  from emp fetch first 10 rows only

```

YR	MIN_YEAR	MOD_YR	RN
2005	2005	0	0
2006	2005	0	1
2007	2005	0	2
2008	2005	0	3
1984	2005	0	4
2010	2005	0	5
2011	2005	0	6

2012	2005	0	7
2013	2005	0	8
2014	2005	0	9

```

select min_year-mod(min_year,10)+rn as yr,
       min_year,
       mod(min_year,10) as mod_yr
       rn
  from (
select (select min(extract(year from hiredate))
        from emp) as min_year,
        id-1 as rn
  from t10
  ) x

```

YR	MIN_YEAR	MOD_YR	RN
-----	-----	-----	-----
2005	2005	0	0
2006	2005	0	1
2007	2005	0	2
2008	2005	0	3
2009	2005	0	4
2010	2005	0	5
2011	2005	0	6
2012	2005	0	7
2013	2005	0	8
2014	2005	0	9

内嵌视图 Y 会返回每个 **HIREDATE** 所属的年份以及该年份聘请的员工数量。

```

select year(hiredate) yr, count(*) cnt
  from emp
 group by year(hiredate)

```

YR	CNT
-----	-----
2005	1
2006	10
2007	2
2008	1

最后，将内嵌视图 **Y** 外连接到内嵌视图 **X**，以便返回所有的年份，包括没有聘请员工的年份。

10.5 生成连续的数字值

1. 问题

你希望有一个“行源生成器”（row source generator），以便在查询中使用它。对于需要透视的查询，行源生成器很有用。例如，你想返回如下结果集，它包含指定的行数。

ID

1
2
3
4
5
6
7
8
9
10
...

如果 RDBMS 提供了动态返回行的内置函数，就不用预先创建包含固定行数的透视表，这就是动态行生成器可以提供极大方便的原因。如果 RDBMS 没有提供这样的函数，你就必须在需要时使用包含固定行数的传统透视表来生成行（这可能还不够）。

2. 解决方案

本解决方案演示如何返回 10 行数据，其中包含的数字从 1 开始不断递增。你可以轻松地修改本解决方案，以返回任意的行数。

能够返回不断递增的值，这种能力打开了通往众多其他解决方案的大门。例如，你可以生成一系列数字，再将它们与特定日期相加，以生成日期序列。你还可以使用这些数字来分析字符串。

DB2 和 SQL Server

使用递归式 **WITH** 子句来生成一系列行，其中包含不断递增的值。实际上，在大多数 RDBMS 中，可以使用递归 CTE 来解决这个问题。

```
1 with x (id)
2 as (
3 select 1
4 union all
5 select id+1
6   from x
7  where id+1 <= 10
8 )
9 select * from x
```

Oracle

在 Oracle Database 中，可以使用 **MODEL** 子句来生成行。

```
1 select array id
2   from dual
3 model
4   dimension by (0 idx)
5   measures(1 array)
6   rules iterate (10) (
7     array[iteration_number] = iteration_number+1
8   )
```

PostgreSQL

使用为生成行而专门设计的函数 **GENERATE_SERIES**。

```
1 select id
2   from generate_series (1, 10) x(id)
```

13. 讨论

DB2 和 SQL Server

递归式 **WITH** 子句将 **ID**（从 1 开始）不断递增，直到满足 **WHERE** 子句指定的条件。首先，必须生成一个值为 1 的行。为此，可以从包含 1 行数据的表中选择 1（**SELECT 1**）。在 **DB2** 中，也可以使用 **VALUES** 子句来创建只包含 1 行数据的结果集。

Oracle

在使用 **MODEL** 子句的解决方案中，有一个显式的 **ITERATE** 命令能够生成多行数据。如果省略 **ITERATE** 子句，则将只返回 1 行数据，因为 **DUAL** 中只有 1 行数据。下面是一个这样的示例。

```
select array id
   from dual
model
  dimension by (0 idx)
  measures(1 array)
  rules ()

ID
--
1
```

MODEL 子句不仅能像访问数组一样访问行，还能轻松地“创建”或返回目标表中没有的行。在该解决方案中，**IDX** 为数组索引（特定值在数组中的位置），而 **ARRAY**（别名为 **ID**）是

元素为行的“数组”。第一行的值默认为 1，可以通过 **ARRAY[0]** 来引用。Oracle 提供了函数 **ITERATION_NUMBER**，以便你跟踪已完成的迭代次数。该解决方案迭代了 10 次，导致 **ITERATION_NUMBER** 的值从 0 逐渐变为 9。通过将这些值加上 1，得到结果值 1~10。

为了弄明白 **MODEL** 子句背后发生的事情，可以执行以下查询。

```
select 'array['||idx||'] = '||array as output
  from dual
 model
   dimension by (0 idx)
   measures(1 array)
   rules iterate (10) (
     array[iteration_number] = iteration_number+1
   )
```

OUTPUT

```
-----
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
array[5] = 6
array[6] = 7
array[7] = 8
array[8] = 9
array[9] = 10
```

PostgreSQL

所有的工作都是由函数 **GENERATE_SERIES** 完成的。这个函数接受 3 个参数，它们全部是数字值。第一个参数为起始值，第二个参数为终止值，第三个参数为可选的“步长”值（指定每次递增多少）。如果省略了第三个参数，则递增量默认为 1。

函数 **GENERATE_SERIES** 非常灵活，允许你不以硬编码的方式指定参数。如果你要返回 5 行数据，并希望起始值、终止值和步长分别为 10、30 和 5，以获得如下结果集：

ID

10
15
20
25
30

那么可以像下面这样创造性地完成这项任务。

```
select id
  from generate_series(
        (select min(deptno) from emp),
        (select max(deptno) from emp),
        5
    ) x(id)
```

注意，编写查询时，我们并不知道将传递给函数 **GENERATE_SERIES** 的值是什么。相反，它们是在主查询执行期间由子查询生成的。

10.6 小结

商业用户经常要求执行涉及区间的查询，这是商业运作方式的必然结果。然而，在有些情况下，必须巧妙地处理区间，而本章的实例演示了如何这样做。

第 11 章 高级查找

实际上，本书前面一直在讨论查找，其中很多查询使用连接、**WHERE** 子句和分组技术来找到并返回所需的结果。然而，有些类型的查找操作不同于其他查找操作，它们体现了一种不同的查找思路，也许你需要以每次一页的方式显示结果集。这个问题由两部分组成，一部分是找出要显示的整个结果集，另一部分则是当用户在结果集中导航时，不断查找下一页以便将其显示出来。乍一看，分页好像不是查找问题，但可以将其视为查找问题，也可以用 SQL 查找这样的方式来解决。本章介绍的都是这种类型的查找解决方案。

11.1 在结果集中翻页

1. 问题

你想在结果集中翻页或“滚动”。例如，你想返回 **EMP** 表中前 5 名员工的薪水，然后再返回接下来 5 名员工的薪水，以此类推。你的目标是，让用户每次查看 5 条记录，且每当用户单击 **Next** 按钮时都向前滚动。

2. 解决方案

在 **SQL** 中，没有“第一个”“最后一个”和“下一个”的概念，因此你必须给行指定排列顺序。仅当指定排列顺序后，才能准确地返回记录区间。

可以使用窗口函数 **ROW_NUMBER OVER** 指定排列顺序，并使用 **WHERE** 子句指定要返回的记录窗口。例如，下面的查询会返回第 1~5 行。

```
select sal
  from (
select row_number() over (order by sal) as rn,
      sal
  from emp
  ) x
 where rn between 1 and 5
```

```
SAL
----
800
950
1100
1250
```

下面的查询会返回第 6~10 行。

```
select sal
  from (
select row_number() over (order by sal) as rn,
      sal
  from emp
  ) x
 where rn between 6 and 10
```

```

  SAL
-----
1300
1500
1600
2450
2850
```

你可以返回任何记录区间，为此只需修改查询中的 **WHERE** 子句。

13. 讨论

内嵌视图 **X** 中的窗口函数 **ROW_NUMBER OVER** 会给每条记录都指定一个独一无二的编号（从 1 开始不断递增）。下面是内嵌视图 **X** 返回的结果集。

```
select row_number() over (order by sal) as rn,
      sal
  from emp
```

```
RN          SAL
--  -
1          800
2          950
3         1100
4         1250
```

5	1250
6	1300
7	1500
8	1600
9	2450
10	2850
11	2975
12	3000
13	3000
14	5000

给记录指定编号后，就可以使用 **RN** 值来指定要返回的记录区间了。

对于 Oracle 用户，还有另一种替代方案：使用函数 **ROWNUM**（而不是函数 **ROW NUMBER OVER**）给行生成序列号。

```
select sal
  from (
select sal, rownum rn
  from (
select sal
  from emp
 order by sal
    )
    )
 where rn between 6 and 10
```

```
    SAL
-----
    1300
    1500
    1600
    2450
    2850
```

使用函数 **ROWNUM** 时，必须增加一层子查询嵌套。最内层的子查询根据薪水对行进行排序，中间的子查询给行添加行号，而最外层的 **SELECT** 返回你查找的数据。

11.2 在表中跳过 n 行数据

1. 问题

你想编写一个查询，以每次跳过一人的方式返回 EMP 表中的员工。换言之，你想返回第一个员工、第三个员工.....例如，对于下面的结果集：

ENAME

ADAMS
ALLEN
BLAKE
CLARK
FORD
JAMES
JONES
KING
MARTIN
MILLER
SCOTT
SMITH
TURNER
WARD

你想从中返回如下结果集。

ENAME

ADAMS
BLAKE
FORD
JONES
MARTIN
SCOTT
TURNER

12. 解决方案

要跳过结果集中的第 2 行、第 4 行或第 n 行，必须先对结果集进行排序，否则将不存在“第一个”“下一个”“第二个”或“第四个”的概念。

使用窗口函数 **ROW_NUMBER OVER** 给每一行都指定一个编号，然后结合使用这些编号和求模函数来跳过不想返回的行。在 DB2、MySQL、PostgreSQL 和 Oracle 中，求模函数为 **MOD**。在 SQL Server 中，使用运算符 **%** 来求模。下面的示例使用了 **MOD** 来跳过偶数行。

```
1 select ename
2   from (
3 select row_number() over (order by ename) rn,
4        ename
5   from emp
6   ) x
7  where mod(rn,2) = 1
```

13. 讨论

在内嵌视图 **X** 中，窗口函数 **ROW_NUMBER OVER** 会给每一行都指定一个排名（即便员工名相同，排名也不同），结果如下所示。

```
select row_number() over (order by ename) rn, ename
   from emp

RN ENAME
-- -----
1 ADAMS
2 ALLEN
3 BLAKE
4 CLARK
```

5	FORD
6	JAMES
7	JONES
8	KING
9	MARTIN
10	MILLER
11	SCOTT
12	SMITH
13	TURNER
14	WARD

最后，使用求模运算每隔一行跳过一行。

11.3 在外连接中使用OR逻辑

1. 问题

你想返回部门编号为 10 和部门编号为 20 的所有员工的名字及其所属部门的信息，以及部门编号为 30 和部门编号为 40 的部门信息（但不返回这些部门的员工信息）。你首先想到的可能是像下面这样做。

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d, emp e
 where d.deptno = e.deptno
    and (e.deptno = 10 or e.deptno = 20)
 order by 2
```

ENAME	DEPTNO	DNAME	LOC
CLARK	10	ACCOUNTING	NEW YORK
KING	10	ACCOUNTING	NEW YORK
MILLER	10	ACCOUNTING	NEW YORK
SMITH	20	RESEARCH	DALLAS
ADAMS	20	RESEARCH	DALLAS
FORD	20	RESEARCH	DALLAS
SCOTT	20	RESEARCH	DALLAS
JONES	20	RESEARCH	DALLAS

由于这个查询使用的是内连接，因此结果集中没有部门编号为 30 和部门编号为 40 的部门信息。

你尝试使用下面的查询将 EMP 表外连接到 DEPT 表，但结果还是不正确。

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d left join emp e
    on (d.deptno = e.deptno)
 where e.deptno = 10
    or e.deptno = 20
```


order by 2			
ENAME	DEPTNO	DNAME	LOC
-----	-----	-----	-----
CLARK	10	ACCOUNTING	NEW YORK
KING	10	ACCOUNTING	NEW YORK
MILLER	10	ACCOUNTING	NEW YORK
SMITH	20	RESEARCH	DALLAS
ADAMS	20	RESEARCH	DALLAS
FORD	20	RESEARCH	DALLAS
SCOTT	20	RESEARCH	DALLAS
JONES	20	RESEARCH	DALLAS

你希望最终的结果集像下面这样。

ENAME	DEPTNO	DNAME	LOC
-----	-----	-----	-----
CLARK	10	ACCOUNTING	NEW YORK
KING	10	ACCOUNTING	NEW YORK
MILLER	10	ACCOUNTING	NEW YORK
SMITH	20	RESEARCH	DALLAS
JONES	20	RESEARCH	DALLAS
SCOTT	20	RESEARCH	DALLAS
ADAMS	20	RESEARCH	DALLAS
FORD	20	RESEARCH	DALLAS
	30	SALES	CHICAGO
	40	OPERATIONS	BOSTON

12. 解决方案

将 OR 条件移到 JOIN 子句中。

```

1  select e.ename, d.deptno, d.dname, d.loc
2    from dept d left join emp e
3      on (d.deptno = e.deptno
4         and (e.deptno=10 or e.deptno=20))
5   order by 2

```

也可以先在内嵌视图中根据 **EMP.DEPTNO** 进行筛选，然后再进行外连接。

```
1 select e.ename, d.deptno, d.dname, d.loc
2   from dept d
3   left join
4       (select ename, deptno
5        from emp
6        where deptno in ( 10, 20 )
7        ) e on ( e.deptno = d.deptno )
8 order by 2
```

13. 讨论

DB2、MySQL、PostgreSQL 和 SQL Server

对于这些 RDBMS，提供了两种解决方案。第一种解决方案是将 **OR** 条件移到 **JOIN** 子句中，使其成为连接条件的一部分。这样便可以对从 **EMP** 表返回的行进行筛选，同时不会丢失从 **DEPT** 表返回的有关部门编号为 30 和部门编号为 40 的部门信息。

第二种解决方案是将筛选器移到内嵌视图中。内嵌视图 **E** 会根据 **EMP.DEPTNO** 进行筛选，并返回感兴趣的 **EMP** 行。然后，将这些行外连接到 **DEPT** 表。由于在这个外连接中锚点表（anchor table）为 **DEPT**，因此将返回包括部门编号为 30 和部门编号为 40 在内的所有部门。

11.4 确定哪些行是互逆的

1. 问题

你有一张包含两次考试结果的表，你想确定哪两组成绩是互逆的。请看如下来自视图 V 的结果集。

```
select *  
  from V
```

TEST1	TEST2
-------	-------

20	20
50	25
20	20
60	30
70	90
80	130
90	70
100	50
110	55
120	60
130	80
140	70

从这些结果可知，TEST1 为 70、TEST2 为 90 与 TEST1 为 90、TEST2 为 70 是互逆的。同理，TEST1 为 80、TEST2 为 130 与 TEST1 为 130、TEST2 为 80 是互逆的。另外，TEST1 为 20、TEST2 为 20 与 TEST2 为 20、TEST1 为 20 也是互逆的。对于两组互逆的成绩，你只想返回其中的一组。换言之，你希望结果集是这样的。

TEST1	TEST2
20	20
70	90
80	130

而不是这样的。

TEST1	TEST2
20	20
20	20
70	90
80	130
90	70
130	80

12. 解决方案

使用自连接找出这样的行，即一行的 **TEST1** 和 **TEST2** 分别与另一行的 **TEST2** 和 **TEST1** 相等。

```
select distinct v1.*
  from V v1, V v2
 where v1.test1 = v2.test2
    and v1.test2 = v2.test1
    and v1.test1 <= v1.test2
```

13. 讨论

自连接会生成笛卡儿积，以使每一行的 **TEST1** 和 **TEST2** 都能与另一行的 **TEST2** 和 **TEST1** 进行比较。下面的查询会找出互逆的行。

```
select v1.*
  from V v1, V v2
 where v1.test1 = v2.test2
    and v1.test2 = v2.test1
```

TEST1	TEST2
-------	-------

20	20
20	20
20	20
20	20
90	70
130	80
70	90
80	130

使用关键字 **DISTINCT** 可以消除最终结果集中的重复行。在 **WHERE** 子句中，最后的筛选器（**and V1.TEST1 <= V1.TEST2**）将确保只返回两组互逆成绩中的一组（**TEST1** 小于或等于 **TEST2** 的那组）。

11.5 返回前 n 条记录

1. 问题

你想限制结果集的规模，使其只包含排名前 n 的记录。例如，你想返回薪水排在前 5 位的员工的名字和薪水。

2. 解决方案

解决这个问题需要使用窗口函数。具体使用哪个窗口函数取决于你在不分伯仲的情况下要怎么办？下面的解决方案使用了函数 **DENSE_RANK**，在薪水相同的情况下，只将总数加 1。

```
1  select ename,sal
2    from (
3  select ename, sal,
4         dense_rank() over (order by sal desc) dr
5    from emp
6       ) x
7  where dr <= 5
```

返回的行数可能超过 5，但这些行只包含 5 个不同的薪水值。如果不管是否存在薪水相同的情况，你都只想返回 5 行数据，那么可以使用函数 **ROW_NUMBER OVER**（因为这个函数不允许平局）。

3. 讨论

所有的工作都是由内嵌视图 **X** 中的窗口函数 **DENSE_RANK**

OVER 完成的。下面显示了应用这个函数得到的完整结果。

```
select ename, sal,  
       dense_rank() over (order by sal desc) dr  
from emp
```

ENAME	SAL	DR
KING	5000	1
SCOTT	3000	2
FORD	3000	2
JONES	2975	3
BLAKE	2850	4
CLARK	2450	5
ALLEN	1600	6
TURNER	1500	7
MILLER	1300	8
WARD	1250	9
MARTIN	1250	9
ADAMS	1100	10
JAMES	950	11
SMITH	800	12

现在只需返回 DR 小于或等于 5 的行。

11.6 找出值最高和最低的记录

1. 问题

你想找出表中的极值。例如，你想在 **EMP** 表中找出薪水最高和薪水最低的员工。

2. 解决方案

DB2、Oracle 和 SQL Server

使用窗口函数 **MIN OVER** 和 **MAX OVER** 分别找出最低薪水和最高薪水。

```
1 select ename
2   from (
3 select ename, sal,
4         min(sal)over() min_sal,
5         max(sal)over() max_sal
6   from emp
7        ) x
8  where sal in (min_sal,max_sal)
```

3. 讨论

DB2、Oracle 和 SQL Server

窗口函数 **MIN OVER** 和 **MAX OVER** 让每一行都能够访问最低薪水和最高薪水。内嵌视图 **X** 返回的结果集如下所示。

```
select ename, sal,
```


min(sal)over() min_sal, max(sal)over() max_sal from emp			
ENAME	SAL	MIN_SAL	MAX_SAL
-----	-----	-----	-----
SMITH	800	800	5000
ALLEN	1600	800	5000
WARD	1250	800	5000
JONES	2975	800	5000
MARTIN	1250	800	5000
BLAKE	2850	800	5000
CLARK	2450	800	5000
SCOTT	3000	800	5000
KING	5000	800	5000
TURNER	1500	800	5000
ADAMS	1100	800	5000
JAMES	950	800	5000
FORD	3000	800	5000
MILLER	1300	800	5000

有了这个结果集后，余下的全部工作就是返回 SAL 等于 MIN_SAL 或 MAX_SAL 的行。

11.7 查看后面的行

1. 问题

你想找出所有这样的员工，即其薪水比他获聘后第一个获聘的员工的薪水低。从下面的结果集可知，**SMITH**、**WARD**、**MARTIN**、**JAMES** 和 **MILLER** 的薪水都比他们获聘后第一个获聘的员工的薪水低，因此你想编写一个返回这些员工的查询。

ENAME	SAL	HIREDATE
-----	-----	-----
SMITH	800	17-DEC-80
ALLEN	1600	20-FEB-81
WARD	1250	22-FEB-81
JONES	2975	02-APR-81
BLAKE	2850	01-MAY-81
CLARK	2450	09-JUN-81
TURNER	1500	08-SEP-81
MARTIN	1250	28-SEP-81
KING	5000	17-NOV-81
JAMES	950	03-DEC-81
FORD	3000	03-DEC-81
MILLER	1300	23-JAN-82
SCOTT	3000	09-DEC-82
ADAMS	1100	12-JAN-83

2. 解决方案

首先要定义“后面”的含义。要确定一行位于另一行后面，必须对结果集进行排序。

可以使用窗口函数 **LEAD OVER** 来访问下一个获聘的员工的薪水，然后再检查这个薪水值是否更高。

```

1 select ename, sal, hiredate
2   from (
3 select ename, sal, hiredate,
4        lead(sal)over(order by hiredate) next_sal
5   from emp
6        ) alias
7  where sal < next_sal

```

13. 讨论

窗口函数 **LEAD OVER** 非常适合解决本实例这样的问题。使用它不仅可以提高查询的可读性，还可以提高解决方案的灵活性，因为你可以给它传递一个参数，指出要查看接下来的第几行（默认为第 1 行）。作为排序依据的列可能包含重复值时，能够往前跳多行很重要。

下面的示例表明，使用 **LEAD OVER** 来查看下一个获聘的员工的薪水易如反掌。

```

select ename, sal, hiredate,
       lead(sal)over(order by hiredate) next_sal
  from emp

```

ENAME	SAL	HIREDATE	NEXT_SAL
SMITH	800	17-DEC-80	1600
ALLEN	1600	20-FEB-81	1250
WARD	1250	22-FEB-81	2975
JONES	2975	02-APR-81	2850
BLAKE	2850	01-MAY-81	2450
CLARK	2450	09-JUN-81	1500
TURNER	1500	08-SEP-81	1250
MARTIN	1250	28-SEP-81	5000
KING	5000	17-NOV-81	950
JAMES	950	03-DEC-81	3000
FORD	3000	03-DEC-81	1300
MILLER	1300	23-JAN-82	3000

SCOTT	3000	09-DEC-82	1100
ADAMS	1100	12-JAN-83	

最后，只返回 **SAL** 比 **NEXT_SAL** 小的行。由于 **LEAD OVER** 默认往前跳 1 行，因此如果在 **EMP** 表中有多名员工的获聘日期相同，那么将比较他们的 **SAL** 值。这可能是你希望的，也可能不是。如果你的目标是将每位员工的 **SAL** 值与下一位获聘的员工的 **SAL** 值进行比较，就需要将同一天获聘的其他员工排除在外，为此可以使用如下解决方案。

```
select ename, sal, hiredate
  from (
select ename, sal, hiredate,
       lead(sal,cnt-rn+1)over(order by hiredate) next_sal
  from (
select ename,sal,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
  )
  )
 where sal < next_sal
```

该解决方案的思路是，找出当前行到要与当前行比较的行的距离。如果有 5 行的值是相同的，那么对于其中的第 1 行，需要向前跳 5 行。**CNT** 表示 **HIREDATE** 值相同的员工数量，**RN** 表示员工排名。排名是按 **HIREDATE** 分区的，因此仅当员工的 **HIREDATE** 值与其他员工相同时，其排名才可能大于 1。排名是基于 **EMPNO** 的。（这种选择是随意的。）知道有多少名员工的 **HIREDATE** 值相同以及这些员工的排名后，计算到下一个 **HIREDATE** 值的距离很简单，只需将 **HIREDATE** 值相同的员工数减去当前排名再加 1（**CNT - RN + 1**）。

14. 另请参阅

有关在存在重复值的情况下使用 **LEAD OVER** 的示例以及这种技术的详细讨论，请参阅 8.7 节和 10.2 节。

11.8 平移行值

1. 问题

你想返回每位员工的名字、薪水以及下一个更高和更低的薪水值。如果没有更高或更低的薪水值，就执行回转操作：如果当前员工的薪水是最低的，就将下一个更低的薪水设置为最高的薪水；如果当前员工的薪水是最高的，就将下一个更高的薪水设置为最低的薪水。你想返回如下结果集。

ENAME	SAL	FORWARD	REWIND
-----	-----	-----	-----
SMITH	800	950	5000
JAMES	950	1100	800
ADAMS	1100	1250	950
WARD	1250	1250	1100
MARTIN	1250	1300	1250
MILLER	1300	1500	1250
TURNER	1500	1600	1300
ALLEN	1600	2450	1500
CLARK	2450	2850	1600
BLAKE	2850	2975	2450
JONES	2975	3000	2850
SCOTT	3000	3000	2975
FORD	3000	5000	3000
KING	5000	800	3000

2. 解决方案

窗口函数 **LEAD OVER** 和 **LAG OVER** 能够轻松解决这个问题，而且编写出来的查询的可读性极高。要访问当前行的前一行和下一行，可以分别使用窗口函数 **LAG OVER** 和 **LEAD OVER**。

```

1  select ename,sal,
2         coalesce(lead(sal)over(order by sal),min(sal)over())
forward,
3         coalesce(lag(sal)over(order by sal),max(sal)over())
rewind
4  from emp

```

13. 讨论

默认情况下，窗口函数 **LAG OVER** 和 **LEAD OVER** 会分别返回前一行的值和后一行的值。在 **OVER** 子句的 **ORDER BY** 部分，可以定义“前一行”和“后一行”。如果查看上述解决方案，你将发现第一步是返回后一行和前一行，而行的排列顺序是基于 **SAL** 的。

```

select ename,sal,
       lead(sal)over(order by sal) forward,
       lag(sal)over(order by sal) rewind
from emp

```

ENAME	SAL	FORWARD	REWIND
SMITH	800	950	
JAMES	950	1100	800
ADAMS	1100	1250	950
WARD	1250	1250	1100
MARTIN	1250	1300	1250
MILLER	1300	1500	1250
TURNER	1500	1600	1300
ALLEN	1600	2450	1500
CLARK	2450	2850	1600
BLAKE	2850	2975	2450
JONES	2975	3000	2850
SCOTT	3000	3000	2975
FORD	3000	5000	3000
KING	5000	3000	

注意，员工 SMITH 的 REWIND 为 NULL，而员工 KING 的 FORWARD 为 NULL，因为这两名员工的薪水分别是最低和最高的。本节“问题”部分指出，如果 FORWARD 或 REWIND 的值为 NULL，就执行回转操作。这意味着对于薪水最高的员工，应将其 FORWARD 值设置为表中最低的薪水值，而对于薪水最低的员工，应将其 REWIND 值设置为表中最高的薪水值。在没有指定分区或窗口的情况下（OVER 子句后面的括号内是空的），窗口函数 MIN OVER 和 MAX OVER 将分别返回表中最低薪水值和最高薪水值。

```
select ename,sal,
       coalesce(lead(sal)over(order by sal),min(sal)over())
forward,
       coalesce(lag(sal)over(order by sal),max(sal)over()) rewind
from emp
```

ENAME	SAL	FORWARD	REWIND
SMITH	800	950	5000
JAMES	950	1100	800
ADAMS	1100	1250	950
WARD	1250	1250	1100
MARTIN	1250	1300	1250
MILLER	1300	1500	1250
TURNER	1500	1600	1300
ALLEN	1600	2450	1500
CLARK	2450	2850	1600
BLAKE	2850	2975	2450
JONES	2975	3000	2850
SCOTT	3000	3000	2975
FORD	3000	5000	3000
KING	5000	800	3000

LAG OVER 和 LEAD OVER 另一个很有用的特征是，可以指定你能向前或向后跳多远。在本例中，你只往前或往后跳了一行。要分别往前跳 3 行、往后跳 5 行很容易，只需指定 3 和 5。


```

select ename,sal,
       lead(sal,3)over(order by sal) forward,
       lag(sal,5)over(order by sal) rewind
from emp

```

ENAME	SAL	FORWARD	REWIND
-----	-----	-----	-----
SMITH	800	1250	
JAMES	950	1250	
ADAMS	1100	1300	
WARD	1250	1500	
MARTIN	1250	1600	
MILLER	1300	2450	800
TURNER	1500	2850	950
ALLEN	1600	2975	1100
CLARK	2450	3000	1250
BLAKE	2850	3000	1250
JONES	2975	5000	1300
SCOTT	3000		1500
FORD	3000		1600
KING	5000		2450

11.9 结果排名

1. 问题

你想对 **EMP** 表中的薪水排名，并保留相同的值。你想返回如下结果集。

RNK	SAL
1	800
2	950
3	1100
4	1250
4	1250
5	1300
6	1500
7	1600
8	2450
9	2850
10	2975
11	3000
11	3000
12	5000

2. 解决方案

窗口函数使得编写排名查询超级简单。对排名来说，有 3 个窗口函数特别有用，即 **DENSE_RANK OVER**、**ROW_NUMBER OVER** 和 **RANK OVER**。

由于你想保留相同的值，因此使用窗口函数 **DENSE_RANK OVER**。

```
1 select dense_rank() over(order by sal) rnk, sal
```

13. 讨论

本例中所有的工作都是由窗口函数 **DENSE_RANK OVER** 完成的。在关键字 **OVER** 后面的括号中，使用一个 **ORDER BY** 子句来指定行排名依据。上述解决方案使用的是 **ORDER BY SAL**，因此将按薪水从低到高的顺序对 **EMP** 表中的行进行排名。

11.10 消除重复行

1. 问题

你想找出 **EMP** 表中不同的职位类型，但不想看到重复的行。结果集如下所示。

```
JOB
-----
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

2. 解决方案

所有 **RDBMS** 都支持关键字 **DISTINCT**，这也是消除结果集中重复行的最简单机制。然而，本实例还将介绍另外两种消除重复行的方法。

使用 **DISTINCT**（或 **GROUP BY**）的传统方法肯定管用。下面是另一种解决方案，使用的是窗口函数 **ROW_NUMBER OVER**。

```
1  select job
2    from (
3  select job,
4         row_number()over(partition by job order by job) rn
5    from emp
6   ) x
7  where rn = 1
```

传统解决方案

使用关键字 **DISTINCT** 来消除结果集中的重复行。

```
select distinct job
  from emp
```

另外，也可以使用 **GROUP BY** 来消除重复行。

```
select job
  from emp
 group by job
```

13. 讨论

DB2、Oracle 和 SQL Server

该解决方案以不同寻常的方式看待分区型窗口函数。通过在 **ROW_NUMBER** 的 **OVER** 子句中使用 **PARTITION BY**，可以在每次遇到新职位时都将 **ROW_NUMBER** 返回的值重置为 1。下面显示了内嵌视图 **X** 返回的结果。

```
select job,
       row_number()over(partition by job order by job) rn
  from emp
```

JOB	RN
ANALYST	1
ANALYST	2
CLERK	1
CLERK	2
CLERK	3
CLERK	4
MANAGER	1
MANAGER	2

MANAGER	3
PRESIDENT	1
SALESMAN	1
SALESMAN	2
SALESMAN	3
SALESMAN	4

以上解决方案给每一行都指定了一个递增的序列号，同时每当遇到新职位时，都将序列号重置为 1。要剔除重复行，保留 RN 为 1 的行即可。

使用 **ROW_NUMBER OVER** 时，**ORDER BY** 子句必不可少（DB2 中除外），但不影响结果。对于包含相同职位的行，返回哪一行无关紧要，只要返回了每种职位。

传统解决方案

第一种传统解决方案演示了如何使用关键字 **DISTINCT** 来消除结果集中的重复行。记住，关键字 **DISTINCT** 针对的是整个 **SELECT** 列表，因此在这个列表中包含更多列将影响结果集。请看下面两个查询之间的差别。

<code>select distinct job</code> <code>from emp</code>	<code>select distinct job, deptno</code> <code>from emp</code>
JOB -----	JOB DEPTNO -----
ANALYST	ANALYST 20
CLERK	CLERK 10
MANAGER	CLERK 20
PRESIDENT	CLERK 30
SALESMAN	MANAGER 10
	MANAGER 20
	MANAGER 30
	PRESIDENT 10
	SALESMAN 30

在 **SELECT** 列表中添加 **DEPTNO** 后，返回的是 **EMP** 表中不同

的 JOB 和 DEPTNO 组合值。

第二种传统解决方案使用 **GROUP BY** 来消除重复行。这种使用 **GROUP BY** 的方式很常见，但别忘了，**GROUP BY** 和 **DISTINCT** 是两个截然不同的子句，并不能互换。此处使用 **GROUP BY** 来消除重复行只是出于完整性考虑，因为你肯定会遇到这种使用 **GROUP BY** 的方式。

11.11 查找马值

1. 问题

你想返回一个结果集，其中包含每位员工的名字、所属的部门、薪水、获聘日期以及所属部门最后聘请的员工的薪水。你想返回的结果集如下所示。

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	MILLER	1300	23-JAN-2007	1300
10	KING	5000	17-NOV-2006	1300
10	CLARK	2450	09-JUN-2006	1300
20	ADAMS	1100	12-JAN-2007	1100
20	SCOTT	3000	09-DEC-2007	1100
20	FORD	3000	03-DEC-2006	1100
20	JONES	2975	02-APR-2006	1100
20	SMITH	800	17-DEC-2005	1100
30	JAMES	950	03-DEC-2006	950
30	MARTIN	1250	28-SEP-2006	950
30	TURNER	1500	08-SEP-2006	950
30	BLAKE	2850	01-MAY-2006	950
30	WARD	1250	22-FEB-2006	950
30	ALLEN	1600	20-FEB-2006	950

其中 **LATEST_SAL** 列的值就是马值（knight value），因为查找这些值的方法与国际象棋中马的走法类似。确定结果的方式与确定马的下一个位置相同：先跳到某一行，然后跳到某一列，如图 11-1 所示。要找到正确的 **LATEST_SAL** 值，必须先定位到（跳到）当前部门中最后的 **HIREDATE** 所在的行，然后跳到该行的 **SAL** 列。

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
30	JAMES	950	03-DEC-2006	950
30	MARTIN	1250	28-SEP-2006	950
30	TURNER	1500	08-SEP-2006	950
30	BLAKE	2850	01-MAY-2006	950
30	WARD	1250	22-FEB-2006	950
30	ALLEN	1600	20-FEB-2006	950

图 11-1：马值是通过跳到特定行再跳到特定列得到的



“马值”这个术语是我的一个非常机灵的同事 Kay Young 杜撰的。在 Kay 帮我审阅完本书的实例后，我跟他说不知道如何命名这个实例。考虑到需要先查看一行，再跳到另一行获取值，Kay 想出了“马值”这个名词。

12. 解决方案

DB2 和 SQL Server

在子查询中使用 CASE 表达式，将每个部门最后聘请的员工的 LATEST_SAL 值设置为其 SAL，将其他员工的 LATEST_SAL 值设置为 0。在外部查询中，使用窗口函数 MAX OVER 将每位员工的 LATEST_SAL 值都设置为其部门的非零 LATEST_SAL 值。

```

1 select deptno,
2         ename,
3         sal,
4         hiredate,
```

```

5      max(latest_sal)over(partition by deptno) latest_sal
6  from (
7  select deptno,
8      ename,
9      sal,
10     hiredate,
11     case
12         when hiredate = max(hiredate)over(partition by
deptno)
13         then sal else 0
14     end latest_sal
15  from emp
16  ) x
17  order by 1, 4 desc

```

Oracle

使用窗口函数 **MAX OVER** 返回每个部门最后聘请的员工的 **SAL**。为此，在 **KEEP** 子句中使用函数 **DENSE_RANK** 和 **LAST**，并按 **HIREDATE** 排序，以返回给定部门中最后聘请的员工的 **SAL**。

```

1  select deptno,
2      ename,
3      sal,
4      hiredate,
5      max(sal)
6          keep(dense_rank last order by hiredate)
7          over(partition by deptno) latest_sal
8  from emp
9  order by 1, 4 desc

```

13. 讨论

DB2 和 SQL Server

在 **CASE** 表达式中使用窗口函数 **MAX OVER**，以找出每个部门

最后获聘的员工的获聘日期。如果员工的 **HIREDATE** 与 **MAX OVER** 返回的值相同，就返回该员工的 **SAL**，否则返回 0。这些操作的结果如下所示。

```
select deptno,
       ename,
       sal,
       hiredate,
       case
         when hiredate = max(hiredate)over(partition by deptno)
         then sal else 0
       end latest_sal
from emp
```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	CLARK	2450	09-JUN-2006	0
10	KING	5000	17-NOV-2006	0
10	MILLER	1300	23-JAN-2007	1300
20	SMITH	800	17-DEC-2005	0
20	ADAMS	1100	12-JAN-2007	1100
20	FORD	3000	03-DEC-2006	0
20	SCOTT	3000	09-DEC-2007	0
20	JONES	2975	02-APR-2006	0
30	ALLEN	1600	20-FEB-2006	0
30	BLAKE	2850	01-MAY-2006	0
30	MARTIN	1250	28-SEP-2006	0
30	JAMES	950	03-DEC-2006	950
30	TURNER	1500	08-SEP-2006	0
30	WARD	1250	22-FEB-2006	0

由于 **LATEST_SAL** 值要么为 0，要么为最后获聘的员工的 **SAL**，因此可以将上述查询放在一个内嵌视图中，并再次使用 **MAX OVER**，但这次返回每个部门的非零 **LATEST_SAL** 值。

```
select deptno,
       ename,
       sal,
```

```

        hiredate,
        max(latest_sal)over(partition by deptno) latest_sal
    from (
select deptno,
       ename,
       sal,
       hiredate,
       case
           when hiredate = max(hiredate)over(partition by deptno)
           then sal else 0
       end latest_sal
    from emp
    ) x
order by 1, 4 desc

```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	MILLER	1300	23-JAN-2007	1300
10	KING	5000	17-NOV-2006	1300
10	CLARK	2450	09-JUN-2006	1300
20	ADAMS	1100	12-JAN-2007	1100
20	SCOTT	3000	09-DEC-2007	1100
20	FORD	3000	03-DEC-2006	1100
20	JONES	2975	02-APR-2006	1100
20	SMITH	800	17-DEC-2005	1100
30	JAMES	950	03-DEC-2006	950
30	MARTIN	1250	28-SEP-2006	950
30	TURNER	1500	08-SEP-2006	950
30	BLAKE	2850	01-MAY-2006	950
30	WARD	1250	22-FEB-2006	950
30	ALLEN	1600	20-FEB-2006	950

Oracle

Oracle 解决方案的关键在于利用 **KEEP** 子句。**KEEP** 子句能够对分组 / 分区操作返回的行进行排名，并使用其中的第一行或最后一行。请看下面这个没有使用 **KEEP** 的解决方案。

```

select deptno,
       ename,
       sal,

```

<pre> hiredate, max(sal) over(partition by deptno) latest_sal from emp order by 1, 4 desc </pre>				
DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL

10	MILLER	1300	23-JAN-2007	5000
10	KING	5000	17-NOV-2006	5000
10	CLARK	2450	09-JUN-2006	5000
20	ADAMS	1100	12-JAN-2007	3000
20	SCOTT	3000	09-DEC-2007	3000
20	FORD	3000	03-DEC-2006	3000
20	JONES	2975	02-APR-2006	3000
20	SMITH	800	17-DEC-2005	3000
30	JAMES	950	03-DEC-2006	2850
30	MARTIN	1250	28-SEP-2006	2850
30	TURNER	1500	08-SEP-2006	2850
30	BLAKE	2850	01-MAY-2006	2850
30	WARD	1250	22-FEB-2006	2850
30	ALLEN	1600	20-FEB-2006	2850

没有使用 **KEEP** 时，**MAX OVER** 返回的是每个部门最高的薪水，而不是最后获聘的员工的薪水。在本实例中，使用 **KEEP** 并指定 **ORDER BY HIREDATE**，可以按 **HIREDATE** 对每个部门的员工进行排序。然后，**DENSE_RANK** 函数会给每一行都指定一个排名。最后，**LAST** 函数会决定将聚合函数应用于哪一行（基于 **DENSE_RANK** 排名的最后一行）。在本实例中，聚合函数 **MAX** 将应用于最后一行的 **HIREDATE** 所对应的 **SAL** 列。这相当于对于每个部门，都获取按 **HIREDATE** 排名时最后一行的 **SAL** 值。

对于每个部门的行，你都要基于 **HIREDATE** 列进行排名，然后将聚合函数 **MAX** 应用于 **SAL** 列。这种根据一个维度进行排名，并对另一个维度进行聚合的功能非常方便，无须像其他解决方案中那样使用连接和内嵌视图。最后，通过在 **KEEP** 子句后面添加 **OVER** 子句，可以在分区的每一行中都返回

KEEP“保留”的 SAL 值。

也可以按 **HIREDATE** 降序排列，并“保留”第一个 **SAL** 值。请比较下面两个查询，它们返回了相同的结果集。

```
select deptno,
       ename,
       sal,
       hiredate,
       max(sal)
         keep(dense_rank last order by hiredate)
         over(partition by deptno) latest_sal
  from emp
 order by 1, 4 desc
```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	MILLER	1300	23-JAN-2007	1300
10	KING	5000	17-NOV-2006	1300
10	CLARK	2450	09-JUN-2006	1300
20	ADAMS	1100	12-JAN-2007	1100
20	SCOTT	3000	09-DEC-2007	1100
20	FORD	3000	03-DEC-2006	1100
20	JONES	2975	02-APR-2006	1100
20	SMITH	800	17-DEC-2005	1100
30	JAMES	950	03-DEC-2006	950
30	MARTIN	1250	28-SEP-2006	950
30	TURNER	1500	08-SEP-2006	950
30	BLAKE	2850	01-MAY-2006	950
30	WARD	1250	22-FEB-2006	950
30	ALLEN	1600	20-FEB-2006	950

```
select deptno,
       ename,
       sal,
       hiredate,
       max(sal)
         keep(dense_rank first order by hiredate desc)
         over(partition by deptno) latest_sal
  from emp
 order by 1, 4 desc
```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
-----	-----	-----	-----	-----
10	MILLER	1300	23-JAN-2007	1300
10	KING	5000	17-NOV-2006	1300
10	CLARK	2450	09-JUN-2006	1300
20	ADAMS	1100	12-JAN-2007	1100
20	SCOTT	3000	09-DEC-2007	1100
20	FORD	3000	03-DEC-2006	1100
20	JONES	2975	02-APR-2006	1100
20	SMITH	800	17-DEC-2005	1100
30	JAMES	950	03-DEC-2006	950
30	MARTIN	1250	28-SEP-2006	950
30	TURNER	1500	08-SEP-2006	950
30	BLAKE	2850	01-MAY-2006	950
30	WARD	1250	22-FEB-2006	950
30	ALLEN	1600	20-FEB-2006	950

11.12 生成简单预测

1. 问题

你想根据当前的数据返回表示未来行动的行和列。例如，请看下面的结果集。

ID	ORDER_DATE	PROCESS_DATE
1	25-SEP-2005	27-SEP-2005
2	26-SEP-2005	28-SEP-2005
3	27-SEP-2005	29-SEP-2005

对于这个结果集中的每一行，你都想返回 3 行（当前行和另外两行）数据。除了额外的行，你还想返回另外两列，其中包含预期的订单处理日期。

从前面的结果集可知，订单将在两天后处理。假设订单处理完成后，下一步是订单核验，最后一步是发货。订单核验在订单处理后一天进行，而发货在订单核验后一天进行。你想返回一个呈现整个过程的结果集。换言之，你想将前面的结果集转换为下面的结果集。

ID	ORDER_DATE	PROCESS_DATE	VERIFIED	SHIPPED
1	25-SEP-2005	27-SEP-2005		
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	29-SEP-2005
2	26-SEP-2005	28-SEP-2005		
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	30-SEP-2005
3	27-SEP-2005	29-SEP-2005		
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	01-OCT-2005

12. 解决方案

关键是使用笛卡儿积为每个订单再生成两行数据，然后使用 **CASE** 表达式来创建所需的列值。

DB2、MySQL 和 SQL Server

使用递归式 **WITH** 子句生成笛卡儿积所需的行。除了用来获取当前日期的函数外，DB2 和 SQL Server 解决方案完全相同。DB2 解决方案使用的是 **CURRENT_DATE**，而 SQL Server 解决方案使用的是 **GETDATE**。MySQL 解决方案使用的是 **CURDATE**，并需要在 **WITH** 后面插入关键字 **RECURSIVE**，以指出这是一个递归式 CTE。下面显示了 SQL Server 解决方案。

```
1 with nrows(n) as (  
2   select 1 from t1 union all  
3   select n+1 from nrows where n+1 <= 3  
4 )  
5 select id,  
6        order_date,  
7        process_date,  
8        case when nrows.n >= 2  
9            then process_date+1  
10         else null  
11        end as verified,  
12        case when nrows.n = 3  
13            then process_date+2  
14         else null  
15        end as shipped  
16   from (  
17   select nrows.n id,  
18         getdate()+nrows.n as order_date,  
19         getdate()+nrows.n+2 as process_date  
20   from nrows  
21   ) orders, nrows  
22  order by 1
```

Oracle

使用分层子句 **CONNECT BY** 生成笛卡儿积所需的 3 行数据。使用 **WITH** 子句，以便重用 **CONNECT BY** 返回的结果，从而避免再次调用它。

```
1 with nrows as (  
2   select level n  
3     from dual  
4   connect by level <= 3  
5 )  
6 select id,  
7        order_date,  
8        process_date,  
9        case when nrows.n >= 2  
10         then process_date+1  
11         else null  
12       end as verified,  
13        case when nrows.n = 3  
14         then process_date+2  
15         else null  
16       end as shipped  
17 from (  
18 select nrows.n id,  
19        sysdate+nrows.n as order_date,  
20        sysdate+nrows.n+2 as process_date  
21   from nrows  
22  ) orders, nrows
```

PostgreSQL

可以使用多种不同的方式生成笛卡儿积，该解决方案使用了 PostgreSQL 函数 **GENERATE_SERIES**。

```
1 select id,  
2        order_date,  
3        process_date,  
4        case when gs.n >= 2  
5         then process_date+1  
6         else null  
7       end as verified,  
8        case when gs.n = 3  
9         then process_date+2
```

```

10          else null
11      end as shipped
12  from (
13  select gs.id,
14         current_date+gs.id as order_date,
15         current_date+gs.id+2 as process_date
16  from generate_series(1,3) gs (id)
17       ) orders,
18       generate_series(1,3)gs(n)

```

MySQL

MySQL 不支持自动生成行的函数。

13. 讨论

DB2、MySQL 和 SQL Server

内嵌视图 **ORDERS** 会返回本节“问题”部分呈现的结果集。

```

with nrows(n) as (
select 1 from t1 union all
select n+1 from nrows where n+1 <= 3
)
select nrows.n id, getdate()+nrows.n as order_date,
       getdate()+nrows.n+2 as process_date
  from nrows

```

ID	ORDER_DATE	PROCESS_DATE
1	25-SEP-2005	27-SEP-2005
2	26-SEP-2005	28-SEP-2005
3	27-SEP-2005	29-SEP-2005

这个查询使用 **WITH** 子句创建了 3 行数据，用于表示要处理的订单。**NROWS** 返回了值 1、2 和 3，然后将 **GETDATE**（在 DB2 中为 **CURRENT_DATE**，在 MySQL 中为 **CURDATE()**）加

上这些数字，以生成订单的日期。本节“问题”部分说过，处理需要花两天时间，因此这个查询给 **ORDER_DATE** 加了两天（给 **GETDATE** 加上 **NROWS** 返回的值，再加上两天）。

有了基本结果集后，下一步是生成笛卡儿积，因为需要给每个订单返回 3 行数据。使用 **NROWS** 生成笛卡儿积，以便给每个订单返回 3 行数据。

```
with nrows(n) as (  
  select 1 from t1 union all  
  select n+1 from nrows where n+1 <= 3  
)  
select nrows.n,  
       orders.*  
  from (  
select nrows.n id,  
       getdate()+nrows.n as order_date,  
       getdate()+nrows.n+2 as process_date  
  from nrows  
    ) orders, nrows  
 order by 2,1
```

N	ID	ORDER_DATE	PROCESS_DATE
1	1	25-SEP-2005	27-SEP-2005
2	1	25-SEP-2005	27-SEP-2005
3	1	25-SEP-2005	27-SEP-2005
1	2	26-SEP-2005	28-SEP-2005
2	2	26-SEP-2005	28-SEP-2005
3	2	26-SEP-2005	28-SEP-2005
1	3	27-SEP-2005	29-SEP-2005
2	3	27-SEP-2005	29-SEP-2005
3	3	27-SEP-2005	29-SEP-2005

每个订单都有 3 行数据后，只需使用 **CASE** 表达式来创建额外的列，用于表示验证和发货状态。

对于每个订单的第一行，**VERIFIED** 和 **SHIPPED** 列都应 **NULL**；对于每个订单的第二行，**SHIPPED** 列应为 **NULL**；对

于每个订单的第三行，每一列都应为非 **NULL** 值。最终的结果集如下所示。

```
with nrows(n) as (  
  select 1 from t1 union all  
  select n+1 from nrows where n+1 <= 3  
)  
select id,  
       order_date,  
       process_date,  
       case when nrows.n >= 2  
            then process_date+1  
            else null  
  
       end as verified,  
       case when nrows.n = 3  
            then process_date+2  
            else null  
       end as shipped  
  from (  
select nrows.n id,  
       getdate()+nrows.n as order_date,  
       getdate()+nrows.n+2 as process_date  
  from nrows  
  ) orders, nrows  
 order by 1
```

ID	ORDER_DATE	PROCESS_DATE	VERIFIED	SHIPPED
1	25-SEP-2005	27-SEP-2005		
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	29-SEP-2005
2	26-SEP-2005	28-SEP-2005		
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	30-SEP-2005
3	27-SEP-2005	29-SEP-2005		
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	01-OCT-2005

最终的结果集呈现了从收到订单到发货的整个订单处理过程。

Oracle

内嵌视图 **ORDERS** 返回了本节“问题”部分呈现的结果集。

```
with nrows as (  
  select level n  
    from dual  
 connect by level <= 3  
)  
select nrows.n id,  
       sysdate+nrows.n order_date,  
       sysdate+nrows.n+2 process_date  
  from nrows
```

ID	ORDER_DATE	PROCESS_DATE
1	25-SEP-2005	27-SEP-2005
2	26-SEP-2005	28-SEP-2005
3	27-SEP-2005	29-SEP-2005

这个查询使用 **CONNECT BY** 创建了 3 行数据，用于表示要处理的订单。使用 **WITH** 子句通过 **NROWS.N** 来引用 **CONNECT BY** 返回的行。**CONNECT BY** 返回了值 1、2 和 3，然后将 **SYSDATE** 加上这些数字，以生成订单的日期。本节“问题”部分说过，处理需要花两天时间，因此这个查询给 **ORDER_DATE** 加了两天（给 **SYSDATE** 加上 **CONNECT BY** 返回的值，再加上两天）。

有了基本结果集后，下一步是生成笛卡儿积，因为需要给每个订单返回 3 行数据。使用 **NROWS** 生成笛卡儿积，以便给每个订单返回 3 行数据。

```
with nrows as (  
  select level n  
    from dual  
 connect by level <= 3  
)  
select nrows.n,
```

```

        orders.*
    from (
select nrow.n id,
        sysdate+nrow.n order_date,
        sysdate+nrow.n+2 process_date
    from nrow
    ) orders, nrow

```

N	ID	ORDER_DATE	PROCESS_DATE
1	1	25-SEP-2005	27-SEP-2005
2	1	25-SEP-2005	27-SEP-2005
3	1	25-SEP-2005	27-SEP-2005
1	2	26-SEP-2005	28-SEP-2005
2	2	26-SEP-2005	28-SEP-2005
3	2	26-SEP-2005	28-SEP-2005
1	3	27-SEP-2005	29-SEP-2005
2	3	27-SEP-2005	29-SEP-2005
3	3	27-SEP-2005	29-SEP-2005

每个订单都有 3 行数据后，只需使用 **CASE** 表达式来创建额外的列，用于表示验证和发货状态。

对于每个订单的第一行，**VERIFIED** 和 **SHIPPED** 列都应为 **NULL**；对于每个订单的第二行，**SHIPPED** 列应为 **NULL**；对于每个订单的第三行，每一列都应为非 **NULL** 值。最终的结果集如下所示。

```

with nrow as (
select level n
  from dual
 connect by level <= 3
)
select id,
       order_date,
       process_date,
       case when nrow.n >= 2
            then process_date+1
            else null
       end as verified,
       case when nrow.n = 3

```

```

        then process_date+2
        else null
    end as shipped
from (
select nrow.n id,
       sysdate+nrow.n order_date,
       sysdate+nrow.n+2 process_date
from nrow
) orders, nrow

```

ID	ORDER_DATE	PROCESS_DATE	VERIFIED	SHIPPED
1	25-SEP-2005	27-SEP-2005		
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	29-SEP-2005
2	26-SEP-2005	28-SEP-2005		
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	30-SEP-2005
3	27-SEP-2005	29-SEP-2005		
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	01-OCT-2005

最终的结果集呈现了从收到订单到发货的整个订单处理过程。

PostgreSQL

内嵌视图 **ORDERS** 返回了本节“问题”部分呈现的结果集。

```

select gs.id,
       current_date+gs.id as order_date,
       current_date+gs.id+2 as process_date
from generate_series(1,3) gs (id)

```

ID	ORDER_DATE	PROCESS_DATE
1	25-SEP-2005	27-SEP-2005
2	26-SEP-2005	28-SEP-2005
3	27-SEP-2005	29-SEP-2005

这个查询使用函数 **GENERATE_SERIES** 创建了 3 行数据，用

于表示要处理的订单。**GENERATE_SERIES** 返回了值 1、2 和 3，然后将 **CURRENT_DATE** 加上这些数字，以生成订单的日期。本节“问题”部分说过，处理需要花两天时间，因此这个查询给 **ORDER_DATE** 加了两天（给 **CURRENT_DATE** 加上 **GENERATE_SERIES** 返回的值，再加上两天）。有了基本结果集后，下一步是生成笛卡儿积，因为需要给每个订单返回 3 行数据。使用 **GENERATE_SERIES** 生成笛卡儿积，以便给每个订单返回 3 行数据。

```
select gs.n,
       orders.*
  from (
select gs.id,
       current_date+gs.id as order_date,
       current_date+gs.id+2 as process_date
  from generate_series(1,3) gs (id)
       ) orders,
       generate_series(1,3)gs(n)
```

N	ID	ORDER_DATE	PROCESS_DATE
1	1	25-SEP-2005	27-SEP-2005
2	1	25-SEP-2005	27-SEP-2005
3	1	25-SEP-2005	27-SEP-2005
1	2	26-SEP-2005	28-SEP-2005
2	2	26-SEP-2005	28-SEP-2005
3	2	26-SEP-2005	28-SEP-2005
1	3	27-SEP-2005	29-SEP-2005
2	3	27-SEP-2005	29-SEP-2005
3	3	27-SEP-2005	29-SEP-2005

每个订单都有 3 行数据后，只需使用 **CASE** 表达式来创建额外的列，用于表示验证和发货状态。

对于每个订单的第一行，**VERIFIED** 和 **SHIPPED** 列都应为 **NULL**；对于每个订单的第二行，**SHIPPED** 列应为 **NULL**；对于每个订单的第三行，每一列都应为非 **NULL** 值。最终的结果集如下所示。

```

select id,
       order_date,
       process_date,
       case when gs.n >= 2
            then process_date+1
            else null
       end as verified,
       case when gs.n = 3
            then process_date+2
            else null
       end as shipped
  from (
select gs.id,
       current_date+gs.id as order_date,
       current_date+gs.id+2 as process_date
  from generate_series(1,3) gs(id)
       ) orders,
       generate_series(1,3)gs(n)

```

ID	ORDER_DATE	PROCESS_DATE	VERIFIED	SHIPPED
---	-----	-----	-----	-----
1	25-SEP-2005	27-SEP-2005		
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	29-SEP-2005
2	26-SEP-2005	28-SEP-2005		
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	30-SEP-2005
3	27-SEP-2005	29-SEP-2005		
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	01-OCT-2005

最终的结果集呈现了从收到订单到发货的整个订单处理过程。

11.13 小结

本章实例呈现的是使用单个函数无法解决的实际问题，它们是商业用户经常会要求你解决的问题。

第 12 章 报表制作和整形

本章介绍有助于制作报表的查询，这些查询通常需要考虑与报表相关的格式设置，还需使用多级聚合。本章的另一个焦点是结果集转置：通过将行转换为列来对数据进行整形。

一般而言，这些解决方案有一个共同特征，就是能够以不同于存储格式和形状的方式呈现数据。熟悉转置后，你肯定会在本章未涉及的场景中找到其用武之地。

12.1 将结果集转置为一行

1. 问题

你想将多行中的值转换为单行中的列。例如，有一个结果集，它显示了每个部门的员工数量。

DEPTNO	CNT
10	3
20	5
30	6

而你要调整输出的格式，让这个结果集展示如下结果。

DEPTNO_10	DEPTNO_20	DEPTNO_30
3	5	6

这是一个经典示例，它以不同于存储形状的方式呈现数据。

2. 解决方案

使用 **CASE** 和聚合函数 **SUM** 来转置这个结果集。

```
1 select sum(case when deptno=10 then 1 else 0 end) as deptno_10,  
2         sum(case when deptno=20 then 1 else 0 end) as deptno_20,  
3         sum(case when deptno=30 then 1 else 0 end) as deptno_30  
4   from emp
```

3. 讨论

本实例很好地说明了转置，其概念很简单：对于非转置查询返回的每一行数据，使用 **CASE** 表达式将行拆分成列。然后，由于这里的问题是计算每个部门的员工数量，因此使用聚合函数 **SUM**。如果不明白其中的工作原理，请执行前面的查询，但删除其中的聚合函数 **SUM**，并在 **SELECT** 列表中包含 **DEPTNO** 以提高可读性。

```
select deptno,
       case when deptno=10 then 1 else 0 end as deptno_10,
       case when deptno=20 then 1 else 0 end as deptno_20,
       case when deptno=30 then 1 else 0 end as deptno_30
from emp
order by 1
```

DEPTNO	DEPTNO_10	DEPTNO_20	DEPTNO_30
-----	-----	-----	-----
10	1	0	0
10	1	0	0
10	1	0	0
20	0	1	0
20	0	1	0
20	0	1	0
20	0	1	0
30	0	0	1
30	0	0	1
30	0	0	1
30	0	0	1
30	0	0	1
30	0	0	1

可以将每个 **CASE** 表达式视为一个标志，以决定当前行属于哪个部门。至此，“将行转换为列”的工作已完成，下一步是分别计算 **DEPTNO_10**、**DEPTNO_20** 和 **DEPTNO_30** 列值的总和，并按 **DEPTNO** 分组，结果如下所示。

```
select deptno,
       sum(case when deptno=10 then 1 else 0 end) as deptno_10,
       sum(case when deptno=20 then 1 else 0 end) as deptno_20,
       sum(case when deptno=30 then 1 else 0 end) as deptno_30
```

from emp group by deptno			
DEPTNO	DEPTNO_10	DEPTNO_20	DEPTNO_30
-----	-----	-----	-----
10	3	0	0
20	0	5	0
30	0	0	6

如果你查看这个结果集，就会明白输出是合理的。例如，在 **DEPTNO 10** 行，**DEPTNO_10** 列的值为 3（10 号部门有 3 位员工），而其他列的值为 0。由于这里的目标是返回一行数据，因此最后一步是删除 **DEPTNO** 和 **GROUP BY** 子句，只计算 **CASE** 表达式的总和。

select sum(case when deptno=10 then 1 else 0 end) as deptno_10, sum(case when deptno=20 then 1 else 0 end) as deptno_20, sum(case when deptno=30 then 1 else 0 end) as deptno_30 from emp		
DEPTNO_10	DEPTNO_20	DEPTNO_30
-----	-----	-----
3	5	6

下面是你可能遇到的解决这种问题的另一种方法。

select max(case when deptno=10 then empcount else null end) as deptno_10 max(case when deptno=20 then empcount else null end) as deptno_20, max(case when deptno=30 then empcount else null end) as deptno_30 from (select deptno, count(*) as empcount from emp group by deptno) x	
---	--

这种方法使用内嵌视图来生成每个部门的员工总数。主查询

中的 **CASE** 表达式将行转换为列，得到如下结果。

DEPTNO_10	DEPTNO_20	DEPTNO_30
-----	-----	-----
3	NULL	NULL
NULL	5	NULL
NULL	NULL	6

然后，**MAX** 函数将多行合并成一行。

DEPTNO_10	DEPTNO_20	DEPTNO_30
-----	-----	-----
3	5	6

12.2 将结果集转置为多行

1. 问题

通过为给定列中每个不同的值都创建一列，你想将行转换为列，但与上一节的实例不同，你想输出多行。与转置为一行一样，转置为多行也是一种基本的数据整形方法。

例如，你想返回每位员工及其职位（**JOB**），但当前使用的查询返回的结果集如下所示。

JOB	ENAME
-----	-----
ANALYST	SCOTT
ANALYST	FORD
CLERK	SMITH
CLERK	ADAMS
CLERK	MILLER
CLERK	JAMES
MANAGER	JONES
MANAGER	CLARK
MANAGER	BLAKE
PRESIDENT	KING
SALESMAN	ALLEN
SALESMAN	MARTIN
SALESMAN	TURNER
SALESMAN	WARD

你想调整结果集的格式，让每种职位各为一列。

CLERKS	ANALYSTS	MGRS	PREZ	SALES
-----	-----	-----	-----	-----
MILLER	FORD	CLARK	KING	TURNER
JAMES	SCOTT	BLAKE		MARTIN
ADAMS		JONES		WARD
SMITH				ALLEN

12. 解决方案

不同于上一节的实例，本实例的结果集包含多行。上一节介绍的方法对本实例不管用，因为按 **JOB** 分组并使用 **MAX(ENAME)** 时，将为每个 **JOB** 都返回一个 **ENAME**，即像上一节的实例一样，将只返回一行。要解决这个问题，必须让每个 **JOB/ENAME** 组合都是独一无二的。这样使用聚合函数来删除 **NULL** 时，就不会丢失任何 **ENAME** 了。

使用排名函数 **ROW_NUMBER OVER** 让每个 **JOB/ENAME** 组合都是独一无二的。使用 **CASE** 表达式和聚合函数 **MAX** 对结果集进行转置，并按窗口函数 **ROW_NUMBER OVER** 返回的值分组。

```
1  select max(case when job='CLERK'
2                then ename else null end) as clerks,
3         max(case when job='ANALYST'
4                then ename else null end) as analysts,
5         max(case when job='MANAGER'
6                then ename else null end) as mgrs,
7         max(case when job='PRESIDENT'
8                then ename else null end) as prez,
9         max(case when job='SALESMAN'
10               then ename else null end) as sales
11  from (
12 select job,
13        ename,
14        row_number()over(partition by job order by ename) rn
15  from emp
16  ) x
17  group by rn
```

13. 讨论

首先，借助窗口函数 **ROW_NUMBER OVER** 让每一个 **JOB/ENAME** 组合都是独一无二的。

```
select job,
       ename,
       row_number()over(partition by job order by ename) rn
from emp
```

JOB	ENAME	RN
ANALYST	FORD	1
ANALYST	SCOTT	2
CLERK	ADAMS	1
CLERK	JAMES	2
CLERK	MILLER	3
CLERK	SMITH	4
MANAGER	BLAKE	1
MANAGER	CLARK	2
MANAGER	JONES	3
PRESIDENT	KING	1
SALESMAN	ALLEN	1
SALESMAN	MARTIN	2
SALESMAN	TURNER	3
SALESMAN	WARD	4

通过给每一个 **ENAME** 指定一个在当前职位中独一无二的“行号”，可以防范因两位员工具有相同名字和职位而带来的问题。这里的目标是，确保在按行号（**RN**）分组的情况下，使用 **MAX** 不会导致结果集中的任何员工丢失。解决这个问题时，这是最重要的步骤。如果没有这一步，外部查询中的聚合就会删除原本要返回的行。请看使用上一节介绍的方法而不使用 **ROW_NUMBER OVER** 时，结果集是什么样的。

```
select max(case when job='CLERK'
               then ename else null end) as clerks,
       max(case when job='ANALYST'
               then ename else null end) as analysts,
       max(case when job='MANAGER'
               then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
```

<pre> then ename else null end) as prez, max(case when job='SALESMAN' then ename else null end) as sales from emp </pre>				
CLERKS	ANALYSTS	MGRS	PREZ	SALES
-----	-----	-----	-----	-----
SMITH	SCOTT	JONES	KING	WARD

对于每种职位，都只返回了一位员工：**MAX ENAME** 返回的值。转置结果集时，应该使用 **MIN** 或 **MAX** 来删除结果集中的 **NULL**，而不是使用它们限制返回的 **ENAME**。随着继续往下阅读，这里的工作原理将变得越来越清晰。

然后，使用 **CASE** 表达式将 **ENAME** 放到合适的职位列中。

<pre> select rn, case when job='CLERK' then ename else null end as clerks, case when job='ANALYST' then ename else null end as analysts, case when job='MANAGER' then ename else null end as mgrs, case when job='PRESIDENT' then ename else null end as prez, case when job='SALESMAN' then ename else null end as sales from (select job, ename, row_number()over(partition by job order by ename) rn from emp) x </pre>					
RN	CLERKS	ANALYSTS	MGRS	PREZ	SALES
--	-----	-----	-----	-----	-----
1		FORD			
2		SCOTT			
1	ADAMS				
2	JAMES				
3	MILLER				

4	SMITH			
1		BLAKE		
2		CLARK		
3		JONES		
1			KING	
1				ALLEN
2				MARTIN
3				TURNER
4				WARD

至此，行转置成了列。最后，删除 **NULL** 值，以提高结果集的可读性。为了删除 **NULL** 值，需要使用聚合函数 **MAX** 并按 **RN** 分组。（也可以使用 **MIN**。这里使用 **MAX** 并无特殊考虑，只要能够将每组聚合为一个值就行。）对于每个 **RN/JOB/ENAME** 组合，都只有一个值。通过按 **RN** 分组并在 **MAX** 调用中使用 **CASE** 表达式，能够确保每次调用 **MAX** 都只从分组中选择一个名字（因为除这个名字外，分组中的其他值都为 **NULL**）。

```
select max(case when job='CLERK'
                then ename else null end) as clerks,
       max(case when job='ANALYST'
                then ename else null end) as analysts,
       max(case when job='MANAGER'
                then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                then ename else null end) as prez,
       max(case when job='SALESMAN'
                then ename else null end) as sales
  from (
select job,
       ename,
       row_number()over(partition by job order by ename) rn
  from emp
  ) x
 group by rn
```

CLERKS	ANALYSTS	MGRS	PREZ	SALES
-----	-----	-----	----	-----
MILLER	FORD	CLARK	KING	TURNER
JAMES	SCOTT	BLAKE		MARTIN

ADAMS	JONES	WARD
SMITH		ALLEN

设置查询结果的格式时，使用 **ROW_NUMBER OVER** 来生成独一无二的组合很有用。请看下面的查询，它生成了一个按 **DEPTNO** 和 **JOB** 列出员工的稀疏报表。

```
select deptno dno, job,
       max(case when deptno=10
                 then ename else null end) as d10,
       max(case when deptno=20
                 then ename else null end) as d20,
       max(case when deptno=30
                 then ename else null end) as d30,
       max(case when job='CLERK'
                 then ename else null end) as clerks,
       max(case when job='ANALYST'
                 then ename else null end) as anals,
       max(case when job='MANAGER'
                 then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                 then ename else null end) as prez,
       max(case when job='SALESMAN'
                 then ename else null end) as sales
  from (
Select deptno,
       job,
       ename,
       row_number()over(partition by job order by ename) rn_job,
       row_number()over(partition by deptno order by ename)
rn_deptno
  from emp
) x
 group by deptno, job, rn_deptno, rn_job
 order by 1
```

DNO	JOB	D10	D20	D30	CLERKS	ANALS	MGRS	PREZ	SALES
10	CLERK	MILLER			MILLER				
10	MANAGER	CLARK					CLARK		
10	PRESIDENT	KING						KING	
20	ANALYST		FORD			FORD			

20 ANALYST	SCOTT		SCOTT	
20 CLERK	ADAMS	ADAMS		
20 CLERK	SMITH	SMITH		
20 MANAGER	JONES		JONES	
30 CLERK		JAMES	JAMES	
30 MANAGER		BLAKE		BLAKE
30 SALESMAN		ALLEN		ALLEN
30 SALESMAN		MARTIN		MARTIN
30 SALESMAN		TURNER		TURNER
30 SALESMAN		WARD		WARD

只需修改分组依据（进而修改 **SELECT** 列表中的非组合列），就可生成格式不同的报表。建议你花时间去做一些实验，尝试调整 **GROUP BY** 子句包含的列，看看这将如何影响报表的格式。

12.3 对结果集进行逆转置

1. 问题

你想将列转换为行。请看下面的结果集：

DEPTNO_10	DEPTNO_20	DEPTNO_30
3	5	6

你想将它转换成下面这样。

DEPTNO	COUNTS_BY_DEPT
10	3
20	5
30	6

你可能注意到了，上面的第一个结果集就是 12.1 节实例的输出。为了在本实例中使用这个结果集，可以使用如下查询将其存储在视图中。

```
create view emp_cnts as
(
select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
       sum(case when deptno=20 then 1 else 0 end) as deptno_20,
       sum(case when deptno=30 then 1 else 0 end) as deptno_30
  from emp
)
```

在接下来的解决方案和讨论中，将引用这个查询创建的视图 EMP_CNTS。

12. 解决方案

从要获得的结果集很容易看出，可以对 **EMP** 表使用 **COUNT** 和 **GROUP BY** 来生成它。但这里假设数据并不是以行的方式存储的：数据可能是非规范化的，而聚合值存储在多列中。

要将列转换为行，可以使用笛卡儿积。我们需要事先知道要转换为行的列数，因为用来生成笛卡儿积的表表达式的基数不能小于要转置的列数。

本解决方案不会创建非规范化数据表，而会使用 12.1 节实例中的解决方案来创建一个“宽”结果集。完整的解决方案如下所示。

```
1 select dept.deptno,  
2        case dept.deptno  
3            when 10 then emp_cnts.deptno_10  
4            when 20 then emp_cnts.deptno_20  
5            when 30 then emp_cnts.deptno_30  
6        end as counts_by_dept  
7  from emp_cnts cross join  
8        (select deptno from dept where deptno <= 30) dept
```

13. 讨论

内嵌视图 **EMP_CNTS** 是你转换为行的非规范化视图（或“宽”结果集）。

DEPTNO_10	DEPTNO_20	DEPTNO_30
-----	-----	-----
3	5	6

由于有 3 列，因此需要创建 3 行。首先，生成内嵌视图 **EMP_CNTS** 与至少有 3 行数据的表表达式的笛卡儿积。下面

的代码使用包含 4 行的 DEPT 表来生成笛卡儿积。

```
select dept.deptno,  
       emp_cnts.deptno_10,  
       emp_cnts.deptno_20,  
       emp_cnts.deptno_30  
  from (  
    Select sum(case when deptno=10 then 1 else 0 end) as deptno_10,  
           sum(case when deptno=20 then 1 else 0 end) as deptno_20,  
           sum(case when deptno=30 then 1 else 0 end) as deptno_30  
    from emp  
      ) emp_cnts,  
   (select deptno from dept where deptno <= 30) dept
```

DEPTNO	DEPTNO_10	DEPTNO_20	DEPTNO_30
10	3	5	6
20	3	5	6
30	3	5	6

笛卡儿积能够为内嵌视图 EMP_CNTS 中的每一列返回一行数据。由于最终的结果集应只包含 DEPTNO 和相应部门的员工数量，因此使用 CASE 表达式将每行 3 列变为每行 1 列。

```
select dept.deptno,  
       case dept.deptno  
         when 10 then emp_cnts.deptno_10  
         when 20 then emp_cnts.deptno_20  
         when 30 then emp_cnts.deptno_30  
       end as counts_by_dept  
  from (  
    emp_cnts  
 cross join (select deptno from dept where deptno <= 30) dept
```

DEPTNO	COUNTS_BY_DEPT
10	3
20	5
30	6

12.4 将结果集逆转置为一列

1. 问题

你想将查询返回的所有列值都放在一列中，并返回它们。例如，你想返回 10 号部门所有员工的 **ENAME**、**JOB** 和 **SAL**，并将这 3 个值放在一列中。为此，你要为每位员工返回 3 行，并在员工之间添加一个空白行。你想返回的结果集如下所示。

EMPS

CLARK
MANAGER
2450
KING
PRESIDENT
5000
MILLER
CLERK
1300

2. 解决方案

关键是结合使用递归 CTE 和笛卡儿积来为每位员工返回 4 行数据。第 10 章介绍过递归 CTE，附录 B 也会对其进行更深入的讨论。使用笛卡儿连接，可以在每一行中返回一个列值，并在员工之间添加空白行。

使用窗口函数 **ROW_NUMBER OVER** 根据 **EMPNO** 给行排名（1~4），然后使用 **CASE** 表达式将 3 列转换为 1 列（在

PostgreSQL 和 MySQL 中，必须在 WITH 后面添加关键字 RECURSIVE）。

```
1  with four_rows (id)
2    as
3  (
4    select 1
5      union all
6    select id+1
7      from four_rows
8     where id < 4
9  )
10 ,
11  x_tab (ename,job,sal,rn )
12    as
13  (
14    select e.ename,e.job,e.sal,
15           row_number()over(partition by e.empno
16                             order by e.empno)
17    from emp e
18    join four_rows on 1=1
19  )
20  select
21    case rn
22      when 1 then ename
23      when 2 then job
24      when 3 then cast(sal as char(4))
25    end emps
26  from x_tab
```

13. 讨论

使用窗口函数 ROW_NUMBER OVER 给 10 号部门的每位员工添加一个排名值。

```
select e.ename,e.job,e.sal,
       row_number()over(partition by e.empno
```

```

                                order by e.empno) rn
from emp e
  where e.deptno=10

```

ENAME	JOB	SAL	RN
CLARK	MANAGER	2450	1
KING	PRESIDENT	5000	1
MILLER	CLERK	1300	1

当前，这个排名值的意义不大。由于你是根据 **EMPNO** 进行的分区，因此 10 号部门全部 3 位员工的排名都是 1。生成笛卡儿积后，排名将初具雏形，如下面的结果所示。

```

with four_rows (id)
  as
  (select 1
   from dual
   union all
   select id+1
   from four_rows
   where id < 4
  )
select e.ename,e.job,e.sal,
row_number()over(partition by e.empno
order by e.empno)
  from emp e
 join four_rows on 1=1

```

ENAME	JOB	SAL	RN
CLARK	MANAGER	2450	1
CLARK	MANAGER	2450	2
CLARK	MANAGER	2450	3
CLARK	MANAGER	2450	4
KING	PRESIDENT	5000	1
KING	PRESIDENT	5000	2
KING	PRESIDENT	5000	3
KING	PRESIDENT	5000	4
MILLER	CLERK	1300	1
MILLER	CLERK	1300	2
MILLER	CLERK	1300	3

此时你应该停下来想一想，弄明白下面两个重点。

- 现在每位员工有 4 行数据，这些行的 RN 为 1~4。这是因为窗口函数是在 **FROM** 和 **WHERE** 子句之后执行的。因此，按 **EMPNO** 分区导致遇到新员工时 RN 将被重置为 1。
- 我们使用了递归 CTE 来确保每位员工有 4 行数据。在 SQL Server 和 DB2 中，不需要使用关键字 **RECURSIVE**，在 Oracle、MySQL 和 PostgreSQL 中则需要。

至此，艰难的工作已经完成，余下的任务是使用 **CASE** 表达式将每位员工的 **ENAME**、**JOB** 和 **SAL** 都放在一列中（需要使用 **CAST** 将 **SAL** 转换为字符串，以免 **CASE** 出问题）。

```
with four_rows (id)
as
(select 1
union all
select id+1
from four_rows
where id < 4
)
,
x_tab (ename,job,sal,rn )
as
(select e.ename,e.job,e.sal,
row_number()over(partition by e.empno
order by e.empno)
from emp e
join four_rows on 1=1)

select case rn
when 1 then ename
when 2 then job
when 3 then cast(sal as char(4))
end emps
from x_tab
EMPS
```

CLARK
MANAGER
2450

KING
PRESIDENT
5000

MILLER
CLERK
1300

12.5 消除结果集中的重复值

1. 问题

你正在制作一张报表，当多行的同一列的值相同时，你希望这个列值只显示一次。例如，你想从 **EMP** 表返回 **DEPTNO** 和 **ENAME** 并按 **DEPTNO** 分组，但对于每个 **DEPTNO**，只想显示一次。你想返回如下结果集。

DEPTNO	ENAME
10	CLARK
	KING
	MILLER
20	SMITH
	ADAMS
	FORD
	SCOTT
	JONES
30	ALLEN
	BLAKE
	MARTIN
	JAMES
	TURNER
	WARD

2. 解决方案

这是一个简单的格式设置问题，使用窗口函数 **LAG OVER** 可以轻而易举地解决。

```
1  select
2      case when
3          lag(deptno)over(order by deptno) = deptno then null
4          else deptno end DEPTNO
```



```
5      , ename
6  from emp
```

Oracle 用户也可以用 **DECODE** 替代 **CASE**。

```
1 select to_number(
2         decode(lag(deptno)over(order by deptno),
3               deptno,null,deptno)
4       ) deptno, ename
5  from emp
```

13. 讨论

使用窗口函数 **LAG OVER** 来返回前一行的 **DEPTNO**。

```
select lag(deptno)over(order by deptno) lag_deptno,
       deptno,
       ename
  from emp
```

LAG_DEPTNO	DEPTNO	ENAME
	10	CLARK
10	10	KING
10	10	MILLER
10	20	SMITH
20	20	ADAMS
20	20	FORD
20	20	SCOTT
20	20	JONES
20	30	ALLEN
30	30	BLAKE
30	30	MARTIN
30	30	JAMES
30	30	TURNER
30	30	WARD

如果查看上面的结果集，你可以很容易发现哪些行的

DEPTNO 与 LAG_DEPTNO 相等。对于这些行，你要将其 DEPTNO 设置为 NULL，为此可以使用 DECODE（使用 TO_NUMBER 将 DEPTNO 转换成了数字）。

```
select to_number(
      CASE WHEN (lag(deptno)over(order by deptno)
= deptno THEN null else deptno END deptno ,
      deptno,null,deptno)
      ) deptno, ename
from emp
```

DEPTNO	ENAME
-----	-----
10	CLARK
	KING
	MILLER
20	SMITH
	ADAMS
	FORD
	SCOTT
	JONES
30	ALLEN
	BLAKE
	MARTIN
	JAMES
	TURNER
	WARD

12.6 转置结果集以简化涉及多行的计算

1. 问题

你要执行的计算涉及来自多行的数据，为简化工作，你想将这些行转置为列，这样你需要的所有数据都会出现在同一行中。

在本书的示例数据中，薪水总额最高的部门是 20 号部门。要确认这一点，可以执行下面的查询。

```
select deptno, sum(sal) as sal
  from emp
 group by deptno
```

DEPTNO	SAL
10	8750
20	10875
30	9400

你想计算 10 号部门的薪水总额和 30 号部门的薪水总额分别比 20 号部门的薪水总额少多少，最终结果如下所示。

d20_10_diff	d20_30_diff
2125	1475

2. 解决方案

使用聚合函数 **SUM** 和 **CASE** 表达式对部门薪水总额进行转置，然后在 **SELECT** 列表中编写计算薪水总额之差的表达式。

```

1 select d20_sal - d10_sal as d20_10_diff,
2        d20_sal - d30_sal as d20_30_diff
3   from (
4 select sum(case when deptno=10 then sal end) as d10_sal,
5        sum(case when deptno=20 then sal end) as d20_sal,
6        sum(case when deptno=30 then sal end) as d30_sal
7   from emp
8        ) totals_by_dept

```

也可以使用 CTE 来编写这个查询，有些人可能觉得这样做可读性更高。

```

with totals_by_dept (d10_sal, d20_sal, d30_sal)
as
(select
        sum(case when deptno=10 then sal end) as d10_sal,
        sum(case when deptno=20 then sal end) as d20_sal,
        sum(case when deptno=30 then sal end) as d30_sal
 from emp)

select  d20_sal - d10_sal as d20_10_diff,
        d20_sal - d30_sal as d20_30_diff
  from totals_by_dept

```

13. 讨论

首先，使用 **CASE** 表达式将各部门的薪水行转置为列。

```

select case when deptno=10 then sal end as d10_sal,
       case when deptno=20 then sal end as d20_sal,
       case when deptno=30 then sal end as d30_sal
  from emp

```

D10_SAL	D20_SA	L D30_SAL
-----	-----	-----
	800	
		1600

		1250
	2975	
		1250
		2850
2450		
	3000	
5000		
		1500
	1100	
		950
	3000	
1300		

然后，对每个 **CASE** 表达式应用聚合函数 **SUM**，计算出每个部门的薪水总额。

<pre>select sum(case when deptno=10 then sal end) as d10_sal, sum(case when deptno=20 then sal end) as d20_sal, sum(case when deptno=30 then sal end) as d30_sal from emp</pre>		
D10_SAL	D20_SAL	D30_SAL
-----	-----	-----
8750	10875	9400

最后，将上述 SQL 放在一个内嵌视图中，并执行减法运算。

12.7 创建尺寸固定的数据桶

1. 问题

你想将数据划分到大小相同的桶中，其中每个桶包含的元素个数是事先定好的。总桶数可能未知，但你要确保每个桶中都包含 5 个元素。例如，你想基于 **EMPNO** 值将 **EMP** 表中的员工分组，每组包含 5 位员工，如下面的结果所示。

GRP	EMPNO	ENAME
1	7369	SMITH
1	7499	ALLEN
1	7521	WARD
1	7566	JONES
1	7654	MARTIN
2	7698	BLAKE
2	7782	CLARK
2	7788	SCOTT
2	7839	KING
2	7844	TURNER
3	7876	ADAMS
3	7900	JAMES
3	7902	FORD
3	7934	MILLER

2. 解决方案

使用排名函数可以极大地简化这个问题的解决方案。将行排名后，要创建大小为 5 的桶，只需执行除法运算并将商向上取整。

使用窗口函数根据 **EMPNO** 对员工进行排名，然后除以 5 以创建分组。（如果你使用的是 SQL Server，请将 **CEIL** 替换为

CEILING。)

```
1 select ceil(row_number()over(order by empno)/5.0) grp,  
2      empno,  
3      ename  
4  from emp
```

13. 讨论

窗口函数 **ROW_NUMBER OVER** 根据 **EMPNO** 进行排序，并会给每一行指定一个排名（或“行号”）。

```
select row_number()over(order by empno) rn,  
      empno,  
      ename  
  from emp
```

RN	EMPNO	ENAME
1	7369	SMITH
2	7499	ALLEN
3	7521	WARD
4	7566	JONES
5	7654	MARTIN
6	7698	BLAKE
7	7782	CLARK
8	7788	SCOTT
9	7839	KING
10	7844	TURNER
11	7876	ADAMS
12	7900	JAMES
13	7902	FORD
14	7934	MILLER

接下来，将函数 **ROW_NUMBER OVER** 返回的值除以 5，再对结果应用函数 **CEIL (CEILING)**。从逻辑上说，除以 5 相当于将行分成规模为 5 的编组，即 5 个值小于或等于 1，5 个

值大于 1 但小于或等于 2。最后一组（包含最后 4 行，因为 EMP 表包含的行数为 14，不是 5 的整数倍）的值大于 2 但小于或等于 3。

函数 **CEIL** 会返回比传递给它的值大的最小整数，确保创建的分组数为整数。下面显示了对除法运算的商应用函数 **CEIL**。（运算是从左到右进行的，从 **RN** 计算出 **DIVISION**，再计算出 **GRP**。）

```
select row_number()over(order by empno) rn,
       row_number()over(order by empno)/5.0 division,
       ceil(row_number()over(order by empno)/5.0) grp,
       empno,
       ename
from emp
```

RN	DIVISION	GRP	EMPNO	ENAME
1	.2	1	7369	SMITH
2	.4	1	7499	ALLEN
3	.6	1	7521	WARD
4	.8	1	7566	JONES
5	1	1	7654	MARTIN
6	1.2	2	7698	BLAKE
7	1.4	2	7782	CLARK
8	1.6	2	7788	SCOTT
9	1.8	2	7839	KING
10	2	2	7844	TURNER
11	2.2	3	7876	ADAMS
12	2.4	3	7900	JAMES
13	2.6	3	7902	FORD
14	2.8	3	7934	MILLER

12.8 创建预定数量的桶

1. 问题

你想将数据划分到数量固定的几个桶中。例如，你想将 **EMP** 表中的员工划分到 4 个桶中，结果如下所示。

GRP	EMPNO	ENAME
---	---	---
1	7369	SMITH
1	7499	ALLEN
1	7521	WARD
1	7566	JONES
2	7654	MARTIN
2	7698	BLAKE
2	7782	CLARK
2	7788	SCOTT
3	7839	KING
3	7844	TURNER
3	7876	ADAMS
4	7900	JAMES
4	7902	FORD
4	7934	MILLER

这是一种组织分类数据的常见方式，因为在很多分析中，将一个集合分成多个规模相同的集合是重要的第一步。例如，通过计算这些分组的平均薪水或其他平均值，或许能够揭示查看各个值时被波动掩盖的趋势。

这个问题与 12.7 节的问题相反。在 12.7 节中，桶的数量是未知的，但每个桶包含的元素个数是事先定好的，而在这个问题中，可能不知道每个桶中有多少个元素，但要创建的桶的个数是固定（已知）的。

12. 解决方案

由于现在很多 RDBMS 支持函数 **NTILE**，因此这个问题的解决方案很简单。**NTILE** 会将一个有序集合划分到指定数量的桶中。如果集合元素无法均分，就将多出来的元素放到前面的桶中，本实例要返回的结果集反映了这一点：第 1 个桶和第 2 个桶分别有 4 行数据，而第 3 个桶和第 4 个桶分别只有 3 行数据。

使用窗口函数 **NTILE** 创建 4 个桶。

```
1 select ntile(4)over(order by empno) grp,  
2        empno,  
3        ename  
4  from emp
```

13. 讨论

所有的工作都是由函数 **NTILE** 完成的。**ORDER BY** 子句会按照指定的顺序排列行，而函数 **NTILE** 本身会给每一行指定一个组号，将前 1/4 的行加入第 1 组，将接下来 1/4 的行加入第二组，以此类推。

12.9 创建水平直方图

1. 问题

你想使用 SQL 来创建沿水平方向延伸的直方图。例如，你想以水平直方图的方式显示每个部门的员工数量，在直方图中每个星号（*）表示一位员工。你想返回的结果集如下所示。

DEPTNO	CNT
10	***
20	*****
30	*****

2. 解决方案

本解决方案的关键是，使用聚合函数 **COUNT** 和 **GROUP BY DEPTNO** 来计算每个部门的员工数量。然后，将 **COUNT** 返回的值传递给一个字符串函数，以生成相应数量的 * 字符。

DB2

使用函数 **REPEAT** 生成直方图。

```
1 select deptno,  
2        repeat('*',count(*)) cnt  
3   from emp  
4  group by deptno
```

Oracle、PostgreSQL 和 MySQL

使用函数 **LPAD** 生成相应数量的 * 字符。

```
1 select deptno,  
2         lpad('*',count(*),'*') as cnt  
3   from emp  
4  group by deptno
```

SQL Server

使用函数 **REPLICATE** 生成直方图。

```
1 select deptno,  
2         replicate('*',count(*)) cnt  
3   from emp  
4  group by deptno
```

13. 讨论

所有解决方案采用的方法都相同，差别在于用来为每位员工返回一个 * 的字符串函数。这里的讨论以 Oracle 解决方案为例，但适用于所有解决方案。

首先，计算每个部门的员工数量。

```
select deptno,  
       count(*)  
  from emp  
 group by deptno
```

DEPTNO	COUNT(*)
10	3
20	5
30	6

接下来，使用 **COUNT** 返回的值来控制为每个部门返回的 * 字符个数。要返回所需数量的 * 字符，只需将 **COUNT(*)** 作为参数传递给字符串函数 **LPAD**。

```
select deptno,  
       lpad('*',count(*),'*') as cnt  
from emp  
group by deptno
```

DEPTNO	CNT
10	***
20	*****
30	*****

如果你使用的是 PostgreSQL，那么可能需要像下面这样使用强制类型转换确保 **COUNT(*)** 返回一个整数。

```
select deptno,  
       lpad('*',count(*)::integer,'*') as cnt  
from emp  
group by deptno
```

DEPTNO	CNT
10	***
20	*****
30	*****

这里的强制类型转换必不可少，因为 PostgreSQL 要求传递给函数 **LPAD** 的数字参数为整数。

12.10 创建垂直直方图

1. 问题

你想生成一个从下往上延伸的直方图。例如，你想以垂直直方图的方式显示每个部门的员工数量，每位员工用一个 * 字符表示。你想返回的结果集如下所示。

```
D10 D20 D30
--- --- ---
          *
        * *
       * *
      * *
 *    * *
 *    * *
 *    * *
```

2. 解决方案

这个问题的解决方案以本章前面使用的一种方案为基础：使用函数 **ROW_NUMBER OVER** 来唯一地表示每个部门的每位员工。使用聚合函数 **MAX** 来转置结果集，并根据函数 **ROW_NUMBER OVER** 返回的值进行分组。（如果你使用的是 SQL Server，请不要在 **ORDER BY** 子句中使用 **DESC**。）

```
1 select max(deptno_10) d10,
2         max(deptno_20) d20,
3         max(deptno_30) d30
4   from (
5 select row_number()over(partition by deptno order by empno) rn,
6        case when deptno=10 then '*' else null end deptno_10,
7        case when deptno=20 then '*' else null end deptno_20,
8        case when deptno=30 then '*' else null end deptno_30
9   from emp
```

```

10      ) x
11  group by rn
12  order by 1 desc, 2 desc, 3 desc

```

13. 讨论

第一步是使用窗口函数 **ROW_NUMBER** 来唯一地表示每个部门的每位员工。对于每个部门的每位员工，使用 **CASE** 表达式返回一个 *****。

```

select row_number()over(partition by deptno order by empno) rn,
       case when deptno=10 then '*' else null end deptno_10,
       case when deptno=20 then '*' else null end deptno_20,
       case when deptno=30 then '*' else null end deptno_30
from emp

```

RN	DEPTNO_10	DEPTNO_20	DEPTNO_30
1	*		
2	*		
3	*		
1		*	
2		*	
3		*	
4		*	
5		*	
1			*
2			*
3			*
4			*
5			*
6			*

下一步也是最后一步，将聚合函数 **MAX** 应用于每个 **CASE** 表达式并按 **RN** 分组，将结果集中的 **NULL** 删除。根据 **RDBMS** 对 **NULL** 的排序方式，将结果集升序或降序排列。

```

select max(deptno_10) d10,
       max(deptno_20) d20,
       max(deptno_30) d30
  from (
select row_number()over(partition by deptno order by empno) rn,
       case when deptno=10 then '*' else null end deptno_10,
       case when deptno=20 then '*' else null end deptno_20,
       case when deptno=30 then '*' else null end deptno_30
  from emp
    ) x
 group by rn
 order by 1 desc, 2 desc, 3 desc

```

```

D10 D20 D30

```

```

--- --- ---
      *
    *  *
    *  *
*   *  *
*   *  *
*   *  *

```


12.11 返回未被用作分组依据的列

1. 问题

你正在编写一个 **GROUP BY** 查询，并要返回未包含在 **GROUP BY** 子句中的列。这通常是不可能的，因为未被用作分组依据的列在各行中不是唯一的。

假设你想找出每个部门中薪水最高和薪水最低的员工，以及每个职位中薪水最高和薪水最低的员工。你要显示每位员工的名字、所属部门、职位和薪水。你想返回的结果集如下所示。

DEPTNO	ENAME	JOB	SAL	DEPT_STATUS			JOB_STATUS		
10	MILLER	CLERK	1300	LOW	SAL	IN DEPT	TOP	SAL	IN JOB
10	CLARK	MANAGER	2450				LOW	SAL	IN JOB
10	KING	PRESIDENT	5000	TOP	SAL	IN DEPT	TOP	SAL	IN JOB
20	SCOTT	ANALYST	3000	TOP	SAL	IN DEPT	TOP	SAL	IN JOB
20	FORD	ANALYST	3000	TOP	SAL	IN DEPT	TOP	SAL	IN JOB
20	SMITH	CLERK	800	LOW	SAL	IN DEPT	LOW	SAL	IN JOB
20	JONES	MANAGER	2975				TOP	SAL	IN JOB
30	JAMES	CLERK	950	LOW	SAL	IN DEPT			
30	MARTIN	SALESMAN	1250				LOW	SAL	IN JOB
30	WARD	SALESMAN	1250				LOW	SAL	IN JOB
30	ALLEN	SALESMAN	1600				TOP	SAL	IN JOB
30	BLAKE	MANAGER	2850	TOP	SAL	IN DEPT			

可惜在 **SELECT** 子句中包含所有这些列将影响分组。来看一个例子。员工 **KING** 的薪水最高，你想使用下面的查询来验证这一点。

```
select ename,max(sal)
  from empgroup by ename
```

这个查询将返回 **EMP** 表中的全部 14 行数据，而不是员工

KING 及其薪水。问题出在分组上：**MAX(SAL)** 被应用于每个 **ENAME**。上述查询的意思好像是找出薪水最高的员工，但它实际上做的是找出 **EMP** 表中每位员工的最高薪水。本实例将介绍如何在 **GROUP BY** 子句中不包含 **ENAME** 列的情况下返回它。

12. 解决方案

使用内嵌视图找出相应部门和职位的最高薪水和最低薪水，然后只保留获得这些薪水的员工。

使用窗口函数 **MAX OVER** 和 **MIN OVER** 找出相应部门和职位的最高薪水和最低薪水，然后保留薪水为相应部门或职位的最高薪水或最低薪水的行。

```
1 select deptno,ename,job,sal,
2         case when sal = max_by_dept
3             then 'TOP SAL IN DEPT'
4             when sal = min_by_dept
5             then 'LOW SAL IN DEPT'
6         end dept_status,
7         case when sal = max_by_job
8             then 'TOP SAL IN JOB'
9             when sal = min_by_job
10            then 'LOW SAL IN JOB'
11        end job_status
12    from (
13 select deptno,ename,job,sal,
14         max(sal)over(partition by deptno) max_by_dept,
15         max(sal)over(partition by job) max_by_job,
16         min(sal)over(partition by deptno) min_by_dept,
17         min(sal)over(partition by job) min_by_job
18    from emp
19       ) emp_sals
20   where sal in (max_by_dept,max_by_job,
21                min_by_dept,min_by_job)
```

13. 讨论

首先，使用窗口函数 **MAX OVER** 和 **MIN OVER** 找出相应部门和职位的最高薪水和最低薪水。

<pre>select deptno,ename,job,sal, max(sal)over(partition by deptno) maxDEPT, max(sal)over(partition by job) maxJOB, min(sal)over(partition by deptno) minDEPT, min(sal)over(partition by job) minJOB from emp</pre>							
DEPTNO	ENAME	JOB	SAL	MAXDEPT	MAXJOB	MINDEPT	MINJOB
10	MILLER	CLERK	1300	5000	1300	1300	800
10	CLARK	MANAGER	2450	5000	2975	1300	2450
10	KING	PRESIDENT	5000	5000	5000	1300	5000
20	SCOTT	ANALYST	3000	3000	3000	800	3000
20	FORD	ANALYST	3000	3000	3000	800	3000
20	SMITH	CLERK	800	3000	1300	800	800
20	JONES	MANAGER	2975	3000	2975	800	2450
20	ADAMS	CLERK	1100	3000	1300	800	800
30	JAMES	CLERK	950	2850	1300	950	800
30	MARTIN	SALESMAN	1250	2850	1600	950	1250
30	TURNER	SALESMAN	1500	2850	1600	950	1250
30	WARD	SALESMAN	1250	2850	1600	950	1250
30	ALLEN	SALESMAN	1600	2850	1600	950	1250
30	BLAKE	MANAGER	2850	2850	2975	950	2450

现在，可以将每位员工的薪水同相应部门和职位的最高薪水和最低薪水进行比较了。注意，**SELECT** 子句中包含多列，但这不会影响 **MIN OVER** 和 **MAX OVER** 的返回值。这就是窗口函数的优点：根据指定的分组或分区计算聚合值，并为每个分组返回多行数据。最后，将窗口函数放到一个内嵌视图中，并只保留那些与窗口函数返回的值匹配的行。为在最终的结果集中显示每位员工的“状态”，这里使用了简单的 **CASE** 表达式。

```

select deptno,ename,job,sal,
       case when sal = max_by_dept
            then 'TOP SAL IN DEPT'
            when sal = min_by_dept
            then 'LOW SAL IN DEPT'
       end dept_status,
       case when sal = max_by_job
            then 'TOP SAL IN JOB'
            when sal = min_by_job
            then 'LOW SAL IN JOB'
       end job_status
  from (
select deptno,ename,job,sal,
       max(sal)over(partition by deptno) max_by_dept,
       max(sal)over(partition by job) max_by_job,
       min(sal)over(partition by deptno) min_by_dept,
       min(sal)over(partition by job) min_by_job
  from emp
 ) x
 where sal in (max_by_dept,max_by_job,
              min_by_dept,min_by_job)

```

DEPTNO	ENAME	JOB	SAL	DEPT_STATUS	JOB_STATUS
-----	-----	-----	-----	-----	-----
10	MILLER	CLERK	1300	LOW SAL IN DEPT	TOP SAL IN JOB
10	CLARK	MANAGER	2450		LOW SAL IN JOB
10	KING	PRESIDENT	5000	TOP SAL IN DEPT	TOP SAL IN JOB
20	SCOTT	ANALYST	3000	TOP SAL IN DEPT	TOP SAL IN JOB
20	FORD	ANALYST	3000	TOP SAL IN DEPT	TOP SAL IN JOB
20	SMITH	CLERK	800	LOW SAL IN DEPT	LOW SAL IN JOB
20	JONES	MANAGER	2975		TOP SAL IN JOB
30	JAMES	CLERK	950	LOW SAL IN DEPT	
30	MARTIN	SALESMAN	1250		LOW SAL IN JOB
30	WARD	SALESMAN	1250		LOW SAL IN JOB
30	ALLEN	SALESMAN	1600		TOP SAL IN JOB
30	BLAKE	MANAGER	2850	TOP SAL IN DEPT	

12.12 计算简单的小计

1. 问题

在本实例中，你想创建一个结果集，其中包含小计（聚合分组的特定列）和总计（聚合整张表的特定列）。一个这样的结果集既包含 **EMP** 表中每种职位的薪水总额，也包含 **EMP** 表中所有薪水的总额，其中每种职位的薪水总额为小计，所有薪水的总额为总计。这种结果集如下所示。

JOB	SAL
-----	-----
ANALYST	6000
CLERK	4150
MANAGER	8275
PRESIDENT	5000
SALESMAN	5600
TOTAL	29025

2. 解决方案

GROUP BY 子句的 **ROLLUP** 扩展完美地解决了这个问题。如果你使用的 **RDBMS** 不支持 **ROLLUP**，则可以使用标量子查询或 **UNION** 查询来解决这个问题，只是难度更大。

DB2 和 Oracle

使用聚合函数 **SUM** 计算薪水总额，并使用 **GROUP BY** 的 **ROLLUP** 扩展将结果组织为小计（针对不同职位）和总计（针对整张表）。

```
1 select case grouping(job)
```

```
2         when 0 then job
3         else 'TOTAL'
4     end job,
5     sum(sal) sal
6 from emp
7 group by rollup(job)
```

SQL Server 和 MySQL

首先，使用聚合函数 **SUM** 计算薪水总额，并使用 **WITH ROLLUP** 将结果组织为小计（针对不同职位）和总计（针对整张表）。然后，使用 **COALESCE** 给总计行提供标签 **TOTAL**。（如果不这样做，该行的 **JOB** 列将为 **NULL**。）

```
1 select coalesce(job,'TOTAL') job,
2        sum(sal) sal
3 from emp
4 group by job with rollup
```

在 SQL Server 中，也可以像 Oracle/DB2 解决方案那样使用函数 **GROUPING**（而不是函数 **COALESCE**）来确定聚合等级。

PostgreSQL

与 SQL Server 和 MySQL 解决方案类似，也可以使用 **GROUP BY** 的 **ROLLUP** 扩展，但语法稍有不同。

```
select coalesce(job,'TOTAL') job,
       sum(sal) sal
  from emp
 group by rollup(job)
```

13. 讨论

DB2 和 Oracle

首先，使用聚合函数 **SUM** 并按 **JOB** 分组，以计算不同职位的薪水总额。

```
select job, sum(sal) sal
  from emp
 group by job
```

JOB	SAL
ANALYST	6000
CLERK	4150
MANAGER	8275
PRESIDENT	5000
SALESMAN	5600

然后，使用 **GROUP BY** 的 **ROLLUP** 扩展，以便在生成不同职位小计的同时生成总计。

```
select job, sum(sal) sal
  from emp
 group by rollup(job)
```

JOB	SAL
ANALYST	6000
CLERK	4150
MANAGER	8275
PRESIDENT	5000
SALESMAN	5600
	29025

最后，使用函数 **GROUPING** 在总计行的 **JOB** 列显示标签。如果 **JOB** 值为 **NULL**，那么函数 **GROUPING** 将返回 1，这表明 **SAL** 值为 **ROLLUP** 生成的总计。如果 **JOB** 值不为 **NULL**，则函数 **GROUPING** 将返回 0，这表明 **SAL** 值是 **GROUP BY**（而不是 **ROLLUP**）生成的。通过在 **CASE** 表达式中调用

GROUPING(JOB)，可以根据情况返回职位名称或标签 **TOTAL**。

```
select case grouping(job)
        when 0 then job
        else 'TOTAL'
      end job,
      sum(sal) sal
  from emp
 group by rollup(job)
```

JOB	SAL
-----	-----
ANALYST	6000
CLERK	4150
MANAGER	8275
PRESIDENT	5000
SALESMAN	5600
TOTAL	29025

SQL Server 和 MySQL

首先，使用聚合函数 **SUM** 并按 **JOB** 将结果分组，以生成不同职位的薪水小计。

```
select job, sum(sal) sal
  from emp
 group by job
```

JOB	SAL
-----	-----
ANALYST	6000
CLERK	4150
MANAGER	8275
PRESIDENT	5000
SALESMAN	5600

然后，使用 **GROUP BY** 的 **ROLLUP** 扩展，以便在生成不同职

位小计的同时生成总计。

```
select job, sum(sal) sal
  from emp
 group by job with rollup
```

JOB	SAL
-----	-----
ANALYST	6000
CLERK	4150
MANAGER	8275
PRESIDENT	5000
SALESMAN	5600
	29025

最后，使用函数 **COALESCE** 来处理 **JOB** 列。如果 **JOB** 值为 **NULL**，就说明 **SAL** 值是 **ROLLUP** 生成的总计。如果 **JOB** 值不为 **NULL**，则说明 **SAL** 值是 **GROUP BY**（而不是 **ROLLUP**）生成的。

```
select coalesce(job,'TOTAL') job,
       sum(sal) sal
  from emp
 group by job with rollup
```

JOB	SAL
-----	-----
ANALYST	6000
CLERK	4150
MANAGER	8275
PRESIDENT	5000
SALESMAN	5600
TOTAL	29025

PostgreSQL

工作原理与 MySQL 和 SQL Server 解决方案相同，唯一不同的是 **ROLLUP** 子句的语法：在 **GROUP BY** 后面加上 **ROLLUP(JOB)**。

12.13 计算各种可能的小计

1. 问题

你想找出不同部门、职位、职位 / 部门组合的薪水小计，同时显示整个 **EMP** 表的薪水总计。换言之，你想返回如下结果集。

DEPTNO	JOB	CATEGORY	SAL

10	CLERK	TOTAL BY DEPT AND JOB	1300
10	MANAGER	TOTAL BY DEPT AND JOB	2450
10	PRESIDENT	TOTAL BY DEPT AND JOB	5000
20	CLERK	TOTAL BY DEPT AND JOB	1900
30	CLERK	TOTAL BY DEPT AND JOB	950
30	SALESMAN	TOTAL BY DEPT AND JOB	5600
30	MANAGER	TOTAL BY DEPT AND JOB	2850
20	MANAGER	TOTAL BY DEPT AND JOB	2975
20	ANALYST	TOTAL BY DEPT AND JOB	6000
	CLERK	TOTAL BY JOB	4150
	ANALYST	TOTAL BY JOB	6000
	MANAGER	TOTAL BY JOB	8275
	PRESIDENT	TOTAL BY JOB	5000
	SALESMAN	TOTAL BY JOB	5600
10		TOTAL BY DEPT	8750
30		TOTAL BY DEPT	9400
20		TOTAL BY DEPT	10875
		GRAND TOTAL FOR TABLE	29025

2. 解决方案

最近几年，对于 **GROUP BY** 语法的扩展使得以上问题解决起来易如反掌。如果你使用的 **RDBMS** 没有提供计算各种小计的扩展，就必须手动计算它们，为此可以使用自连接或标量子查询。

DB2

在 DB2 中，需要使用 **CAST** 将 **GROUPING** 返回的结果转换为数据类型 **CHAR(1)**。

```
1 select deptno,
2         job,
3         case cast(grouping(deptno) as char(1))||
4              cast(grouping(job) as char(1))
5             when '00' then 'TOTAL BY DEPT AND JOB'
6             when '10' then 'TOTAL BY JOB'
7             when '01' then 'TOTAL BY DEPT'
8             when '11' then 'TOTAL FOR TABLE'
9         end category,
10        sum(sal)
11  from emp
12 group by cube(deptno,job)
13 order by grouping(job),grouping(deptno)
```

Oracle

结合使用 **GROUP BY** 子句的 **CUBE** 扩展和拼接运算符 **||**。

```
1 select deptno,
2         job,
3         case grouping(deptno)||grouping(job)
4             when '00' then 'TOTAL BY DEPT AND JOB'
5             when '10' then 'TOTAL BY JOB'
6             when '01' then 'TOTAL BY DEPT'
7             when '11' then 'GRAND TOTALFOR TABLE'
8         end category,
9         sum(sal) sal
10  from emp
11 group by cube(deptno,job)
12 order by grouping(job),grouping(deptno)
```

SQL Server

使用 **GROUP BY** 子句的 **CUBE** 扩展。在 SQL Server 中，需要

使用 **CAST** 将 **GROUPING** 返回的结果转换为数据类型 **CHAR(1)**，并使用拼接运算符 **+**（而不像在 Oracle 中那样使用运算符 **||**）。

```
1 select deptno,
2         job,
3         case cast(grouping(deptno)as char(1))+
4               cast(grouping(job)as char(1))
5             when '00' then 'TOTAL BY DEPT AND JOB'
6             when '10' then 'TOTAL BY JOB'
7             when '01' then 'TOTAL BY DEPT'
8             when '11' then 'GRAND TOTAL FOR TABLE'
9         end category,
10        sum(sal) sal
11  from emp
12 group by deptno,job with cube
13 order by grouping(job),grouping(deptno)
```

PostgreSQL

PostgreSQL 解决方案与 SQL Server 解决方案类似，但 **CUBE** 扩展和拼接运算符的语法稍有不同。

```
select deptno,job
,case concat(
cast (grouping(deptno) as char(1)),cast (grouping(job) as char(1))
)
when '00' then 'TOTAL BY DEPT AND JOB'
      when '10' then 'TOTAL BY JOB'
      when '01' then 'TOTAL BY DEPT'
      when '11' then 'GRAND TOTAL FOR TABLE'
end category
, sum(sal) as sal
from emp
group by cube(deptno,job)
```

MySQL

MySQL 只提供了前述解决方案使用的部分功能，准确地说

是没有提供函数 **CUBE**。因此，在 **MySQL** 中，需要使用多个 **UNION ALL** 来生成各种可能的小计。

```
1 select deptno, job,
2         'TOTAL BY DEPT AND JOB' as category,
3         sum(sal) as sal
4   from emp
5  group by deptno, job
6 union all
7 select null, job, 'TOTAL BY JOB', sum(sal)
8   from emp
9  group by job
10 union all
11 select deptno, null, 'TOTAL BY DEPT', sum(sal)
12   from emp
13  group by deptno
14 union all
15 select null,null,'GRAND TOTAL FOR TABLE', sum(sal)
16   from emp
```

13. 讨论

Oracle、DB2 和 SQL Server

在这 3 种 **RDBMS** 中，解决方案几乎是相同的。首先，使用聚合函数 **SUM** 并按 **DEPTNO** 和 **JOB** 分组，以找出各个 **JOB** 和 **DEPTNO** 组合的薪水小计。

```
select deptno, job, sum(sal) sal
  from emp
 group by deptno, job
```

DEPTNO	JOB	SAL
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	CLERK	1900

20	ANALYST	6000
20	MANAGER	2975
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600

然后，生成不同 JOB 和不同 DEPTNO 的薪水小计，以及整张表的薪水总计。为此，使用了 GROUP BY 的 CUBE 扩展。

```
select deptno,
       job,
       sum(sal) sal
from emp
group by cube(deptno,job)
```

DEPTNO	JOB	SAL

		29025
	CLERK	4150
	ANALYST	6000
	MANAGER	8275
	SALESMAN	5600
	PRESIDENT	5000
10		8750
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20		10875
20	CLERK	1900
20	ANALYST	6000
20	MANAGER	2975
30		9400
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600

接下来，结合使用函数 GROUPING 和 CASE 表达式设置结果的格式，让输出的含义更明确。根据 SAL 值是 GROUP BY 还是 CUBE 生成的，GROUPING(JOB) 将返回 1 或 0：如果结果是 CUBE 生成的，那么 GROUPING(JOB) 将返回 1，否则就返

回 0。GROUPING(DEPTNO) 的返回值与此类似。从解决方案的第一步可知，分组是根据 DEPTNO 和 JOB 进行的，因此对于表示特定 DEPTNO 和 JOB 组合对应的小计的行，GROUPING 返回的结果为 0。下面的查询证明了这一点。

```
select deptno,
       job,
       grouping(deptno) is_deptno_subtotal,
       grouping(job) is_job_subtotal,
       sum(sal) sal
  from emp
 group by cube(deptno,job)
 order by 3,4
```

DEPTNO	JOB	IS_DEPTNO_SUBTOTAL	IS_JOB_SUBTOTAL	SAL
10	CLERK	0	0	1300
10	MANAGER	0	0	2450
10	PRESIDENT	0	0	5000
20	CLERK	0	0	1900
30	CLERK	0	0	950
30	SALESMAN	0	0	5600
30	MANAGER	0	0	2850
20	MANAGER	0	0	2975
20	ANALYST	0	0	6000
10		0	1	8750
20		0	1	10875
30		0	1	9400
	CLERK	1	0	4150
	ANALYST	1	0	6000
	MANAGER	1	0	8275
	PRESIDENT	1	0	5000
	SALESMAN	1	0	5600
		1	1	29025

最后，使用 CASE 表达式根据 GROUPING(JOB) 和 GROUPING(DEPTNO) 返回的结果来确定各行所属的类别。

```
select deptno,
       job,
       case grouping(deptno)||grouping(job)
```

when '00' then 'TOTAL BY DEPT AND JOB' when '10' then 'TOTAL BY JOB' when '01' then 'TOTAL BY DEPT' when '11' then 'GRAND TOTAL FOR TABLE' end category, sum(sal) sal from emp group by cube(deptno,job) order by grouping(job),grouping(deptno)				
DEPTNO	JOB	CATEGORY		SAL
10	CLERK	TOTAL BY DEPT AND JOB		1300
10	MANAGER	TOTAL BY DEPT AND JOB		2450
10	PRESIDENT	TOTAL BY DEPT AND JOB		5000
20	CLERK	TOTAL BY DEPT AND JOB		1900
30	CLERK	TOTAL BY DEPT AND JOB		950
30	SALESMAN	TOTAL BY DEPT AND JOB		5600
30	MANAGER	TOTAL BY DEPT AND JOB		2850
20	MANAGER	TOTAL BY DEPT AND JOB		2975
20	ANALYST	TOTAL BY DEPT AND JOB		6000
	CLERK	TOTAL BY JOB		4150
	ANALYST	TOTAL BY JOB		6000
	MANAGER	TOTAL BY JOB		8275
	PRESIDENT	TOTAL BY JOB		5000
	SALESMAN	TOTAL BY JOB		5600
10		TOTAL BY DEPT		8750
30		TOTAL BY DEPT		9400
20		TOTAL BY DEPT		10875
		GRAND TOTAL FOR TABLE		29025

在 Oracle 解决方案中，隐式地将函数 **GROUPING** 返回的值转换成了字符类型，以便能够将两个值拼接起来。在 DB2 和 SQL Server 中，需要使用 **CAST** 显式地将函数 **GROUPING** 的结果转换为 **CHAR(1)**，如上述解决方案所示。另外，在 SQL Server 中，将两个 **GROUPING** 调用的结果拼接为一个字符串时，必须使用运算符 **+**（而不能使用运算符 **||**）。

在 Oracle 和 DB2 中，还有一个名为 **GROUPING SETS** 的 **GROUP BY** 扩展。这个扩展很有用，例如，可以使用它来生

成与 CUBE 类似的输出。（与使用 CUBE 的解决方案一样，在 DB2 和 SQL Server 中，需要使用 CAST 将函数 GROUPING 返回的值转换为正确的数据类型。）

```
select deptno,
       job,
       case grouping(deptno)||grouping(job)
         when '00' then 'TOTAL BY DEPT AND JOB'
         when '10' then 'TOTAL BY JOB'
         when '01' then 'TOTAL BY DEPT'
         when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
from emp
group by grouping sets ((deptno),(job),(deptno,job),())
```

DEPTNO	JOB	CATEGORY	SAL
-----	-----	-----	-----
10	CLERK	TOTAL BY DEPT AND JOB	1300
20	CLERK	TOTAL BY DEPT AND JOB	1900
30	CLERK	TOTAL BY DEPT AND JOB	950
20	ANALYST	TOTAL BY DEPT AND JOB	6000
10	MANAGER	TOTAL BY DEPT AND JOB	2450
20	MANAGER	TOTAL BY DEPT AND JOB	2975
30	MANAGER	TOTAL BY DEPT AND JOB	2850
30	SALESMAN	TOTAL BY DEPT AND JOB	5600
10	PRESIDENT	TOTAL BY DEPT AND JOB	5000
	CLERK	TOTAL BY JOB	4150
	ANALYST	TOTAL BY JOB	6000
	MANAGER	TOTAL BY JOB	8275
	SALESMAN	TOTAL BY JOB	5600
	PRESIDENT	TOTAL BY JOB	5000
10		TOTAL BY DEPT	8750
20		TOTAL BY DEPT	10875
30		TOTAL BY DEPT	9400
		GRAND TOTAL FOR TABLE	29025

GROUPING SETS 的优点在于允许你自定义分组。在上面的查询中，GROUPING SETS 子句按 DEPTNO、JOB 以及 DEPTNO 和 JOB 的组合分组，最后的空白括号生成了总计。在生成包含各种聚合等级的报告时，GROUPING SETS 提供了极大的灵

活性。例如，在上面的例子中，如果不想包含 **GRAND TOTAL**，只需将 **GROUPING SETS** 子句中的那对空白括号删除。

```
/* 不包含总计 */

select deptno,
       job,
       case grouping(deptno)||grouping(job)
         when '00' then 'TOTAL BY DEPT AND JOB'
         when '10' then 'TOTAL BY JOB'
         when '01' then 'TOTAL BY DEPT'
         when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
  from emp
 group by grouping sets ((deptno),(job),(deptno,job))
```

DEPTNO	JOB	CATEGORY	SAL
10	CLERK	TOTAL BY DEPT AND JOB	1300
20	CLERK	TOTAL BY DEPT AND JOB	1900
30	CLERK	TOTAL BY DEPT AND JOB	950
20	ANALYST	TOTAL BY DEPT AND JOB	6000
10	MANAGER	TOTAL BY DEPT AND JOB	2450
20	MANAGER	TOTAL BY DEPT AND JOB	2975
30	MANAGER	TOTAL BY DEPT AND JOB	2850
30	SALESMAN	TOTAL BY DEPT AND JOB	5600
10	PRESIDENT	TOTAL BY DEPT AND JOB	5000
	CLERK	TOTAL BY JOB	4150
	ANALYST	TOTAL BY JOB	6000
	MANAGER	TOTAL BY JOB	8275
	SALESMAN	TOTAL BY JOB	5600
	PRESIDENT	TOTAL BY JOB	5000
10		TOTAL BY DEPT	8750
20		TOTAL BY DEPT	10875
30		TOTAL BY DEPT	9400

还可以排除特定的小计（比如不同 **DEPTNO** 的小计），只需将前述 **GROUPING SETS** 子句中的 **(DEPTNO)** 删除。

```
/* 不包含不同DEPTNO的小计 */
```

```
select deptno,  
       job,  
       case grouping(deptno)||grouping(job)  
         when '00' then 'TOTAL BY DEPT AND JOB'  
         when '10' then 'TOTAL BY JOB'  
         when '01' then 'TOTAL BY DEPT'  
         when '11' then 'GRAND TOTAL FOR TABLE'  
       end category,  
       sum(sal) sal  
from emp  
group by grouping sets ((job),(deptno,job),())  
order by 3
```

DEPTNO	JOB	CATEGORY	SAL
		GRAND TOTAL FOR TABLE	29025
10	CLERK	TOTAL BY DEPT AND JOB	1300
20	CLERK	TOTAL BY DEPT AND JOB	1900
30	CLERK	TOTAL BY DEPT AND JOB	950
20	ANALYST	TOTAL BY DEPT AND JOB	6000
20	MANAGER	TOTAL BY DEPT AND JOB	2975
30	MANAGER	TOTAL BY DEPT AND JOB	2850
30	SALESMAN	TOTAL BY DEPT AND JOB	5600
10	PRESIDENT	TOTAL BY DEPT AND JOB	5000
10	MANAGER	TOTAL BY DEPT AND JOB	2450
	CLERK	TOTAL BY JOB	4150
	SALESMAN	TOTAL BY JOB	5600
	PRESIDENT	TOTAL BY JOB	5000
	MANAGER	TOTAL BY JOB	8275
	ANALYST	TOTAL BY JOB	6000

如你所见，使用 **GROUPING SETS** 可以轻松地生成不同的小计和总计，从不同的角度呈现数据。

MySQL

首先，使用聚合函数 **SUM**，并按照 **DEPTNO** 和 **JOB** 进行分组。

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job
```

DEPTNO	JOB	CATEGORY	SAL
10	CLERK	TOTAL BY DEPT AND JOB	1300
10	MANAGER	TOTAL BY DEPT AND JOB	2450
10	PRESIDENT	TOTAL BY DEPT AND JOB	5000
20	CLERK	TOTAL BY DEPT AND JOB	1900
20	ANALYST	TOTAL BY DEPT AND JOB	6000
20	MANAGER	TOTAL BY DEPT AND JOB	2975
30	CLERK	TOTAL BY DEPT AND JOB	950
30	MANAGER	TOTAL BY DEPT AND JOB	2850
30	SALESMAN	TOTAL BY DEPT AND JOB	5600

然后，使用 **UNION ALL** 添加不同职位的薪水小计（**TOTAL BY JOB**）。

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job
 union all
select null, job, 'TOTAL BY JOB', sum(sal)
  from emp
 group by job
```

DEPTNO	JOB	CATEGORY	SAL
10	CLERK	TOTAL BY DEPT AND JOB	1300
10	MANAGER	TOTAL BY DEPT AND JOB	2450
10	PRESIDENT	TOTAL BY DEPT AND JOB	5000
20	CLERK	TOTAL BY DEPT AND JOB	1900
20	ANALYST	TOTAL BY DEPT AND JOB	6000
20	MANAGER	TOTAL BY DEPT AND JOB	2975
30	CLERK	TOTAL BY DEPT AND JOB	950
30	MANAGER	TOTAL BY DEPT AND JOB	2850
30	SALESMAN	TOTAL BY DEPT AND JOB	5600
	ANALYST	TOTAL BY JOB	6000

CLERK	TOTAL BY JOB	4150
MANAGER	TOTAL BY JOB	8275
PRESIDENT	TOTAL BY JOB	5000
SALESMAN	TOTAL BY JOB	5600

接下来，使用 **UNION ALL** 添加不同 **DEPTNO** 的薪水小计。

```

select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job
 union all
select null, job, 'TOTAL BY JOB', sum(sal)
  from emp
 group by job
 union all
select deptno, null, 'TOTAL BY DEPT', sum(sal)
  from emp
 group by deptno

```

DEPTNO	JOB	CATEGORY	SAL
-----	-----	-----	-----
10	CLERK	TOTAL BY DEPT AND JOB	1300
10	MANAGER	TOTAL BY DEPT AND JOB	2450
10	PRESIDENT	TOTAL BY DEPT AND JOB	5000
20	CLERK	TOTAL BY DEPT AND JOB	1900
20	ANALYST	TOTAL BY DEPT AND JOB	6000
20	MANAGER	TOTAL BY DEPT AND JOB	2975
30	CLERK	TOTAL BY DEPT AND JOB	950
30	MANAGER	TOTAL BY DEPT AND JOB	2850
30	SALESMAN	TOTAL BY DEPT AND JOB	5600
	ANALYST	TOTAL BY JOB	6000
	CLERK	TOTAL BY JOB	4150
	MANAGER	TOTAL BY JOB	8275
	PRESIDENT	TOTAL BY JOB	5000
	SALESMAN	TOTAL BY JOB	5600
10		TOTAL BY DEPT	8750
20		TOTAL BY DEPT	10875
30		TOTAL BY DEPT	9400

最后，使用 **UNION ALL** 添加整张表的薪水总计。

```

select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job
 union all
select null, job, 'TOTAL BY JOB', sum(sal)
  from emp
 group by job
 union all
select deptno, null, 'TOTAL BY DEPT', sum(sal)
  from emp
 group by deptno
 union all
select null,null, 'GRAND TOTAL FOR TABLE', sum(sal)
  from emp

```

DEPTNO	JOB	CATEGORY	SAL
-----	-----	-----	-----
10	CLERK	TOTAL BY DEPT AND JOB	1300
10	MANAGER	TOTAL BY DEPT AND JOB	2450
10	PRESIDENT	TOTAL BY DEPT AND JOB	5000
20	CLERK	TOTAL BY DEPT AND JOB	1900
20	ANALYST	TOTAL BY DEPT AND JOB	6000
20	MANAGER	TOTAL BY DEPT AND JOB	2975
30	CLERK	TOTAL BY DEPT AND JOB	950
30	MANAGER	TOTAL BY DEPT AND JOB	2850
30	SALESMAN	TOTAL BY DEPT AND JOB	5600
	ANALYST	TOTAL BY JOB	6000
	CLERK	TOTAL BY JOB	4150
	MANAGER	TOTAL BY JOB	8275
	PRESIDENT	TOTAL BY JOB	5000
	SALESMAN	TOTAL BY JOB	5600
10		TOTAL BY DEPT	8750
20		TOTAL BY DEPT	10875
30		TOTAL BY DEPT	9400
		GRAND TOTAL FOR TABLE	29025

12.14 标出非小计行

1. 问题

在使用 **GROUP BY** 子句的 **CUBE** 扩展生成的报表中，你想将 **GROUP BY** 子句生成的行同 **CUBE** 或 **ROLLUP** 生成的行区分开。

下面的结果集是使用 **GROUP BY** 子句的 **CUBE** 扩展生成的，其显示了 **EMP** 表中的薪水分布情况。

DEPTNO	JOB	SAL
		29025
	CLERK	4150
	ANALYST	6000
	MANAGER	8275
	SALESMAN	5600
	PRESIDENT	5000
10		8750
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20		10875
20	CLERK	1900
20	ANALYST	6000
20	MANAGER	2975
30		9400
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600

这张报表包含不同 **DEPTNO** 和 **JOB** 组合的薪水小计，不同 **DEPTNO** 的薪水小计，不同 **JOB** 的薪水小计以及整个 **EMP** 表的薪水总计，而你想清晰地指出不同的聚合等级。换言之，你想标出各个聚合值所属的类别，即 **SAL** 列值表示的是不同 **DEPTNO** 的小计，不同 **JOB** 的小计，还是总计？你希望返回

的结果集如下所示。

DEPTNO	JOB	SAL	DEPTNO_SUBTOTALS	JOB_SUBTOTALS
		29025	1	1
	CLERK	4150	1	0
	ANALYST	6000	1	0
	MANAGER	8275	1	0
	SALESMAN	5600	1	0
	PRESIDENT	5000	1	0
10		8750	0	1
10	CLERK	1300	0	0
10	MANAGER	2450	0	0
10	PRESIDENT	5000	0	0
20		10875	0	1
20	CLERK	1900	0	0
20	ANALYST	6000	0	0
20	MANAGER	2975	0	0
30		9400	0	1
30	CLERK	950	0	0
30	MANAGER	2850	0	0
30	SALESMAN	5600	0	0

12. 解决方案

使用函数 **GROUPING** 来指出哪些值是 **CUBE** 或 **ROLLUP** 生成的小计（超级聚合值）。下面的解决方案适用于 PostgreSQL、DB2 和 Oracle。

```
1 select deptno, job, sal,
2         grouping(deptno) deptno_subtotals,
3         grouping(job) job_subtotals
4   from emp
5  group by cube(deptno, job)
```

与 DB2 和 Oracle 解决方案相比，SQL Server 解决方案的唯一不同之处是 **CUBE/ROLLUP** 子句的书写方式。


```
1 select deptno, job, sum(sal) sal,  
2         grouping(deptno) deptno_subtotals,  
3         grouping(job) job_subtotals  
4   from emp  
5  group by deptno,job with cube
```

本实例旨在突出 **CUBE** 和 **GROUPING** 在生成小计方面的用途。本书撰写之时，MySQL 不支持 **CUBE** 和 **GROUPING**。

13. 讨论

如果 **DEPTNO_SUBTOTALS** 为 0，并且 **JOB_SUBTOTALS** 为 1（**JOB** 值为 **NULL**），则 **SAL** 值指的是 **CUBE** 生成的特定 **DEPTNO** 的薪水小计。如果 **JOB_SUBTOTALS** 为 0，并且 **DEPTNO_SUBTOTALS** 为 1（**DEPTNO** 值为 **NULL**），则 **SAL** 值指的是 **CUBE** 生成的特定 **JOB** 的薪水小计。在 **DEPTNO_SUBTOTALS** 和 **JOB_SUBTOTALS** 都为 0 的行中，**SAL** 值指的是常规聚合值（特定 **DEPTNO/JOB** 组合的薪水小计）。

12.15 使用CASE表达式来标识行

1. 问题

你想将特定列（比如 EMP 表中的 JOB 列）的值转换为“布尔”标志。例如，你想返回如下结果集。

ENAME	IS_CLERK	IS_SALES	IS_MGR	IS_ANALYST	IS_PREZ
KING	0	0	0	0	1
SCOTT	0	0	0	1	0
FORD	0	0	0	1	0
JONES	0	0	1	0	0
BLAKE	0	0	1	0	0
CLARK	0	0	1	0	0
ALLEN	0	1	0	0	0
WARD	0	1	0	0	0
MARTIN	0	1	0	0	0
TURNER	0	1	0	0	0
SMITH	1	0	0	0	0
MILLER	1	0	0	0	0
ADAMS	1	0	0	0	0
JAMES	1	0	0	0	0

这样的结果集很有用，不仅可以帮助调试，还可以提供不同于典型结果集的数据视图。

2. 解决方案

使用 CASE 表达式来评估每位员工的 JOB，并返回 1 或 0，以指出员工的职位。对于每种可能的职位，都需要编写一个 CASE 表达式，以生成相应的列值。

```
1 select ename,
```

```
2      case when job = 'CLERK'
3          then 1 else 0
4      end as is_clerk,
5      case when job = 'SALESMAN'
6          then 1 else 0
7      end as is_sales,
8      case when job = 'MANAGER'
9          then 1 else 0
10     end as is_mgr,
11     case when job = 'ANALYST'
12         then 1 else 0
13     end as is_analyst,
14     case when job = 'PRESIDENT'
15         then 1 else 0
16     end as is_prez
17 from emp
18 order by 2,3,4,5,6
```

13. 讨论

上述解决方案中代码的含义不言自明。如果你理解不了，那么可以在 **SELECT** 子句中添加 **JOB**，这可以帮助理解。

```
select ename,
       job,
       case when job = 'CLERK'
           then 1 else 0
       end as is_clerk,
       case when job = 'SALESMAN'
           then 1 else 0
       end as is_sales,
       case when job = 'MANAGER'
           then 1 else 0
       end as is_mgr,
       case when job = 'ANALYST'
           then 1 else 0
       end as is_analyst,
       case when job = 'PRESIDENT'
           then 1 else 0
       end as is_prez
```

from emp
order by 2

ENAME	JOB	IS_CLERK	IS_SALES	IS_MGR	IS_ANALYST	IS_PREZ
SCOTT	ANALYST	0	0	0	1	0
FORD	ANALYST	0	0	0	1	0
SMITH	CLERK	1	0	0	0	0
ADAMS	CLERK	1	0	0	0	0
MILLER	CLERK	1	0	0	0	0
JAMES	CLERK	1	0	0	0	0
JONES	MANAGER	0	0	1	0	0
CLARK	MANAGER	0	0	1	0	0
BLAKE	MANAGER	0	0	1	0	0
KING	PRESIDENT	0	0	0	0	1
ALLEN	SALESMAN	0	1	0	0	0
MARTIN	SALESMAN	0	1	0	0	0
TURNER	SALESMAN	0	1	0	0	0
WARD	SALESMAN	0	1	0	0	0

12.16 创建稀疏矩阵

1. 问题

你想创建一个稀疏矩阵，比如下面这个对 EMP 表中 DEPTNO 列和 JOB 列进行转置而得到的稀疏矩阵。

D10	D20	D30	CLERKS	MGRS	PREZ	ANALS	SALES
	SMITH		SMITH				
		ALLEN					ALLEN
		WARD					WARD
	JONES			JONES			
		MARTIN					MARTIN
		BLAKE		BLAKE			
CLARK				CLARK			
	SCOTT					SCOTT	
KING					KING		
		TURNER					TURNER
	ADAMS		ADAMS				
		JAMES	JAMES				
	FORD					FORD	
MILLER			MILLER				

2. 解决方案

使用 CASE 表达式来创建稀疏的行列转换结果。

```
1 select case deptno when 10 then ename end as d10,
2        case deptno when 20 then ename end as d20,
3        case deptno when 30 then ename end as d30,
4        case job when 'CLERK' then ename end as clerks,
5        case job when 'MANAGER' then ename end as mgrs,
6        case job when 'PRESIDENT' then ename end as prez,
7        case job when 'ANALYST' then ename end as anal,
8        case job when 'SALESMAN' then ename end as sales
```

13. 讨论

要将 **DEPTNO** 行和 **JOB** 行转换为列，只需使用 **CASE** 表达式来评估这些行返回的每个可能值。另外，如果要删除 **NULL** 行以生成密集型报表，那么需要根据某种标准进行分组。例如，使用窗口函数 **ROW_NUMBER OVER** 对每个部门的员工进行排名，然后再使用聚合函数 **MAX** 删除某些 **NULL** 值。

```
select max(case deptno when 10 then ename end) d10,
       max(case deptno when 20 then ename end) d20,
       max(case deptno when 30 then ename end) d30,
       max(case job when 'CLERK' then ename end) clerks,
       max(case job when 'MANAGER' then ename end) mgrs,
       max(case job when 'PRESIDENT' then ename end) prez,
       max(case job when 'ANALYST' then ename end) anals,
       max(case job when 'SALESMAN' then ename end) sales
  from (
select deptno, job, ename,
       row_number()over(partition by deptno order by empno) rn
  from emp
  ) x
 group by rn
```

D10	D20	D30	CLERKS	MGRS	PREZ	ANALS	SALES
-----	-----	-----	-----	-----	-----	-----	-----
CLARK	SMITH	ALLEN	SMITH	CLARK			ALLEN
KING	JONES	WARD		JONES	KING		WARD
MILLER	SCOTT	MARTIN	MILLER		SCOTT		MARTIN
	ADAMS	BLAKE	ADAMS	BLAKE			
	FORD	TURNER			FORD		TURNER
		JAMES	JAMES				

12.17 按时间分组

1. 问题

你想按时段汇总数据。例如，你有一个交易日志，你想以 5 秒为间隔对交易数据进行分组和汇总。**TRX_LOG** 表包含的数据行如下所示。

```
select trx_id,
       trx_date,
       trx_cnt
  from trx_log
```

TRX_ID	TRX_DATE	TRX_CNT
1	28-JUL-2020 19:03:07	44
2	28-JUL-2020 19:03:08	18
3	28-JUL-2020 19:03:09	23
4	28-JUL-2020 19:03:10	29
5	28-JUL-2020 19:03:11	27
6	28-JUL-2020 19:03:12	45
7	28-JUL-2020 19:03:13	45
8	28-JUL-2020 19:03:14	32
9	28-JUL-2020 19:03:15	41
10	28-JUL-2020 19:03:16	15
11	28-JUL-2020 19:03:17	24
12	28-JUL-2020 19:03:18	47
13	28-JUL-2020 19:03:19	37
14	28-JUL-2020 19:03:20	48
15	28-JUL-2020 19:03:21	46
16	28-JUL-2020 19:03:22	44
17	28-JUL-2020 19:03:23	36
18	28-JUL-2020 19:03:24	41
19	28-JUL-2020 19:03:25	33
20	28-JUL-2020 19:03:26	19

而你想返回如下结果集。

GRP	TRX_START	TRX_END	TOTAL
---	-----	-----	-----

1	28-JUL-2020	19:03:07	28-JUL-2020	19:03:11	141
2	28-JUL-2020	19:03:12	28-JUL-2020	19:03:16	178
3	28-JUL-2020	19:03:17	28-JUL-2020	19:03:21	202
4	28-JUL-2020	19:03:22	28-JUL-2020	19:03:26	173

12. 解决方案

将全部数据按条目进行分桶，每桶包含 5 行数据。实现这种逻辑分组的方式有多种，本实例使用的是 12.7 节介绍的方法，即用 **TRX_ID** 值除以 5。

创建好“分组”后，使用聚合函数 **MIN**、**MAX** 和 **SUM** 找出每个分组的起始时间、终止时间和交易总数。（在 SQL Server 中，应该使用 **CEILING** 而不是 **CEIL**。）

```

1 select ceil(trx_id/5.0) as grp,
2       min(trx_date)    as trx_start,
3       max(trx_date)    as trx_end,
4       sum(trx_cnt)      as total
5   from trx_log
6  group by ceil(trx_id/5.0)

```

13. 讨论

首先，对行进行逻辑分组，这是整个解决方案的关键。通过除以 5，并获取不小于所得商的最小整数，可以创建逻辑分组。

```

select trx_id,
       trx_date,
       trx_cnt,
       trx_id/5.0 as val,
       ceil(trx_id/5.0) as grp

```


from trx_log					
TRX_ID	TRX_DATE		TRX_CNT	VAL	GRP

1	28-JUL-2020 19:03:07		44	.20	1
2	28-JUL-2020 19:03:08		18	.40	1
3	28-JUL-2020 19:03:09		23	.60	1
4	28-JUL-2020 19:03:10		29	.80	1
5	28-JUL-2020 19:03:11		27	1.00	1
6	28-JUL-2020 19:03:12		45	1.20	2
7	28-JUL-2020 19:03:13		45	1.40	2
8	28-JUL-2020 19:03:14		32	1.60	2
9	28-JUL-2020 19:03:15		41	1.80	2
10	28-JUL-2020 19:03:16		15	2.00	2
11	28-JUL-2020 19:03:17		24	2.20	3
12	28-JUL-2020 19:03:18		47	2.40	3
13	28-JUL-2020 19:03:19		37	2.60	3
14	28-JUL-2020 19:03:20		48	2.80	3
15	28-JUL-2020 19:03:21		46	3.00	3
16	28-JUL-2020 19:03:22		44	3.20	4
17	28-JUL-2020 19:03:23		36	3.40	4
18	28-JUL-2020 19:03:24		41	3.60	4
19	28-JUL-2020 19:03:25		33	3.80	4
20	28-JUL-2020 19:03:26		19	4.00	4

最后，使用合适的聚合函数找出各个 5 秒时段的交易总数，以及起始时间和终止时间。

<pre> select ceil(trx_id/5.0) as grp, min(trx_date) as trx_start, max(trx_date) as trx_end, sum(trx_cnt) as total from trx_log group by ceil(trx_id/5.0) </pre>					
GRP	TRX_START		TRX_END		TOTAL

1	28-JUL-2020 19:03:07	28-JUL-2005	19:03:11		141
2	28-JUL-2020 19:03:12	28-JUL-2005	19:03:16		178
3	28-JUL-2020 19:03:17	28-JUL-2005	19:03:21		202
4	28-JUL-2020 19:03:22	28-JUL-2005	19:03:26		173

如果你的数据稍有不同（可能没有 **ID** 列），那么可以将 **TRX_DATE** 的分钟数和秒数部分除以 5 来创建类似的分组。

然后，获取 **TRX_DATE** 的小时部分，并根据小时部分和逻辑分组（**GRP**）进行分组。下面就是一个这样的示例。（这里使用的是 Oracle 函数 **TO_CHAR** 和 **TO_NUMBER**，在其他 RDBMS 中，需要使用相应的日期和字符格式设置函数。）

```
select trx_date, trx_cnt,
       to_number(to_char(trx_date, 'hh24')) hr,
       ceil(to_number(to_char(trx_date-1/24/60/60, 'miss'))/5.0)
  grp
from trx_log
```

TRX_DATE	20 TRX_CNT	HR	GRP
28-JUL-2020 19:03:07	44	19	62
28-JUL-2020 19:03:08	18	19	62
28-JUL-2020 19:03:09	23	19	62
28-JUL-2020 19:03:10	29	19	62
28-JUL-2020 19:03:11	27	19	62
28-JUL-2020 19:03:12	45	19	63
28-JUL-2020 19:03:13	45	19	63
28-JUL-2020 19:03:14	32	19	63
28-JUL-2020 19:03:15	41	19	63
28-JUL-2020 19:03:16	15	19	63
28-JUL-2020 19:03:17	24	19	64
28-JUL-2020 19:03:18	47	19	64
28-JUL-2020 19:03:19	37	19	64
28-JUL-2020 19:03:20	48	19	64
28-JUL-2020 19:03:21	46	19	64
28-JUL-2020 19:03:22	44	19	65
28-JUL-2020 19:03:23	36	19	65
28-JUL-2020 19:03:24	41	19	65
28-JUL-2020 19:03:25	33	19	65
28-JUL-2020 19:03:26	19	19	65

这里的关键是将每个 5 秒的时段作为一组，**GRP** 的实际值是什么则无关紧要。然后就可以像上述解决方案那样使用聚合函数了。

```
select hr, grp, sum(trx_cnt) total
  from (
```

```

select trx_date, trx_cnt,
       to_number(to_char(trx_date, 'hh24')) hr,
       ceil(to_number(to_char(trx_date-1/24/60/60, 'miss'))/5.0)
grp
  from trx_log
     ) x
  group by hr, grp

```

HR	GRP	TOTAL
19	62	141
19	63	178
19	64	202
19	65	173

在交易日志覆盖的时间超过 1 小时时，将小时部分作为分组依据的一部分很有用。在 DB2 和 Oracle 中，也可以使用窗口函数 **SUM OVER** 来生成同样的结果。下面的查询会返回 **TRX_LOG** 表中所有的行，以及逻辑分组中的 **TRX_CNT** 移动总计和 **TRX_CNT** 总计。

```

select trx_id, trx_date, trx_cnt,
       sum(trx_cnt)over(partition by ceil(trx_id/5.0)
                        order by trx_date
                        range between unbounded preceding
                               and current row) runing_total,
       sum(trx_cnt)over(partition by ceil(trx_id/5.0)) total,
       case when mod(trx_id,5.0) = 0 then 'X' end grp_end
  from trx_log

```

TRX_ID	TRX_DATE	TRX_CNT	RUNING_TOTAL	TOTAL
GRP_END				
1	28-JUL-2020 19:03:07	44	44	141
2	28-JUL-2020 19:03:08	18	62	141
3	28-JUL-2020 19:03:09	23	85	141
4	28-JUL-2020 19:03:10	29	114	141
5	28-JUL-2020 19:03:11	27	141	141 X
6	28-JUL-2020 19:03:12	45	45	178
7	28-JUL-2020 19:03:13	45	90	178
8	28-JUL-2020 19:03:14	32	122	178

9	28-JUL-2020	19:03:15	41	163	178
10	28-JUL-2020	19:03:16	15	178	178 X
11	28-JUL-2020	19:03:17	24	24	202
12	28-JUL-2020	19:03:18	47	71	202
13	28-JUL-2020	19:03:19	37	108	202
14	28-JUL-2020	19:03:20	48	156	202
15	28-JUL-2020	19:03:21	46	202	202 X
16	28-JUL-2020	19:03:22	44	44	173
17	28-JUL-2020	19:03:23	36	80	173
18	28-JUL-2020	19:03:24	41	121	173
19	28-JUL-2020	19:03:25	33	154	173
20	28-JUL-2020	19:03:26	19	173	173 X

12.18 同时对不同的分组/分区进行聚合

1. 问题

你想同时聚合不同的维度。例如，你想返回一个结果集，其中列出了每位员工的名字、所属部门、所属部门的员工数、所属职位的员工数以及整个 **EMP** 表中的员工总数。结果集如下所示。

ENAME	DEPTNO	DEPTNO_CNT	JOB	JOB_CNT	TOTAL
MILLER	10	3	CLERK	4	14
CLARK	10	3	MANAGER	3	14
KING	10	3	PRESIDENT	1	14
SCOTT	20	5	ANALYST	2	14
FORD	20	5	ANALYST	2	14
SMITH	20	5	CLERK	4	14
JONES	20	5	MANAGER	3	14
ADAMS	20	5	CLERK	4	14
JAMES	30	6	CLERK	4	14
MARTIN	30	6	SALESMAN	4	14
TURNER	30	6	SALESMAN	4	14
WARD	30	6	SALESMAN	4	14
ALLEN	30	6	SALESMAN	4	14
BLAKE	30	6	MANAGER	3	14

2. 解决方案

使用窗口函数 **COUNT OVER**，并指定要进行聚合的数据分区（分组）。

```
select ename,
       deptno,
       count(*)over(partition by deptno) deptno_cnt,
       job,
```

```
count(*)over(partition by job) job_cnt,  
count(*)over() total  
from emp
```

13. 讨论

本实例充分展示了窗口函数的威力和便利之处。只需指定要聚合的各种数据分区（分组），就可以制作非常详细的报表，既不需要反复自连接，也不需要编写 **SELECT** 列表中编写烦琐（还可能性能低下）的子查询。所有的工作都由窗口函数 **COUNT OVER** 完成。为了弄明白输出结果，请专注于每个 **COUNT** 操作的 **OVER** 子句。

```
count(*)over(partition by deptno)  
  
count(*)over(partition by job)  
  
count(*)over()
```

请记住 **OVER** 子句的主要部分，即 **PARTITION BY** 和 **ORDER BY**，前者指定分区依据，后者指定逻辑顺序。请看第一个 **COUNT**，它根据 **DEPTNO** 进行分区。因此，按 **DEPTNO** 将 **EMP** 表中的行分组，并对每个分组执行 **COUNT** 操作。由于没有指定框架或窗口子句（没有 **ORDER BY**），因此需要计算每个分组的总行数。**PARTITION BY** 子句会找出所有独特的 **DEPTNO** 值，而函数 **COUNT** 会计算包含特定 **DEPTNO** 值的行数。在 **COUNT(*)OVER(PARTITION BY DEPTNO)** 中，**PARTITION BY** 子句找出了分区（分组）值 10、20 和 30。

第二个 **COUNT** 操作按 **JOB** 分区，但工作原理一样。最后一个 **COUNT** 操作没有进行分区，因为括号内是空的。空括号表示“整张表”。因此，前两个 **COUNT** 操作根据指定的分组（分

区) 执行聚合, 而最后一个 **COUNT** 操作会计算 **EMP** 表的总行数。



别忘了, 窗口函数是在 **WHERE** 子句后执行的。如果以某种方式对结果集进行了过滤 (比如将 10 号部门的员工都排除在外), 那么 **TOTAL** 值将为 11, 而不是 14。如果在执行完窗口函数后还想做一些结果过滤, 则必须将窗口查询放在一个内嵌视图中, 再对这个视图返回的结果进行过滤。

12.19 聚合移动值区间

1. 问题

你想计算移动聚合，比如 EMP 表的移动薪水总计。你想从第一位员工的 HIREDATE 开始，计算 90 天内的薪水总计，以便了解从第一位员工获聘到最近一位员工获聘期间，90 天内的薪水波动情况。你想返回如下结果集。

HIREDATE	SAL	SPENDING_PATTERN
-----	----	-----
17-DEC-200	800	800
20-FEB-2011	1600	2400
22-FEB-2011	1250	3650
02-APR-2011	2975	5825
01-MAY-2011	2850	8675
09-JUN-2011	2450	8275
08-SEP-2011	1500	1500
28-SEP-2011	1250	2750
17-NOV-2011	5000	7750
03-DEC-2011	950	11700
03-DEC-2011	3000	11700
23-JAN-2012	1300	10250
09-DEC-2012	3000	3000
12-JAN-2013	1100	4100

2. 解决方案

如果你使用的 RDBMS 支持相关的窗口函数，并允许在框架或窗口子句中指定移动窗口，那么这个问题解决起来将易如反掌。关键是在窗口函数中按 HIREDATE 排序，然后再指定一个 90 天的移动窗口（起点为第一位员工的获聘日期）。计算的总薪水为当前员工获聘日期前 90 天内所有获聘员工（包括当前员工）的薪水总和。如果没有可供使用的窗口函

数，则可以使用标量子查询，但解决方案将更复杂。

DB2 和 Oracle

在 DB2 和 Oracle 中，使用窗口函数 **SUM OVER**，并按 **HIREDATE** 排序。在窗口或框架子句中，指定 90 天的区间，以计算当前员工以及之前 90 天内获聘的其他所有员工的薪水总和。在 DB2 中，不能在窗口函数的 **ORDER BY** 子句中指定 **HIREDATE**，必须按 **DAYS(HIREDATE)** 排序，如下述代码的第 3 行所示。

```
1 select hiredate,
2         sal,
3         sum(sal)over(order by days(hiredate)
4                       range between 90 preceding
5                             and current row) spending_pattern
6 from emp e
```

相比于 DB2 解决方案，Oracle 解决方案更简单，因为在 Oracle 中，可以在窗口函数中将日期时间数据类型作为排序依据。

```
1 select hiredate,
2         sal,
3         sum(sal)over(order by hiredate
4                       range between 90 preceding
5                             and current row) spending_pattern
6 from emp e
```

MySQL

使用窗口函数 **SUM OVER**，但语法稍有不同。

```
1 select hiredate,
2         sal,
3         sum(sal)over(order by hiredate
4                       range interval 90 day preceding ) spending_pattern
```

```
5 from emp e
```

PostgreSQL 和 SQL Server

使用标量子查询来计算当前员工获聘日期前 90 天内获聘的所有员工的薪水总和。

```
1 select e.hiredate,  
2       e.sal,  
3       (select sum(sal) from emp d  
4         whered.hiredate between e.hiredate-90  
5                               and e.hiredate) as  
6       spending_pattern  
7 from emp e  
8 order by 1
```

13. 讨论

DB2、MySQL 和 Oracle

DB2、MySQL 和 Oracle 解决方案基于相同的逻辑，它们之间只有细微的差别，就是在窗口函数的 **ORDER BY** 子句中指定 **HIREDATE** 的方式，以及在 MySQL 中指定时间段的语法。本书撰写之时，在 DB2 中，如果设置窗口的范围时使用的是数字值，就不能在 **ORDER BY** 子句中使用 **DATE** 值，例如，使用 **RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW** 指定窗口范围时，可以按日期排序，但使用 **RANGE BETWEEN 90 PRECEDING AND CURRENT ROW** 指定窗口范围时，不能这样做。

要弄明白整个解决方案所做的工作，只需明白窗口子句所做的工作。你定义的窗口会根据 **HIREDATE** 将员工排序，然后，窗口函数会计算总和。计算的总和并不是所有员工的薪

水总和。具体处理过程如下。

- a. 评估获聘的第一位员工的薪水。由于之前没有聘请其他员工，因此此时的薪水总和为第一位员工的薪水。
- b. 根据 **HIREDATE** 确定下一位员工，并评估其薪水。将这位员工的薪水和 90 天内获聘的其他员工的薪水相加，得到移动总和。

第一位员工的 **HIREDATE** 为 2010 年 12 月 17 日，而第二位员工的 **HIREDATE** 为 2011 年 2 月 20 日。聘请第一位员工后，未过 90 天就聘请了第二位员工，因此对于第二位员工，移动总和为 2400（1600 + 800）。如果你不明白 **SPENDING_PATTERN** 列的值是怎么来的，请看下面的查询及其结果集。

```
select distinct
    dense_rank()over(order by e.hiredate) window,
    e.hiredate current_hiredate,
    d.hiredate hiredate_within_90_days,
    d.sal sals_used_for_sum
from emp e,
     emp d
where d.hiredate between e.hiredate-90 and e.hiredate
```

	WINDOW	CURRENT_HIREDATE	HIREDATE_WITHIN_90_DAYS	SALS_USED_FOR_SUM
1	17-DEC-2010	17-DEC-2010		800
2	20-FEB-2011	17-DEC-2010		800
2	20-FEB-2011	20-FEB-2011		1600
3	22-FEB-2011	17-DEC-2010		800
3	22-FEB-2011	20-FEB-2011		1600
3	22-FEB-2011	22-FEB-2011		1250
4	02-APR-2011	20-FEB-2011		1600
4	02-APR-2011	22-FEB-2011		1250
4	02-APR-2011	02-APR-2011		2975
5	01-MAY-2011	20-FEB-2011		1600
5	01-MAY-2011	22-FEB-2011		1250
5	01-MAY-2011	02-APR-2011		2975
5	01-MAY-2011	01-MAY-2011		2850

6	09-JUN-2011	02-APR-2011	2975
6	09-JUN-2011	01-MAY-2011	2850
6	09-JUN-2011	09-JUN-2011	2450
7	08-SEP-2011	08-SEP-2011	1500
8	28-SEP-2011	08-SEP-2011	1500
8	28-SEP-2011	28-SEP-2011	1250
9	17-NOV-2011	08-SEP-2011	1500
9	17-NOV-2011	28-SEP-2011	1250
9	17-NOV-2011	17-NOV-2011	5000
10	03-DEC-2011	08-SEP-2011	1500
10	03-DEC-2011	28-SEP-2011	1250
10	03-DEC-2011	17-NOV-2011	5000
10	03-DEC-2011	03-DEC-2011	950
10	03-DEC-2011	03-DEC-2011	3000
11	23-JAN-2012	17-NOV-2011	5000
11	23-JAN-2012	03-DEC-2011	950
11	23-JAN-2012	03-DEC-2011	3000
11	23-JAN-2012	23-JAN-2012	1300
12	09-DEC-2012	09-DEC-2012	3000
13	12-JAN-2013	09-DEC-2012	3000
13	12-JAN-2013	12-JAN-2013	1100

计算总和时，只考虑 **WINDOW** 列值相同的行。例如，对于窗口（**WINDOW**）3，计算其总和时只考虑薪水 800、1600 和 1250，结果为 3650。如果你查看本节“问题”部分的最终结果集，将发现对于日期 2011 年 2 月 22 日（对应于窗口 3），**SPENDING_PATTERN** 值为 3650。要核实前面的自连接查询在定义的窗口中包含正确的薪水值，只需按 **CURRENT_DATE** 分组，并计算各个分组的 **SALS_USED_FOR_SUM** 总和。结果应该与本节“问题”部分显示的结果集相同（这里将日期为 2011 年 12 月 3 日的重复行排除在外了）。

```
select current_hiredate,
       sum(sals_used_for_sum) spending_pattern
  from (
select distinct
       dense_rank()over(order by e.hiredate) window,
       e.hiredate current_hiredate,
       d.hiredate hiredate_within_90_days,
```

```

        d.sal sals_used_for_sum
    from emp e,
        emp d
    where d.hiredate between e.hiredate-90 and e.hiredate
        ) x
    group by current_hiredate

```

```

CURRENT_HIREDATE  SPENDING_PATTERN
-----

```

```

17-DEC-2010      800
20-FEB-2011     2400
22-FEB-2011     3650
02-APR-2011     5825
01-MAY-2011     8675
09-JUN-2011     8275
08-SEP-2011     1500
28-SEP-2011     2750
17-NOV-2011     7750
03-DEC-2011    11700
23-JAN-2012    10250
09-DEC-2012     3000
12-JAN-2013     4100

```

PostgreSQL 和SQL Server

该解决方案的关键是使用标量子查询（或自连接），并使用聚合函数 **SUM** 计算当前员工的 **HIREDATE** 之前 90 天内的薪水总和。如果不明白其中的工作原理，那么可以将该解决方案转换为自连接，并查看计算总和时涉及哪些行。请看下面的查询，它返回了与前面的解决方案相同的结果集。

```

select e.hiredate,
       e.sal,
       sum(d.sal) as spending_pattern
  from emp e, emp d
 where d.hiredate
        between e.hiredate-90 and e.hiredate
 group by e.hiredate,e.sal
 order by 1\

```

```

HIREDATE      SAL  SPENDING_PATTERN

```

17-DEC-2010	800	800
20-FEB-2011	1600	2400
22-FEB-2011	1250	3650
02-APR-2011	2975	5825
01-MAY-2011	2850	8675
09-JUN-2011	2450	8275
08-SEP-2011	1500	1500
28-SEP-2011	1250	2750
17-NOV-2011	5000	7750
03-DEC-2011	950	11700
03-DEC-2011	3000	11700
23-JAN-2012	1300	10250
09-DEC-2012	3000	3000
12-JAN-2013	1100	4100

如果还不明白，则可以将聚合函数删除，并从笛卡儿积开始。先使用 **EMP** 表生成笛卡儿积，以便将每个 **HIREDATE** 同其他所有的 **HIREDATE** 进行比较。这里只显示了结果集的一部分，因为 **EMP** 表的笛卡儿积包含 196（14 × 14）行数据。

<pre>select e.hiredate, e.sal, d.sal, d.hiredate from emp e, emp d</pre>			
HIREDATE	SAL	SAL	HIREDATE
17-DEC-2010	800	800	17-DEC-2010
17-DEC-2010	800	1600	20-FEB-2011
17-DEC-2010	800	1250	22-FEB-2011
17-DEC-2010	800	2975	02-APR-2011
17-DEC-2010	800	1250	28-SEP-2011
17-DEC-2010	800	2850	01-MAY-2011
17-DEC-2010	800	2450	09-JUN-2011
17-DEC-2010	800	3000	09-DEC-2012
17-DEC-2010	800	5000	17-NOV-2011
17-DEC-2010	800	1500	08-SEP-2011
17-DEC-2010	800	1100	12-JAN-2013
17-DEC-2010	800	950	03-DEC-2011
17-DEC-2010	800	3000	03-DEC-2011

17-DEC-2010	800	1300	23-JAN-2012
20-FEB-2011	1600	800	17-DEC-2010
20-FEB-2011	1600	1600	20-FEB-2011
20-FEB-2011	1600	1250	22-FEB-2011
20-FEB-2011	1600	2975	02-APR-2011
20-FEB-2011	1600	1250	28-SEP-2011
20-FEB-2011	1600	2850	01-MAY-2011
20-FEB-2011	1600	2450	09-JUN-2011
20-FEB-2011	1600	3000	09-DEC-2012
20-FEB-2011	1600	5000	17-NOV-2011
20-FEB-2011	1600	1500	08-SEP-2011
20-FEB-2011	1600	1100	12-JAN-2013
20-FEB-2011	1600	950	03-DEC-2011
20-FEB-2011	1600	3000	03-DEC-2011
20-FEB-2011	1600	1300	23-JAN-2012

从这个结果集可知，没有其他 **HIREDATE** 在 2010 年 12 月 17 日之前的 90 天内，因此在这个日期对应的行中，总和为 800。如果查看下一个 **HIREDATE**（2012 年 2 月 20 日），你将发现只有一个 **HIREDATE** 落在 90 天的窗口内（在之前的 90 天内），它就是 2010 年 12 月 17 日。如果将 2010 年 12 月 17 日获聘的员工的 **SAL** 同 2012 年 2 月 20 日获聘员工的 **SAL** 相加（因为要查找的是在当前 **HIREDATE** 之前 90 天内的 **HIREDATE**），结果将为 2400，这就是 **HIREDATE** 为 2012 年 2 月 20 日的行的最终总和。

知道工作原理后，在 **WHERE** 子句中使用筛选器返回每个 **HIREDATE** 及其之前 90 天内的 **HIREDATE**。

```

select e.hiredate,
       e.sal,
       d.sal sal_to_sum,
       d.hiredate within_90_days
  from emp e, emp d
 where d.hiredate
       between e.hiredate-90 and e.hiredate
 order by 1
HIREDATE      SAL  SAL_TO_SUM WITHIN_90_DAYS
-----

```

17-DEC-2010	800	800	17-DEC-2010
20-FEB-2011	1600	800	17-DEC-2010
20-FEB-2011	1600	1600	20-FEB-2011
22-FEB-2011	1250	800	17-DEC-2010
22-FEB-2011	1250	1600	20-FEB-2011
22-FEB-2011	1250	1250	22-FEB-2011
02-APR-2011	2975	1600	20-FEB-2011
02-APR-2011	2975	1250	22-FEB-2011
02-APR-2011	2975	2975	02-APR-2011
01-MAY-2011	2850	1600	20-FEB-2011
01-MAY-2011	2850	1250	22-FEB-2011
01-MAY-2011	2850	2975	02-APR-2011
01-MAY-2011	2850	2850	01-MAY-2011
09-JUN-2011	2450	2975	02-APR-2011
09-JUN-2011	2450	2850	01-MAY-2011
09-JUN-2011	2450	2450	09-JUN-2011
08-SEP-2011	1500	1500	08-SEP-2011
28-SEP-2011	1250	1500	08-SEP-2011
28-SEP-2011	1250	1250	28-SEP-2011
17-NOV-2011	5000	1500	08-SEP-2011
17-NOV-2011	5000	1250	28-SEP-2011
17-NOV-2011	5000	5000	17-NOV-2011
03-DEC-2011	950	1500	08-SEP-2011
03-DEC-2011	950	1250	28-SEP-2011
03-DEC-2011	950	5000	17-NOV-2011
03-DEC-2011	950	950	03-DEC-2011
03-DEC-2011	950	3000	03-DEC-2011
03-DEC-2011	3000	1500	08-SEP-2011
03-DEC-2011	3000	1250	28-SEP-2011
03-DEC-2011	3000	5000	17-NOV-2011
03-DEC-2011	3000	950	03-DEC-2011
03-DEC-2011	3000	3000	03-DEC-2011
23-JAN-2012	1300	5000	17-NOV-2011
23-JAN-2012	1300	950	03-DEC-2011
23-JAN-2012	1300	3000	03-DEC-2011
23-JAN-2012	1300	1300	23-JAN-2012
09-DEC-2012	3000	3000	09-DEC-2012
12-JAN-2013	1100	3000	09-DEC-2012
12-JAN-2013	1100	1100	12-JAN-2013

知道要在移动汇总窗口中包含哪些 **SAL** 值后，只需使用聚合函数 **SUM** 来生成更明确的结果集。


```

select e.hiredate,
       e.sal,
       sum(d.sal) as spending_pattern
  from emp e, emp d
 where d.hiredate
        between e.hiredate-90 and e.hiredate
 group by e.hiredate,e.sal
 order by 1

```

如果将前一个查询的结果集与这个查询（解决方案中使用的查询）的结果集进行比较，你将发现它们是一样的。

```

select e.hiredate,
       e.sal,
       (select sum(sal) from emp d
        where d.hiredate between e.hiredate-90
                               and e.hiredate) as spending_pattern
  from emp e
 order by 1

```

HIREDATE	SAL	SPENDING_PATTERN
-----	-----	-----
17-DEC-2010	800	800
20-FEB-2011	1600	2400
22-FEB-2011	1250	3650
02-APR-2011	2975	5825
01-MAY-2011	2850	8675
09-JUN-2011	2450	8275
08-SEP-2011	1500	1500
28-SEP-2011	1250	2750
17-NOV-2011	5000	7750
03-DEC-2011	950	11700
03-DEC-2011	3000	11700
23-JAN-2012	1300	10250
09-DEC-2012	3000	3000
12-JAN-2013	1100	4100

12.20 转置包含小计的结果集

1. 问题

你想制作一张包含小计的报表，再对其进行转置，以生成可读性更强的报表。例如，你被要求制作一张报表，其中列出了每个部门、每个部门的管理者以及各个管理者下属的员工的薪水总和。另外，你还想返回两种小计：每个部门由管理者管理的员工的薪水总和；结果集中的所有薪水之和（部门小计之和）。当前，你已经有如下报表：

DEPTNO	MGR	SAL
10	7782	1300
10	7839	2450
10		3750
20	7566	6000
20	7788	1100
20	7839	2975
20	7902	800
20		10875
30	7698	6550
30	7839	2850
30		9400
		24025

但要提供一张可读性更强的报表，因此要将上面的结果集转换成下面这样，让报表的含义更清晰。

MGR	DEPT10	DEPT20	DEPT30	TOTAL
7566	0	6000	0	
7698	0	0	6550	
7782	1300	0	0	
7788	0	1100	0	
7839	2450	2975	2850	
7902	0	800	0	

3750	10875	9400	24025
------	-------	------	-------

12. 解决方案

首先，使用 **GROUP BY** 扩展 **ROLLUP** 来生成小计。然后，执行经典转置操作（使用聚合和 **CASE** 表达式）来生成所需的报表列。使用函数 **GROUPING** 可以轻松地确定哪些值为小计（由 **ROLLUP** 生成的）。根据你使用的 **RDBMS** 对 **NULL** 值的排序方式，可能需要在解决方案中添加 **ORDER BY**，让最终的输出类似于前面的结果集。

DB2 和 Oracle

使用 **GROUP BY** 扩展 **ROLLUP** 来生成小计，然后使用 **CASE** 表达式将数据转换为可读性更强的报表。

```

1 select mgr,
2         sum(case deptno when 10 then sal else 0 end) dept10,
3         sum(case deptno when 20 then sal else 0 end) dept20,
4         sum(case deptno when 30 then sal else 0 end) dept30,
5         sum(case flag when '11' then sal else null end) total
6   from (
7 select deptno,mgr,sum(sal) sal,
8        cast(grouping(deptno) as char(1))||
9        cast(grouping(mgr) as char(1)) flag
10  from emp
11 where mgr is not null
12 group by rollup(deptno,mgr)
13        ) x
14 group by mgr

```

SQL Server

使用 **GROUP BY** 扩展 **ROLLUP** 来生成小计，然后使用 **CASE** 表达式将数据转换为可读性更强的报表。

```

1 select mgr,
2         sum(case deptno when 10 then sal else 0 end) dept10,
3         sum(case deptno when 20 then sal else 0 end) dept20,
4         sum(case deptno when 30 then sal else 0 end) dept30,
5         sum(case flag when '11' then sal else null end) total
6   from (
7 select deptno,mgr,sum(sal) sal,
8         cast(grouping(deptno) as char(1))+
9         cast(grouping(mgr) as char(1)) flag
10  from emp
11 where mgr is not null
12 group by deptno,mgr with rollup
13        ) x
14 group by mgr

```

PostgreSQL

使用 **GROUP BY** 扩展 **ROLLUP** 来生成小计，然后使用 **CASE** 表达式将数据转换为可读性更强的报表。

```

1 select mgr,
2         sum(case deptno when 10 then sal else 0 end) dept10,
3         sum(case deptno when 20 then sal else 0 end) dept20,
4         sum(case deptno when 30 then sal else 0 end) dept30,
5         sum(case flag when '11' then sal else null end) total
6   from (
7 select deptno,mgr,sum(sal) sal,
8         concat(cast (grouping(deptno) as char(1)),
9         cast(grouping(mgr) as char(1))) flag
10  from emp
11 where mgr is not null
12 group by rollup (deptno,mgr)
13        ) x
14 group by mgr

```

MySQL

使用 **GROUP BY** 扩展 **ROLLUP** 来生成小计，然后使用 **CASE** 表达式将数据转换为可读性更强的报表。

```

1  select mgr,
2      sum(case deptno when 10 then sal else 0 end) dept10,
3      sum(case deptno when 20 then sal else 0 end) dept20,
4      sum(case deptno when 30 then sal else 0 end) dept30,
5      sum(case flag when '11' then sal else null end) total
6  from (
7      select deptno,mgr,sum(sal) sal,
8          concat( cast(grouping(deptno) as char(1)) ,
9              cast(grouping(mgr) as char(1))) flag
10     from emp
11    where mgr is not null
12   group by deptno,mgr with rollup
13          ) x
14   group by mgr

```

13. 讨论

这里提供的解决方案基本相同，唯一的差别是拼接字符串和指定 **GROUPING** 的方式。鉴于这些解决方案非常相似，接下来将只讨论 SQL Server 解决方案，目标是突出中间结果集，但这些讨论也适用于 DB2 和 Oracle 解决方案。

首先，生成一个结果集，其中包含各个部门的每位管理者下属的员工的薪水总和，旨在呈现特定管理者下属的特定部门员工的收入情况。例如，下面的查询让你能够对 KING 下属的 10 号部门和 30 号部门的员工的薪水总和进行比较。

```

select deptno,mgr,sum(sal) sal
  from emp
 where mgr is not null
  group by mgr,deptno
 order by 1,2

```

DEPTNO	MGR	SAL
10	7782	1300
10	7839	2450

20	7566	6000
20	7788	1100
20	7839	2975
20	7902	800
30	7698	6550
30	7839	2850

然后，使用 **GROUP BY** 扩展 **ROLLUP** 来生成每个部门的薪水小计和全部员工的薪水总计（计算这些值，只考虑有管理者管理的员工）。

```

select deptno,mgr,sum(sal) sal
  from emp
 where mgr is not null
 group by deptno,mgr with rollup

```

DEPTNO	MGR	SAL
10	7782	1300
10	7839	2450
10		3750
20	7566	6000
20	7788	1100
20	7839	2975
20	7902	800
20		10875
30	7698	6550
30	7839	2850
30		9400
		24025

生成各种小计后，需要确定哪些是 **ROLLUP** 生成的，哪些是 **GROUP BY** 生成的。为此，使用函数 **GROUPING** 生成位映射（bitmap），以标出常规聚合生成的小计。

```

select deptno,mgr,sum(sal) sal,
       cast(grouping(deptno) as char(1))+
       cast(grouping(mgr) as char(1)) flag
  from emp
 where mgr is not null

```

group by deptno,mgr with rollup			
DEPTNO	MGR	SAL	FLAG
10	7782	1300	00
10	7839	2450	00
10		3750	01
20	7566	6000	00
20	7788	1100	00
20	7839	2975	00
20	7902	800	00
20		10875	01
30	7698	6550	00
30	7839	2850	00
30		9400	01
		24025	11

在上述结果集中，**FLAG** 值为 00 的行包含的是常规聚合结果。**FLAG** 值为 01 的行包含的是 **ROLLUP** 聚合不同部门的 **SAL** 得到的结果。（这是因为在 **ROLLUP** 中，先列出的是 **DEPTNO**。如果改变排列顺序，比如使用 **GROUP BY MGR,DEPTNO WITH ROLLUP**，那么结果将截然不同。）**FLAG** 值为 11 的行包含的是 **ROLLUP** 聚合所有行中的 **SAL** 得到的结果。

至此，“万事俱备，只欠东风”，只需使用 **CASE** 表达式就可制作出漂亮的报表。这里的目标是制作一张报表，显示每位管理者在每个部门管理的所有员工的薪水总和。对于特定的管理者，如果他在特定部门没有下属，就返回 0；否则，就返回该部门的所有下属的薪水总和。另外，你还想添加一个 **TOTAL** 列，用于显示所有员工的薪水总和。下面的解决方案可满足上述所有要求。

```
select mgr,
       sum(case deptno when 10 then sal else 0 end) dept10,
       sum(case deptno when 20 then sal else 0 end) dept20,
       sum(case deptno when 30 then sal else 0 end) dept30,
       sum(case flag when '11' then sal else null end) total
```

```

from (
select deptno,mgr,sum(sal) sal,
       cast(grouping(deptno) as char(1))+
       cast(grouping(mgr) as char(1)) flag
  from emp
 where mgr is not null
 group by deptno,mgr with rollup
        ) x
 group by mgr
 order by coalesce(mgr,9999)

```

MGR	DEPT10	DEPT20	DEPT30	TOTAL
-----	-----	-----	-----	-----
7566	0	6000	0	
7698	0	0	6550	
7782	1300	0	0	
7788	0	1100	0	
7839	2450	2975	2850	
7902	0	800	0	
	3750	10875	9400	24025

12.21 小结

数据库用于存储数据，但我们迟早需要从中检索数据并将其在其他地方呈现出来。本章的实例演示了各种对数据进行整形（重新设置格式）以满足用户需求的重要方式。这些方法不仅能够以所需的形式将数据提供给用户，在数据库所有者需要创建数据仓库时，也将发挥重要作用。

等你支持企业用户的经验更为丰富后，将能够得心应手地扩展本章的实例，以更优雅的方式呈现数据。

第 13 章 分层查询

数据中可能存在层次关系，本章介绍表达这种关系的实例。对于层次数据，相比于对其进行存储，对其进行检索并以层次方式呈现出来通常更难。

几年前，MySQL 引入了递归式 CTE，现在大多数 RDBMS 支持这种功能。因此，使用递归式 CTE 已成为编写分层查询的标准方法。本章将大量使用这项功能，以提供将数据的层次结构呈现出来的解决方案。

介绍实例前，先来看看 EMP 表中 EMPNO 和 MGR 之间的层次关系。

```
select empno,mgr
  from emp
 order by 2
```

EMPNO	MGR
7788	7566
7902	7566
7499	7698
7521	7698
7900	7698
7844	7698
7654	7698
7934	7782
7876	7788
7566	7839
7782	7839
7698	7839
7369	7902
7839	

如果仔细观察，你将发现每个 MGR 值都是一个 EMPNO，这意味着 EMP 表中的每位管理者也同样是员工，且未被存储在其他地

方。**MGR** 和 **EMPNO** 之间为父子关系，因为 **EMPNO** 对应的 **MGR** 值是它的直接父节点。（对于特定的员工，其管理者之上可能还有管理者，而这些管理者之上也有管理者，以此类推，形成 n 层层次结构。）对于没有管理者的员工，其 **MGR** 值为 **NULL**。

13.1 呈现父子关系

1. 问题

你想在返回子记录中数据的同时，返回父记录中的信息。例如，你想显示每位员工的名字以及其管理者的名字。换言之，你想返回如下结果集。

```
EMPS_AND_MGRS
-----
FORD works for JONES
SCOTT works for JONES
JAMES works for BLAKE
TURNER works for BLAKE
MARTIN works for BLAKE
WARD works for BLAKE
ALLEN works for BLAKE
MILLER works for CLARK
ADAMS works for SCOTT
CLARK works for KING
BLAKE works for KING
JONES works for KING
SMITH works for FORD
```

2. 解决方案

基于 **MGR** 和 **EMPNO** 相等自连接 **EMP** 表，以找出每位员工的管理者的名字。然后，使用 **RDBMS** 提供的字符串拼接函数生成所需的字符串。

DB2、Oracle 和 PostgreSQL

自连接 **EMP** 表，然后使用表示拼接运算符的双竖线（**||**）。

```
1 select a.ename || ' works for ' || b.ename as emps_and_mgrs
2   from emp a, emp b
3  where a.mgr = b.empno
```

MySQL

自连接 **EMP** 表，然后使用拼接函数 **CONCAT**。

```
1 select concat(a.ename, ' works for ',b.ename) as emps_and_mgrs
2   from emp a, emp b
3  where a.mgr = b.empno
```

SQL Server

自连接 **EMP** 表，然后使用表示拼接运算符的加号（+）。

```
1 select a.ename + ' works for ' + b.ename as emps_and_mgrs
2   from emp a, emp b
3  where a.mgr = b.empno
```

13. 讨论

所有解决方案的实现都几乎相同，唯一不同的是字符串拼接方法，因此讨论一种解决方案后，其他解决方案都不言自明了。

关键是基于 **MGR** 和 **EMPNO** 相等来自连接 **EMP** 表。首先，将 **EMP** 表连接到自身以生成笛卡儿积（下面只显示了笛卡儿积返回值的一部分）。

```
select a.empno, b.empno
   from emp a, emp b
```

EMPNO	MGR
-------	-----

7369	7369
7369	7499
7369	7521
7369	7566
7369	7654
7369	7698
7369	7782
7369	7788
7369	7839
7369	7844
7369	7876
7369	7900
7369	7902
7369	7934
7499	7369
7499	7499
7499	7521
7499	7566
7499	7654
7499	7698
7499	7782
7499	7788
7499	7839
7499	7844
7499	7876
7499	7900
7499	7902
7499	7934

如你所见，通过生成笛卡儿积，可以返回每种可能的 **EMPNO/EMPNO** 组合。（使得好像 **EMPNO** 为 7369 的员工的经理是表中所有员工的经理者。）

然后，对结果进行筛选，只返回每位员工及其经理者的 **EMPNO**。为此，可以基于 **MGR** 和 **EMPNO** 相等进行连接。

<pre> 1 select a.empno, b.empno mgr 2 from emp a, emp b 3 where a.mgr = b.empno </pre>	
EMPNO	MGR

7902	7566
7788	7566
7900	7698
7844	7698
7654	7698
7521	7698
7499	7698
7934	7782
7876	7788
7782	7839
7698	7839
7566	7839
7369	7902

有了每位员工及其管理者的 **EMPNO** 后，就可以返回管理者的名字了，为此只需选择 **B.ENAME**（而不是 **B.EMPNO**）。如果还不明白其中的工作原理，则可以将自连接替换为标量子查询，以帮助理解。

```
select a.ename,
       (select b.ename
        from emp b
        where b.empno = a.mgr) as mgr
  from emp a
```

ENAME	MGR
-----	-----
SMITH	FORD
ALLEN	BLAKE
WARD	BLAKE
JONES	KING
MARTIN	BLAKE
BLAKE	KING
CLARK	KING
SCOTT	JONES
KING	
TURNER	BLAKE
ADAMS	SCOTT
JAMES	BLAKE
FORD	JONES
MILLER	CLARK

这个标量子查询版本与自连接版本等效，但存在细微的差别：在标量子查询版本的结果集中，包含表示员工 KING 的行，在自连接版本的结果集中则没有这一行。你可能会问，这是为什么呢？别忘了，NULL 与任何值都不相等，包括 NULL 本身。在自连接解决方案中，其连接基于 EMPNO 和 MGR 相等，这将把 MGR 值为 NULL 的员工排除在外。在使用自连接时，如果要返回员工 KING，则必须使用外连接，如下面的两种查询所示。在这两种查询中，第一种查询使用的是 ANSI 外连接语法，第二种查询则使用了 Oracle 外连接语法。这两种查询的输出结果相同，如第二种查询后面的结果所示。

```
/* ANSI */
select a.ename, b.ename mgr
  from emp a left join emp b
    on (a.mgr = b.empno)
```

```
/* Oracle */
select a.ename, b.ename mgr
  from emp a, emp b
 where a.mgr = b.empno (+)
```

ENAME	MGR
-----	-----
FORD	JONES
SCOTT	JONES
JAMES	BLAKE
TURNER	BLAKE
MARTIN	BLAKE
WARD	BLAKE
ALLEN	BLAKE
MILLER	CLARK
ADAMS	SCOTT
CLARK	KING
BLAKE	KING
JONES	KING
SMITH	FORD
KING	

13.2 呈现子-父-祖父关系

1. 问题

员工 CLARK 是 KING 的下属，要表示这种关系，可以使用上一节中的解决方案。如果员工 CLARK 还是另一位员工的管理者，那么该如何表示这种关系呢？请看下面的查询。

```
select ename,empno,mgr
  from emp
 where ename in ('KING','CLARK','MILLER')
```

ENAME	EMPNO	MGR
CLARK	7782	7839
KING	7839	
MILLER	7934	7782

如你所见，员工 MILLER 是 CLARK 的下属，而 CLARK 是 KING 的下属。你要呈现从 MILLER 到 KING 的完整层次结构。换言之，你想返回如下结果集。

```
LEAF__BRANCH__ROOT
-----
MILLER-->CLARK-->KING
```

然而，上一节使用的单次自连接方法无法呈现上述完整关系。虽然可以编写执行两次自连接的查询，但使用遍历层次结构的通用方法更佳。

2. 解决方案

本实例不同于上一个实例，因为它要呈现的关系包含 3 层。

Oracle 提供了遍历树型数据的功能，如果你使用的 RDBMS 没有提供这种功能，则可以使用 CTE 来解决这个问题。

DB2 和 SQL Server

使用递归式 WITH 找出 MILLER 的管理者 CLARK，再找出 CLARK 的管理者 KING。下面的解决方案使用的是 SQL Server 字符串拼接运算符 +。

```
1  with x (tree,mgr,depth)
2  as (
3  select cast(ename as varchar(100)),
4         mgr, 0
5  from emp
6  where ename = 'MILLER'
7  union all
8  select cast(x.tree+'-->' + e.ename as varchar(100)),
9         e.mgr, x.depth+1
10 from emp e, x
11 where x.mgr = e.empno
12 )
13 select tree leaf__branch__root
14 from x
15 where depth = 2
```

只要修改拼接运算符，就可以将该解决方案用于其他数据库。换言之，用于 DB2 时，可以将拼接运算符改为 ||。

MySQL 和 PostgreSQL

MySQL 和 PostgreSQL 解决方案与上述解决方案类似，只是需要添加关键字 RECURSIVE。

```
1  with recursive x (tree,mgr,depth)
2  as (
3  select cast(ename as varchar(100)),
4         mgr, 0
5  from emp
```

```

6  where ename = 'MILLER'
7  union all
8  select cast(concat(x.tree,'-->',emp.ename) as char(100)),
9         e.mgr, x.depth+1
10 from emp e, x
11 where x.mgr = e.empno
12 )
13 select tree leaf__branch__root
14 from x
15 where depth = 2

```

Oracle

使用函数 `SYS_CONNECT_BY_PATH` 返回 MILLER、MILLER 的管理者 CLARK 以及 CLARK 的管理者 KING，并使用 `CONNECT BY` 子句遍历树。

```

1  select ltrim(
2         sys_connect_by_path(ename,'-->'),
3         '-->') leaf__branch__root
4  from emp
5  where level = 3
6  start with ename = 'MILLER'
7  connect by prior mgr = empno

```

13. 讨论

DB2、SQL Server、PostgreSQL 和 MySQL

该解决方案的做法是从叶子节点开始，向上遍历到根节点。（尝试沿相反的方向遍历，这是一个不错的练习。）**UNION ALL** 的上半部分会找出表示员工 MILLER 的行（叶子节点），而下半部分会找出 MILLER 的管理者以及这位管理者的管理者（查找管理者的管理者的过程不断重复，直到找到最大的管理者，即根节点）。**DEPTH** 的初始值为 0，而每次

往上查找管理者时，其值都自动加 1。当你执行递归查询时，DB2 将自动调整 DEPTH 的值。



有关递归式 **WITH** 子句的深入而有趣的介绍，请参阅 Jonathan Gennick 撰写的文章“Understanding the WITH Clause”。

接下来，**UNION ALL** 的第二个查询会将递归视图 **X** 连接到 **EMP** 表，以定义父子关系。在这个查询中，使用的是 SQL Server 拼接运算符。

```
with x (tree,mgr,depth)
  as (
select cast(ename as varchar(100)),
      mgr, 0
  from emp
 where ename = 'MILLER'
 union all
select cast(x.tree+'-->' + e.ename as varchar(100)),
      e.mgr, x.depth+1
  from emp e, x
 where x.mgr = e.empno
 )
select tree leaf__branch__root
  from x
```

TREE	DEPTH
MILLER	0
CLARK	1
KING	2

至此，问题的核心部分已经解决（从 **MILLER** 开始从下往上遍历，以返回完整的层次关系），余下的只有格式设置了。由于树遍历是递归式的，因此只需将 **EMP** 表中的当前 **ENAME** 与前一个 **ENAME** 拼接起来，这将生成如下结果集。

```

with x (tree,mgr,depth)
  as (
select cast(ename as varchar(100)),
      mgr, 0
  from emp
 where ename = 'MILLER'
 union all
select cast(x.tree+'-->' +e.ename as varchar(100)),
      e.mgr, x.depth+1
  from emp e, x
 where x.mgr = e.empno
 )
select depth, tree
  from x

DEPTH TREE
-----
0 MILLER
1 MILLER-->CLARK
2 MILLER-->CLARK-->KING

```

最后，只留下层次结构中的最后一行。完成这种任务的方法有多种，该解决方案根据 **DEPTH** 来判断是否到达了根节点。（显然，如果 **KING** 之上还有管理者，那么将需要修改基于 **DEPTH** 的筛选器。有关不使用这种筛选器的更通用的解决方案，请参阅下一节。）

Oracle

在 Oracle 解决方案中，**CONNECT BY** 子句完成了所有的工作。无须使用任何连接，你就可以从 **MILLER** 遍历到 **KING**。**CONNECT BY** 子句中的表达式定义了数据之间的关系以及遍历树的方式。

```

select ename
  from emp
 start with ename = 'MILLER'
connect by prior mgr = empno

```

```
ENAME
-----
MILLER
CLARK
KING
```

关键字 **PRIOR** 能够访问层次结构中前一条记录的值。因此，对于任何给定的 **EMPNO**，都可以使用 **PRIOR MGR** 来访问这位员工的管理者的编号。对于子句 **CONNECT BY PRIOR MGR = EMPNO**，可以将其视为一种父子连接。



要更详细地了解 **CONNECT BY** 及其在分层查询中的用途，请参阅“Hierarchical Queries in Oracle”。

至此，你成功地呈现了从 **MILLER** 到 **KING** 的整个层次结构，从而解决了问题的很大一部分，余下的只是格式设置了。为此，使用函数 **SYS_CONNECT_BY_PATH** 将每个 **ENAME** 附加到前一个 **ENAME** 后面。

```
select sys_connect_by_path(ename,'-->') tree
      from emp
      start with ename = 'MILLER'
 connect by prior mgr = empno

TREE
-----
-->MILLER
-->MILLER-->CLARK
-->MILLER-->CLARK-->KING
```

由于你只想返回完整的层次结构，因此可以根据伪列 **LEVEL** 进行筛选（有关更通用的方法，请参阅下一节）。

```
select sys_connect_by_path(ename,'-->') tree
      from emp
     where level = 3
      start with ename = 'MILLER'
```

```
connect by prior mgr = empno
```

```
TREE
```

```
-----
```

```
-->MILLER-->CLARK-->KING
```

最后，使用函数 **LTRIM** 从结果集中删除开头的 -->。

13.3 创建基于表的分层视图

1. 问题

你想返回一个结果集，将整张表的层次结构呈现出来。在 **EMP** 表中，员工 **KING** 之上没有管理者，因此 **KING** 为根节点。你想从 **KING** 开始，显示其所有下属以及这些下属的所有下属。换言之，你想返回如下结果集。

EMP_TREE	

KING	
KING - BLAKE	
KING - BLAKE - ALLEN	
KING - BLAKE - JAMES	
KING - BLAKE - MARTIN	
KING - BLAKE - TURNER	
KING - BLAKE - WARD	
KING - CLARK	
KING - CLARK - MILLER	
KING - JONES	
KING - JONES - FORD	
KING - JONES - FORD - SMITH	
KING - JONES - SCOTT	
KING - JONES - SCOTT - ADAMS	

2. 解决方案

DB2、PostgreSQL 和 SQL Server

使用递归式 **WITH** 子句生成一个层次结构，其中包含 **KING** 及其管理的所有员工。下面展示的是 **DB2** 解决方案（使用的是 **DB2** 拼接运算符 **||**）。要将该解决方案用于 **SQL Server** 和 **MySQL**，只需在其中分别使用拼接运算符 **+** 和拼接函数

CONCAT。

```
1  with x (ename,empno)
2    as (
3  select cast(ename as varchar(100)),empno
4    from emp
5   where mgr is null
6   union all
7  select cast(x.ename||' - '||e.ename as varchar(100)),
8         e.empno
9    from emp e, x
10   where e.mgr = x.empno
11  )
12  select ename as emp_tree
13    from x
14   order by 1
```

MySQL

在 MySQL 中，还需添加关键字 RECURSIVE。

```
1  with recursive x (ename,empno)
2    as (
3  select cast(ename as varchar(100)),empno
4    from emp
5   where mgr is null
6   union all
7  select cast(concat(x.ename,' - ',e.ename) as varchar(100)),
8         e.empno
9    from emp e, x
10   where e.mgr = x.empno
11  )
12  select ename as emp_tree
13    from x
14   order by 1
```

Oracle

使用函数 CONNECT BY 定义层次结构，并使用函数 SYS_CONNECT_BY_PATH 设置输出的格式。

```
1 select ltrim(  
2         sys_connect_by_path(ename,' - '),  
3         ' - ') emp_tree  
4   from emp  
5   start with mgr is null  
6  connect by prior empno=mgr  
7   order by 1
```

相比于上一节的解决方案，该解决方案的不同之处在于没有使用基于伪列 **LEVEL** 的筛选器。删除这个筛选器后，将显示所有可能的树（符合条件 **PRIOR EMPNO=MGR** 的树）。

13. 讨论

DB2、MySQL、PostgreSQL 和 SQL Server

首先，找出根行（员工 **KING**），如递归视图 **X** 中 **UNION ALL** 的上半部分所示。然后，找出 **KING** 的下属以及这些下属的下属，这是通过将递归视图 **X** 连接到 **EMP** 表实现的。递归将不断进行，直到返回所有的员工为止。在没有设置格式的情况下，递归视图 **X** 返回的结果集如下所示。

```
with x (ename,empno)  
  as (  
select cast(ename as varchar(100)),empno  
  from emp  
  where mgr is null  
  union all  
select cast(e.ename as varchar(100)),e.empno  
  from emp e, x  
  where e.mgr = x.empno  
  )  
  select ename emp_tree  
  from x  
  
EMP_TREE
```

```
-----  
KING  
JONES  
SCOTT  
ADAMS  
FORD  
SMITH  
BLAKE  
ALLEN  
WARD  
MARTIN  
TURNER  
JAMES  
CLARK  
MILLER
```

层次结构中的所有行都被返回了，这很有用，但如果不进一步设置格式，将无法看出哪些员工是管理者。通过将每位员工同其管理者拼接起来，可以返回更有意义的输出。要生成前面要求的输出，只需在递归视图 X 中 **UNION ALL** 下半部分的 **SELECT** 子句中使用如下代码。

```
cast(x.ename+', '+e.ename as varchar(100))
```

解决这类问题时，**WITH** 子句很有用，因为即便层次结构发生变化（比如叶子节点变成分支节点），也无须对查询做任何修改。

Oracle

CONNECT BY 子句返回了层次结构中的行。**START WITH** 子句定义了根行。如果你删除该解决方案中的 **SYS_CONNECT_BY_PATH** 部分，然后再执行它，将发现它返回了正确的行（这很有用），但没有通过格式设置将行之间的关系呈现出来。

```
select ename emp_tree
```

```

    from emp
    start with mgr is null
connect by prior empno = mgr

```

EMP_TREE

KING
 JONES
 SCOTT
 ADAMS
 FORD
 SMITH
 BLAKE
 ALLEN
 WARD
 MARTIN
 TURNER
 JAMES
 CLARK
 MILLER

使用伪列 **LEVEL** 和函数 **LPAD**，可以将层次结构更清晰地呈现出来，进而明白为什么 **SYS_CONNECT_BY_PATH** 会返回前面呈现的输出。

```

select lpad(' ',2*level,' ')||ename emp_tree
    from emp
    start with mgr is null
connect by prior empno = mgr

```

EMP_TREE

..KING
JONES
SCOTT
ADAMS
FORD
SMITH
BLAKE
ALLEN
WARD
MARTIN

```
.....TURNER
.....JAMES
....CLARK
.....MILLER
```

上述输出通过缩进呈现了上下属关系，进而指出了哪些员工是管理者。例如，KING 没有上级管理者，JONES 的上级管理者是 KING，SCOTT 的上级管理者是 JONES，而 ADAMS 的上级管理者是 SCOTT。

如果查看使用 `SYS_CONNECT_BY_PATH` 时解决方案返回的行，你就会明白是 `SYS_CONNECT_BY_PATH` 将层次结构中的关系呈现了出来。找到特定的员工后，你将看到他所有的上级管理者。

```
KING
KING - JONES
KING - JONES - SCOTT
KING - JONES - SCOTT - ADAMS
```

13.4 找出给定父行的所有子行

1. 问题

你想找出 JONES 的所有下属，包括直接下属和间接下属（JONES 的下属的下属）。下面列出了 JONES 及其所有下属。

```
ENAME
-----
JONES
SCOTT
ADAMS
FORD
SMITH
```

2. 解决方案

能够定位到树的顶部或底部很有用。在本解决方案中，不需要特殊的格式设置。这里的目标很简单，就是返回 JONES 下属的所有员工，包括 JONES 自己。这种查询充分展示了递归式 SQL 扩展（比如 Oracle 的 **CONNECT BY** 以及 SQL Server 和 DB2 的 **WITH** 子句）的威力。

DB2、PostgreSQL 和 SQL Server

使用递归式 **WITH** 子句找出是 JONES 下属的所有员工。从 JONES 开始，在 **UNION ALL** 上半部分的查询中指定 **WHERE ENAME = JONES**。

```
1  with x (ename,empno)
2    as (
```

```
3 select ename,empno
4   from emp
5  where ename = 'JONES'
6  union all
7 select e.ename, e.empno
8   from emp e, x
9  where x.empno = e.mgr
10 )
11 select ename
12   from x
```

Oracle

使用 **CONNECT BY** 子句并指定 **START WITH ENAME = JONES**，以找出 JONES 下属的所有员工。

```
1 select ename
2   from emp
3  start with ename = 'JONES'
4 connect by prior empno = mgr
```

13. 讨论

DB2、MySQL、PostgreSQL 和 SQL Server

递归式 **WITH** 子句使得这个问题解决起来比较容易。**WITH** 子句的第一部分（**UNION ALL** 的上半部分）返回了所有与 JONES 相关的行。你需要返回 **ENAME**（用于显示员工的名字）和 **EMPNO**（以便使用它来指定连接条件）。**UNION ALL** 的下半部分递归地将 **EMP.MGR** 连接到了 **X.EMPNO**。这个连接条件将被不断应用，直到穷尽整个结果集。

Oracle

START WITH 子句会将 JONES 指定为根节点。**CONNECT BY**

子句中的条件驱动着树遍历过程不断进行下去，直到这个条件不再成立才会停止。

13.5 确定叶子节点、分支节点和根节点

1. 问题

你想判断给定的行是哪种类型的节点：叶子节点、分支节点还是根节点。在本实例中，叶子节点指的是不是管理者的员工，分支节点指的是自己是管理者且还有上级管理者的员工，而根节点指的是没有上级管理者的员工。对于层次结构中的每一行，你都要返回 1（TRUE）或 0（FALSE），以指出其状态。你希望返回的结果集如下所示。

ENAME	IS_LEAF	IS_BRANCH	IS_ROOT
KING	0	0	1
JONES	0	1	0
SCOTT	0	1	0
FORD	0	1	0
CLARK	0	1	0
BLAKE	0	1	0
ADAMS	1	0	0
MILLER	1	0	0
JAMES	1	0	0
TURNER	1	0	0
ALLEN	1	0	0
WARD	1	0	0
MARTIN	1	0	0
SMITH	1	0	0

2. 解决方案

EMP 表建立的是树型层次结构，而不是递归层次结构，因为根节点的 **MGR** 为 **NULL**，认识到这一点很重要。如果 **EMP** 建立的是递归层次结构，那么根节点将指向自己（也就是说，员工 **KING** 的 **MGR** 值将为他的 **EMPNO**）。我们发现，指向自

已是不合常理的，因此将根节点的 MGR 设置为了 NULL。使用 Oracle 的 CONNECT BY 以及 DB2 和 SQL Server 的 WITH 子句时，你会发现树型层次结构比递归层次结构更容易处理，效率也更高。使用 CONNECT BY 或 WITH 处理递归层次结构时务必小心，因为最终编写的 SQL 代码可能包含循环。如果处理递归层次结构时出现问题，那么请务必检查这种循环。

DB2、PostgreSQL、MySQL 和 SQL Server

使用 3 个标量子查询在每个节点类型列中返回正确的“布尔”值（1 或 0）。

```
1 select e.ename,
2       (select sign(count(*)) from emp d
3        where 0 =
4              (select count(*) from emp f
5               where f.mgr = e.empno)) as is_leaf,
6       (select sign(count(*)) from emp d
7        where d.mgr = e.empno
8        and e.mgr is not null) as is_branch,
9       (select sign(count(*)) from emp d
10        where d.empno = e.empno
11        and d.mgr is null) as is_root
12 from emp e
13 order by 4 desc,3 desc
```

Oracle

上述子查询解决方案也适用于 Oracle。如果你使用的是 Oracle Database 10g 以前的版本，那么也应该使用这种解决方案。下面的解决方案使用了 Oracle 提供的内置函数 CONNECT_BY_ROOT 和 CONNECT_BY_ISLEAF（这些内置函数是 Oracle Database 10g 引入的）来找出根行和叶子行。

```
1 select ename,
2        connect_by_isleaf is_leaf,
```

```

3      (select count(*) from emp e
4      where e.mgr = emp.empno
5      and emp.mgr is not null
6      and rownum = 1) is_branch,
7      decode(ename,connect_by_root(ename),1,0) is_root
8  from emp
9  start with mgr is null
10 connect by prior empno = mgr
11 order by 4 desc, 3 desc

```

13. 讨论

DB2、PostgreSQL、MySQL 和 SQL Server

DB2、PostgreSQL、MySQL 和 SQL Server 该解决方案根据本节“问题”部分指定的规则来确定叶子节点、分支节点和根节点。首先，判断员工是否是叶子节点。如果员工不是管理者（没有下属），那他就是叶子节点。下面显示了第一个标量子查询（IS_LEAF）。

```

select e.ename,
       (select sign(count(*)) from emp d
        where 0 =
          (select count(*) from emp f
           where f.mgr = e.empno)) as is_leaf
  from emp e
 order by 2 desc

```

ENAME	IS_LEAF
-----	-----
SMITH	1
ALLEN	1
WARD	1
ADAMS	1
TURNER	1
MARTIN	1
JAMES	1
MILLER	1

JONES	0
BLAKE	0
CLARK	0
FORD	0
SCOTT	0
KING	0

由于 IS_LEAF 的输出应该是 0 或 1，因此必须确定 COUNT(*) 操作的符号，否则对于叶子节点，返回的将不是 1 而是 14。也可以使用只有一行的表来计数，因为要返回的是 0 或 1。

<pre> select e.ename, (select count(*) from t1 d where not exists (select null from emp f where f.mgr = e.empno)) as is_leaf from emp e order by 2 desc </pre>	
ENAME	IS_LEAF
-----	-----
SMITH	1
ALLEN	1
WARD	1
ADAMS	1
TURNER	1
MARTIN	1
JAMES	1
MILLER	1
JONES	0
BLAKE	0
CLARK	0
FORD	0
SCOTT	0
KING	0

然后，找出分支节点。如果员工是管理者（有下属），同时还有上级管理者，那他就是分支节点。下面显示了标量子查询 IS_BRANCH 的结果。

```

select e.ename,
       (select sign(count(*)) from emp d
        where d.mgr = e.empno
          and e.mgr is not null) as is_branch
  from emp e
 order by 2 desc

```

ENAME	IS_BRANCH
JONES	1
BLAKE	1
SCOTT	1
CLARK	1
FORD	1
SMITH	0
TURNER	0
MILLER	0
JAMES	0
ADAMS	0
KING	0
ALLEN	0
MARTIN	0
WARD	0

同样，必须确定 **COUNT(*)** 操作返回的结果的符号，否则对于分支节点，返回的值可能大于 1。与标量子查询 **IS_LEAF** 一样，也可以使用一张只有一行的表，以避免使用 **SIGN**。下面的解决方案使用了 **T1** 表。

```

select e.ename,
       (select count(*) from t1 t
        where exists (
          select null from emp f
          where f.mgr = e.empno
            and e.mgr is not null)) as is_branch
  from emp e
 order by 2 desc

```

ENAME	IS_BRANCH
-------	-----------

JONES	1
BLAKE	1
SCOTT	1
CLARK	1
FORD	1
SMITH	0
TURNER	0
MILLER	0
JAMES	0
ADAMS	0
KING	0
ALLEN	0
MARTIN	0
WARD	0

最后，找出根节点。根节点指的是没有上级管理者且自己是管理者的员工。在 **EMP** 表中，只有 **KING** 是根节点。下面显示了标量子查询 **IS_ROOT**。

```
select e.ename,
       (select sign(count(*)) from emp d
        where d.empno = e.empno
          and d.mgr is null) as is_root
  from emp e
 order by 2 desc
```

ENAME	IS_ROOT
-----	-----
KING	1
SMITH	0
ALLEN	0
WARD	0
JONES	0
TURNER	0
JAMES	0
MILLER	0
FORD	0
ADAMS	0
MARTIN	0
BLAKE	0
CLARK	0

由于 **EMP** 是一张小型表，只有 14 行数据，因此很容易看出只有员工 **KING** 是根节点。有鉴于此，获取 **COUNT(*)** 操作结果的符号并非必不可少。在可能存在多个根节点的情况下，可以使用 **SIGN**，也可以在标量子查询中使用只有一行的表，这在前面讨论 **IS_BRANCH** 和 **IS_LEAF** 时介绍过。

Oracle

如果使用的是 Oracle Database 10g 之前的 Oracle 版本，那么可以参阅前面针对其他 RDBMS 的讨论，因为它们的解决方案无须修改就适用于 Oracle。如果使用的是 Oracle Database 10g 或更高的版本，那么你可能想利用两个内置函数，即 **CONNECT_BY_ROOT** 和 **CONNECT_BY_ISLEAF**，它们让你能够轻松地找出根节点和叶子节点。在本书撰写之时，如果想使用 **CONNECT_BY_ROOT** 和 **CONNECT_BY_ISLEAF**，则必须在 SQL 语句中使用 **CONNECT BY**。首先，使用 **CONNECT_BY_ISLEAF** 找出叶子节点。

```
select ename,
       connect_by_isleaf is_leaf
  from emp
 start with mgr is null
 connect by prior empno = mgr
 order by 2 desc
```

ENAME	IS_LEAF
ADAMS	1
SMITH	1
ALLEN	1
TURNER	1
MARTIN	1
WARD	1
JAMES	1
MILLER	1

KING	0
JONES	0
BLAKE	0
CLARK	0
FORD	0
SCOTT	0

然后，使用标量子查询找出分支节点。分支节点指的是自己为管理者且还有上级管理者的员工。

```

select ename,
       (select count(*) from emp e
        where e.mgr = emp.empno
          and emp.mgr is not null
          and rownum = 1) is_branch
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc

```

ENAME	IS_BRANCH
-----	-----
JONES	1
SCOTT	1
BLAKE	1
FORD	1
CLARK	1
KING	0
MARTIN	0
MILLER	0
JAMES	0
TURNER	0
WARD	0
ADAMS	0
ALLEN	0
SMITH	0

基于 ROWNUM 的筛选器必不可少，它能确保返回的值为 1 或 0。

接下来，使用函数 CONNECT_BY_ROOT 找出根节点。该解决

方案会确定行的 **ENAME**，并将其与查询返回的所有行进行比较。如果有匹配的行，那么该行就是根节点。

```
select ename,
       decode(ename,connect_by_root(ename),1,0) is_root
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc
```

ENAME	IS_ROOT
-----	-----
KING	1
JONES	0
SCOTT	0
ADAMS	0
FORD	0
SMITH	0
BLAKE	0
ALLEN	0
WARD	0
MARTIN	0
TURNER	0
JAMES	0
CLARK	0
MILLER	0

函数 **SYS_CONNECT_BY_PATH** 会从根节点开始，将层次结构中的关系呈现出来。

```
select ename,
       ltrim(sys_connect_by_path(ename,','),',') path
  from emp
 start with mgr is null
connect by prior empno=mgr
```

ENAME	PATH
-----	-----
KING	KING
JONES	KING,JONES
SCOTT	KING,JONES,SCOTT
ADAMS	KING,JONES,SCOTT,ADAMS

FORD	KING, JONES, FORD
SMITH	KING, JONES, FORD, SMITH
BLAKE	KING, BLAKE
ALLEN	KING, BLAKE, ALLEN
WARD	KING, BLAKE, WARD
MARTIN	KING, BLAKE, MARTIN
TURNER	KING, BLAKE, TURNER
JAMES	KING, BLAKE, JAMES
CLARK	KING, CLARK
MILLER	KING, CLARK, MILLER

为了获得根行，只需提取 PATH 中的第一个 ENAME。

```
select ename,
       substr(root,1,instr(root,',')-1) root
  from (
select ename,
       ltrim(sys_connect_by_path(ename,','),'') root
  from emp
 start with mgr is null
 connect by prior empno=mgr
        )
```

ENAME	ROOT
-----	-----
KING	
JONES	KING
SCOTT	KING
ADAMS	KING
FORD	KING
SMITH	KING
BLAKE	KING
ALLEN	KING
WARD	KING
MARTIN	KING
TURNER	KING
JAMES	KING
CLARK	KING
MILLER	KING

最后，根据 ROOT 列标记结果。如果该列的值为 NULL，那么它就是根行。

13.6 小结

当前，所有 RDBMS 都支持 CTE，这使得编写适用于所有 RDBMS 的分层查询比以前容易得多。鉴于很多数据包含层次关系（虽然这种关系可能不是有意包含的），而查询时必须考虑这种关系，因此 CTE 的面世是个巨大的进步。

第 14 章 杂项

本章包含不适合放在其他章的查询，原因可能是查询所属的章篇幅太长，也可能是查询解决的问题太有趣。本章是非常有趣的一章，其中的解决方案你可能会用到，也可能用不到，但这些查询都很有趣，而我们很想在这本书中介绍它们。

14.1 使用SQL Server运算符PIVOT创建交叉报表

1. 问题

你想创建一张交叉报表，将结果集中的行转换为列。你熟悉传统的转置方法，但想尝试点儿新鲜的东西。具体地说，你想在不使用 **CASE** 表达式和连接的情况下，返回如下结果集。

DEPT_10	DEPT_20	DEPT_30	DEPT_40
3	5	6	0

2. 解决方案

使用 **PIVOT** 运算符生成所需的结果集，而不使用 **CASE** 表达式或额外的连接操作。

```
1 select [10] as dept_10,  
2        [20] as dept_20,  
3        [30] as dept_30,  
4        [40] as dept_40  
5 from (select deptno, empno from emp) driver  
6 pivot (  
7     count(driver.empno)  
8     for driver.deptno in ( [10],[20],[30],[40] )  
9 ) as empPivot
```

3. 讨论

乍一看，**PIVOT** 运算符有点儿怪怪的，但在上述解决方案

中，它执行的操作与下面你所熟悉的转置查询相同。

```
select sum(case deptno when 10 then 1 else 0 end) as dept_10,
       sum(case deptno when 20 then 1 else 0 end) as dept_20,
       sum(case deptno when 30 then 1 else 0 end) as dept_30,
       sum(case deptno when 40 then 1 else 0 end) as dept_40
from emp
```

DEPT_10	DEPT_20	DEPT_30	DEPT_40
3	5	6	0

大致知道发生的情况后，下面来详细介绍 **PIVOT** 运算符所做的工作。在上述解决方案中，第 5 行是一个名为 **DRIVER** 的内嵌视图。

```
from (select deptno, empno from emp) driver
```

之所以使用别名 **DRIVER**，是因为这个内嵌视图（表表达式）返回的行将作为输入直接提供给 **PIVOT** 操作。**PIVOT** 运算符会评估第 8 行的 **FOR** 列表中列出的项，将行转换为列。

```
for driver.deptno in ( [10],[20],[30],[40] )
```

评估过程如下。

- a. 如果有 **DEPTNO** 值为 10 的行，就对这些行执行指定的聚合操作（**COUNT(DRIVER.EMPNO)**）。
- b. 对 **DEPTNO** 为 20、30 和 40 的行重复上述过程。

第 8 行的方括号内列出的项不仅指定了聚合操作针对的行所包含的值，还将在结果集中用作列名（不包括方括号）。上述解决方案的 **SELECT** 子句引用了 **FOR** 列表中指定的项，并给它们指定了别名。如果没有给 **FOR** 列表中的项指定别名，那么列名将与 **FOR** 列表项相同（不包括方括号）。

有趣的是，由于内嵌视图 **DRIVER** 只是一个内嵌视图而已，因此你可能想在这里使用更复杂的 SQL。假设你要修改结果集，在其中将实际的部门名称用作列名。下面列出了 **DEPT** 表中包含的行。

select * from dept		
DEPTNO	DNAME	LOC

10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

你想使用 **PIVOT** 返回如下结果集。

ACCOUNTING	RESEARCH	SALES	OPERATIONS

3	5	6	0

由于内嵌视图 **DRIVER** 可以是任何合法的表表达式，因此可以将 **EMP** 表连接到 **DEPT** 表，再使用 **PIVOT** 来评估返回的行。下面的查询将返回所需的结果集。

select [ACCOUNTING] as ACCOUNTING,	
[SALES] as SALES,	
[RESEARCH] as RESEARCH,	
[OPERATIONS] as OPERATIONS	
from (
select d.dname, e.empno	
from emp e,dept d	
where e.deptno=d.deptno	
) driver	
pivot (
count(driver.empno)	
for driver.dname in ([ACCOUNTING],[SALES],[RESEARCH],	
[OPERATIONS])	
) as empPivot	

如你所见，在转置结果集方面，**PIVOT** 提供了一种有趣的快捷途径。无论你是否要使用它来替代传统的转置方法，工具箱中多了一个工具都是一件好事。

14.2 使用SQL Server运算符UNPIVOT逆转置交叉报表

1. 问题

你有一个转置得到的结果集（或事实表），你想对其进行逆转置。例如，当前结果集为 1 行 4 列，而你想将其转换为 4 行 2 列。对于上一节实例生成的结果集：

ACCOUNTING	RESEARCH	SALES	OPERATIONS
3	5	6	0

你想将其转换成下面这样。

DNAME	CNT
ACCOUNTING	3
RESEARCH	5
SALES	6
OPERATIONS	0

2. 解决方案

SQL Server 不仅提供了 PIVOT，也提供了 UNPIVOT。要逆转置结果集，只需将它作为驱动表（driver），并让 UNPIVOT 运算符来完成所有工作。你需要做的全部工作就是指定列名。

```
1 select DNAME, CNT
2   from (
3     select [ACCOUNTING] as ACCOUNTING,
4            [SALES]      as SALES,
```

```

5          [RESEARCH]    as RESEARCH,
6          [OPERATIONS] as OPERATIONS
7      from (
8          select d.dname, e.empno
9              from emp e,dept d
10             where e.deptno=d.deptno
11
12         ) driver
13     pivot (
14         count(driver.empno)
15         for driver.dname in ([ACCOUNTING],[SALES],[RESEARCH],
16 [OPERATIONS])
17     ) as empPivot
18 unpivot (cnt for dname in
19 (ACCOUNTING,SALES,RESEARCH,OPERATIONS)
20 ) as un_pivot

```

阅读本实例前，应该先阅读上一个实例，因为内嵌视图 **NEW_DRIVER** 与上一个实例的解决方案代码完全相同（如果看不懂，请参阅上一个实例）。第 3~16 行的代码在前面介绍过，这里唯一的新语法是使用 **UNPIVOT** 的第 18 行。

UNPIVOT 命令会查看 **NEW_DRIVER** 返回的结果集，并对其中的每一行和每一列进行评估。例如，**UNPIVOT** 运算符会评估 **NEW_DRIVER** 返回的列名，当它遇到 **ACCOUNTING** 时，会将列名 **ACCOUNTING** 转换为（位于 **DNAME** 列的）行值，它还会获取 **NEW_DRIVER** 中的 **ACCOUNTING** 列值（3），并将其作为 **ACCOUNTING** 行中的一个值（**CNT** 列值）返回。对于 **FOR** 列表中指定的每一项，**UNPIVOT** 都会执行上述操作，并将每一项作为一行返回。

新的结果集更紧凑，为 2 列（**DNAME** 列和 **CNT** 列）4 行。

```

select DNAME, CNT
  from (
    select [ACCOUNTING] as ACCOUNTING,
           [SALES] as SALES,

```

```

        [RESEARCH] as RESEARCH,
        [OPERATIONS] as OPERATIONS
    from (
        select d.dname, e.empno
        from emp e,dept d
        where e.deptno=d.deptno

        ) driver
    pivot (
        count(driver.empno)
        for driver.dname in ( [ACCOUNTING],[SALES],[RESEARCH],
[OPERATIONS] )
        ) as empPivot
    ) new_driver
    unpivot (cnt for dname in (ACCOUNTING,SALES,RESEARCH,OPERATIONS)
    ) as un_pivot

```

DNAME	CNT
ACCOUNTING	3
RESEARCH	5
SALES	6
OPERATIONS	0

14.3 使用Oracle子句MODEL转置结果集

1. 问题

与 14.1 节的实例一样，你想找到一种替代方案，用于替代本书前面介绍过的传统转置方法。你想尝试使用 Oracle 子句 **MODEL**。不同于 SQL Server 运算符 **PIVOT**，Oracle 子句 **MODEL** 并非是为转置结果集而生的。事实上，将 **MODEL** 子句用于转置属于滥用，这也不是 **MODEL** 子句的初衷。尽管如此，**MODEL** 子句还是提供了一种解决常见问题的有趣方法。在这里，你想对下面的结果集进行转换。

```
select deptno, count(*) cnt
  from emp
 group by deptno
```

DEPTNO	CNT
10	3
20	5
30	6

结果如下。

D10	D20	D30
3	5	6

2. 解决方案

就像传统转置方法那样，在 **MODEL** 子句中使用聚合函数和 **CASE** 表达式。主要区别在于，在 **MODEL** 子句中，我们会使用数组来存储聚合结果，并将结果集中的数组返回。

```

select max(d10) d10,
       max(d20) d20,
       max(d30) d30
  from (
select d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
  measures(deptno, cnt d10, cnt d20, cnt d30)
  rules(
    d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0
  end,
    d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0
  end,
    d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0
  end
  )
  )

```

13. 讨论

MODEL 子句在 Oracle 工具箱中新增了一个功能强大的 SQL 工具。着手使用 **MODEL** 后，你将发现它提供了很多有用的功能，比如迭代，以数组形式访问行值，在结果集中更新插入（**upsert**）行以及构建参考模型。本实例并没有利用 **MODEL** 子句提供的上述任何功能，但从多个角度考虑问题并以出乎意料的方式使用功能是一件好事。（这至少有助于了解在什么情况下，某些功能比其他功能更有用。）

为弄明白上述解决方案，先来看看 **FROM** 子句中的内嵌视图。这个内嵌视图会计算 **EMP** 表中每个部门的员工数量，结果如下所示。

```

select deptno, count(*) cnt
  from emp
 group by deptno

```

DEPTNO	CNT
10	3
20	5
30	6

这个结果集被提供给 **MODEL** 进行处理。在 **MODEL** 子句中，有 3 个非常显眼的子句：**DIMENSION BY**、**MEASURES** 和 **RULES**。先来说说 **MEASURES**。

MEASURES 列表中的项就是我们为这个查询声明的数组。这个查询使用了 4 个数组：**DEPTNO**、**D10**、**D20** 和 **D30**。与 **SELECT** 列表中的列一样，**MEASURES** 列表中的数组也可以有别名。如你所见，这 4 个数组中有 3 个实际上是内嵌视图返回的 **CNT**。

如果 **MEASURES** 列表中包含了数组，那么 **DIMENSION BY** 子句指定的项就为数组索引。数组 **D10** 是 **CNT** 的别名。从前面显示的内嵌视图结果集可知，**CNT** 有 3 个值，即 3、5 和 6。当你创建数组 **CNT** 时，便创建了一个包含 3 个元素的数组，这 3 个元素分别是整数 3、5 和 6。那么如何逐一访问这个数组中的这些值呢？答案是使用数组索引。索引是在 **DIMENSION BY** 子句中定义的，其值分别为 10、20 和 30（来自前面显示的结果集）。因此，如下表达式的结果为 3，因为它访问的是数组 **D10** 中与 **DEPTNO 10** 对应的 **CNT** 值（3）：

d10[10]

由于数组 **D10**、**D20** 和 **D30** 中包含的都是来自 **CNT** 的值，因此它们的值相同。那么如何将合适的计数放入正确的数组呢？答案是使用 **RULES** 子句。从前面显示的内嵌视图返回的结果集可知，**DEPTNO** 的值分别为 10、20 和 30。在 **RULES**

子句中，包含 **CASE** 的表达式会分别检查 **DEPTNO** 数组中的每个值。

- 如果值为 10，就在 **D10[10]** 中存储与 **DEPTNO 10** 对应的 **CNT** 值，否则存储 0。
- 如果值为 20，就在 **D20[20]** 中存储与 **DEPTNO 20** 对应的 **CNT** 值，否则存储 0。
- 如果值为 30，就在 **D30[30]** 中存储与 **DEPTNO 30** 对应的 **CNT** 值，否则存储 0。

如果你感觉很迷惑，请不用担心，只需停止阅读，并执行至此讨论过的所有代码。下面显示了这些代码生成的结果集。在有些情况下，阅读正文后看看相关代码执行的任务，然后再回过头去阅读对你会有所帮助。执行代码后，就很容易理解结果为什么是那样的了。

```
select deptno, d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
  measures(deptno, cnt d10, cnt d20, cnt d30)
  rules(
    d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0
end,
    d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0 end,
    d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0 end
  )
```

DEPTNO	D10	D20	D30
10	3	0	0
20	0	5	0
30	0	0	6

如你所见，**RULES** 子句修改了每一个数组中的值。如果你还感到迷惑，只需再次执行这个查询，但将 **RULES** 子句中的表达式注释掉。

```

select deptno, d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
  measures(deptno, cnt d10, cnt d20, cnt d30)
  rules(
    /*
      d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0
end,
      d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0
end,
      d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0
end
    */
  )

```

DEPTNO	D10	D20	D30
10	3	3	3
20	5	5	5
30	6	6	6

上述输出清楚地表明，现在 **MODEL** 子句返回的结果集与内嵌视图相同，只是给 **COUNT** 操作指定了别名 **D10**、**D20** 和 **D30**。下面的查询证明了这一点。

```

select deptno, count(*) d10, count(*) d20, count(*) d30
  from emp
 group by deptno

```

DEPTNO	D10	D20	D30
10	3	3	3
20	5	5	5
30	6	6	6

因此，**MODEL** 子句所做的事情只是将 **DEPTNO** 和 **CNT** 值加入数组中，然后确保每个数组都只包含一个 **DEPTNO** 值对应的 **CNT** 值。至此，数组 **D10**、**D20** 和 **D30** 都只包含一个非零值，该非零值为相应 **DEPTNO** 值对应的 **CNT** 值。对结果集进

行转置后，余下的唯一工作就是使用聚合函数 **MAX**（也可以使用聚合函数 **MIN** 或 **SUM**，就本例而言，这不会有任何不同）让结果集只包含一行数据。

```
select max(d10) d10,
       max(d20) d20,
       max(d30) d30
  from (
select d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
  measures(deptno, cnt d10, cnt d20, cnt d30)
  rules(
    d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0
end,
    d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0
end,
    d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0
end
  )
 )
```

D10	D20	D30
-----	-----	-----
3	5	6

14.4 从不固定的位置提取子串

1. 问题

你有一个字符串字段，其中包含序列化的日志数据，你想对字符串进行分析，从中提取意义重大的信息。可惜这些信息在字符串中的位置并不固定。因此，要提取这些信息，必须利用这样一个事实：这些信息前后是一些独特的字符。例如，请看下面的字符串。

```
xxxxxabc[867]xxx[-]xxxx[5309]xxxxx
xxxxxtime:[11271978]favnum:[4]id:[Joe]xxxxx
call:[F_GET_ROWS()]b1:[ROSEWOOD...SIR]b2:[44400002]77.90xxxxx
film:[non_marked]qq:[unit]tailpipe:[withabanana?]80sxxxxx
```

你想提取包含在方括号内的值，进而返回如下结果集。

FIRST_VAL	SECOND_VAL	LAST_VAL
-----	-----	-----
867	-	5309
11271978	4	Joe
F_GET_ROWS()	ROSEWOOD...SIR	44400002
non_marked	unit	withabanana?

2. 解决方案

虽然不知道你感兴趣的值在字符串中的准确位置，但你知道它们位于方括号（[]）内，且这样的值有 3 个。使用 Oracle 内置函数 **INSTR** 确定方括号的位置，然后使用内置函数 **SUBSTR** 从字符串中提取所需的值。视图 **V** 包含要分析的字符串，其定义如下所示（这里使用它只是为了提高可读性）。

```

create view V
as
select 'xxxxxabc[867]xxx[-]xxxx[5309]xxxxx' msg
  from dual
  union all
  select 'xxxxxtime:[11271978]favnum:[4]id:[Joe]xxxxx' msg
  from dual
  union all
  select 'call:[F_GET_ROWS()]b1:[ROSEWOOD...SIR]b2:
[44400002]77.90xxxxxx' msg
  from dual
  union all
  select 'film:[non_marked]qq:[unit]tailpipe:
[withabanana?]80sxxxxxx' msg
  from dual

1 select substr(msg,
2       instr(msg,['',1,1)+1,
3       instr(msg,']',1,1)-instr(msg,['',1,1)-1) first_val,
4       substr(msg,
5       instr(msg,['',1,2)+1,
6       instr(msg,']',1,2)-instr(msg,['',1,2)-1) second_val,
7       substr(msg,
8       instr(msg,['',-1,1)+1,
9       instr(msg,']',-1,1)-instr(msg,['',-1,1)-1) last_val
10  from V

```

13. 讨论

使用 Oracle 内置函数 **INSTR**，可以非常容易地解决这个问题。由于你知道要获取的值包含在方括号（**[]**）内，同时知道有 3 对方括号，因此解决方案的第一步是，使用 **INSTR** 确定每个字符串中 **[]** 的位置。下面的示例会在每一行中显示左方括号和右方括号的位置。

```

select instr(msg,['',1,1) "1st_[",
       instr(msg,']',1,1) ""]_1st",
       instr(msg,['',1,2) "2nd_[",

```

instr(msg,']',1,2) "]"_2nd", instr(msg,'[',-1,1) "3rd_[" , instr(msg,']',-1,1) "]"_3rd" from V					
1st_[]_1st		2nd_[]_2nd		3rd_[]_3rd	
9	13	17	19	24	29
11	20	28	30	34	38
6	19	23	38	42	51
6	17	21	26	36	49

至此，最棘手的工作已经完成，余下的全部工作就是调用 **SUBSTR**，并根据这些位置从 **MSG** 中提取子串。你可能注意到了，在完整的解决方案中，对 **INSTR** 返回的值做了一些简单的算术运算，具体地说是 + 1 和 -1。这些运算必不可少，旨在确保最终的结果集中不包含左方括号（[）。在这里列出的解决方案中，没有对 **INSTR** 返回的值执行这些算术运算，因此提取的每个值开头都有一个左方括号。

select substr(msg, instr(msg,'[',1,1), instr(msg,']',1,1)-instr(msg,'[',1,1)) first_val, substr(msg, instr(msg,'[',1,2), instr(msg,']',1,2)-instr(msg,'[',1,2)) second_val, substr(msg, instr(msg,'[',-1,1), instr(msg,']',-1,1)-instr(msg,'[',-1,1)) last_val from V		
FIRST_VAL	SECOND_VAL	LAST_VAL
[867	[-	[5309
[11271978	[4	[Joe
[F_GET_ROWS()	[ROSEWOOD...SIR	[44400002
[non_marked	[unit	[withabanana?

从上面的结果集可知，左方括号没有被删除。你可能会想：给 **INSTR** 返回的值加上 1 好了，这样就能将开头的左方括号

删除了。但为什么要减去 1 呢？原因是在给 **INSTR** 返回的值加上 1 后，如果不减去 1，提取的值末尾将有一个右方括号。

```
select substr(msg,
              instr(msg,'[',1,1)+1,
              instr(msg,']',1,1)-instr(msg,'[',1,1)) first_val,
       substr(msg,
              instr(msg,'[',1,2)+1,
              instr(msg,']',1,2)-instr(msg,'[',1,2)) second_val,
       substr(msg,
              instr(msg,'[',-1,1)+1,
              instr(msg,']',-1,1)-instr(msg,'[',-1,1)) last_val
from V
```

FIRST_VAL	SECOND_VAL	LAST_VAL
-----	-----	-----
867]	-]	5309]
11271978]	4]	Joe]
F_GET_ROWS()]	ROSEWOOD...SIR]	44400002]
non_marked]	unit]	withabanana?]

现在你应该很清楚了：为了避免包括左方括号和右方括号，必须给起始索引加 1，并给终止索引减 1。

14.5 确定特定年份有多少天（另一种**Oracle**解决方案）

1. 问题

你想确定特定年份有多少天。



本实例为 9.2 节提供了一种替代解决方案，但只适用于 Oracle。

2. 解决方案

获取给定日期所属年份的最后一天，再使用函数 **TO_CHAR** 将该日期转换为一个 3 位数，这个数字指出了该日期是当前年份的第几天。

```
1 select 'Days in 2021: '||
2         to_char(add_months(trunc(sysdate,'y'),12)-1,'DDD')
3         as report
4   from dual
5 union all
6 select 'Days in 2020: '||
7         to_char(add_months(trunc(
8         to_date('01-SEP-2020'),'y'),12)-1,'DDD')
9   from dual
```

REPORT

```
-----
Days in 2021: 365
Days in 2020: 366
```

13. 讨论

首先，使用函数 **TRUNC** 返回给定日期所属年份的第一天。

```
select trunc(to_date('01-SEP-2020'),'y')
       from dual

TRUNC(TO_DA
-----
01-JAN-2020
```

然后，使用函数 **ADD_MONTHS** 给截断获得的日期加上 1 年（12 个月），再减去 1 天，得到给定日期所属年份的最后一天。

```
select add_months(
       trunc(to_date('01-SEP-2020'),'y'),
       12) before_subtraction,
       add_months(
       trunc(to_date('01-SEP-2020'),'y'),
       12)-1 after_subtraction
       from dual

BEFORE_SUBT AFTER_SUBTR
-----
01-JAN-2021 31-DEC-2020
```

最后，获得给定日期所属年份的最后一天后，使用 **TO_CHAR** 返回一个 3 位数，这个数字指出了该天是其所属年份的第几天（比如第 1 天、第 50 天等）。

```
select to_char(
       add_months(
       trunc(to_date('01-SEP-2020'),'y'),
       12)-1,'DDD') num_days_in_2020
       from dual

NUM
---
```


14.6 找出同时包含字母和数字的字符串

1. 问题

你有一个包含字母和数字数据的列，而你想返回这样的行，即其也包含字母和数字。换言之，如果该列值只包含字母或只包含数字，则不返回相应的行。返回的值应该同时包含字母和数字。请看下面的数据。

```
STRINGS
-----
1010 switch
333
3453430278
ClassSummary
findRow 55
threes
```

最终的结果集应该只包含那些同时有字母和数字的行。

```
STRINGS
-----
1010 switch
findRow 55
```

2. 解决方案

使用内置函数 **TRANSLATE** 将每个字母都转换为某种字符，并将每个数字都转换为另一种字符。然后，只保留同时包含前述两种字符的字符串。本解决方案使用的是 **Oracle** 语法，但 **DB2** 和 **PostgreSQL** 也支持 **TRANSLATE**，因此只需做简单修改，本解决方案就将适用于这两种 **RDBMS**。

```

with v as (
select 'ClassSummary' strings from dual union
select '3453430278'      from dual union
select 'findRow 55'      from dual union
select '1010 switch'     from dual union
select '333'             from dual union
select 'threes'          from dual
)
select strings
  from (
select strings,
translate(
strings,
'abcdefghijklmnopqrstuvwxyz0123456789',
rpad('#',26,'#')||rpad('*',10,'*')) translated
from v
) x
where instr(translated,'#') > 0
and instr(translated,'*') > 0

```



也可以不使用 **WITH** 子句，而使用内嵌视图或直接创建一个视图。

13. 讨论

函数 **TRANSLATE** 使得这个问题解决起来易如反掌。首先使用 **TRANSLATE** 将每个字母和数字都分别转换为井号（#）和星号（*）。内嵌视图返回的中间结果如下所示。

```

with v as (
select 'ClassSummary' strings from dual union
select '3453430278'      from dual union
select 'findRow 55'      from dual union
select '1010 switch'     from dual union
select '333'             from dual union
select 'threes'          from dual
)

```

```

select strings,
       translate(
         strings,
         'abcdefghijklmnopqrstuvwxyz0123456789',
         rpad('#',26,'#')||rpad('*',10,'*')) translated
  from v

```

STRINGS	TRANSLATED
1010 switch	**** #####
333	***
3453430278	*****
ClassSummary	C####S#####
findRow 55	####R## **
threes	#####

现在，只需保留那些至少包含一个 **#** 和一个 ***** 的行。为此，使用函数 **INSTR** 判断字符串中是否包含 **#** 和 *****。如果字符串中包含这两种字符，那么 **INSTR** 返回的值将大于 0。为清晰起见，下面显示了最终返回的字符串以及对其进行转换后得到的结果。

```

with v as (
select 'ClassSummary' strings from dual union
select '3453430278'          from dual union
select 'findRow 55'          from dual union
select '1010 switch'         from dual union
select '333'                 from dual union
select 'threes'              from dual
)
select strings, translated
  from (
select strings,
       translate(
         strings,
         'abcdefghijklmnopqrstuvwxyz0123456789',
         rpad('#',26,'#')||rpad('*',10,'*')) translated
  from v
)
 where instr(translated,'#') > 0
    and instr(translated,'*') > 0

```

STRINGS	TRANSLATED
-----	-----
1010 switch	**** #####
findRow 55	####R## **

14.7 在Oracle中将整数转换为其二进制表示

1. 问题

在 Oracle 系统中，你想将整数转换为其二进制表示。例如，你想返回 EMP 表中所有薪水的二进制表示，如下面的结果集所示。

ENAME	SAL	SAL_BINARY
SMITH	800	1100100000
ALLEN	1600	11001000000
WARD	1250	10011100010
JONES	2975	101110011111
MARTIN	1250	10011100010
BLAKE	2850	101100100010
CLARK	2450	100110010010
SCOTT	3000	101110111000
KING	5000	1001110001000
TURNER	1500	10111011100
ADAMS	1100	10001001100
JAMES	950	1110110110
FORD	3000	101110111000
MILLER	1300	10100010100

2. 解决方案

MODEL 子句提供了迭代以及以数组方式访问行值的功能，因此使用它来解决这个问题是一种自然而然的想法。（这里假设必须使用 SQL 来解决这个问题，如若不然，使用存储函数则更合适。）与本书的其他解决方案一样，即便找不到本解决方案的实际用途，其中介绍的技巧也很有用。**MODEL** 子句可以执行过程性任务，同时具备 SQL 基于集合的特征和强大威力，明白这一点很有用。因此，即便你认为自己绝不可能

在 SQL 中这样做，也没有关系。这里无意于建议你应该怎么做，不该怎么做，而只想让你专注于其中的技巧，以便在合适的情况下付诸应用。

下面的解决方案会返回 **EMP** 表中所有的 **ENAME** 和 **SAL**，同时在标量子查询中调用 **MODEL**。（从某种意义上说，可以将其作为 **EMP** 表中一个独立的函数，它像其他函数一样接受输入，启动处理过程，并返回一个值。）

```
1 select ename,
2       sal,
3       (
4       select bin
5       from dual
6       model
7       dimension by ( 0 attr )
8       measures ( sal num,
9                  cast(null as varchar2(30)) bin,
10                  '0123456789ABCDEF' hex
11              )
12       rules iterate (10000) until (num[0] <= 0) (
13       bin[0] =
14 substr(hex[cv()],mod(num[cv()],2)+1,1)||bin[cv()],
15       num[0] = trunc(num[cv()]/2)
16       ) sal_binary
17 from emp
```

13. 讨论

上面“解决方案”部分说过，使用存储函数来解决这个问题可能是更好的选择。实际上，本解决方案借鉴了一个函数，具体地说，它是根据 Oracle 员工 Tom Kyte 编写的一个名为 **TO_BASE** 的函数改编而成的。与本书其他你可能不会使用的解决方案一样，虽然你可能不会使用本解决方案，但它很好

地演示了 **MODEL** 子句的一些功能，比如迭代以及以数组方式访问行值。

为了帮助你理解本解决方案，来看看前述包含 **MODEL** 子句的子查询的演变形式。下面的代码与上述解决方案中的子查询几乎相同，唯一不同的是，它返回的是值 2 的二进制表示。

```
select bin
  from dual
 model
 dimension by ( 0 attr )
 measures ( 2 num,
            cast(null as varchar2(30)) bin,
            '0123456789ABCDEF' hex
          )
 rules iterate (10000) until (num[0] <= 0) (
   bin[0] = substr (hex[cv()],mod(num[cv()],2)+1,1) || bin[cv()],
   num[0] = trunc(num[cv()]/2)
 )

BIN
-----
10
```

在上述查询定义的 **RULES** 中，第一次迭代的结果与下面的查询相同。

```
select 2 start_val,
       '0123456789ABCDEF' hex,
       substr('0123456789ABCDEF',mod(2,2)+1,1) ||
       cast(null as varchar2(30)) bin,
       trunc(2/2) num
  from dual
```

START_VAL	HEX	BIN	NUM
2	0123456789ABCDEF	0	1

START_VAL 是要转换为二进制表示的数字，这里为 2；**BIN**

是对 0123456789ABCDEF（在本解决方案中为 HEX）执行子串提取操作的结果；NUM 用于确定是否要退出循环。

从上述结果集可知，执行循环的第一次迭代后，BIN 的值为 0，而 NUM 的值为 1。由于 NUM 不小于或等于 0，因此将再执行一次循环迭代。下面的 SQL 语句显示了这次迭代的结果。

```
select num start_val,
       substr('0123456789ABCDEF',mod(1,2)+1,1) || bin bin,
       trunc(1/2) num
  from (
select 2 start_val,
       '0123456789ABCDEF' hex,
       substr('0123456789ABCDEF',mod(2,2)+1,1) ||
       cast(null as varchar2(30)) bin,
       trunc(2/2) num
  from dual
 )
```

START_VAL	BIN	NUM
1	10	0

在这次循环迭代中，对 HEX 执行子串提取操作，结果为 1，再将前一个 BIN 值（0）附加到它后面。测试变量 NUM 的值为 0，因此这是最后一次迭代，返回值 10 是数字 2 的二进制表示。熟悉其中的工作原理后，可以将 MODEL 子句中的迭代删除，通过逐行遍历来弄明白规则是如何应用的。

```
select 2 orig_val, num, bin
  from dual
 model
 dimension by ( 0 attr )
 measures ( 2 num,
           cast(null as varchar2(30)) bin,
           '0123456789ABCDEF' hex
         )
 rules (
   bin[0] = substr (hex[cv()],mod(num[cv()],2)+1,1)||bin[cv()],
```



```
num[0] = trunc(num[cv()]/2),  
bin[1] = substr (hex[0],mod(num[0],2)+1,1)||bin[0],  
num[1] = trunc(num[0]/2)  
)
```

ORIG_VAL	NUM	BIN
2	1	0
2	0	10

14.8 对经过排名的结果集进行转置

1. 问题

你想对表中的值进行排名，然后将结果集转置为 3 列。这样做旨在分别显示前 3 名、接下来的 3 名以及其余各行记录。例如，你想根据 **SAL** 对 **EMP** 表中的员工进行排名，然后将结果转置为 3 列，以得到如下结果集。

TOP_3		NEXT_3		REST
-----		-----		-----
KING	(5000)	BLAKE	(2850)	TURNER (1500)
FORD	(3000)	CLARK	(2450)	MILLER (1300)
SCOTT	(3000)	ALLEN	(1600)	MARTIN (1250)
JONES	(2975)			WARD (1250)
				ADAMS (1100)
				JAMES (950)
				SMITH (800)

2. 解决方案

本解决方案的关键在于，先使用窗口函数 **DENSE_RANK OVER** 根据 **SAL** 对员工进行排名，并允许排名相同。使用 **DENSE_RANK OVER**，可以轻松地获悉最高的 3 个薪水值、接下来的 3 个薪水值以及其他薪水值。

然后，使用窗口函数 **ROW_NUMBER OVER** 分别对各个分组（薪水前 3 组、薪水第 4~6 组和其他薪水组）中的员工进行排名。接下来，只需执行传统的转置并利用 **RDBMS** 提供的内置字符串函数，就可以获得漂亮的结果。下面的解决方案使用的是 **Oracle** 语法，但现在所有的 **RDBMS** 都支持窗口函数，因此很容易对该解决方案进行修改，以用于其他

RDBMS。

```
1 select max(case grp when 1 then rpad(ename,6) ||
2           ' ('|| sal ||')' end) top_3,
3           max(case grp when 2 then rpad(ename,6) ||
4           ' ('|| sal ||')' end) next_3,
5           max(case grp when 3 then rpad(ename,6) ||
6           ' ('|| sal ||')' end) rest
7   from (
8 select ename,
9        sal,
10       rnk,
11       case when rnk <= 3 then 1
12            when rnk <= 6 then 2
13            else 3
14       end grp,
15       row_number()over (
16         partition by case when rnk <= 3 then 1
17                        when rnk <= 6 then 2
18                        else 3
19         end
20         order by sal desc, ename
21       ) grp_rnk
22   from (
23 select ename,
24        sal,
25        dense_rank()over(order by sal desc) rnk
26   from emp
27   ) x
28   ) y
29  group by grp_rnk
```

13. 讨论

使用窗口函数，只需编写少量的代码就可以完成大量的工作，本实例淋漓尽致地展示了这一点。上述解决方案看似复杂，但将其从内向外进行分解后，你会惊讶地发现它非常简单。下面先来执行内嵌视图 **x**。

```
select ename,
       sal,
       dense_rank()over(order by sal desc) rnk
from emp
```

ENAME	SAL	RNK
KING	5000	1
SCOTT	3000	2
FORD	3000	2
JONES	2975	3
BLAKE	2850	4
CLARK	2450	5
ALLEN	1600	6
TURNER	1500	7
MILLER	1300	8
WARD	1250	9
MARTIN	1250	9
ADAMS	1100	10
JAMES	950	11
SMITH	800	12

从上面的结果集可知，内嵌视图 **X** 根据 **SAL** 对员工进行排名，同时允许排名相同。（由于本解决方案使用的是 **DENSE_RANK** 而不是 **RANK**，因此排名可以相同且不跳跃。）

下一步是使用 **CASE** 表达式来评估 **DENSE_RANK** 返回的排名，以便对内嵌视图 **X** 返回的行进行分组。此外，使用窗口函数 **ROW_NUMBER OVER** 根据 **SAL** 对各组员工（这些分组是使用 **CASE** 表达式创建的）分别进行排名。所有这些都是在内嵌视图 **Y** 中进行的。

```
select ename,
       sal,
       rnk,
       case when rnk <= 3 then 1
            when rnk <= 6 then 2
            else 3
       end grp,
       row_number()over (
         partition by case when rnk <= 3 then 1
```

```

                when rnk <= 6 then 2
                else 3
            end
            order by sal desc, ename
        ) grp_rnk
    from (
select ename,
       sal,
       dense_rank()over(order by sal desc) rnk
    from emp
    ) x

```

ENAME	SAL	RNK	GRP	GRP_RNK
KING	5000	1	1	1
FORD	3000	2	1	2
SCOTT	3000	2	1	3
JONES	2975	3	1	4
BLAKE	2850	4	2	1
CLARK	2450	5	2	2
ALLEN	1600	6	2	3
TURNER	1500	7	3	1
MILLER	1300	8	3	2
MARTIN	1250	9	3	3
WARD	1250	9	3	4
ADAMS	1100	10	3	5
JAMES	950	11	3	6
SMITH	800	12	3	7

至此，查询已初具雏形，如果从内往外（从内嵌视图 **X** 开始）看，你将发现它并不复杂。到目前为止，上述查询返回了每位员工及其 **SAL**、**RNK**（员工的 **SAL** 在所有员工中的排名）、**GRP**（根据 **SAL** 指出员工属于哪个分组）和 **GRP_RANK**（员工的 **SAL** 在组内的排名）。

接下来，对 **ENAME** 执行传统转置，并使用 Oracle 拼接运算符 **||** 将 **SAL** 附加到 **ENAME** 后面，其中函数 **RPAD** 确保括号内的数字值是对齐的。最后，根据 **GRP_RNK** 进行分组，确保在结果集中显示每位员工。最终的结果集如下所示。

```

select max(case grp when 1 then rpad(ename,6) ||
           ' ('|| sal ||')' end) top_3,
       max(case grp when 2 then rpad(ename,6) ||
           ' ('|| sal ||')' end) next_3,
       max(case grp when 3 then rpad(ename,6) ||
           ' ('|| sal ||')' end) rest
  from (
select ename,
       sal,
       rnk,
       case when rnk <= 3 then 1
            when rnk <= 6 then 2
            else 3
       end grp,
       row_number()over (
         partition by case when rnk <= 3 then 1
                        when rnk <= 6 then 2
                        else 3
         end
         Order by sal desc, ename
       ) grp_rnk
  from (
select ename,
       sal,
       dense_rank()over(order by sal desc) rnk
  from emp
  ) x
  ) y
group by grp_rnk

```

TOP_3		NEXT_3		REST	

KING	(5000)	BLAKE	(2850)	TURNER	(1500)
FORD	(3000)	CLARK	(2450)	MILLER	(1300)
SCOTT	(3000)	ALLEN	(1600)	MARTIN	(1250)
JONES	(2975)			WARD	(1250)
ADAMS	(1100)				
				JAMES	(950)
				SMITH	(800)

如果仔细研究前面介绍的各个查询，你将发现只访问了 **EMP** 表一次。使用窗口函数的优点之一是，只需访问数据一次就能完成大量的工作。无须使用自连接，也不用创建临时表，

只需获取所需的行，然后让窗口函数来完成其余的工作。在内嵌视图 **X** 中访问 **EMP** 表，然后以所需的方式传递结果集即可。创建这种报表时，如果只需访问数据库表一次，则意味着性能将得到极大的提升。

14.9 给经过两次转置的结果集添加列标题

1. 问题

你想合并两个结果集，并将它们转置为两列。另外，你还想给各组添加列“标题”。例如，你有两张表，分别包含公司中从事不同领域（比如研究领域和应用领域）开发工作的员工的信息。

```
select * from it_research
```

```
DEPTNO ENAME
```

```
-----
```

```
100 HOPKINS
100 JONES
100 TONEY
200 MORALES
200 P.WHITAKER
200 MARCIANO
200 ROBINSON
300 LACY
300 WRIGHT
300 J.TAYLOR
```

```
select * from it_apps
```

```
DEPTNO ENAME
```

```
-----
```

```
400 CORRALES
400 MAYWEATHER
400 CASTILLO
400 MARQUEZ
400 MOSLEY
500 GATTI
500 CALZAGHE
600 LAMOTTA
600 HAGLER
600 HEARNS
600 FRAZIER
```


700 GUINN
700 JUDAH
700 MARGARITO

你想制作一张报表，在两列中分别列出这两张表中的员工。你还想显示返回每个部门的编号（DEPTNO），并在部门编号后面显示相应部门所有员工的名字（ENAME）。换言之，你想返回如下结果集。

RESEARCH	APPS
-----	-----
100	400
JONES	MAYWEATHER
TONEY	CASTILLO
HOPKINS	MARQUEZ
200	MOSLEY
P.WHITAKER	CORRALES
MARCIANO	500
ROBINSON	CALZAGHE
MORALES	GATTI
300	600
WRIGHT	HAGLER
J.TAYLOR	HEARNS
LACY	FRAZIER
	LAMOTTA
	700
	JUDAH
	MARGARITO
	GUINN

12. 解决方案

从很大程度上说，本解决方案只需执行合并和转置，然后再做一些细微的调整：在每个部门的员工 ENMAE 前面加上相应的 DEPTNO。这里使用笛卡儿积来为每个 DEPTNO 多生成一行数据，以便显示部门的所有员工以及 DEPTNO 对应的行。本解决方案使用的是 Oracle 语法，但由于 DB2 支持计算移动

窗口的窗口函数（框架子句），因此很容易对本解决方案进行转换，使其适用于 DB2。由于只在本实例中使用了 **IT_RESEARCH** 表和 **IT_APPS** 表，因此下面的解决方案中也会包含创建这些表的语句。

```
create table IT_research (deptno number, ename varchar2(20))
```

```
insert into IT_research values (100,'HOPKINS')
insert into IT_research values (100,'JONES')
insert into IT_research values (100,'TONEY')
insert into IT_research values (200,'MORALES')
insert into IT_research values (200,'P.WHITAKER')
insert into IT_research values (200,'MARCIANO')
insert into IT_research values (200,'ROBINSON')
insert into IT_research values (300,'LACY')
insert into IT_research values (300,'WRIGHT')
insert into IT_research values (300,'J.TAYLOR')
```

```
create table IT_apps (deptno number, ename varchar2(20))
```

```
insert into IT_apps values (400,'CORRALES')
insert into IT_apps values (400,'MAYWEATHER')
insert into IT_apps values (400,'CASTILLO')
insert into IT_apps values (400,'MARQUEZ')
insert into IT_apps values (400,'MOSLEY')
insert into IT_apps values (500,'GATTI')
insert into IT_apps values (500,'CALZAGHE')
insert into IT_apps values (600,'LAMOTTA')
insert into IT_apps values (600,'HAGLER')
insert into IT_apps values (600,'HEARNS')
insert into IT_apps values (600,'FRAZIER')
insert into IT_apps values (700,'GUINN')
insert into IT_apps values (700,'JUDAH')
insert into IT_apps values (700,'MARGARITO')
```

```
1 select max(decode(flag2,0,it_dept)) research,
2         max(decode(flag2,1,it_dept)) apps
3   from (
4 select sum(flag1)over(partition by flag2
5                        order by flag1,rownum) flag,
6        it_dept, flag2
```

```

7   from (
8 select 1 flag1, 0 flag2,
9        decode(rn,1,to_char(deptno),' '||ename) it_dept
10  from (
11 select x.*, y.id,
12        row_number()over(partition by x.deptno order by y.id) rn
13  from (
14 select deptno,
15        ename,
16        count(*)over(partition by deptno) cnt
17  from it_research
18       ) x,
19       (select level id from dual connect by level <= 2) y
20       )
21  where rn <= cnt+1
22 union all
23 select 1 flag1, 1 flag2,
24        decode(rn,1,to_char(deptno),' '||ename) it_dept
25  from (
26 select x.*, y.id,
27        row_number()over(partition by x.deptno order by y.id) rn
28  from (
29 select deptno,
30        ename,
31        count(*)over(partition by deptno) cnt
32  from it_apps
33       ) x,
34       (select level id from dual connect by level <= 2) y
35       )
36  where rn <= cnt+1
37       ) tmp1
38       ) tmp2
39  group by flag

```

13. 讨论

与其他众多的数据仓库 / 报表型查询一样，上述解决方案看起来也非常复杂，但经过分解后你将发现，它只执行了合并和转置，并使用笛卡儿积进行了调整。为了分解这个查询，

可以先查看 **UNION ALL** 的组成部分，然后再查看合并和转置操作。先从 **UNION ALL** 的后半部分开始。

```
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
    ) x,
    (select level id from dual connect by level <= 2) y
    ) z
 where rn <= cnt+1
```

FLAG1	FLAG2	IT_DEPT
1	1	400
1	1	MAYWEATHER
1	1	CASTILLO
1	1	MARQUEZ
1	1	MOSLEY
1	1	CORRALES
1	1	500
1	1	CALZAGHE
1	1	GATTI
1	1	600
1	1	HAGLER
1	1	HEARNS
1	1	FRAZIER
1	1	LAMOTTA
1	1	700
1	1	JUDAH
1	1	MARGARITO
1	1	GUINN

来看看这个结果集是如何生成的。如果将以上查询分解为最简单的组成部分，那么我们将得到内嵌视图 **X**，它会返回 **IT_APPS** 表中每位员工的 **DEPTNO** 和 **ENAME**，以及每个

DEPTNO 的员工数量。

```
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
from it_apps
```

DEPTNO	ENAME	CNT
400	CORRALES	5
400	MAYWEATHER	5
400	CASTILLO	5
400	MARQUEZ	5
400	MOSLEY	5
500	GATTI	2
500	CALZAGHE	2
600	LAMOTTA	4
600	HAGLER	4
600	HEARNS	4
600	FRAZIER	4
700	GUINN	3
700	JUDAH	3
700	MARGARITO	3

下一步是使用 **CONNECT BY** 从 **DUAL** 表中返回两行数据，并生成这两行数据与内嵌视图 **X** 返回行的笛卡儿积。

```
select *
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
    ) x,
    (select level id from dual connect by level <= 2) y
order by 2
```

DEPTNO	ENAME	CNT	ID
500	CALZAGHE	2	1
500	CALZAGHE	2	2
400	CASTILLO	5	1

400	CASTILLO	5	2
400	CORRALES	5	1
400	CORRALES	5	2
600	FRAZIER	4	1
600	FRAZIER	4	2
500	GATTI	2	1
500	GATTI	2	2
700	GUINN	3	1
700	GUINN	3	2
600	HAGLER	4	1
600	HAGLER	4	2
600	HEARNS	4	1
600	HEARNS	4	2
700	JUDAH	3	1
700	JUDAH	3	2
600	LAMOTTA	4	1
600	LAMOTTA	4	2
700	MARGARITO	3	1
700	MARGARITO	3	2
400	MARQUEZ	5	1
400	MARQUEZ	5	2
400	MAYWEATHER	5	1
400	MAYWEATHER	5	2
400	MOSLEY	5	1
400	MOSLEY	5	2

从上面的结果集可知，通过生成与内嵌视图 Y 的笛卡儿积，内嵌视图 X 中的每一行都被返回了两次。为什么要生成笛卡儿积呢？稍后你会明白的。下一步是在当前结果集中，根据 DEPTNO 对员工进行分组，再根据 ID 值对员工进行组内排名（笛卡儿积返回的 ID 值为 1 或 2）。这个排名操作的结果如下述查询的输出所示。

```

select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
    ) x,

```

(select level id from dual connect by level <= 2) y				
DEPTNO	ENAME	CNT	ID	RN
400	CORRALES	5	1	1
400	MAYWEATHER	5	1	2
400	CASTILLO	5	1	3
400	MARQUEZ	5	1	4
400	MOSLEY	5	1	5
400	CORRALES	5	2	6
400	MOSLEY	5	2	7
400	MAYWEATHER	5	2	8
400	CASTILLO	5	2	9
400	MARQUEZ	5	2	10
500	GATTI	2	1	1
500	CALZAGHE	2	1	2
500	GATTI	2	2	3
500	CALZAGHE	2	2	4
600	LAMOTTA	4	1	1
600	HAGLER	4	1	2
600	HEARNS	4	1	3
600	FRAZIER	4	1	4
600	LAMOTTA	4	2	5
600	HAGLER	4	2	6
600	FRAZIER	4	2	7
600	HEARNS	4	2	8
700	GUINN	3	1	1
700	JUDAH	3	1	2
700	MARGARITO	3	1	3
700	GUINN	3	2	4
700	JUDAH	3	2	5
700	MARGARITO	3	2	6

以上查询对每位员工都进行了排名，也对表示员工的重复行进行了排名。对于 **IT_APP** 表中的每位员工，结果集中都包含相应的两行，而每一行都包含员工在部门内的排名。为什么要生成这些多余的行呢？这是因为你要在结果集的 **ENAME** 列中显示 **DEPTNO**。如果只生成 **IT_APPS** 表和一张单行表的笛卡儿积，那么就不会有额外的行，因为将任何表的基数乘以 1 的结果都是该表的基数。

下一步是对至此返回的结果集进行转置，将所有 **ENAME** 都放在一列中，并在它们前面加上 **DEPTNO**。下面的查询演示了这项工作是如何完成的。

```
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
  ) x,
  (select level id from dual connect by level <= 2) y
  ) z
 where rn <= cnt+1
```

FLAG1	FLAG2	IT_DEPT
1	1	400
1	1	MAYWEATHER
1	1	CASTILLO
1	1	MARQUEZ
1	1	MOSLEY
1	1	CORRALES
1	1	500
1	1	CALZAGHE
1	1	GATTI
1	1	600
1	1	HAGLER
1	1	HEARNS
1	1	FRAZIER
1	1	LAMOTTA
1	1	700
1	1	JUDAH
1	1	MARGARITO
1	1	GUINN

FLAG1 和 **FLAG2** 的作用到后面才会显现出来，现在可以暂时忽略。请将注意力放在 **IT_DEPT** 列。对于每个 **DEPTNO**，返

回的行数为 **CNT*2**，但实际上只需要 **CNT+1** 行。**WHERE** 子句中的筛选器删除了多余的行，只保留排名值小于或等于 **CNT+1** 的行，即显示部门中所有员工后，再多显示一名员工（在部门中排名第一的员工）。这个额外行将用于显示 **DEPTNO**。通过使用 **DECODE**（一个较老的 Oracle 函数，作用与 **CASE** 表达式大致相同）来评估 **RN** 的值，可以将 **DEPTNO** 加入结果集中。原来根据 **RN** 值排在最前面的员工依然显示在结果集中，但现在位于相应部门的最后面（由于排列顺序无关紧要，因此这不是问题）。至此，**UNION ALL** 的后半部分就介绍完了。

对于 **UNION ALL** 的前半部分，处理方式与后半部分相同，因此没必要再解释。下面来看看合并得到的结果集。

```
select 1 flag1, 0 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_research
    ) x,
    (select level id from dual connect by level <= 2) y
    )
 where rn <= cnt+1
union all
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
```

```

) x,
(select level id from dual connect by level <= 2) y
)
where rn <= cnt+1

```

FLAG1	FLAG2	IT_DEPT

1	0	100
1	0	JONES
1	0	TONEY
1	0	HOPKINS
1	0	200
1	0	P.WHITAKER
1	0	MARCIANO
1	0	ROBINSON
1	0	MORALES
1	0	300
1	0	WRIGHT
1	0	J.TAYLOR
1	0	LACY
1	1	400
1	1	MAYWEATHER
1	1	CASTILLO
1	1	MARQUEZ
1	1	MOSLEY
1	1	CORRALES
1	1	500
1	1	CALZAGHE
1	1	GATTI
1	1	600
1	1	HAGLER
1	1	HEARNS
1	1	FRAZIER
1	1	LAMOTTA
1	1	700
1	1	JUDAH
1	1	MARGARITO
1	1	GUINN

当前，**FLAG1** 的用途还不清楚，但 **FLAG2** 指出了当前行是 **UNION ALL** 的哪部分返回的（0 表示前半部分，1 表示后半部分）。

下一步是将合并后的结果集放在一个内嵌视图中，并生成基于 **FLAG1** 的移动总计（**FLAG1** 的用途终于被揭示出来了），用于表示不同部分中行的排名。包含排名（移动总计）的结果集如下所示。

```
select sum(flag1)over(partition by flag2
                      order by flag1,rownum) flag,
       it_dept, flag2
  from (
select 1 flag1, 0 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_research
  ) x,
       (select level id from dual connect by level <= 2) y
  )
 where rn <= cnt+1
union all
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
  ) x,
       (select level id from dual connect by level <= 2) y
  )
 where rn <= cnt+1
  ) tmp1
```

FLAG	IT_DEPT	FLAG2
1	100	0

2	JONES	0
3	TONEY	0
4	HOPKINS	0
5	200	0
6	P.WHITAKER	0
7	MARCIANO	0
8	ROBINSON	0
9	MORALES	0
10	300	0
11	WRIGHT	0
12	J.TAYLOR	0
13	LACY	0
1	400	1
2	MAYWEATHER	1
3	CASTILLO	1
4	MARQUEZ	1
5	MOSLEY	1
6	CORRALES	1
7	500	1
8	CALZAGHEe	1
9	GATTI	1
10	600	1
11	HAGLER	1
12	HEARNS	1
13	FRAZIER	1
14	LAMOTTA	1
15	700	1
16	JUDAH	1
17	MARGARITO	1
18	GUINN	1

最后一步是根据 **FLAG2** 对 **TMP1** 返回的值进行转置，并根据 **FLAG**（**TMP1** 生成的移动总计）进行分组。转置 **TMP1** 返回结果的操作放在内嵌视图 **TMP2** 中。最终的解决方案和结果集如下所示。

```
select max(decode(flag2,0,it_dept)) research,
       max(decode(flag2,1,it_dept)) apps
  from (
select sum(flag1)over(partition by flag2
                      order by flag1,rownum) flag,
       it_dept, flag2
```

```

from (
select 1 flag1, 0 flag2,
      decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
      row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
      ename,
      count(*)over(partition by deptno) cnt
  from it_research
    ) x,
    (select level id from dual connect by level <= 2) y
    )
 where rn <= cnt+1
union all
select 1 flag1, 1 flag2,
      decode(rn,1,to_char(deptno),' '||ename) it_dept
  from (
select x.*, y.id,
      row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
      ename,
      count(*)over(partition by deptno) cnt
  from it_apps
    ) x,
    (select level id from dual connect by level <= 2) y
    )
 where rn <= cnt+1
    ) tmp1
    ) tmp2
group by flag

```

RESEARCH	APPS
-----	-----
100	400
JONES	MAYWEATHER
TONEY	CASTILLO
HOPKINS	MARQUEZ
200	MOSLEY
P.WHITAKER	CORRALES
MARCIANO	500
ROBINSON	CALZAGHE
MORALES	GATTI

300

WRIGHT
J. TAYLOR
LACY

600

HAGLER
HEARNS
FRAZIER
LAMOTTA

700

JUDAH
MARGARITO
GUINN

14.10 在Oracle中将标量子查询转换为复合子

查询

1. 问题

标量子查询只能返回一个值，而你想绕过这种限制。例如，你试图执行如下查询：

```
select e.deptno,
       e.ename,
       e.sal,
       (select d.dname,d.loc,sysdate today
        from dept d
        where e.deptno=d.deptno)
from emp e
```

但以出错告终，因为 **SELECT** 列表中的子查询只能返回一个值。

2. 解决方案

必须承认，在实际工作中，不太可能遇到这种问题，因为只要将 **EMP** 表和 **DEPT** 表连接起来，就可以从 **DEPT** 表中返回任意数量的值。然而，这里的重点是掌握技巧以及如何在合适的场景中使用它。嵌套在 **SELECT** 子句中的 **SELECT** 子句被称为标量子查询，它们只能返回一个值，要绕过这种限制，关键是使用 Oracle 对象类型：定义包含多个属性的对象，再将对象作为单个实体并分别引用其中的元素。实际上，这样做根本没有绕过前述限制，因为确实只返回了一个值，即一个包含众多属性的对象。

本解决方案将使用如下对象类型。

```
create type generic_obj
as object (
  val1 varchar2(10),
  val2 varchar2(10),
  val3 date
);
```

定义这个对象类型后，便可执行如下查询。

```
1 select x.deptno,
2       x.ename,
3       x.multival.val1 dname,
4       x.multival.val2 loc,
5       x.multival.val3 today
6 from (
7 select e.deptno,
8       e.ename,
9       e.sal,
10      (select generic_obj(d.dname,d.loc,sysdate+1)
11       from dept d
12       where e.deptno=d.deptno) multival
13 from emp e
14      ) x
```

DEPTNO	ENAME	DNAME	LOC	TODAY
20	SMITH	RESEARCH	DALLAS	12-SEP-2020
30	ALLEN	SALES	CHICAGO	12-SEP-2020
30	WARD	SALES	CHICAGO	12-SEP-2020
20	JONES	RESEARCH	DALLAS	12-SEP-2020
30	MARTIN	SALES	CHICAGO	12-SEP-2020
30	BLAKE	SALES	CHICAGO	12-SEP-2020
10	CLARK	ACCOUNTING	NEW YORK	12-SEP-2020
20	SCOTT	RESEARCH	DALLAS	12-SEP-2020
10	KING	ACCOUNTING	NEW YORK	12-SEP-2020
30	TURNER	SALES	CHICAGO	12-SEP-2020
20	ADAMS	RESEARCH	DALLAS	12-SEP-2020
30	JAMES	SALES	CHICAGO	12-SEP-2020
20	FORD	RESEARCH	DALLAS	12-SEP-2020
10	MILLER	ACCOUNTING	NEW YORK	12-SEP-2020

13. 讨论

上述解决方案的关键是使用对象的构造函数（构造函数默认与对象同名）。由于对象本身就是单个标量值，因此返回它没有违反标量子查询规则。

```
select e.deptno,
       e.ename,
       e.sal,
       (select generic_obj(d.dname,d.loc,sysdate-1)
        from dept d
        where e.deptno=d.deptno) multival
from emp e
```

DEPTNO	ENAME	SAL	MULTIVAL(VAL1, VAL2, VAL3)
20	SMITH	800	GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
30	ALLEN	1600	GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
30	WARD	1250	GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
20	JONES	2975	GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
30	MARTIN	1250	GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
30	BLAKE	2850	GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
10	CLARK	2450	GENERIC_OBJ('ACCOUNTING', 'NEW YORK', '12-SEP-2020')
20	SCOTT	3000	GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
10	KING	5000	GENERIC_OBJ('ACCOUNTING', 'NEW YORK', '12-SEP-2020')
30	TURNER	1500	GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
20	ADAMS	1100	GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
30	JAMES	950	GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2020')
20	FORD	3000	GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2020')
10	MILLER	1300	GENERIC_OBJ('ACCOUNTING', 'NEW YORK', '12-SEP-2020')

将这个查询放在一个内嵌视图中，并提取对象的属性。



与其他 RDBMS 不同，在 Oracle 中，通常不需要给内嵌视图命名，但这里必须给内嵌视图命名，否则将无法引用对象的属性。

14.11 将序列化数据转换为行

1. 问题

你想对序列化数据（存储在字符串中的数据）进行分析，并将其作为行返回。例如，你存储了如下数据。

```
STRINGS
-----
entry:stewiegriffin:lois:brian:
entry:moe::sizlack:
entry:petergriffin:meg:chris:
entry:willie:
entry:quagmire:mayorwest:cleveland:
entry::flanders:
entry:robo:tchi:ken:
```

对于这些序列化字符串，你想将其转换为如下结果集。

VAL1	VAL2	VAL3

moe		sizlack
petergriffin	meg	chris
quagmire	mayorwest	cleveland
robo	tchi	ken
stewiegriffin	lois	brian
willie		
		flanders

2. 解决方案

本例中每个序列化字符串最多可以存储 3 个值，值之间用冒号分隔。不一定每个字符串都包含 3 个值，也可能更少。如果一个字符串包含的值不到 3 个，你必须小心处理，将各个

值放在结果集的正确列中。例如，请看下面的字符串。

```
entry:::flanders:
```

这个字符串缺失了前两个值，只有第三个值。因此，如果查看本节“问题”部分的结果集，你将发现在 **FLANDERS** 所在的行中，**VAL1** 列和 **VAL2** 列的值都为 **NULL**。

本解决方案的关键是先对字符串进行分析，然后再执行简单的转置。本解决方案使用了视图 **V** 返回的行，该视图的定义如下所示。

```
create view V
as
select 'entry:stewiegriffin:lois:brian:' strings
  from dual
union all
select 'entry:moe::sizlack:'
  from dual
union all
select 'entry:petergriffin:meg:chris:'
  from dual
union all
select 'entry:willie:'
  from dual
union all
select 'entry:quagmire:mayorwest:cleveland:'
  from dual
union all
select 'entry:::flanders:'
  from dual
union all
select 'entry:robo:tchi:ken:'
  from dual
```

本解决方案对视图 **V** 提供的示例数据进行了分析，如以下代码所示。这里使用的是 **Oracle** 语法，但由于只涉及字符串分析函数，因此很容易对该解决方案进行转换，使其适用于其

他 RDBMS。

```
1  with cartesian as (  
2    select level id  
3      from dual  
4    connect by level <= 100  
5  )  
6  select max(decode(id,1,substr(strings,p1+1,p2-1))) val1,  
7         max(decode(id,2,substr(strings,p1+1,p2-1))) val2,  
8         max(decode(id,3,substr(strings,p1+1,p2-1))) val3  
9    from (  
10 select v.strings,  
11        c.id,  
12        instr(v.strings,':',1,c.id) p1,  
13        instr(v.strings,':',1,c.id+1)-  
instr(v.strings,':',1,c.id) p2  
14    from v, cartesian c  
15   where c.id <= (length(v.strings)-  
length(replace(v.strings,':')))-1  
16      )  
17  group by strings  
18  order by 1
```

13. 讨论

首先，遍历序列化字符串。

```
with cartesian as (  
select level id  
  from dual  
 connect by level <= 100  
)  
select v.strings,  
       c.id  
  from v, cartesian c  
 where c.id <= (length(v.strings)-  
length(replace(v.strings,':')))-1
```

STRINGS

ID

```

-----
entry:::flanders:          1
entry:::flanders:          2
entry:::flanders:          3
entry:moe::sizlack:        1
entry:moe::sizlack:        2
entry:moe::sizlack:        3
entry:petergriffin:meg:chris: 1
entry:petergriffin:meg:chris: 3
entry:petergriffin:meg:chris: 2
entry:quagmire:mayorwest:cleveland: 1
entry:quagmire:mayorwest:cleveland: 3
entry:quagmire:mayorwest:cleveland: 2
entry:robo:tchi:ken:        1
entry:robo:tchi:ken:        2
entry:robo:tchi:ken:        3
entry:stewiegriffin:lois:brian: 1
entry:stewiegriffin:lois:brian: 3
entry:stewiegriffin:lois:brian: 2
entry:willie:               1

```

然后，使用函数 **INSTR** 确定每个冒号在字符串中的位置。由于要提取的每个值都在两个冒号之间，因此给数字值指定了别名 **P1** 和 **P2**，它们分别表示位置 1 和位置 2。

```

with cartesian as (
select level id
  from dual
 connect by level <= 100
)
select v.strings,
       c.id,
       instr(v.strings,':',1,c.id) p1,
       instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id)
p2
  from v,cartesian c
 where c.id <= (length(v.strings)-
length(replace(v.strings,':')))-1
 order by 1

```

STRINGS	ID	P1	P2
-----	----	-----	-----

entry:::flanders:	1	6	1
entry:::flanders:	2	7	1
entry:::flanders:	3	8	9
entry:moe::sizlack:	1	6	4
entry:moe::sizlack:	2	10	1
entry:moe::sizlack:	3	11	8
entry:petergriffin:meg:chris:	1	6	13
entry:petergriffin:meg:chris:	3	23	6
entry:petergriffin:meg:chris:	2	19	4
entry:quagmire:mayorwest:cleveland:	1	6	9
entry:quagmire:mayorwest:cleveland:	3	25	10
entry:quagmire:mayorwest:cleveland:	2	15	10
entry:robo:tchi:ken:	1	6	5
entry:robo:tchi:ken:	2	11	5
entry:robo:tchi:ken:	3	16	4
entry:stewiegriffin:lois:brian:	1	6	14
entry:stewiegriffin:lois:brian:	3	25	6
entry:stewiegriffin:lois:brian:	2	20	5
entry:willie:	1 6 7		

确定了字符串中每对冒号的位置后，只需将这些信息传递给函数 **SUBSTR**，以提取冒号之间的值。由于要返回的结果集包含 3 列，因此使用 **DECODE** 来评估笛卡儿积返回的 **ID**。

```

with cartesian as (
  select level id
    from dual
   connect by level <= 100
)
select decode(id,1,substr(strings,p1+1,p2-1)) val1,
       decode(id,2,substr(strings,p1+1,p2-1)) val2,
       decode(id,3,substr(strings,p1+1,p2-1)) val3
  from (
select v.strings,
       c.id,
       instr(v.strings,':',1,c.id) p1,
       instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id)
p2
  from v,cartesian c
 where c.id <= (length(v.strings)-
length(replace(v.strings,':')))-1
)

```

order by 1		
VAL1	VAL2	VAL3

moe		
petergriffin		
quagmire		
robo		
stewiegriffin		
willie		
	lois	
	meg	
	mayorwest	
	tchi	
		brian
		sizlack
		chris
		cleveland
		flanders
		ken

最后，对 **SUBSTR** 返回的值应用聚合函数，并按 **ID** 分组，让结果集易于阅读。

```

with cartesian as (
select level id
  from dual
 connect by level <= 100
)
select max(decode(id,1,substr(strings,p1+1,p2-1))) val1,
       max(decode(id,2,substr(strings,p1+1,p2-1))) val2,
       max(decode(id,3,substr(strings,p1+1,p2-1))) val3
  from (
select v.strings,
       c.id,
       instr(v.strings,':',1,c.id) p1,
       instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id)
p2
  from v,cartesian c
 where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1
  )

```


group by strings
order by 1

VAL1	VAL2	VAL3
-----	-----	-----
moe		sizlack
petergriffin	meg	chris
quagmire	mayorwest	cleveland
robo	tchi	ken
stewiegriffin	lois	brian
willie		flanders

14.12 计算占总计的百分比

1. 问题

你想制作一张报表，其中包含一系列数字值，并呈现每个值占总计的百分比。假设你使用的是 Oracle，你想返回一个结果集，呈现薪水在不同职位之间的分布情况，以确定哪种职位给公司带来的开销最高。为避免误导，你还想在结果集中呈现各种职位的员工数量。换言之，你想制作如下报表。

JOB	NUM_EMPS	PCT_OF_ALL_SALARIES
CLERK	4	14
ANALYST	2	20
MANAGER	3	28
SALESMAN	4	19
PRESIDENT	1	17

如你所见，如果没有在报表中呈现各种职位的员工数量，那么总裁的薪水在薪水总计中的占比将看起来很低。在加入员工数量后，我们获悉总裁一个人的薪水占了总薪水的 17%。

2. 解决方案

仅当你使用的是 Oracle 时，才能使用内置函数 **RATIO_TO_REPORT** 来妥善地解决上述问题。在其他 RDBMS 中，要计算占总计的百分比，可以使用除法运算，这在 7.11 节中介绍过。

```
1 select job,num_emps,sum(round(pct)) pct_of_all_salaries
2   from (
3 select job,
4        count(*)over(partition by job) num_emps,
```

```

5      ratio_to_report(sal)over()*100 pct
6  from emp
7      )
8  group by job,num_emps

```

13. 讨论

首先，使用窗口函数 **COUNT OVER** 返回各种职位的员工数量。然后，使用 **RATIO_TO_REPORT** 返回各位员工的薪水占薪水总计的百分比（返回的值为小数）。

```

select job,
       count(*)over(partition by job) num_emps,
       ratio_to_report(sal)over()*100 pct
  from emp

```

JOB	NUM_EMPS	PCT
ANALYST	2	10.3359173
ANALYST	2	10.3359173
CLERK	4	2.75624462
CLERK	4	3.78983635
CLERK	4	4.4788975
CLERK	4	3.27304048
MANAGER	3	10.2497847
MANAGER	3	8.44099914
MANAGER	3	9.81912145
PRESIDENT	1	17.2265289
SALESMAN	4	5.51248923
SALESMAN	4	4.30663221
SALESMAN	4	5.16795866
SALESMAN	4	4.30663221

最后，使用聚合函数 **SUM** 将 **RATIO_TO_REPORT** 返回的值相加。务必按 **JOB** 和 **NUM_EMPS** 进行分组。通过乘以 100，可以返回表示百分比值的整数（例如，占比为 25% 时，将返回 25，而不是 0.25）。

```

select job,num_emps,sum(round(pct)) pct_of_all_salaries
  from (
select job,
       count(*)over(partition by job) num_emps,
       ratio_to_report(sal)over()*100 pct
  from emp
    )
group by job,num_emps

```

JOB	NUM_EMPS	PCT_OF_ALL_SALARIES
-----	-----	-----
CLERK	4	14
ANALYST	2	20
MANAGER	3	28
SALESMAN	4	19
PRESIDENT	1	17

14.13 确定编组是否包含指定的值

1. 问题

对于给定的行，你想创建一个布尔标志，指出在该行所属的编组中，是否至少有 1 行包含特定的值。来看一个例子：某位学生在给定时段（3 个月）参加特定次数（3 次）考试。只要该学生通过了其中一次考试，便满足了要求，应返回一个指出这一点的标志。在 3 个月期间的 3 次考试中，如果该学生一次都没通过，就返回另一个标志，以指出这一点。请看下面的示例（这里生成示例行时使用的是 Oracle 语法，如果你使用的是其他 RDBMS，则必须做细微的修改）。

```
create view V
as
select 1 student_id,
       1 test_id,
       2 grade_id,
       1 period_id,
       to_date('02/01/2020','MM/DD/YYYY') test_date,
       0 pass_fail
  from dual union all
select 1, 2, 2, 1, to_date('03/01/2020','MM/DD/YYYY'), 1 from dual
union all
select 1, 3, 2, 1, to_date('04/01/2020','MM/DD/YYYY'), 0 from dual
union all
select 1, 4, 2, 2, to_date('05/01/2020','MM/DD/YYYY'), 0 from dual
union all
select 1, 5, 2, 2, to_date('06/01/2020','MM/DD/YYYY'), 0 from dual
union all
select 1, 6, 2, 2, to_date('07/01/2020','MM/DD/YYYY'), 0 from dual

select *
  from V
```

STUDENT_ID	TEST_ID	GRADE_ID	PERIOD_ID	TEST_DATE	PASS_FAIL
1	1	2	1	01-FEB-2020	0

1	2	2	1	01-MAR-2020	1
1	3	2	1	01-APR-2020	0
1	4	2	2	01-MAY-2020	0
1	5	2	2	01-JUN-2020	0
1	6	2	2	01-JUL-2020	0

从上面的结果集可知，该学生在两个学期（每个学期 3 个月）内参加了 6 次考试。该学生通过了其中一次考试（1 表示通过，0 表示未通过），因此满足了第一学期的要求。由于该学生没有通过第二学期（接下来的 3 个月）的任何一次考试，因此对于第二学期的全部 3 次考试，**PASS_FAIL** 值都为 0。你想返回一个结果集，指出某位学生是否通过了给定学期的考试。

STUDENT_ID	TEST_ID	GRADE_ID	PERIOD_ID	TEST_DATE	METREQ	IN_PROGRESS
1	1	2	1	01-FEB-2020	+	0
1	2	2	1	01-MAR-2020	+	0
1	3	2	1	01-APR-2020	+	0
1	4	2	2	01-MAY-2020	-	0
1	5	2	2	01-JUN-2020	-	0
1	6	2	2	01-JUL-2020	-	1

在这个结果集中，**METREQ** 的值为 + 或-，表示某位学生是否满足了如下要求：在跨度为 3 个月的学期内，是否至少通过了一次考试。对于给定的学期，如果该学生至少通过了其中的一次考试，那么 **IN_PROGRESS** 应为 0。否则，在表示该学期最后一次考试的行中，**IN_PROGRESS** 的值应为 1。

12. 解决方案

上述问题看起来很棘手，因为必须将编组中的行作为一个整体进行处理，而不能分别处理。请看本节“问题”部分中 **PASS_FAIL** 列的值，如果逐行进行评估，好像除 **TEST_ID** 值为 2 的行外，其他各行的 **METREQ** 值都应为-，但情况并非如此。你必须将编组中的所有行作为一个整体进行评估。使用窗口函数 **MAX OVER**，可以轻松地确定某位学生在特定学期内是否至少通过了一次考试。确定这一点后，只需使用 **CASE** 表达式就可以生成正确的布尔值。

```
1  select student_id,
2         test_id,
3         grade_id,
4         period_id,
5         test_date,
6         decode( grp_p_f,1,lpad('+',6),lpad('-',6) ) metreq,
7         decode( grp_p_f,1,0,
8                 decode( test_date,last_test,1,0 ) ) in_progress
9  from (
10 select V.*,
11        max(pass_fail)over(partition by
12                           student_id,grade_id,period_id) grp_p_f,
13        max(test_date)over(partition by
14                           student_id,grade_id,period_id) last_test
15  from V
16  ) x
```

13. 讨论

本解决方案的关键是使用窗口函数 **MAX OVER** 返回每个编组中最大的 **PASS_FAIL** 值。由于 **PASS_FAIL** 列的值只能是 1 或 0，因此如果某位学生至少通过了一次考试，那么 **MAX OVER** 将为整个编组返回 1。下面的代码演示了这一点。

```
STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE PASS_FAIL
GRP_PASS_FAIL
```

	1	1	2	1 01-FEB-2020	0
1	1	2	2	1 01-MAR-2020	1
1	1	3	2	1 01-APR-2020	0
1	1	4	2	2 01-MAY-2020	0
0	1	5	2	2 01-JUN-2020	0
0	1	6	2	2 01-JUL-2020	0

```
select V.*,
       max(pass_fail)over(partition by
                           student_id,grade_id,period_id) grp_p_f,
       max(test_date)over(partition by
                           student_id,grade_id,period_id) last_test
from V
```

STUDENT_ID	TEST_ID	GRADE_ID	PERIOD_ID	TEST_DATE	PASS_FAIL
GRP_P_F	LAST_TEST				
1	1	2	1	01-FEB-2020	0

1	01-APR-2020	1	2	2	1	01-MAR-2020	1
1	01-APR-2020	1	3	2	1	01-APR-2020	0
1	01-APR-2020	1	4	2	2	01-MAY-2020	0
0	01-JUL-2020	1	5	2	2	01-JUN-2020	0
0	01-JUL-2020	1	6	2	2	01-JUL-2020	0
0	01-JUL-2020						

在确定该学生都通过了哪些学期的考试，以及每个学期最后一次考试的日期后，最后一步是做些格式设置，让结果集看起来更漂亮。在最终的解决方案中，使用了 Oracle 函数 **DECODE**（**CASE** 表达式升级版）来创建 **METREQ** 和 **IN_PROGRESS** 列，并使用了 **LPAD** 函数来让 **METREQ** 值右对齐。

```

select student_id,
       test_id,
       grade_id,
       period_id,
       test_date,
       decode( grp_p_f,1,lpad('+',6),lpad('-',6) ) metreq,
       decode( grp_p_f,1,0,
               decode( test_date,last_test,1,0 ) ) in_progress
  from (
select V.*,
       max(pass_fail)over(partition by
                           student_id,grade_id,period_id) grp_p_f,
       max(test_date)over(partition by
                           student_id,grade_id,period_id) last_test
  from V
  ) x

```

STUDENT_ID	TEST_ID	GRADE_ID	PERIOD_ID	TEST_DATE	METREQ	IN_PROGRESS
1	1	2	1	01-FEB-2020	+	

0						
	1	2	2	1	01-MAR-2020	+
0						
	1	3	2	1	01-APR-2020	+
0						
	1	4	2	2	01-MAY-2020	-
0						
	1	5	2	2	01-JUN-2020	-
0						
	1	6	2	2	01-JUL-2020	-
1						

14.14 小结

SQL 威力强大，超乎很多人的想象。展示不同寻常的 SQL 应用方式的实例遍布本书，而本章直奔极端情况，力图展示如何最大限度地利用 SQL（包括标准特性和某些 RDBMS 特有的特性）。

附录 A 温习窗口函数

本书中的实例充分利用了 2003 年发布的 ISO SQL 标准引入的窗口函数以及 RDBMS 特有的窗口函数。本附录将概述窗口函数的工作原理。窗口函数让很多原本很棘手的任务（使用标准 SQL 难以解决的任务）完成起来易如反掌。有关完整的窗口函数列表及其语法和工作原理的全面介绍，请参阅你使用的 RDBMS 提供的文档。

A.1 分组

学习窗口函数前，必须先弄明白 SQL 分组的工作原理。SQL 分组的概念可能难以掌握，要掌握这个概念，必须全面认识 **GROUP BY** 子句的工作原理，明白使用这种子句的查询返回特定结果的原因。

简单地说，分组是一种将类似的行组织在一起的方式。当你在查询中使用 **GROUP BY** 时，结果集中的每一行都是一个分组，表示的是指定列有相同值的一行或多行。

分组是独特的行实例，表示的是指定列有相同值的一行或多行。就 **EMP** 表而言，分组的例子包括：10 号部门的所有员工（这些员工之所以属于同一个分组，是因为他们的 **DEPTNO** 都为 10）；所有的文员（这些员工之所以属于同一个分组，是因为他们的 **JOB** 都为 **CLERK**）。请看下面的查询，其中第一个查询列出了 10 号部门的所有员工，第二个查询将 10 号部门的员工放在了一组，并返回了这个分组包含的行数（成员个数）、最高薪水和最低薪水。

```
select deptno,ename
  from emp
 where deptno=10

DEPTNO ENAME
-----
    10 CLARK
    10 KING
    10 MILLER

select deptno,
       count(*) as cnt,
       max(sal) as hi_sal,
       min(sal) as lo_sal
  from emp
 where deptno=10
```

group by deptno			
DEPTNO	CNT	HI_SAL	LO_SAL
-----	-----	-----	-----
10	3	5000	1300

如果不能将 10 号部门的员工放在一组中，那么要获取第二个查询提供的信息，就必须人工检查表示该部门员工的各行。在只有 3 行数据的情况下，这是小菜一碟，但如果有 300 万行数据呢？为什么要分组呢？这样做的原因很多，可能是想知道有多少个不同的分组，也可能是想知道各个分组都有多少个成员。从这个简单的示例可知，通过分组可以获悉有关表中很多行的信息，而无须逐一检查这些行。

A.1.1 SQL分组的定义

在数学中，群¹的定义为 (G, \bullet, e) ，其中 G 是一个集合， \bullet 是 G 中的一个二元运算，而 e 是 G 的一个成员。我们将以这个定义为基础，对 SQL 分组做出定义。SQL 分组的定义为 (G, e) ，其中 G 为使用 **GROUP BY** 的独立查询的结果集， e 为 G 的一个成员。对于 SQL 分组，存在如下公理：

¹“群”和“分组”的英文相同，都是 group。——译者注

- G 中的每个 e 都是独一无二的，表示一个或多个 e 实例；
- 对于 G 中的每个 e ，聚合函数 **COUNT** 返回的结果都大于 0。



前面的 SQL 分组定义中包含结果集，这突出了这样一个事实，即这里定义的是编写查询时涉及的 SQL 分组。因此，在前述公理中，完全可以将 e 替换为行，因为结果集中的行就是分组。

由于上述公理定义的性质是 SQL 分组的基础，因此必须证明这些公理是正确的（接下来将通过一些 SQL 查询来证明）。

1. 分组不能是空的

根据定义，分组必须至少包含一个成员（行）。如果承认这一点是天经地义的，那么就可以做出如下推断：使用空表无法创建分组。要证明这个命题为真，可以尝试证明此命题为假。下面的示例首先创建了一张空表，然后尝试使用 3 个针对这张空表的查询来创建分组。

```
create table fruits (name varchar(10))
```

```
select name
  from fruits
 group by name
```

```
(no rows selected)
```

```
select count(*) as cnt
  from fruits
 group by name
```

```
(no rows selected)
```

```
select name, count(*) as cnt
  from fruits
 group by name
```

```
(no rows selected)
```

如你所见，无法使用空表创建 SQL 分组。

2. 分组是独一无二的

下面来证明使用包含 **GROUP BY** 子句的查询创建的分组是独一无二的。以下示例在 **FRUITS** 表中插入了 5 行数据，然后

使用这些行来创建分组。

```
insert into fruits values ('Oranges')
insert into fruits values ('Oranges')
insert into fruits values ('Oranges')
insert into fruits values ('Apple')
insert into fruits values ('Peach')
```

```
select *
  from fruits
```

NAME

Oranges
Oranges
Oranges
Apple
Peach

```
select name
  from fruits
 group by name
```

NAME

Apple
Oranges
Peach

```
select name, count(*) as cnt
  from fruits
 group by name
```

NAME	CNT
------	-----

-----	-----
-------	-------

Apple	1
Oranges	3
Peach	1

第一个查询表明，“Oranges”在 **FRUITS** 表中出现了 3 次。不过，第二个查询和第三个查询（它们使用了 **GROUP BY**）只返回了一个“Oranges”实例。这两个查询证明了这样一点：结

果集中的行（在前面的 SQL 分组定义中，是 G 中的 e ）是独一无二的，每个 **NAME** 值都表示 **FRUITS** 表中一个或多个 **NAME** 值实例。

知道分组是独一无二的很重要，因为这意味着如果在查询中使用了 **GROUP BY** 子句，通常就不在 **SELECT** 列表中使用关键字 **DISTINCT**。



GROUP BY 和 **DISTINCT** 不是一回事，它们是两个完全不同的概念。然而，对于在 **GROUP BY** 子句中列出的项，其在结果集中将是独一无二的，因此在使用了 **GROUP BY** 的情况下，再使用 **DISTINCT** 无异于叠床架屋。

弗雷格公理和罗素悖论

弗雷格抽象公理基于 Cantor 对无穷集合成员资格的定义，该公理指出：给定特定的分辨性质，存在一个集合，其中只包含那些具备该性质的元素。这个公理存在漏洞，原因如 Robert Stoll 所言：“没有对抽象原则的使用进行限制”。为反驳这个公理，罗素让弗雷格考虑这样的集合：其成员为集合，而分辨性质为不是自身的成员。

正如罗素指出的，抽象公理给出的前提条件太宽松，只通过指定条件或性质来定义集合成员资格，因此存在自相矛盾的地方。为了更好地阐述其中的矛盾，罗素设计了如下理发匠问题。

在某个小镇，有一位男性理发匠，他给所有不自己刮胡子的人刮胡子，且只给这些人刮胡子。如果这

种说法为真，那么谁给理发匠刮胡子呢？

来看一个更具体的例子。

在集合 y 中，所有成员 x 都满足特定的条件 P 。

这个集合的数学表示如下。

$$\{x \in y | P(x)\}$$

这个集合的定义是，集合 y 只包含满足条件 P 的成员 x 。你可能发现，更直观的定义是，当且仅当 x 满足条件 P 时，它才是集合 y 的成员。

接下来，将条件 P 定义为不是自己的成员。

$$(x \notin x)$$

这样，集合定义将变成当且仅当 x 不是 x 的成员时， x 才是集合 y 的成员。

$$\{x \in y | (x \notin x)\}$$

你现在可能还没明白罗素悖论，但不妨问问自己如下问题：前述集合 y 可以是其自身的成员吗？假设 $x = y$ ，并重新审视这个集合。下面的集合可描述为当且仅当 y 不是 y 的成员时， y 才是 y 的成员。

$$\{y \in y | (y \notin y)\}$$

简单地说，罗素悖论将我们置于这样的境地：存在一个集合，它既是自身的成员，又不是自身的成员，这相互矛盾。如果凭直觉思考，你将认为这根本不是问题。事实上，一个集合怎么可能是自身的成员呢？毕竟由所有图书组成的集合不是一本图书。那么，这个悖论有何意义，在什么情况下会是一个问题呢？当你抽象地应用集

合论时，罗素悖论就是一个问题，例如，当你考虑由所有集合组成的集合时，罗素悖论就会显现出来。如果允许这样的概念存在，那么根据定义，由所有集合组成的集合必然是自身的成员（毕竟它是由所有集合组成的集合）。如果将前面的条件 $P(x)$ 应用于由所有集合组成的集合，那么结果将如何呢？简单地说，将出现罗素悖论：当且仅当由所有集合组成的集合不是自身的成员时，它才是自身的成员，这显然相互矛盾。

Ernst Zermelo 后来提出了分离公理模式（也被称为“子集公理模式”或“分类公理”），巧妙地消除了公理集合论中存在的罗素悖论。

13. COUNT的结果不可能为 0

前面的查询和结果还证明了最后一个公理：在针对非空表且包含 **GROUP BY** 的查询中使用 **COUNT** 时，返回的值不可能为 0。对于特定的分组，返回的计数不可能为 0，这没什么奇怪的。前面证明过，使用空表无法创建分组，因此分组至少包含一行数据。既然至少包含一行数据，计数就至少为 1。



别忘了，这里说的是将 **COUNT** 与 **GROUP BY** 结合起来使用的情形，而非单独使用 **COUNT**。在针对空表的查询中使用 **COUNT** 时，如果没有 **GROUP BY** 子句，那么返回值当然为 0。

A.1.2 悖论

罗素发现弗雷格的抽象公理存在相互矛盾的地方后，弗雷格在回应中这样说：

工作完成后，却发现其科学大厦的地基发生了动摇，对于科学工作者来说，没什么比这更糟糕了。这正是我在收到罗素先生的来信后的处境，因为那时我的作品的印制工作已接近尾声。

悖论通常提供了否定既有理论或理念的场景，在很多情况下，这种否定是局部性的且能够“规避”，或者其适用的情形不多，完全可以忽略。

至此，你可能猜到了，这里讨论悖论旨在指出前面有关 SQL 分组的定义存在悖论，必须予以解决。虽然当前的重点是分组，但最终要讨论的是 SQL 查询。在查询的 **GROUP BY** 子句中，可以指定的值种类繁多，比如常量、表达式，但最常见的是列。由于在 SQL 中 **NULL** 也是合法的“值”，因此我们将为这种灵活性付出代价。**NULL** 会带来问题，因为它会被聚合函数忽略。如果一张表只有一行，且该行的值为 **NULL**，那么在针对该表的 **GROUP BY** 查询中，聚合函数 **COUNT** 将返回什么值呢？根据定义，结合使用 **GROUP BY** 和聚合函数 **COUNT** 时，返回的值必然大于或等于 1。如果值被函数（如 **COUNT**）忽略，那么结果将如何呢？这对分组定义来说意味着什么呢？请看下面的示例，它揭示了 **NULL** 分组悖论（这里使用函数 **COALESCE** 旨在提高可读性）。

```
select *
  from fruits

NAME
-----
Oranges
Oranges
Oranges
Apple
Peach

insert into fruits values (null)
insert into fruits values (null)
```

```
insert into fruits values (null)
insert into fruits values (null)
insert into fruits values (null)
```

```
select coalesce(name,'NULL') as name
  from fruits
```

```
NAME
-----
Oranges
Oranges
Oranges
Apple
Peach
NULL
NULL
NULL
NULL
NULL
```

```
select coalesce(name,'NULL') as name,
       count(name) as cnt
  from fruits
 group by name
```

```
NAME          CNT
-----
Apple          1
NULL           0
Oranges        3
Peach          1
```

看起来由于表中包含 **NULL** 值，导致 **SQL** 分组定义存在矛盾（悖论）。所幸对于这种矛盾，无须过于关注，因为与其说它与定义相关，不如说它与聚合函数的实现相关。来看上述示例中的最后一个查询，与之对应的问题陈述如下。

计算每个名称在 **FRUITS** 表中出现的次数或计算每个分组包含的成员数。

从前面的 **INSERT** 语句可知，有 5 行的值为 **NULL**，这意味着存

在一个 **NULL** 分组，它包含 5 个成员。



NULL 无疑具备使其不同于其他值的性质，但它毕竟也是一个值，可以形成分组。

如何编写最后那个查询，使其返回我们所需的信息（计数 5 而不是 0），同时遵守分组定义呢？下面的示例是一个规避方案，演示了如何应对 **NULL** 分组悖论。

```
select coalesce(name,'NULL') as name,  
       count(*) as cnt  
  from fruits  
 group by name
```

NAME	CNT
-----	-----
Apple	1
Oranges	3
Peach	1
NULL	5

规避方案是使用 **COUNT(*)** 而不是 **COUNT(NAME)**，以避免 **NULL** 分组悖论。将列名传递给聚合函数时，聚合函数会忽略指定列中的 **NULL** 值，因此使用 **COUNT** 时，为避免返回 0，不要传入列名，而要传入一个星号（*）。* 会导致函数 **COUNT** 计算行数，而不是值的个数，因此实际值是否为 **NULL** 无关紧要。

结果集中每个分组（ G 中的每个 e ）都是独一无二的这一公理也存在一个悖论。将 **SQL** 结果集和表称为多重集合或“袋”或许更准确（因为它们可以包含重复的行），有鉴于此，可以返回包含重复分组的结果集。请看下面的查询。

```
select coalesce(name,'NULL') as name,  
       count(*) as cnt  
  from fruits  
 group by name
```

```

union all
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name

```

NAME	CNT
-----	-----
Apple	1
Oranges	3
Peach	1
NULL	5
Apple	1
Oranges	3
Peach	1
NULL	5

```

select x.*
  from (
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name
    ) x,
    (select deptno from dept) y

```

NAME	CNT
-----	-----
Apple	1
Apple	1
Apple	1
Apple	1
Oranges	3
Oranges	3
Oranges	3
Oranges	3
Peach	1
Peach	1
Peach	1
Peach	1
NULL	5
NULL	5
NULL	5
NULL	5

如你所见，在这些查询的最终结果中，分组实际上是重复的。所幸对此无须过于担心，因为它只是一个部分悖论（**partial paradox**）。前面将分组定义成了 (G, e) ，其中 G 是使用 **GROUP BY** 的单个或独立查询的结果集。简单地说，任何 **GROUP BY** 查询返回的结果集本身都复合了前述分组定义。仅当你将两个 **GROUP BY** 查询的结果集合并以创建多重集合时，才可能出现分组重复的情况。在前面的示例中，第一个查询使用了 **UNION ALL**，这不是集合运算，而是多重集合运算，它还调用了 **GROUP BY** 两次，这相当于执行了两个查询。



如果使用集合运算 **UNION**，那么将不会出现重复的分组。

在前面的集合中，第二个查询使用了笛卡儿积，这要求你先将分组具体化（**materialize**），然后再生成笛卡儿积。因此，独立的 **GROUP BY** 查询是符合分组定义的。这两个示例都没有否定前面的 **SQL** 分组定义，此处提供它们只是出于完整性考虑，让你知道在 **SQL** 中一切皆有可能。

A.1.3 **SELECT**和**GROUP BY**之间的关系

定义并证明分组概念后，该介绍与 **GROUP BY** 查询相关的实用知识了。在 **SQL** 中分组时，弄明白 **SELECT** 子句和 **GROUP BY** 子句之间的关系很重要。使用如 **COUNT** 等聚合函数时，对于包含在 **SELECT** 列表中的项，如果没有将其作为参数传递给聚合函数，就必须在 **GROUP BY** 子句中包含它，请务必牢记这一点。如果你编写了如下 **SELECT** 子句：

```
select deptno, count(*) as cnt
  from emp
```


则必须在 **GROUP BY** 子句中包含 **DEPTNO**。

```
select deptno, count(*) as cnt
  from emp
 group by deptno
```

DEPTNO	CNT
10	3
20	5
30	6

对于这条规则，常量、用户定义函数返回的标量值、窗口函数和非相关子查询是例外情况。由于 **SELECT** 子句是在 **GROUP BY** 子句之后执行的，因此这些元素位于 **SELECT** 列表中时，无须（在有些情况下是不能）包含在 **GROUP BY** 子句中。

```
select 'hello' as msg,
       1 as num,
       deptno,
       (select count(*) from emp) as total,
       count(*) as cnt
  from emp
 group by deptno
```

MSG	NUM	DEPTNO	TOTAL	CNT
hello	1	10	14	3
hello	1	20	14	5
hello	1	30	14	6

不要被这个查询弄糊涂了。出现在 **SELECT** 列表中、但未出现在 **GROUP BY** 子句中的那些项，不会改变每个 **DEPTNO** 的 **CNT** 值，也不会改变 **DEPTNO** 的值。基于上述查询的结果，可以对有关 **SELECT** 列表和 **GROUP BY** 子句的项匹配规则做出更精确的定义。

对于 **SELECT** 列表中的项，如果其可以改变分组或聚合函数

返回的值，就必须在 **GROUP BY** 子句中包含它们。

在前面的 **SELECT** 列表中，除 **DEPTNO** 外的其他项都不会改变任何分组（**DEPTNO**）的**CNT**值，也不会改变分组本身。

接下来的问题是，**SELECT** 列表中的哪些项可以改变分组或聚合函数返回的值呢？答案很简单：在 **FROM** 子句中指定的表中的其他列。如果在前面的查询中加上 **JOB** 列，结果将如何呢？

<pre>select deptno, job, count(*) as cnt from emp group by deptno, job</pre>		
DEPTNO	JOB	CNT
-----	-----	----
10	CLERK	1
10	MANAGER	1
10	PRESIDENT	1
20	CLERK	2
20	ANALYST	2
20	MANAGER	1
30	CLERK	1
30	MANAGER	1
30	SALESMAN	4

在 **SELECT** 列表中包含 **EMP** 表的另一列（**JOB**）时，将改变分组和结果集。因此，在 **GROUP BY** 子句中，必须同时包含 **JOB** 和 **DEPTNO**，否则这个查询将以失败告终。在 **SELECT/GROUP BY** 子句中包含 **JOB** 后，查询将从“各个部门都有多少位员工”变成“各个部门都有多少种不同类型的员工”。注意，分组依然是独一无二的，因为虽然 **DEPTNO** 值和 **JOB** 值都不是独一无二的，但它们的组合（**GROUP BY** 和 **SELECT** 列表中包含它们，因此分组中也包含它们）是独一无二的（例如，10 和 **CLERK** 的组合只出现一次）。

如果 **SELECT** 列表中只包含聚合函数，则可以在 **GROUP BY** 子句中包含任何有效的列。下面的两个查询说明了这一点。

```

select count(*)
  from emp
 group by deptno

COUNT(*)
-----
      3
      5
      6

select count(*)
  from emp
 group by deptno,job

COUNT(*)
-----
      1
      1
      1
      2
      2
      1
      1
      1
      1
      4

```

在 **SELECT** 列表中，并非必须包含除聚合函数外的其他项，但这样做通常可以提高结果集的可读性和有用性。



通常，同时使用 **GROUP BY** 和聚合函数时，对于 **SELECT** 列表中的项，如果它是 **FROM** 子句中指定表中的列，且没有作为聚合函数的参数，就必须包含在 **GROUP BY** 子句中。然而，MySQL 提供了一项“特性”，让你能够违反这条规则：对于 **SELECT** 列表中的项，即便它是 **FROM** 子句中指定表中的列，且没有作为聚合函数的参数，也可以不在 **GROUP BY** 子句中包含它。这里之所以使用术语特性，是因为这样做可能发生 bug。实际上，如果你使用的是 MySQL 且很在乎查询的准确性，建议你敦促 MySQL 开发

方删除这个所谓的“特性”。

A.2 窗口函数

弄明白 SQL 分组概念和聚合函数的用法后，窗口函数理解起来就很容易了。与聚合函数一样，窗口函数也对指定的行集（分组）执行聚合操作，但不是为每个分组返回一个值，而是返回多个值。要对其进行聚合的行分组被称为窗口。对于这样的函数，DB2 称之为联机分析处理（OLAP）函数，Oracle 称之为分析函数，而 ISO SQL 称之为窗口函数，因此本书使用此术语。

A.2.1 一个简单示例

假设要计算所有部门的员工总数，传统的做法是对整张 EMP 表执行一个 COUNT(*) 查询。

```
select count(*) as cnt
  from emp

  CNT
-----
   14
```

这很容易，但你经常需要在并不表示聚合或表示其他聚合的行中显示这种聚合数据，窗口函数能够轻松地解决这种问题。例如，下面的查询演示了如何使用窗口函数在细节行（每位员工一行）中显示聚合数据（员工总数）。

```
select ename,
       deptno,
       count(*) over() as cnt
  from emp
 order by 2

ENAME      DEPTNO      CNT
-----
CLARK       10         14
KING        10         14
```

MILLER	10	14
SMITH	20	14
ADAMS	20	14
FORD	20	14
SCOTT	20	14
JONES	20	14
ALLEN	30	14
BLAKE	30	14
MARTIN	30	14
JAMES	30	14
TURNER	30	14
WARD	30	14

这里调用的窗口函数为 **COUNT(*) OVER()**，其中的关键字 **OVER** 指出 **COUNT** 为窗口函数，而不是聚合函数。一般而言，**SQL** 标准允许将所有聚合函数都用作窗口函数，并通过关键字 **OVER** 来区分这两种用法。

那么，在上面的查询中，窗口函数 **COUNT(*) OVER()** 到底做了什么呢？在查询返回的每一行中，它都返回表中的总行数。**OVER** 后面的空括号表明，可以在其中使用额外的子句来指定窗口函数考虑的范围。在没有指定任何子句的情况下，窗口函数考虑的是结果集中所有的行，这就是在前面的输出中，每一行都显示 14 的原因。

你可能开始意识到窗口函数的强大威力了，这种威力使你能够在同一行中执行多级聚合。随着往下阅读，你将发现这种功能很有用。

A.2.2 执行顺序

深入探讨 **OVER** 子句前，需要指出的是，窗口函数是 **ORDER BY** 子句执行前 **SQL** 处理的最后一步。为了证明这一点，在上一节的查询中添加一个 **WHERE** 子句，将 20 号部门的员工和 30 号部门的员工排除在外。

```

select  ename,
        deptno,
        count(*) over() as cnt
  from emp
 where deptno = 10
 order by 2

```

ENAME	DEPTNO	CNT
CLARK	10	3
KING	10	3
MILLER	10	3

现在，每一行的 **CNT** 值都不再是 14，而是 3。在这个示例中，**WHERE** 子句将结果集限制为 3 行，因此窗口函数计算得到的行数为 3。（执行到这个查询的 **SELECT** 部分时，提供给窗口函数的只有 3 行。）从以上示例可知，窗口函数是在 **WHERE** 和 **GROUP BY** 等子句执行完毕后才执行其计算的。

A.2.3 分区

可以使用 **PARTITION BY** 子句指定要对其执行聚合的分区（行组）。正如前面看到的，如果括号是空的，那么窗口函数将把整个结果集作为要聚合的分组。可以将 **PARTITION BY** 子句视为“移动 **GROUP BY**”，因为与传统的 **GROUP BY** 不同，**PARTITION BY** 创建的分组在结果集中并不是独一无二的。可以使用 **PARTITION BY** 来聚合指定的行组（遇到新组后将重置），并返回每一个值，而不是返回表示所有相应值实例的分组。请看下面的查询。

```

select  ename,
        deptno,
        count(*) over(partition by deptno) as cnt
  from emp
 order by 2

```

ENAME	DEPTNO	CNT
CLARK	10	3
KING	10	3
MILLER	10	3

CLARK	10	3
KING	10	3
MILLER	10	3
SMITH	20	5
ADAMS	20	5
FORD	20	5
SCOTT	20	5
JONES	20	5
ALLEN	30	6
BLAKE	30	6
MARTIN	30	6
JAMES	30	6
TURNER	30	6
WARD	30	6

这个查询依然返回 14 行，但 **COUNT** 现在聚合了各个部门，这是由 **PARTITION BY DEPTNO** 子句指定的。同一个部门（分区）的每位员工的 **CNT** 值都相同，因为遇到新部门后，才会重置（重新计算）聚合结果。注意，返回的是每个分组的信息，以及每个分组的成员。上述查询与如下查询等效，但效率更高。

<pre>select e.ename, e.deptno, (select count(*) from emp d where e.deptno=d.deptno) as cnt from emp e order by 2</pre>		
ENAME	DEPTNO	CNT
-----	-----	-----
CLARK	10	3
KING	10	3
MILLER	10	3
SMITH	20	5
ADAMS	20	5
FORD	20	5
SCOTT	20	5
JONES	20	5
ALLEN	30	6
BLAKE	30	6
MARTIN	30	6
JAMES	30	6

TURNER	30	6
WARD	30	6

另外，**PARTITION BY** 子句的优点在于，它以独立于 **SELECT** 语句中其他窗口函数的方式执行其计算：根据不同的列进行分区。请看下面的查询，它返回了每位员工的名字、所属部门、所属部门的员工数量、职位以及具有相同职位的员工数量。

<pre>select ename, deptno, count(*) over(partition by deptno) as dept_cnt, job, count(*) over(partition by job) as job_cnt from emp order by 2</pre>				
ENAME	DEPTNO	DEPT_CNT	JOB	JOB_CNT
MILLER	10	3	CLERK	4
CLARK	10	3	MANAGER	3
KING	10	3	PRESIDENT	1
SCOTT	20	5	ANALYST	2
FORD	20	5	ANALYST	2
SMITH	20	5	CLERK	4
JONES	20	5	MANAGER	3
ADAMS	20	5	CLERK	4
JAMES	30	6	CLERK	4
MARTIN	30	6	SALESMAN	4
TURNER	30	6	SALESMAN	4
WARD	30	6	SALESMAN	4
ALLEN	30	6	SALESMAN	4
BLAKE	30	6	MANAGER	3

从这个结果集可知，所属部门相同的员工的 **DEPT_CNT** 值相同，职位相同的员工的 **JOB_CNT** 值相同。

至此，你应该清楚地知道，**PARTITION BY** 子句的工作原理与 **GROUP BY** 子句类似，但不受 **SELECT** 子句中其他元素的影响，且不要求查询中包含 **GROUP BY** 子句。

A.2.4 NULL的影响

与 **GROUP BY** 子句一样，**PARTITION BY** 子句也将所有 **NULL** 值编入一个分组（分区）。因此，使用 **PARTITION BY** 时，**NULL** 所带来的影响与使用 **GROUP BY** 时类似。下面的查询使用了一个窗口函数来计算业务提成相同的员工数量（为了提高可读性，将 **NULL** 替换成了-1）。

```
select coalesce(comm,-1) as comm,  
       count(*)over(partition by comm) as cnt  
from emp
```

COMM	CNT
0	1
300	1
500	1
1400	1
-1	10
-1	10
-1	10
-1	10
-1	10
-1	10
-1	10
-1	10
-1	10
-1	10
-1	10

由于使用的是 **COUNT(*)**，因此计算的是行数。从结果集可知，有 10 位员工的业务提成为 **NULL**。然而，如果将 **COUNT(*)** 替换为 **COUNT(COMM)**，那么结果将截然不同。

```
select coalesce(comm,-1) as comm,  
       count(comm)over(partition by comm) as cnt  
from emp
```

COMM	CNT
0	1

300	1
500	1
1400	1
-1	0
-1	0
-1	0
-1	0
-1	0
-1	0
-1	0
-1	0
-1	0
-1	0
-1	0

这个查询使用的是 `COUNT(COMM)`，这意味着只计算 `COMM` 列不是 `NULL` 的行数。有一位员工的业务提成为 0，一位员工的业务提成为 300，等等。但请注意业务提成为 `NULL` 的员工数量，结果为 0。怎么会这样呢？因为聚合函数忽略了 `NULL` 值，更准确地说是聚合函数只计算非 `NULL` 值的个数。



使用 `COUNT` 时，先想想是否要将 `NULL` 值包含在内。如果不想将 `NULL` 值包含在内，就使用 `COUNT(column)`；如果想将 `NULL` 值包含在内，就使用 `COUNT(*)`（因为它计算的不是列值个数，而是行数）。

A.2.5 排列顺序很重要时

在有些情况下，窗口函数以什么样的顺序处理行将影响查询返回的结果。有鉴于此，窗口函数支持使用 `ORDER BY` 子句，你可以将其放在 `OVER` 子句中。`ORDER BY` 子句指定了分区内行的排列顺序。（别忘了，在没有 `PARTITION BY` 子句的情况下，“分区”为整个结果集。）



有些窗口函数要求指定分区中行的排列顺序，因此对这些窗口函数来说，`ORDER BY` 子句必不可少。本书撰写之

时，SQL Server 不允许在聚合窗口函数的 **OVER** 子句中使用 **ORDER BY**，但允许在窗口排名函数的 **OVER** 子句中使用 **ORDER BY**。

在窗口函数的 **OVER** 子句中使用 **ORDER BY** 子句时，对如下两方面做了规定：

- 如何对分区中的行进行排序；
- 计算时将考虑哪些行。

请看下面的查询，它会计算 10 号部门员工的移动薪水总计。

```
select deptno,
       ename,
       hiredate,
       sal,
       sum(sal)over(partition by deptno) as total1,
       sum(sal)over() as total2,
       sum(sal)over(order by hiredate) as running_total
from emp
where deptno=10
```

DEPTNO	ENAME	HIREDATE	SAL	TOTAL1	TOTAL2	RUNNING_TOTAL
10	CLARK	09-JUN-1981	2450	8750	8750	2450
10	KING	17-NOV-1981	5000	8750	8750	7450
10	MILLER	23-JAN-1982	1300	8750	8750	8750



为了让你保持警惕，上面的查询中使用了 **SUM(SAL) OVER()**。注意，**TOTAL1** 和 **TOTAL2** 的值相同，这是为什么呢？原因在于窗口函数的执行顺序。**WHERE** 子句会对结果集进行筛选，使得计算总计时只考虑 10 号部门的员工的薪水。在这里，只有一个分区，那就是整个结果集，它只包含 10 部门的员工的薪水，因此 **TOTAL1** 和 **TOTAL2** 的值相同。

通过查看 **SAL** 列的值，很容易弄明白 **RUNNING_TOTAL** 列的值是怎么来的：可以将 **SAL** 列的值相加，来得到移动总计。但更重要的是，在生成移动总计的 **OVER** 子句中，为什么要包含一个 **ORDER BY** 子句呢？这是因为在 **OVER** 子句中使用 **ORDER BY** 时，在分区中指定了一个默认的“移动”窗口或“滑动”窗口，虽然你看不到它。**ORDER BY HIREDATE** 子句让汇总计算在遇到当前行的 **HIREDATE** 后结束。

下面的查询与前面的查询等效，但使用（稍后将介绍的）**RANGE BETWEEN** 子句显式地指定了 **ORDER BY HIREDATE** 的默认行为。

```
select deptno,
       ename,
       hiredate,
       sal,
       sum(sal)over(partition by deptno) as total1,
       sum(sal)over() as total2,
       sum(sal)over(order by hiredate
                    range between unbounded preceding
                           and current row) as running_total
from emp
where deptno=10
```

DEPTNO	ENAME	HIREDATE	SAL	TOTAL1	TOTAL2	RUNNING_TOTAL
10	CLARK	09-JUN-1981	2450	8750	8750	2450
10	KING	17-NOV-1981	5000	8750	8750	7450
10	MILLER	23-JAN-1982	1300	8750	8750	8750

在 **ANSI** 中，这个查询使用的 **RANGE BETWEEN** 子句被称为框架子句（**framing clause**），本书将使用这个术语。至此，你应该很容易明白在 **OVER** 子句中指定 **ORDER BY** 时，将生成移动总计的原因：默认地让查询计算当前行和之前所有行的总计（**ORDER BY** 指定了确定“之前”行的依据，在这里，行是根据 **HIREDATE** 排列的）。

A.2.6 框架子句

下面来执行前面查询中的框架子句，从聘请的第一位员工 CLARK 开始。

1. 首先获取 CLARK 的薪水 2450，并将在 CLARK 之前聘请的所有员工都包含进来，然后计算总计。由于在 10 号部门 CLARK 是最先聘请的员工，因此总计为 CLARK 的薪水 2450，这是 `RUNNING_TOTAL` 返回的第一个值。
2. 转到根据 `HIREDATE` 排在第二位的员工 KING，并再次执行这个框架子句。从当前行的薪水值（KING 的薪水值）5000 开始，将之前的所有行（在 KING 之前聘请的所有员工）都包含进来，并计算 `SAL` 总计。在 KING 之前聘请的员工只有 CLARK，因此总计为 $5000 + 2450$ （7450），这是 `RUNNING_TOTAL` 返回的第二个值。
3. 转到 MILLER，即根据 `HIREDATE` 排列时位于分区最后的员工，并再次执行这个框架子句。从当前行的薪水（MILLER 的薪水）1300 开始，将之前的所有行（在 MILLER 之前聘请的所有员工）都包含进来，并计算 `SAL` 总计。CLARK 和 KING 都是在 MILLER 之前聘请的，因此需要将他们的薪水包含在 MILLER 的 `RUNNING_TOTAL` 中： $2450 + 5000 + 1300$ （8750，这是 `RUNNING_TOTAL` 给 MILLER 返回的值）。

如你所见，移动总计实际上是由框架子句生成的。`ORDER BY` 指定了计算顺序，同时指定使用默认框架。

一般而言，框架子句能够指定要在计算中包含的数据子窗口。指定这种子窗口的方式有很多，请看下面的查询。

```
select deptno,
       ename,
       sal,
       sum(sal)over(order by hiredate
```

```

        range between unbounded preceding
        and current row) as run_total1,
sum(sal)over(order by hiredate
        rows between 1 preceding
        and current row) as run_total2,
sum(sal)over(order by hiredate
        range between current row
        and unbounded following) as run_total3,
sum(sal)over(order by hiredate
        rows between current row
        and 1 following) as run_total4

from emp
where deptno=10

```

DEPTNO	ENAME	SAL	RUN_TOTAL1	RUN_TOTAL2	RUN_TOTAL3	RUN_TOTAL4
10	CLARK	2450	2450	2450	8750	7450
10	KING	5000	7450	7450	6300	6300
10	MILLER	1300	8750	6300	1300	1300

可别被这个查询吓到，它并没有看起来那么恐怖，其中的 **RUN_TOTAL1** 你已经见过，框架子句 **UNBOUNDED PRECEDING AND CURRENT ROW** 的作用你也明白。下面简要描述一下其他的框架子句。

RUN_TOTAL2

此框架子句指定的不是关键字 **RANGE**，而是 **ROWS**，这意味着我们将通过计算行数的方式来构建框架（窗口）。**1 PRECEDING** 意味着框架将从当前行的前一行开始，再延伸到当前行（**CURRENT ROW**）。因此，**RUN_TOTAL2** 返回的是当前员工的薪水与其前一位员工（这是根据 **HIREDATE** 确定的）的薪水之和。

```
[[sqlckbk-APP-A-NOTE-11]]
```



对于员工 CLARK 和 KING，**RUN_TOTAL1** 和

RUN_TOTAL2 的值相同。为什么会这样呢？请想一想，对于这两位员工，这两个窗口函数计算的是哪些值的和。只要仔细想想，你就会明白的。

RUN_TOTAL3

与生成 **RUN_TOTAL1** 的窗口函数相反，生成 **RUN_TOTAL3** 的窗口函数不是从当前行开始、将之前的所有行都包含进来再计算总计，而是从当前行开始、将后面的所有行都包含进来并计算总计。

RUN_TOTAL4

与 **RUN_TOTAL2** 相反，**RUN_TOTAL4** 不是从当前行开始、将前一行包含进来再计算总计，而是从当前行开始、将后一行包含进来并计算总计。



只要能够理解前面阐述的内容，就能弄明白本书的所有实例。但如果你还没弄明白，可以尝试使用自己的数据编写这种查询。面对新功能，与阅读使用它的代码相比，动手编写使用它的代码将更容易学会。

A.2.7 最后一个框架子句示例

来看最后一个有关框架子句如何影响输出的示例，如下面的查询所示。

```
select ename,  
       sal,  
       min(sal)over(order by sal) min1,  
       max(sal)over(order by sal) max1,  
       min(sal)over(order by sal
```



```

        range between unbounded preceding
        and unbounded following) min2,
max(sal)over(order by sal
        range between unbounded preceding
        and unbounded following) max2,
min(sal)over(order by sal
        range between current row
        and current row) min3,
max(sal)over(order by sal
        range between current row
        and current row) max3,
max(sal)over(order by sal
        rows between 3 preceding
        and 3 following) max4
from emp

```

ENAME	SAL	MIN1	MAX1	MIN2	MAX2	MIN3	MAX3	MAX4
SMITH	800	800	800	800	5000	800	800	1250
JAMES	950	800	950	800	5000	950	950	1250
ADAMS	1100	800	1100	800	5000	1100	1100	1300
WARD	1250	800	1250	800	5000	1250	1250	1500
MARTIN	1250	800	1250	800	5000	1250	1250	1600
MILLER	1300	800	1300	800	5000	1300	1300	2450
TURNER	1500	800	1500	800	5000	1500	1500	2850
ALLEN	1600	800	1600	800	5000	1600	1600	2975
CLARK	2450	800	2450	800	5000	2450	2450	3000
BLAKE	2850	800	2850	800	5000	2850	2850	3000
JONES	2975	800	2975	800	5000	2975	2975	5000
SCOTT	3000	800	3000	800	5000	3000	3000	5000
FORD	3000	800	3000	800	5000	3000	3000	5000
KING	5000	800	5000	800	5000	5000	5000	5000

下面详细解读一下这个查询。

MIN1

生成该列的窗口函数没有指定框架子句，因此将使用默认框架子句 **UNBOUNDED PRECEDING AND CURRENT ROW**。为什么在所有的行中，**MIN1** 的值都为 800 呢？这是因为薪水最低的员工排在最前面（**ORDER BY SAL**），且该薪水始终是最低的。

MAX1

MAX1 的值与 **MIN1** 的值有天壤之别，这是为什么呢？依然是因为框架子句默认为 **UNBOUNDED PRECEDING AND CURRENT ROW**。这个框架子句连同 **ORDER BY SAL**，使得 **MAX1** 的值与当前行的 **SAL** 值相同。

请看表示员工 **SMITH** 的第一行。通过将 **SMITH** 的薪水同之前的所有薪水比较，以确定 **SMITH** 的 **MAX1** 值时，结果为 **SMITH** 的薪水，因为之前没有任何薪水值。转到表示 **JAMES** 的下一行，将 **JAMES** 的薪水与之前的所有薪水（**SMITH** 的薪水）比较时，结果是 **JAMES** 的薪水更高，因此 **MAX1** 的值为 **JAMES** 的薪水。如果将这种逻辑应用于所有行，将发现每行的 **MAX1** 值都是该行所表示的员工的薪水。

MIN2 和 MAX2

生成这两列时，指定的框架子句为 **UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING**，这与指定空括号等效。因此，计算 **MIN2** 和 **MAX2** 时，将考虑结果集中的所有行。正如你所预期的，整个结果集的 **MIN** 值和 **MAX** 值为常量，因此这两列的值也为常量。

MIN3 和 MAX3

生成这两列时，指定的框架子句为 **CURRENT ROW AND CURRENT ROW**，这意味着计算 **MIN3** 和 **MAX3** 的值时，只考虑当前员工的薪水。因此，在每一行中，**MIN3** 和 **MAX3** 的值都与 **SAL** 的值相同。这很容易理解，不是吗？

MAX4

生成此列时，指定的框架子句为 **3 PRECEDING AND 3**

FOLLOWING，这意味着对于每一行，都将考虑它前面 3 行、后面 3 行以及它自己，因此这个 **MAX(SAL)** 调用将返回这些行中的最高薪水值。

只要看看员工 **MARTIN** 的 **MAX4** 值，就能明白这个框架子句是如何起作用的。**MARTIN** 的薪水为 1250，他前面的 3 位员工为 **WARD**、**ADAMS** 和 **JAMES**，这些员工的薪水分别为 1250、1100 和 950。**MARTIN** 后面的 3 位员工为 **MILLER**、**TURNER** 和 **ALLEN**，这些员工的薪水分别为 1300、1500 和 1600。在包括 **MARTIN** 在内的所有这些员工中，薪水最高的是 **ALLEN**，因此 **MARTIN** 的 **MAX4** 值为 1600。

A.2.8 可读性 + 性能 = 威力强大

如你所见，窗口函数威力极其强大，使你能够编写同时返回明细信息和聚合信息的查询。与使用多个自连接和 / 或标量子查询相比，使用窗口函数编写的查询更紧凑且效率更高。请看下面的查询，它轻而易举地回答了如下问题：各个部门都有多少位员工？在每个部门中，各种职位的员工分别有多少位（比如 10 号部门有多少位文员）？**EMP** 表中总共有多少位员工？

<pre>select deptno, job, count(*) over (partition by deptno) as emp_cnt, count(job) over (partition by deptno,job) as job_cnt, count(*) over () as total from emp</pre>				
DEPTNO	JOB	EMP_CNT	JOB_CNT	TOTAL

10	CLERK	3	1	14
10	MANAGER	3	1	14
10	PRESIDENT	3	1	14
20	ANALYST	5	2	14
20	ANALYST	5	2	14
20	CLERK	5	2	14
20	CLERK	5	2	14

20	MANAGER	5	1	14
30	CLERK	6	1	14
30	MANAGER	6	1	14
30	SALESMAN	6	4	14
30	SALESMAN	6	4	14
30	SALESMAN	6	4	14
30	SALESMAN	6	4	14

如果要在不使用窗口函数的情况下返回这个结果集，则需要编写的代码将更多些。

```

select a.deptno, a.job,
       (select count(*) from emp b
        where b.deptno = a.deptno) as emp_cnt,
       (select count(*) from emp b
        where b.deptno = a.deptno and b.job = a.job) as job_cnt,
       (select count(*) from emp) as total
  from emp a
 order by 1,2

```

DEPTNO	JOB	EMP_CNT	JOB_CNT	TOTAL
10	CLERK	3	1	14
10	MANAGER	3	1	14
10	PRESIDENT	3	1	14
20	ANALYST	5	2	14
20	ANALYST	5	2	14
20	CLERK	5	2	14
20	CLERK	5	2	14
20	MANAGER	5	1	14
30	CLERK	6	1	14
30	MANAGER	6	1	14
30	SALESMAN	6	4	14
30	SALESMAN	6	4	14
30	SALESMAN	6	4	14
30	SALESMAN	6	4	14

显然，不使用窗口函数的解决方案编写起来不难，但肯定不如使用窗口函数的版本那么清晰，效率也更低。（对于只包含 14 行数据的表，性能没有明显的差异，但如果一张表包含 1000 行数据乃至 10 000 行数据，那么与使用多个自连接和标量子查询

相比，使用窗口函数的优势将会显现出来。)

A.2.9 打下基础

除了可读性和性能方面的优势，在为复杂的报表型查询打下基础方面，窗口函数也很有用。例如，请看下面的报表型查询，它在一个内嵌视图中使用了窗口函数，然后在一个外部查询中聚合结果。使用窗口函数，可以同时返回明细数据和聚合数据，这在制作报表时很有用。下面的报表使用了窗口函数来计算不同分区的计数。由于聚合被应用于多行，因此内嵌视图会返回 **EMP** 表中的所有行，供外部 **CASE** 表达式来执行转置并创建格式良好的报表。

```
select deptno,
       emp_cnt as dept_total,
       total,
       max(case when job = 'CLERK'
                then job_cnt else 0 end) as clerks,
       max(case when job = 'MANAGER'
                then job_cnt else 0 end) as mgrs,
       max(case when job = 'PRESIDENT'
                then job_cnt else 0 end) as prez,
       max(case when job = 'ANALYST'
                then job_cnt else 0 end) as anals,
       max(case when job = 'SALESMAN'
                then job_cnt else 0 end) as smen
  from (
select deptno,
       job,
       count(*) over (partition by deptno) as emp_cnt,
       count(job) over (partition by deptno,job) as job_cnt,
       count(*) over () as total
  from emp
 ) x
 group by deptno, emp_cnt, total
```

DEPTNO	DEPT_TOTAL	TOTAL	CLERKS	MGRS	PREZ	ANALS	SMEN
10	3	14	1	1	1	0	0
20	5	14	2	1	0	2	0

这个查询返回了每个部门、每个部门的员工数量、EMP 表中的员工总数以及每个部门不同职位的员工数。所有这些都是在一个查询中完成的，没有使用连接，也没有使用临时表。

作为使用窗口函数可以轻松回答的多个问题的最后一个示例，请看下面的查询。

<pre>select ename as name, sal, max(sal)over(partition by deptno) as hiDpt, min(sal)over(partition by deptno) as loDpt, max(sal)over(partition by job) as hiJob, min(sal)over(partition by job) as loJob, max(sal)over() as hi, min(sal)over() as lo, sum(sal)over(partition by deptno order by sal,empno) as dptRT, sum(sal)over(partition by deptno) as dptSum, sum(sal)over() as ttl from emp order by deptno,dptRT</pre>										
NAME	SAL	HIDPT	LODPT	HIJOB	LOJOB	HI	LO	DPTRT	DPTSUM	TTL
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----
MILLER	1300	5000	1300	1300	800	5000	800	1300	8750	29025
CLARK	2450	5000	1300	2975	2450	5000	800	3750	8750	29025
KING	5000	5000	1300	5000	5000	5000	800	8750	8750	29025
SMITH	800	3000	800	1300	800	5000	800	800	10875	29025
ADAMS	1100	3000	800	1300	800	5000	800	1900	10875	29025
JONES	2975	3000	800	2975	2450	5000	800	4875	10875	29025
SCOTT	3000	3000	800	3000	3000	5000	800	7875	10875	29025
FORD	3000	3000	800	3000	3000	5000	800	10875	10875	29025
JAMES	950	2850	950	1300	800	5000	800	950	9400	29025
WARD	1250	2850	950	1600	1250	5000	800	2200	9400	29025
MARTIN	1250	2850	950	1600	1250	5000	800	3450	9400	29025
TURNER	1500	2850	950	1600	1250	5000	800	4950	9400	29025
ALLEN	1600	2850	950	1600	1250	5000	800	6550	9400	29025
BLAKE	2850	2850	950	2975	2450	5000	800	9400	9400	29025

这个查询轻松、高效地回答了下面的问题，可读性强且没有连接到 **EMP** 表。只需将员工同结果集中不同行的薪水关联起来，就可确定：

- 在所有员工中，谁的薪水最高（**HI**）；
- 在所有员工中，谁的薪水最低（**LO**）；
- 谁的薪水在整个部门中最低（**HIDPT**）；
- 谁的薪水在整个部门中最高（**LODPT**）；
- 谁的薪水在担任相同职位的所有员工中最高（**HIJOB**）；
- 谁的薪水在担任相同职位的所有员工中最低（**LOJOB**）；
- 所有员工的薪水总计是多少（**TTL**）；
- 各个部门的薪水总计是多少（**DPTSUM**）；
- 各个部门的移动薪水总计是什么样的（**DPTRT**）。

附录 B 通用表表达式

本书的很多查询无法使用典型的数据库表来实现，尤其在涉及聚合函数和窗口函数时。因此，在有些查询中，必须创建一张派生表——子查询或通用表表达式（CTE）。

B.1 子查询

如果想创建虚拟表，以便对其执行包含窗口函数或聚合函数的查询，那么最简单的做法无疑是使用子查询。为此，只需编写一个查询并将其放在括号内，然后再编写另一个使用它的查询。下面的代码演示了如何使用子查询来实现双重聚合：你想找出每种职位的员工数量，然后再找出其中最大的员工数量，但在标准查询中，不能嵌套聚合函数。

```
select max(HeadCount) as HighestJobHeadCount from
(select job,count(empno) as HeadCount
from emp
group by job) head_count_tab
```

使用子查询时需要注意的一点是，有些 RDBMS 要求给子查询表指定别名，有些则不要求这样做。上面的示例使用的是 MySQL 语法，必须给子查询表指定别名。（这里的别名为 HEAD_COUNT_TAB，是在右括号后面指定的。）

PostgreSQL 和 SQL Server 也要求给子查询表指定别名，但 Oracle 不要求这样做。

B.2 通用表表达式

为了克服子查询的一些局限性，我们引入了 CTE，但它最著名的功能是能够编写 SQL 递归查询。实际上，引入 CTE 的主要目的是让 SQL 支持递归。

以下示例的结果与前面的子查询版本相同，即执行双重聚合。

```
with head_count_tab (job,HeadCount) as

(select job,count(empno)
from emp
group by job)

select max(HeadCount) as HighestJobHeadCount
from head_count_tab
```

以上查询解决的问题虽然简单，但演示了 CTE 的重要特性。我们使用 **WITH** 子句引入派生表（在括号内指定列名），并将生成派生表的查询放在括号内。如果想同时定义多个派生表，则可以将它们用逗号分隔，并在生成派生表的查询前面指定派生表的名称（这与在 SQL 中指定别名的方式相反）。

由于内部查询是在外部查询之前定义的，因此在很多情况下，CTE 的可读性更高，让你能够更容易研究查询中的各个元素，进而弄明白查询的逻辑。当然，与编码的其他方面一样，CTE 的可读性是否更高取决于具体情况，在有些情况下，子查询的可读性更高。

鉴于引入 CTE 的主要目的是支持递归，因此演示 CTE 功能的最佳做法是进行递归查询。

下面的查询使用了递归 CTE 来计算前 20 个斐波那契数。注意，在 **anchor** 查询的前半部分，可以初始化虚拟表第一行的

值。

```
with recursive workingTable (fibNum, NextNumber, index1)
as
(select 0,1,1
union all
select fibNum+nextNumber,fibNum,index1+1
from anchor
where index1<20)

select fibNum from workingTable as fib
```

下一个斐波那契数为当前斐波那契数与前一个斐波那契数之和。为执行这种计算，可以使用 **LAG**，但这里使用了表示当前斐波那契数与前一个斐波那契数的两列来模拟 **LAG**。请注意关键字 **RECURSIVE**，它在 MySQL、Oracle 和 PostgreSQL 中必不可少，但在 SQL Server 和 DB2 中是可选的。在这个查询中，**index1** 列显得有点儿多余，因为计算斐波那契数时并没有用到它。这里包含它旨在简化在 **WHERE** 子句中指定返回行数的工作。在递归 CTE 中，**WHERE** 子句至关重要，如果没有它，查询将不会终止。（但在这个示例中，如果将这个子句删除，那么 RDBMS 很可能在数字大到超出了相应数据类型的范围时引发溢出错误。）

从功能的角度说，最简单的 CTE 与子查询差别不大，它们都能够编写引用派生表的复杂查询。然而，嵌套大量子查询时，其可读性将急剧下降，因为在后续查询层中，各种变量的含义将晦涩不明。相比之下，CTE 垂直地排列各个元素，让你更容易弄明白这些元素的含义。

B.3 小结

派生表的使用极大地拓展了 SQL 的功能。本书的很多地方使用了子查询和 CTE，因此弄明白它们的工作原理至关重要，同时你必须掌握它们的独特语法，这样才能成功地使用它们。当前，本书涉及的 RDBMS 都支持递归 CTE，它提供了众多额外的可能性，是最重要的 SQL 扩展之一。

关于作者

安东尼·莫利纳罗（**Anthony Molinaro**），美国强生公司数据科学家，现任杨森公司研发部观察性医疗数据分析小组经理。主要研究领域包括非参数方法、时序分析及大型数据库特征化与变换。安东尼是 OHDSI 开源科学社区的成员，拥有纽约城市大学亨特学院数学学士学位以及应用数学和统计学硕士学位。他与妻子和两个女儿居住在美国纽约。

罗伯特·德·格拉夫（**Robert de Graaf**），大学毕业后担任制造行业工程师，其间发现了统计学在解决问题方面的强大威力，因此攻读了统计学硕士学位。自 2013 年起担任 RightShip 公司数据科学家。另著有 *Managing Your Data Science Projects*。

关于封面

本书封面上的动物是盾尾岩蜥（rougtail rock agama）。这种动物分布在埃及、土耳其、希腊和其他地中海周边国家，通常出现在气候为干旱或半干旱的石质山地和海滨地区。盾尾岩蜥属昼行性动物，常出没在岩石、树林、建筑及其他隐蔽且可以爬行的地方。

盾尾岩蜥一次产蛋 3~12 颗，成年体长 30~35 厘米。这种蜥蜴的特征是大腿强壮，同时与众多其他的飞龙科蜥蜴一样，能够根据情绪和环境温度改变颜色。无论雌雄，盾尾岩蜥的身体通常为灰色或棕色，背部和两侧有彩色斑点。不同于其他蜥蜴，盾尾岩蜥等飞龙科蜥蜴的尾巴掉后不能再生。

盾尾岩蜥虽然容易受惊，但一般不会攻击人，如果从小饲养，通常非常温驯。盾尾岩蜥常作为宠物饲养，可以投喂昆虫和各种绿叶植物。如果容器足够大，可以将为数不多的盾尾岩蜥一起饲养，但雄性必须分开，以免相互打斗。

盾尾岩蜥的数量比较稳定，世界自然保护联盟（IUCN）并未将其列为需要关注的物种，但 O'Reilly 图书封面上的许多动物濒临灭绝，它们对世界很重要。

封面插图来自 Karen Montgomery 的作品，根据来源不明的黑白活动版画而作。

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：
ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
 - 微博 @图灵社区：电子书和好文章的消息
 - 微博 @图灵新知：图灵教育的科普小组
 - 微信 图灵访谈：ituring_interview，讲述码农精彩人生
 - 微信 图灵教育：turingbooks
-