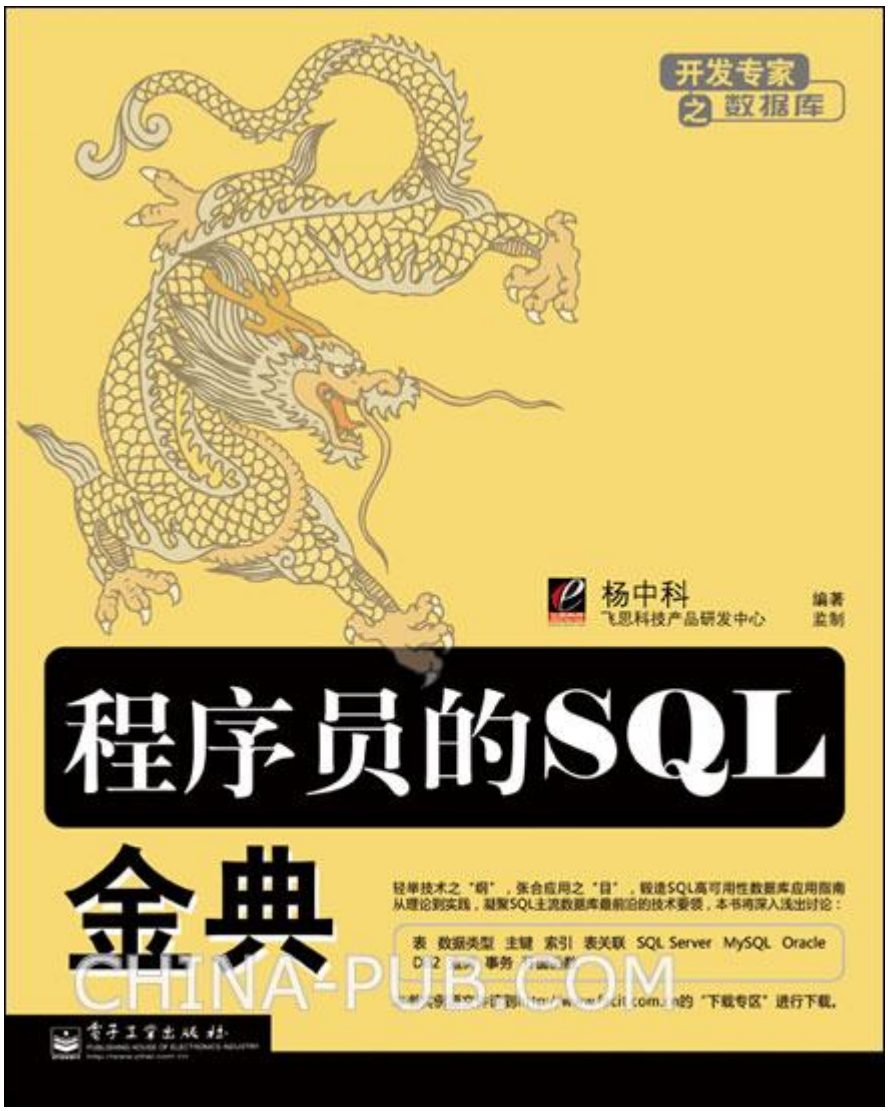
传智播客.net 培训内部使用，禁止外传

For Internal Only



作者 杨中科

如鹏网 www.rupeng.com

传智播客.net 培训 www.itcast.cn

第一章 数据库入门

本章介绍数据库的入门知识，首先介绍什么是数据库，然后介绍数据库中的一些基本概念，接着介绍操纵数据库的不同方式，最后介绍操纵数据库时使用的语言 SQL，在章节中我们还将穿插一些非常有趣的话题。

1.1 数据库概述

广义上来讲，数据库就是“数据的仓库”，计算机系统经常用来处理各种各样大量的数据，比如使用计算机系统收集一个地区的人口信息、检索符合某些条件的当地人口信息、当一个人去世后还要从系统中删除此人的相关信息。我们可以自定义一个文件格式，然后把人口数据按照这个格式保存到文件中，当需要对已经存入的数据进行检索或者修改的时候就重新读取这个文件然后进行相关操作。这种数据处理方式存在很多问题，比如需要开发人员熟悉操作磁盘文件的函数、开发人员必须编写复杂的搜寻算法才能快速的把数据从文件中检索出来、当数据格式发生变化的时候要编写复杂的文件格式升级程序、很难控制并发修改。

在计算机系统在各个行业开始普遍应用以后，计算机专家也遇到了同样的问题，因此他们提出了数据库理论，从而大大简化了开发信息系统的难度。数据库理论的鼻祖是 Charles W.Bachman，他也因此获得了 1973 年的图灵奖。IBM 的 Ted Codd 则首先提出了关系数据库理论，并在 IBM 研究机构开发原型，这个项目就是 R 系统，并且使用 SQL 做为存取数据表的语言，R 系统对后来的 Oracle、Ingres 和 DB2 等关系型数据库系统都产生了非常重要的影响。

1.1.1 “数据库”与“数据库管理系统”

前面我们讲到数据库就是“数据的仓库”，我们还需要一套系统来帮助我们管理这些数据，比如帮助我们查询到我们需要的数据、帮我们将过时的数据删除，这样的系统我们称之为数据库管理系统（Database Management System，DBMS）。有时候很多人也将 DBMS 简称为“数据库”，但是一定要区分“数据库”的这两个不同的意思。

数据库管理系统是一种操纵和管理数据库的系统软件，是用于建立、使用和维护数据库。它对数据库进行统一的管理和控制，以保证数据库的安全性和完整性。用户通过 DBMS 访

问数据库中的数据，数据库管理员也通过 DBMS 进行数据库的维护工作。它提供多种功能，可使多个应用程序和用户用不同的方法在同时或不同时刻去建立，修改和询问数据库。它使用户能方便地定义和操纵数据，维护数据的安全性和完整性，以及进行多用户下的并发控制和恢复数据库。通俗的说，DBMS 就是数据库的大管家，需要维护什么数据、查找什么数据的话找它告诉他了，它会帮你办的干净利落。

1.1.2 数据库能做什么

数据库能够帮助你储存、组织和检索数据。数据库以一定的逻辑方式组织数据，当我们要对数据进行增删改查的时候数据库能非常快速的完成所要求的操作；同时数据库隐藏了数据的组织形式，我们只要对数据的属性进行描述就可以了，当我们要对数据库中的数据进行操作的时候只要告诉“做什么”（What to do）就可以了，DBMS 会决定一个比较好的完成操作的方式，也就是我们无需关心“怎么做”（How to do），这样我们就能从数据存储的底层中脱身出来，把更多精力投入到业务系统的开发中。

数据库允许我们创建规则，以确保在增加、更新以及删除数据的时候保证数据的一致性；数据库允许我们指定非常复杂的数据过滤机制，这样无论业务规则多么复杂，我们都能轻松应对；数据库可以处理多用户并发修改问题；数据库提供了操作的事务性机制，这样可以保证业务数据的万无一失。

1.1.3 主流数据库管理系统介绍

目前有许多 DBMS 产品，如 DB2、Oracle、Microsoft SQL Server、Sybase SQLServer、Informix、MySQL 等，它们在数据库市场上各自占有一席之地。下面简要介绍几种常用的数据库管理系统。

（1）DB2

DB2 第一种使用使用 SQL 的数据库产品。DB2 于 1982 年首次发布，现在已经可以用在许多操作系统平台上，它除了可以运行在 OS/390 和 VM 等大型机操作系统以及中等规模的 AS/400 系统之外，IBM 还提供了跨平台（包括基于 UNIX 的 LINUX，HP-UX，Sun Solaris，以及 SCO UnixWare；还有用于个人电脑的 Windows 2000 系统）的 DB2 产品。应用程序可以通过使用微软的 ODBC 接口、Java 的 JDBC 接口或者 CORBA 接口代理来访问 DB2 数据库。

DB2 有不同的版本，比如 DB2 Everyplace 是为移动用户提供的一个内存占用小且性能出色的版本；DB2 for z/OS 则是为主机系统提供的版本；Enterprise Server Edition(ESE)是一种适用于中型和大型企业的版本； Workgroup Server Edition(WSE)主要适用于小型和中型企业，它提供除大型机连接之外的所有 ESE 特性；而 DB2 Express 则是为开发人员提供的可以免费使用的版本。

IBM 是最早进行关系数据库理论研究和产品开发的公司，在关系数据库理论方面一直走在业界的前列，所以 DB2 的功能和性能都是非常优秀的，不过对开发人员的要求也比其他数据库系统更高，使用不当很容易造成宕机、死锁等问题；DB2 在 SQL 的扩展方面比较保守，很多其他数据库系统支持的 SQL 扩展特性在 DB2 上都无法使用；同时 DB2 对数据的类型要求也非常严格，在数据类型不匹配的时候会报错而不是进行类型转换，而且如果发生精度溢出、数据超长等问题的时候也会直接报错，这虽然保证了数据的正确性，但是也使得基于 DB2 的开发更加麻烦。因此，很多开发人员称 DB2 为“最难用的数据库系统”。

（2）Oracle

Oracle 是和 DB2 同时期发展起来的数据库产品，也是第二个采用 SQL 的数据库产品。Oracle 从 DB2 等产品中吸取到了很多优点，同时又避免了 IBM 的官僚体制与过度学术化，大胆的引进了许多新的理论与特性，所以 Oracle 无论是功能、性能还是可用性都是非常好的。

（3）Microsoft SQL Server

Microsoft SQL Server 是微软推出的一款数据库产品。细心的读者也许已经发现我们前面提到了另外一个名字非常相似的 Sybase SQLServer，这里的名字相似并不是一种巧合，这还要从 Microsoft SQL Server 的发展史谈起。

微软当初要进军图形化操作系统，所以就开始和 IBM“合作”开发 OS/2，最终当然无疾而终，但是微软就很快的推出了自己的新一代视窗操作系统；而当微软发现数据库系统这块新的市场的时候，微软没有自己重头开发一个数据库系统，而是找到了 Sybase 来“合作”开发基于 OS/2 的数据产品，当然微软达到目的以后就立即停止和 Sybase 的合作了，于 1995 年推出了自己的 Microsoft SQL Server6.0，经过几年的发展终于在 1998 年推出了轰动一时的 Microsoft SQL Server7.0，也正是这一个版本使得微软在数据库产品领域有了一席之地。正因为这段“合作”历史，所以使得 Microsoft SQL Server 和 Sybase SQLServer 在很多地方非常类似，比如底层采用的 TDS 协议、支持的语法扩展、函数等等。

微软在 2000 年推出了 Microsoft SQL Server 2000，这个版本继续稳固了 Microsoft SQL Server 的市场地位，由于 Windows 操作系统在个人计算机领域的普及，Microsoft SQL Server 理所当然的成为了很多数据库开发人员的接触的第一个而且有可能也是唯一一个数据库产品，很多人甚至在“SQL Server”和“数据库”之间划上了等号，而且用“SQL”一次来专指 Microsoft SQL Server，可见微软的市场普及做的还是非常好的。做足足够的市场以后，微软在 2005 年“审时度势”的推出了 Microsoft SQL Server 2005，并将于 2008 年发布新一代的 Microsoft SQL Server 2008。

Microsoft SQL Server 的可用性做的非常好，提供了很多了外围工具来帮助用户对数据库进行管理，用户甚至无需直接执行任何 SQL 语句就可以完成数据库的创建、数据表的创建、数据的备份/恢复等工作；Microsoft SQL Server 的开发者社区也是非常庞大的，因此有众多可以参考的学习资料，学习成本非常低，这是其他数据库产品做不具有的优势；同时从 Microsoft SQL Server 2005 开始开发人员可以使用任何支持.Net 的语言来编写存储过程，这进一步降低了 Microsoft SQL Server 的使用门槛。

不过正如微软产品的一贯风格，Microsoft SQL Server 的劣势也是非常明显的：只能运行于 Windows 操作系统，因此我们无法在 Linux、Unix 上运行它；不管微软给出什么样的测试数据，在实际使用中 Microsoft SQL Server 在大数据量和大交易量的环境中的表现都是不尽人意的，当企业的业务量到达一个水平后就要考虑升级到 Oracle 或者 DB2 了。

（4）MySQL

MySQL 是一个小型关系型数据库管理系统，开发者为瑞典 MySQL AB 公司。目前 MySQL 被广泛地应用在中小型系统中，特别是在网络应用中用户群更多。MySQL 没有提供一些中小型系统中很少使用的功能，所以 MySQL 的资源占用非常小，更加易于安装、使用和管理。

由于 MySQL 是开源的，所以在 PHP 和 Java 开发人员心中更是首选的数据库开发搭档，目前 Internet 上流行的网站构架方式是 LAMP（Linux+Apache+MySQL+PHP），即使用 Linux 作为操作系统，Apache 作为 Web 服务器，MySQL 作为数据库，PHP 作为服务器端脚本解释器。

MySQL 目前还很难用于支撑大业务量的系统，所以目前 MySQL 大部分还是用来运行非核心业务；同时由于 MySQL 在国内没有足够的技术支持力量，所以对 MySQL 的技术支持工作是由 ISV 或者系统集成商来承担，这也导致部分客户对 MySQL 比较抵制，他们更倾向于使用有更强技术支持力量的数据库产品。

1.2 数据库基础概念

要想使用数据库，我们必须熟悉一些基本概念，这些概念包括：Catalog、表、列、数

据类型、记录、主键以及表关联等等。

1.2.1 Catalog

数据库就是数据的仓库，而 DBMS 是数据库的“管理员”。一些企业即生产食品又生产农用物资，这些产品都要保存到仓库中，同时企业内部也有一些办公用品需要保存到仓库中。如果这些物品都保存到同一个仓库中的话会造成下面的问题：

- 不便于管理。食品的保存和复印纸的保存需要的保存条件是不同的，食品需要低温保鲜而复印纸则需要除湿，不同类的物品放在一起加大了管理的难度；
- 可能会造成货位冲突。食品要防止阳光直射造成的变质，因此要摆放到背阴面，同时为了防止受潮，也要把它们摆放到高处；办公用胶片也要避免阳光直射，所以同样要摆放到背阴面，而且胶片也要防潮，所以同样要把它们摆放到高处。这就造成两种货物占据的货位相冲突了。
- 会有安全问题。由于所有物品都放到一个仓库中没有进行隔离，所以来仓库领取办公用品的人员可能会顺手牵羊将食品偷偷带出仓库。

既然都是“仓库”，那么数据库系统也存在类似问题。如果企业将人力资源数据和核心业务数据都保存到一个数据库中同样会造成下面的问题：

- 不便于管理。为了防止数据丢失，企业需要对数据进行定期备份，不过和核心业务数据比起来人力资源数据的重要性要稍差，所以人力资源数据只要一个月备份一次就可以了，而核心业务数据则需要每天都备份。如果将这两种数据保存在一个数据库中会给备份工作带来麻烦。
- 可能会造成命名冲突。比如人力资源数据中需要将保存员工数据的表命名为 Persons，而核心业务数据也要将保存客户数据的表也命名为 Persons，这就会相冲突了。
- 会有数据安全问题。由于所有的数据都保存在一个数据库中，这样人力资源系统的用户也可以访问核心业务系统中的数据，很容易造成数据安全问题。

显而易见，对于上边提到的多种物品保存在一个仓库中的问题，最好的解决策略就是使用多个仓库，食品保存在食品仓库中，农用物资保存在农用物资仓库中，而办公用品则保存在办公用品仓库中，这样就可以解决问题了。问了解决同样的问题，DBMS 也采用了多数据库的方式来保存不同类别的数据，一个 DBMS 可以管理多个数据库，我们将人力资源数据保存在 HR 数据库中，而将核心业务数据保存在 BIZ 数据库中，我们将这些不同数据库叫做 Catalog（在有的 DBMS 中也称为 Database，即数据库）。采用多 Catalog 以后可以给我们带来如下好处：

- 便于对各个 Catalog 进行个性化管理。DBMS 都允许我们指定将不同的 Catalog 保存在不同的磁盘上，由于人力资源数据相对次要一些，因此我们可以将 HR 保存在普通硬盘上，而将 BIZ 保存在 RAID 硬盘上。我们还可以对每个 Catalog 所能占据的最大磁盘空间、日志大小甚至优先级进行指定，这样就可以针对不同的业务数据进行个性化定制了。
- 避免了命名冲突。同一个 Catalog 中的表名是不允许重复的，而不同 Catalog 中的表名则是可以重复的，这样 HR 中可以有 Persons 表，而 BIZ 中也可以有 Persons 表，二者结构可以完全不相同，保存的数据也不会互相干扰。
- 安全性更高。DBMS 允许为不同的 Catalog 指定不同的用户，并且可以限定用户能访问的 Catalog。比如用户 hr123 只能访问 HR，而用户 sales001 只能访问 BIZ。这就大大加强了系统数据的安全性。

1.2.2 表（Table）

虽然我们已经将不同用途的物品保存在不同的仓库中了，但是在同一个仓库中数据的保

存仍然存在问题。比如食品分为熟食、生肉、大米等，如果把他们随意的堆放在一起，就会造成我们无法很容易的对这些食品进行管理，当要对大米进行提货的话就必须在一堆的食品中翻来翻去。解决这个问题的方法就是将仓库划分为不同的区域，熟食保存在熟食区，生肉保存在生肉区，而大米则保存在大米区。

DBMS 中也存在类似的问题，虽然我们将核心业务数据保存在 BIZ 数据库中了，但是核心业务数据也有很多不同类型的数据，比如客户资料、商品资料、销售员资料等，如果将这些数据混杂在一起的话将会管理起来非常麻烦，比如我们要查询所有客户资料的话就必须将所有数据查询一遍。解决这个问题的方法就是将不同类型的资料放到不同的“区域”中，我们将这种区域叫做“表”（Table）。客户资料保存到 Customers 表中，将商品资料保存在 Goods 表中，而将销售员资料保存在 SalesMen 表中，这样当需要查找商品的时候只要到 Goods 表中查找就可以了。

1.2.3 列（Column）

同样是生肉，不同的生肉又有不同的特性，有的生肉是里脊肉，有的生肉是前臀尖，这块生肉是 18 公斤，而那块生肉是 12 公斤，这块生肉是 12.2 元/公斤，而那块生肉是 13.6 元/公斤。每块肉都有各自的不同的特性，这些特性包括取肉部位、重量、单价。如果不对每块肉标注这些特性数据的话，当提货人要我们将所有里脊肉出库的话我们就非常麻烦了。解决这个问题的方法就是制作一些标签，在这个标签上标明取肉部位、重量、单价，这样要提取货物就会非常方便了。

不仅如此，标签的格式也要统一，如果第一块生肉的标签内容是：

这块肉是 15.6 公斤的里脊肉，13.2 元/公斤。

另一块生肉的标签内容是：

每市斤 8.6 元，前臀尖，13.6 公斤的，。

采用这种标签由于没有统一的格式，所以阅读起来非常麻烦，要靠人工去分辨，错误率非常高。如果我们规定一个统一的标签格式，比如下面的标签：

取肉部位	
重量	
单价（元/公斤）	

这样每块肉的标签就可以按照这个格式来填写了：

取肉部位	里脊肉
重量	15.6
单价（元/公斤）	13.2

这种格式阅读起来非常方便，如果引入自动识别设备的话，甚至可以实现自动化的物品分拣。

在数据库的表中保存的数据也有类似问题，如果不规定格式的话，表中的数据也会非常

阅读，如果一个员工的资料在表中保存的内容为：

2003年5月入职，是产品开发部的，姓名马小虎。

另外一个员工的资料在表中保存的内容为：

王二小，技术支持部，入职是 2005 年 7 月。

通常，以这种不标准的格式保存造成数据十分混乱，想要从数据库中取出合适的数据仍然非常麻烦。为了解决这个问题，我们规定下面这种标准的格式：

姓名	
部门	
入职时间	

这里的“姓名”、“部门”和“入职时间”就被称为员工表的列（Column），有时候也叫做字段（Field），每个列描述了数据的一个特性。

1.2.4 数据类型（DataType）

上面我们为员工表规定了“姓名”、“部门”和“入职时间”三个列，这样只要按照这个格式进行数据填写就可以了，但是这里仍然有一个问题，那就是我们没法限定用户向表中填写什么数据，比如用户填写成下面的格式：

姓名	33
部门	12.3
入职时间	信息中心

显然姓名不应该为一个数字 33；不可能有一个名称为“12.3”的部门；入职时间更不可能是“信息中心”。因此我们必须规则每一列中填写的数据的格式：姓名必须填写汉字，最短 2 个汉字，最长 5 个汉字；部门必须填写“产品开发部”、“技术支持部”、“产品实施部”、“人力资源部”中的一个；入职时间必须填写为正确的时间格式。

这里就规定了各个列的数据类型（DataType），数据类型规定了一个列中能填写什么类型的数据，减少了不规范数据出现的几率。

除了可以对数据进行规范之外，数据类型还有下面的作用：

- 提高效率。对不同的数据赋予不同的类型能够使得数据库更好的对数据进行存储和管理，从而减少空间占用并且提供数据的访问速度。比如，如果将数字 123454321 以文本类型存储的话将会占用 9 字节的存储空间，而以整数类型保存的话将只需要占用 4 字节的存储空间。
- 能够确定对数据进行操作所需要的正确处理方式。比如如果是整数类型，那么 123+234 被解释为两个整数的加法运算，所以其结果是 357；如果是文本类型，那么 123+234 则会被解释为两个字符串的相连操作，所以其结果是 123234。

1.2.5 记录（Record）

记录有可以被称为行（Row），可以通俗的认为它是数据表中的一行数据。以员工表为

例，一个公司的员工表中的数据是这样的：

姓名	部门	入职时间
马小虎	产品开发部	2003 年 5 月 22 日
王二小	技术支持部	2005 年 7 月 17 日
白展堂	后勤部	1998 年 3 月 27 日
钱长贵	销售部	2001 年 3 月 3 日
李达最	后勤部	2005 年 11 月 11 日

这里每一行数据就代表一个员工的资料，这样的一行数据就叫做一条记录。表是由行和列组成的一张二维表，这就是关系数据库中最基本的数据模型。

1.2.6 主键（PrimaryKey）

员工表中的每一行记录代表了一个员工，一般员工的名字就能唯一标识这一个员工，但是名字也是有可能重复的，这时我们就要为每一名员工分配一个唯一的工号：

工号	姓名	部门	入职时间
001	马小虎	产品开发部	2003 年 5 月 22 日
002	王二小	技术支持部	2005 年 7 月 17 日
003	白展堂	后勤部	1998 年 3 月 27 日
004	钱长贵	销售部	2001 年 3 月 3 日
005	李达最	后勤部	2005 年 11 月 11 日
006	王二小	产品开发部	2005 年 3 月 22 日

这样就可以通过这个工号来唯一标识一名员工了。当老板下令说“把王二小提升为副总”的时候，我们就要问“公司有两个王二小，您要提升哪一个？”，老板可以说“技术支持部的王二小”，但是更好的方式，那就是说“提升工号为的 002 员工为副总”，因为只有 002 这个工号才能唯一标识一名员工。这里的“工号”被称为员工表的“主键”（PrimaryKey），所以我们可以说能唯一标识一行记录的字段就是此表的主键。

有的公司比较懒惰，不想为员工分配工号，只是硬性规定：一个部门中员工的姓名不能重复，有姓名重复的必须调换到其它部门。这样“部门”和“姓名”这两个字段加在一起就能唯一标识一名员工了，这里的“部门”和“姓名”两个字段就被称为“复合主键”，也就是任何一个字段都不能唯一标识一行数据，只有构成“复合主键”的所有字段组合起来才能唯一标识这一行数据。

在大多数 DBMS 中并没有强制规定一个表必须有主键，也就是一个表可以没有主键，但是为一个数据表指定一个主键是一个非常好的习惯。在后边的章节我们将提到用一个无意义的字段做主键将会更加有利于系统的可扩展性。

1.2.7 索引（Index）

无索引的表就是一个无序的行集。比如下面的人员表中有一些数据：

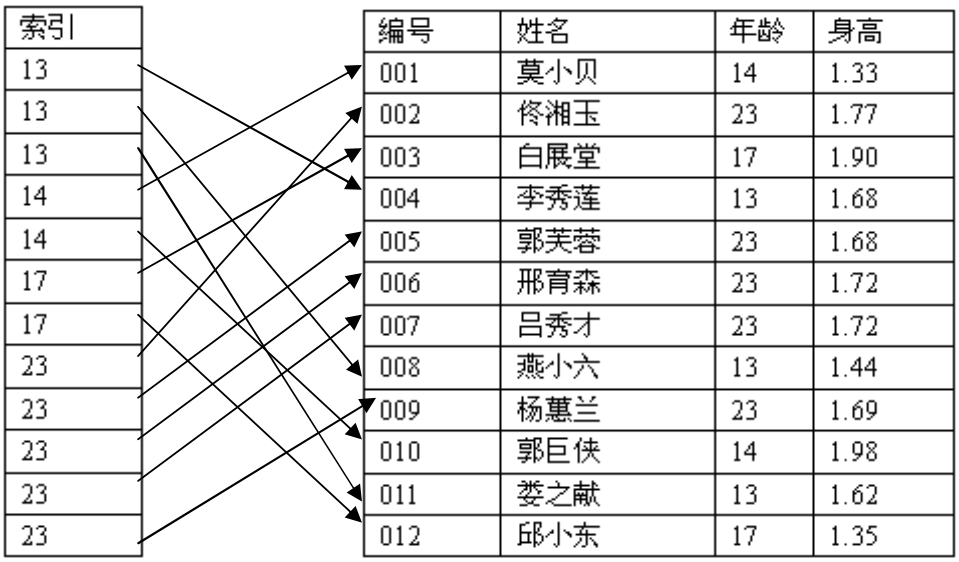
编号	姓名	年龄	身高
001	莫小贝	14	1.33
002	佟湘玉	23	1.77
003	白展堂	17	1.90
004	李秀莲	13	1.68
005	郭芙蓉	23	1.68
006	邢育森	23	1.72
007	吕秀才	23	1.72
008	燕小六	13	1.44

009	杨蕙兰	23	1.69
010	郭巨侠	14	1.98
011	娄之献	13	1.62
012	邱小东	17	1.35

这个表上没有索引，因此如果我们查找年龄等于 17 的人员时，必须查看表中的每一行，看它是否与所需的值匹配。这是一个全表扫描，很慢，如果表中只有少数几个记录与搜索条件相匹配，则其效率是相当低的。

如果我们经常要查询某个年龄的人员，必须查看表中的每一行，看它是否与所需的值匹配。这是一个全表扫描，很慢，如果表中只有少数几个记录与搜索条件相匹配，则其效率是相当低的。

如果我们为年龄列创建一个索引，注意这里的索引所采用的值是排序的：



索引
13
13
13
14
14
17
17
23
23
23
23
23

编号	姓名	年龄	身高
001	莫小贝	14	1.33
002	佟湘玉	23	1.77
003	白展堂	17	1.90
004	李秀莲	13	1.68
005	郭芙蓉	23	1.68
006	邢育森	23	1.72
007	吕秀才	23	1.72
008	燕小六	13	1.44
009	杨蕙兰	23	1.69
010	郭巨侠	14	1.98
011	娄之献	13	1.62
012	邱小东	17	1.35

假如我们要查找年龄为 13 岁的所有人员，那么可以扫描索引，结果得出前 3 行，当到达年龄为 14 的行的时候，我们发现它是一个比我们正在查找的年龄要大的人员。索引值是排序的，因此在读到包含 14 的记录时，我们知道不会再有匹配的记录，可以退出了。如果查找一个值，它在索引表中某个中间点以前不会出现，那么也有找到其第一个匹配索引项的定位算法，而不用进行表的顺序扫描（如二分查找法）。这样，可以快速定位到第一个匹配的值，以节省大量搜索时间。

可以把索引想像成汉语字典的的按笔画查找的目录。汉语字典中的汉字是按拼音的顺序排列在书页中的，如果要查询笔画数为 18 的汉字的话就必须挨个查找每个汉字来比较每个汉字的笔画数，这种速度是让人无法忍受的。而如果我们创建一个按笔画查找的目录：将笔画为 5 的汉字列出来，将笔画为 6 的汉字列出来……，这样当我们要查询笔画数为 18 的汉字的话只要来查找这个目录就可以非常快速的查找到需要的数据了。

虽然索引可以提高数据查询的速度，但是任何事物都是双刃剑，它也有一些缺点：
索引占据一定磁盘空间，就像有按笔画查找的目录的书会比没有这种目录的书页数要多一些。

索引减慢了数据插入和删除的速度。因为每次插入和删除的时候都需要更新索引，一个表拥有的索引越多，则写操作的平均性能下降就越大。

1.2.8 表关联

我们来为货物建一张表，其中包含规格、名称、生产厂家等等信息，如下：

编号	名称	规格	生产厂家	厂家地址	厂家电话
001	生肉	优质	七侠镇肉联厂	西凉河路 3 号	5555-123456
002	玉米肠	简装	七侠镇肉联厂	西凉河路 3 号	5555-123456
003	尿素	60 公斤装	六扇门化工厂	汉中工业区	5555-654321
004	打印纸	16 开	钱氏纸业	县政府对过	5555-123654
005	磷酸二铵	30 公斤装	六扇门化工厂	汉中工业区	5555-654321

可以看到这里存在大量冗余信息，比如厂家的名称、地址、电话等就在表中重复多次，这会带来如下的问题：

- 信息冗余占据空间。数据的存储是占据一定的空间的，如果存在过多冗余信息将会使得存储系统的利用率过低。
- 信息冗余使得新数据的加入变得麻烦。每次录入新的货物的话必须把厂家地址、厂家电话等信息重新录入一次。
- 信息冗余使得维护数据的正确性变得困难。如果七侠镇肉联厂迁址了，那么必须将表中所有七侠镇肉联厂的厂家地址都要更新一遍。

解决的方法就是即将厂家的信息在一个新的表中维护。我们创建下边的厂家表：

厂家编号	厂家名称	厂家地址	厂家电话
001	七侠镇肉联厂	西凉河路 3 号	5555-123456
002	六扇门化工厂	汉中工业区	5555-654321
003	钱氏纸业	县政府对过	5555-123654

这里我们为每个厂家指定了一个厂家编号做为主键，这个编号就可以唯一标识一个厂家。

有了厂家信息表，货物表就可以修改成如下的新的格式了：

编号	名称	规格	生产厂家编号
001	生肉	优质	001
002	玉米肠	简装	001
003	尿素	60 公斤装	002
004	打印纸	16 开	003
005	磷酸二铵	30 公斤装	002

在货物表中只保留了指向厂家表的主键的字段“生产厂家编号”，这样就避免了数据冗余的问题。当进行查询的时候，只要根据“生产厂家编号”到厂家信息表中查询就可以知道厂家的详细信息了；当厂家迁址的时候，只要修改厂家信息表中的一条数据就可以了。

这种将两张表通过字段关联起来的方式就被称为“表关联”，关联到其他表主键的字段被称为“外键”，上边例子中货物表中的“生产厂家编号”字段就是外键。表关联也是关系数据库的核心理念，它使得数据库中的数据不再互相孤立，通过表关联我们可以表达非常复杂的数据关系。

1.2.9 数据库的语言——SQL

DBMS 是一种系统软件，我们要与它交互的时候就必须使用某种语言，在数据库发展

初期每一种 DBMS 都有自己的特有的语言，不过逐渐的 SQL 成为了所有 DBMS 都支持的主流语言。SQL 是专为数据库而建立的操作命令集，是一种功能齐全的数据库语言。在使用它时，只需要发出“做什么”的命令，“怎么做”是不用使用者考虑的。SQL 功能强大、简单易学、使用方便，已经成为了数据库操作的基础，并且现在几乎所有的数据库均支持 SQL。

SQL 的英文全称是 Structured Query Language，它是 1974 年由 Boyce 和 Chamberlin 提出的，并且首先在 IBM 的关系数据库原型产品 R 系统（SYSTEM R）上实现。它的前身是 1972 提出的 SQUARE（Specifying Queries As Relational Expressesion）语言，在 1974 年做了修改，并且改名为 SEQUEL（Structured English Query Language）语言，后来 SEQUEL 简化为 SQL。

SQL 是高级的非过程化编程语言，允许用户在高层数据结构上工作。使用它，用户无需指定对数据的存放方法，也不需要用户了解具体的数据存放方式，所以具有完全不同底层结构的不同数据库系统可以使用相同的 SQL 语言作为数据输入与管理的接口。它以记录集合作为操纵对象，所有 SQL 语句接受集合作为输入，返回集合作为输出，这种集合特性允许一条 SQL 语句的输出作为另一条 SQL 语句的输入，所以 SQL 语言可以嵌套，这使它具有极大的灵活性和强大的功能，在多数情况下，在其他语言中需要一大段程序实现的一个单独事件只需要一个 SQL 语句就可以达到目的，这也意味着用 SQL 语言可以写出非常复杂的语句。

SQL 具有下面 4 个主要的功能：创建数据库并定义表的结构；查询需要的数据；更新或者删除指定的数据；控制数据库的安全。使用 SQL 我们可以完成和 DBMS 的几乎所有交互任务。

比如我们要查找年龄小于 18 岁的员工信息，那么我们只要执行下面的 SQL 就可以：

```
SELECT * from Employees where age<18
```

比如我们要将所有职位为“名誉总裁”的员工删除，那么就可以执行下面的 SQL：

```
DELETE from Employees where position=’名誉总裁’
```

可以看到我们只是描述了我们要做什么，至于怎么去做则由 DBMS 来决定。可以想想如果要是自己编程去实现类似的功能，则需要编写非常复杂的算法才能完成，而且性能也不一定会非常好。

我们可以通过三种方式执行 SQL：

- 在工具中执行。各个 DBMS 几乎都提供了工具用于执行 SQL 语句，比如 Microsoft SQL Server 的 Management Studio、DB2 的命令中心、Oracle 的 SqlPlus 或者 MySQL 的 Query Browser。在这些工具中我们只要输入要执行的 SQL 然后点击【执行】按钮就可以得到执行结果。
- 以编译的方式嵌入到语言中。在这种方式中我们可以把 SQL 直接写到代码中，在编译的时候由编译器来决定和数据库的交互方式。比如 PowerBuild、C 等就采用这种方式。
- 以字符串的形式嵌入到语言中。在这种方式中 SQL 语句只是以字符串的形式写到代码中，然后由代码将其提交到 DBMS，并且分析返回的结果。目前这是大部分支持数据库操作的语言采用的方式，比如 C#、Java、Python、Delphi 和 VB 等。

由于嵌入到语言中的执行方式是严重依赖宿主语言的，而本书不假定用户使用任何编程语言，为了能够使得使用任何语言的读者都能学习本书中的知识点，本书将主要以在工具中执行的方式来执行 SQL 语句，读者可以根据自己使用的编程语言来灵活运用这些知识点。不熟悉用工具执行 SQL 的读者可以参考附录 A 中的介绍。

IBM 是 SQL 语言的发明者，但是其他的数据库厂商都在 IBM 的 SQL 基础上提出了自己的扩展语法，因此形成了不同的 SQL 语法，对于开发人员来说，使用这些有差异的语法

是非常头疼的时候。因此在 1986 年美国国家标准化协会（ANSI）为 SQL 制定了标准，并且在 1987 年国际标准化组织（ISO）也为 SQL 指定了标准，迄今为止已经推出 SQL-86、SQL-89、SQL-92、SQL-99、SQL-2003 等版本的标准。

虽然已经有了国际标准，但是由于种种原因，各个数据库产品的 SQL 语法仍然有着很大差异，在数据库 A 上能成功执行的 SQL 放到数据库 B 上就会执行失败。为了方便使用不同数据库产品的读者都能成功运行本书中的例子，我们会介绍各种数据库 SQL 的差异性，并且给出解决方案，而且本书将会安排专门章节讲解跨数据库程序开发的技术。

1.2.10 DBA 与程序员

如果你是一个数据库开发技术的初学者的话，你会发现到了书店里有很多数据库相关的书你看不懂，你会发现互联网有一些搞数据库的人的 Blog 上说的东西你感觉很陌生，他们都是在谈论数据库的恢复、数据库的调优、调整数据库的安全性，难道他们搞的是更深层次的东西吗？不是的，他们就是数据库系统管理员（Database Administrator，DBA）。围绕在 DBMS 周围的技术人员有两类：数据库系统管理员和开发人员。使用数据库进行程序开发的人员是程序员，而对数据库系统进行管理、维护、调优的则是数据库系统管理员。

作为一名开发人员，我们不必知道如何安装和配置数据库系统，这应该是 DBA 的任务；当规划数据库的备份策略的时候，不要去问开发人员，这也是 DBA 的任务；当数据库系统崩溃的时候，请立即给 DBA 打电话，如果打给开发人员的话，你得到的回答通常是“怎么会呢？天知道怎么恢复！”。正如一个公司的网络系统是由网管来负责的一样，一个公司的数据库系统也是由 DBA 来进行管理的，它们的主要工作如下：

- 安装和配置数据库，创建数据库以及帐户；
- 监视数据库系统，保证数据库不宕机；
- 收集系统统计和性能信息以便进行调整；
- 发现性能糟糕的 SQL，并给开发人员提出调优建议；
- 管理数据库安全性；
- 备份数据库，当发生故障时要及时恢复；
- 升级 DBMS 并且在必要时为系统安装补丁；
- 执行存储和物理设计，均衡设计问题以完成性能优化；

DBA 大部分时间是在监视系统、备份/恢复系统、优化系统，而开发人员则无需精通这些技能；开发人员大部分时间是在用 SQL 实现业务逻辑。二者知识的重合点就是 SQL，一个开发人员如果不熟悉 SQL 的话就无法很好的实现业务逻辑，而一个 DBA 如果不熟悉 SQL 的话就无法完成数据库的调优工作。所以无论你是想成为开发人员还是成为 DBA，那么都首先来学好 SQL 吧！

进行数据库的备份/恢复、权限管理等操作也经常需要使用 SQL 命令来完成，不过这些 SQL 命令都是与特定的 DBMS 产品相关的，而且不同产品的使用方式也是差别很大的，所以本书不会讲解数据库的备份/恢复、权限管理相关的 SQL，有兴趣的读者可以去参考相关的资料。

第二章 数据表的创建和管理

数据表是数据库中的基本元素，一个没有数据表的数据库是没有任何意义的，没有数据表任何 SQL 也是几乎无法执行的。因此本章将介绍数据表的创建、数据表的修改以及数据表的删除等相关知识，在创建数据表之前有可能需要创建 Schema，创建 Schema 的方法请参考附录 A。

2.1 数据类型

由于创建数据表之前必须确定每个列的数据类型，而每种数据库所支持的数据类型都是有一些差别的，所以这节我们首先介绍主流数据库中支持的数据类型。

ANSISQL 规定了数据类型的标准，但是各种主流的数据库系统产品并没有完全的遵守这个标准，其不同主要体现在如下几点：

- 同一数据类型的名称不同。比如长度可变的字符串在 MSSQLServer 中称为 Varchar，在 Oracle 中称为 Varchar2；数值类型在 MSSQLServer 中称为 numeric，在 DB2 中称为 decimal。
- 各种数据库都有自己特性的数据类型：MSSQLServer 中的有可以用来存储布尔型数据的数据类型 bit，而在其他数据库中则没有对应的数据类型。

数据库系统中的数据类型大致可以分为五类：整数、数值、字符相关、日期时间以及二进制。

2.1.1 整数类型

整数类型可以表示-2147483648 到 2147483647 之间的整数[1]。整数数值全部由数字组成，不含有小数点。它们可以用来用作唯一主键，特别是在 MSSQLServer 中整数类型的字段可以指定为“标识列”，标识列的数值将会在新增条目的时候自动增长。

除了标准的整数类型，很多数据库都对整数类型做了扩展。下表是主流数据库系统中对整数类型的扩展支持：

数据库系统	类型	说明
MYSQL	tinyint [unsigned][2]	一个很小的整数。有符号的范围是 **-128** 到 **127**，无符号的范围是 **0** 到 **255**。
	smallint [unsigned]	一个小整数。有符号的范围是**-32768** 到 **32767**，无符号的范围是 **0** 到 **65535**。
	mediumint [unsigned]	一个中等大小整数。有符号的范围是 **-8388608** 到 **8388607**，无符号的范围是 **0** 到 **16777215**。
	int [unsigned]	一个正常大小整数。有符号的范围是 **-2147483648** 到 **2147483647**，无符号的范围是 **0** 到 **4294967295**。
	integer [unsigned]	**integer** 是 **int** 的同义词。
	bigint [unsigned]	一个大整数。有符号的范围是 **-9223372036854775808** 到 **9223372036854775807**，无符号的范围是 **0** 到 **18446744073709551615**。
MSSQLServer	bit	其值只能是 **0**、**1** 或空值。这种数据类型用于存储只有两种可能值的数据，如 **Yes** 或 **No**、**True** 或 **Fa lse** 、**On** 或 **Off**。

[1]这里指的是在 32 位系统下，本书中如果没有特殊说明都是指的 32 位系统。

[2] UNSIGNED 表示是否是有符号数。这里的[]表示 UNSIGNED 是可以省略的，本书中其他部分也将使用这种表示法。

	int	正常大小整数，取值范围是 **-2147483648** 到 **2147483647**。
	smallint	可以存储从**-32768** 到 **32767** 之间的整数。这种数据类型对存储一些常限定在特定范围内的数值型数据非常有用。这种数据类型在数据库里占用 **2** 字节空间。
	tinyint	能存储从 **0** 到 **255** 之间的整数。它在你只打算存储有限数目的数值时很有用。 这种数据类型在数据库中占用 **1** 个字节。
	bigint	可以精确的表示从－2^{63} 到 $2^{63}-1$（即从 **-9,223,372,036,854,775,808** 到 **9,223,372,036,854,775,807**）之间的整数，它占用了八个字节的存储空间。
Oracle	number(10)	**Oracle** 中没有专门的整数类型，因此需要使用 **Number(10)**来表示整形。
DB2	smallint	小整型是两个字节的整数，精度为 **5** 位。小整型的范围从 **-32,768** 到 **32,767**。
	integer	普通整型是四个字节的整数，精度为 **10** 位。大整型的范围从 **-2,147,483,648** 到 **2,147,483,647**。
	bigint	大整型是八个字节的整数，精度为 **19** 位。巨整型的范围从 **-9,223,372,036,854,775,808** 到 **9,223,372,036,854,775,807**。

2.1.2 数值类型

整数类型不能表示含有小数部分，如果需要表示金额等信息的话就必须使用数值类型。各种数据库系统对数值类型的支持也各不相同，必须注意选用合适的数值类型，以防止出现数据错误。下表是主流数据库系统中对数值类型的支持：

数据库系统	类型	说明
MYSQL	**float[(m,d)]**	单精密浮点数字。取值范围是 **-3.402823466E+38** 到 **-1.175494351E-38**，**0** 和 **1.175494351E-38** 到 **3.402823466E+38**。**m** 是显示宽度、而 **d** 是小数的位数。没有参数的 **float** 或有**<24** 的一个参数表示一个单精密浮点数字。
	double[(m,d)]	双精密浮点数字。取值范围是 **-1.7976931348623157E+308** 到

		-2.2250738585072014E-308、 **0** 和 **2.2250738585072014E-308** 到 **1.7976931348623157E+308**。 **m** 是显示宽度、而 **d** 是小数位数。没有参数的 **double** 代表一个双精密浮点数字。
	real[(m,d)]	**real** 是 **double** 同义词。
	decimal[(m[,d])]	一个未压缩的浮点数字，数字作为一个字符串被存储，值的每一位使用一个字符。小数点，并且对于负数，"**-**"符号不在 **M** 中计算。如果 **D** 是 **0**，值将没有小数点或小数部分。**decimal** 值的最大范围与 **double** 相同，但是对一个给定的 **decimal** 列，实际的范围可以通过 **m** 和 **d** 的选择被限制。如果 **d** 被省略，它被默认设置为 **0**。如果 **m** 被省掉，它被默认设置为 **10**。
	numeric(m,d)	**numeric** 是 **decimal** 的一个同义词。
MSSQLServer	decimal (p,s)	数字数据，p 为固定精度，s 为宽度。 decimal 数据类型能用来存储从 $-10^{38}-1$ 到 $10^{38}-1$ 的固定精度和范围的数值型数据。使用这种数据类型时，必须指定范围和精度。 范围是小数点左右所能存储的数字的总位数。精度是小数点右边存储的数字的位数
	numeric	numeric 是 decimal 的同义词。
	money	货币型。用来表示货币值。这种数据类型能存储从-9220 亿到 9220 亿之间的数据，精确到货币单位的万分之一。
	smallmoney	货币型。用来表示货币值。这种数据类型能存储从 -214748.3648 到 214748.3647 之间的数据，精确到货币单位的万分之一
	float	近似数值型。float 数据类型是一种近似数值类型，供浮点数使用。浮点数可以是从-1.79E+308 到 1.79E+308 之间的任意数。
	real	real 数据类型像浮点数一样，它可以表示数值在-3.40E+38 到 3.40E+38 之间的浮点数
Oracle	number(m,n)	数值型，m 是所有有效数字的位数，n 是小数点以后的位数。 如：number(5,2)，则这个字段的最大值是 99,999，如果数值超出了位数限制就

		会被截取多余的位数。 如：number(5,2)，但在一行数据中的这个字段输入 575.316，则真正保存到字段中的数值是 575.32。 如：number(3,0)，输入 575.316，真正保存的数据是 575。
DB2	decimal(p,s)	小数型的值，它是一种压缩十进制数，它有一个隐含的小数点。压缩十进制数将以 BCD 码来存储。小数点的位置取决于数字的精度（p）和小数位（s）。小数型的范围从 $-10^{31}+1$ 到 $10^{31}-1$。
	numeric(p,s)	numeric (p,s)是 decimal(p,s)的同义词。
	real	单精度浮点数，它是实数的 32 位近似值。数字可以为零，或者在从 -3.402E+38 到 -1.175E-37 或从 1.175E-37 到 3.402E+38 的范围内。
	double	双精度浮点数是实数的 64 位近似值。数字可以为零，或者在从 -1.79769E+308 到 -2.225E-307 或从 2.225E-307 到 1.79769E+308 的范围内。

从上表中我们可以发现虽然每种数据类型的名称不同，但是这些类型可以分为可以指定精度、宽度的类型和不能指定精度宽度的类型，其中 decimal(p,s)、numeric(p,s)属于前者，而 real、double、float 则属于后者。

2.1.3 字符相关类型

如果需要存储一个或者多个字符的话就需要使用字符相关类型字段。任意多个字符（也可以是 0 个）组合在一起也可以称作字符串。主流数据库系统提供了下面几种类型供使用：固定长度、可变长度、国际化可变长度以及大字符串。

顾名思义，固定长度字符类型用来保存具有固定长度的字符串，我们可以指定字段所能保存的字符串的长度。比如设定字段的长度为 100，那么如果我们将一个长度为 100 的字符串保存到这个字段的时候将恰好能够填满字段；如果设定的字符串长度不满 100，那么剩余部分将以空格填充。在大部分数据库中固定长度字符类型的名称为 char。

使用固定长度字符类型保存数据的时候，由于剩余部分会以空格填充，那么在读取的字段值的时候就会将后面填充的空格也读取出来，这有的时候是很不方便的的。这时就可以使用可变长度字符类型。可变长度字符类型一般也需要指定一个长度，但是这个长度指的是此字段所能保存的字符串的最大长度，如果保存的字符串的长度没超过最大长度的话，数据库将不会将剩余部分用空格填充。在大部分数据库中可变长度字符类型的名称为 varchar。

固定长度字符类型和可变长度字符类型都只能存储基于 ASCII 的字符，这样对于使用中文、韩文、日文等 Unicode 字符集的程序来讲将会造成储存问题。为了解决这个问题，我们可以使用国际化可变长度字符类型，这种类型可以用两个字节来保存一个字符，这样就可以解决中韩日等字符串保存的问题了。在大部分数据库中可变长度字符类型的名称为

nvarchar。

固定长度字符类型和可变长度字符类型一般都不能指定过于大的长度，比如长度超过 1024 是不允许的，但是在需要保存一些文章、合同等场合的时候经常有长度大于所允许最大长度的字符串，为了保存这些大字符串我们必须使用大字符串类型字段。

固定长度字符类型比固定长度字符类型占用空间要大，但是由于进行字段值设置的时候固定长度字符类型无需进行长度处理就可以进行，因此它的处理速度更快。所以对于长度相对固定的数据来讲，使用固定长度字符类型将会提高系统的性能。

由于 Unicode 覆盖了 ASCII 的表示范围，因此国际化可变长度字符类型不仅可以存储双字节字符，也可以存储普通的英文字符串。不过由于国际化可变长度字符类型采用两个字节来存储一个字符，所以如果字段中没有存放双字节字符的话尽量不要使用国际化可变长度字符类型。

大字符串类型字段可以保存非常多的字符，但是对于这种类型的数据 DBMS 经常将它们保存到单独的空间中，这样就导致数据的保存和加载的速度非常慢，因此除非确实有大字符串需要保存，否则尽量不要使用大字符串类型字段。

下表是主流数据库系统中对字符相关类型的支持：

数据库系统	类型	说明
MYSQL	char(m)	固定长度字符串，长度为 m
	varchar(m)	可变长度字符串，最大长度为 m
	tinytext	小的可变长度字符串，最大长度 2^8- 1 字节
	text	可变长度大字符串，最大长度 2^{16} - 1 字节
	mediumtext	中等可变长度字符串，最大长度 2^{24}- 1 字节
	longtext	大文本可变长度字符串，最大长度 2^{32}- 1 字节
	enum(“value1”, “value2”, ...)	枚举字符串，列可被赋予某个枚举成员
	set (“value1”, “value2”, ...)	集合字符串；列可被赋予多个集合成员
MSSQLServer	char(m)	固定长度字符串，长度为 m
	varchar(m)	可变长度字符串，最大长度为 m
	text	可变长度字符串，最大长度 2^{31} - 1 字节
	nchar(m)	固定长度国际化字符串，长度为 m
	nvarchar(m)	可变长度国际化字符串，最大长度为 m
	ntext	可变长度国际化大字符串，其最大长度为 **2^30 - 1 (1,073,741,823)** 个字符。
Oracle	char(m)	固定长度字符串，长度为 m

	varchar2(m)	可变长度字符串，最大长度为 m
	nvarchar2(m)	可变长度国际化字符串，最大长度为 4GB
	clob	可变长度大字符串，最大长度 2^{16} - 1 字节
	nclob	可变长度国际化大字符串，最大长度为 4GB
DB2	CHARACTER(m)	固定长度字符串，长度为 m
	VARCHAR(m)	可变长度字符串，最大长度为 m
	LONG VARCHAR	可变长度字符串，最长可达 32,700 字节
	CLOB	变长大字符串，最长可以达到 2,147,483,647 字节
	GRAPHIC[(m)]	固定长度图形字符串，长度为 m，如果没有指定长度，就认为是 1 个双字节字符
	VARGRAPHIC(m)	可变长度图形字符串，最大长度为 m
	LONG VARGRAPHIC	可变长度图形字符串

2.1.4 日期时间类型

信息系统中经常需要处理一些日期时间相关的数据，比如审批时间、开户日期等等，我们可以使用字符串来保存这些数据，比如“2008-08-08”，但是使用字符串表示日期使得很难保证数据的正确性，而且进行数据检索的时候也会非常麻烦以及低效，为此必须使用数据库系统提供的日期时间类型数据。

主流数据库系统中对日期时间类型的支持差别非常大，有的数据库系统提供了能够单独表示时间的 Time 类型，而有的数据库系统中日期时间必须组合在一起使用，没有单独的日期类型或者时间类型。下表是主流数据库系统中对日期时间类型的支持：

数据库系统	类型	说明
MYSQL	date	“yyyy-mm-dd”格式表示的日期值。取值范围：“1000-01-01”到“9999-12-31”
	time	“hh:mm:ss”格式表示的时间值。取值范围：“-838:59:59”到“838:59:59”
	datetime	“yyyy-mm-dd hh:mm:ss”格式表示的日期时间值。取值范围：“1000-01-01 00:00:00”到“9999-12-31 23:59:59”
	timestamp	“yyyymmddhhmmss”格式表示的时间戳值。取值范围：19700101000000 到 2037 年的某个时刻

	year	“yyyy”格式的年份值。取值范围：1901 到 2155
MSSQLServer	**datetime**	从 **1753** 年 **1** 月 **1** 日到 **9999** 年 **12** 月 **31** 日的日期时间数据，精确到百分之三秒
	smalldatetime	从 **1900** 年 **1** 月 **1** 日到 **2079** 年 **6** 月 **6** 日的日期和时间数据，精确到分钟
	timestamp	时间戳
Oracle	date	日期时间数据
	timestamp	时间戳
DB2	DATE	日期值。取值范围：“0001-01-01 00:00:00” 到 “9999-12-31 23:59:59”
	TIME	时间值。
	TIMESTAMP	时间戳

2.1.5 二进制类型

如果我们要将一幅图片或者一段视频存入数据库的话就需要使用二进制类型的字段，这种类型的字段通常能够保存非常大的、没有固定结构的数据，而设置、读取这些数据也通常需要宿主语言的辅助。

下表是主流数据库系统中对二进制类型的支持：

数据库系统	类型	说明
MYSQL	blob	
MSSQLServer	**image**	虽然类型名为 **image**，但是并不意味着只能保存图片二进制数据，实际上它可以保存任何二进制数据。
Oracle	blob	
DB2	blob	

2.2 通过 SQL 语句管理数据表

各个主流的 DBMS 都提供了图形化的工具用来创建数据表，我们只需轻点鼠标就可以创建一个数据表，附录 A 中介绍了通过图形化工具创建数据表的方式，因此本节我们将重点介绍通过 SQL 语句创建数据表的方式。我们将介绍如何创建一个新数据表，如何修改已有的数据表，以及如何删除我们不再需要的数据表。

2.2.1 创建数据表

SQL 语句 CREATE TABLE 用于创建数据表，其基本语法如下：

```
CREATE TABLE 表名
(
    字段名 1 字段类型,
    字段名 2 字段类型,
    字段名 3 字段类型,
    ……
    约束定义 1,
    约束定义 2,
```

……

)

这里的 CREATE TABLE 语句告诉数据库系统我们要创建一张数据表，CREATE TABLE 语句后紧跟着表名，这个表名不能与数据库中已有的表名重复。

括号中是一条或者多条表定义，表定义包括字段定义和约束定义两种，一张表中至少要有一个字段定义，而约束定义则是可选的。约束定义包括主键定义、外键定义以及唯一约束定义等。下面用例子来演示这个语句的使用。

下面的 SQL 语句创建了一个用于保存人员信息的数据表：

```
CREATE TABLE T_Person
(
    FName VARCHAR(20),
    FAge INT
)
```

注意：上边的 SQL 在 MYSQL、MSSQLServer 以及 DB2 下可以正常运行，不过由于各个主流数据库系统中数据类型的差异，所以在其他数据库中可能需要改写。

下面是此 SQL 在 Oracle 下的写法：

```
CREATE TABLE T_Person
(
    FName VARCHAR2(20),
    FAge NUMBER (10)
)
```

可以看到这里将人员数据表的名称为 T_Person，并且拥有两个字段，一个字段为记录姓名的字段 FName，另一个为记录年龄的 FAge。姓名为长度不确定的字符串类型，因此这里我们使用最大长度为 20 的可变长度字符串 VS Studio 的插件开发来定义 FName 字段；年龄为整数，所以使用 INT 来定义 FAge 字段。需要用逗号来分隔开每一个字段的定义，我们这里将每个字段都在单独一行中定义，这并不是强制要求的，我们可以将所有字段定义在一行中，如下：

```
CREATE TABLE T_Person(FName VARCHAR(20),FAge INT);
```

这样的 SQL 语句也是合法的，不过当字段数量比较多的时候这样写就会显得过于杂乱，因此推荐使用每行一个字段定义的方式，这样容易阅读，而且出现错误的时候也容易调试，因为很多数据库系统都是根据行号来提示错误信息的。

字段定义只能限制一个字段中所能填充的数据类型，对于“字段值必须唯一、录入的年龄必须介于 18 到 26 岁之间、姓名不能为空”这样的需求则无法满足，这也就是约束定义的工作。为了降低初学者的学习难度，约束定义相关的内容我们将在第五章介绍，通过本章的学习读者能够掌握简单的数据表的创建就可以了。

2.2.2 定义非空约束

我们在注册一些网站的会员的时候都需要填写一些表格，这些表格中有一些属于必填内容，如果不填写的话会无法完成注册。同样我们在设计数据表的时候也希望某些字段为必填值，比如学生信息表中的学号、姓名、年龄字段是必填的，而个人爱好、家庭电话号码等字段则选填，所以我们如下设计建表 SQL：

MYSQL、MSSQLServer、DB2：

```
CREATE TABLE T_Student (FNumber VARCHAR(20) NOT NULL ,FName VARCHAR(20) NOT NULL ,FAge INT NOT NULL ,FFavorite VARCHAR(20),FPhoneNumber VARCHAR(20))
```

Oracle：

CREATE TABLE T_Student (FNumber VARCHAR2(20) NOT NULL ,FName VARCHAR2(20) NOT NULL ,FAge NUMBER (10) NOT NULL ,FFavorite VARCHAR2(20),FPhoneNumber VARCHAR2(20))

可以看到，与普通字段定义不同的地方是，非空字段的定义在类型定义后增加了“NOT NULL”，其他定义方式与普通字段相同。

2.2.3 定义默认值

我们在定义字段的时候为字段设置一个默认值，当向表中插入数据的时候如果没有为这个字段赋值则这个字段的值会取值为这个默认值。比如我们希望设置教师信息表中的是否班主任字段 FISMaster 的默认值为“NO”，那么只要如下设计建表 SQL：

MYSQL、MSSQLServer、DB2：

CREATE TABLE T_Teacher (FNumber VARCHAR(20),FName VARCHAR(20),FAge INT,FISMaster VARCHAR(5) DEFAULT 'NO')

Oracle：

CREATE TABLE T_Teacher (FNumber VARCHAR2(20),FName VARCHAR2(20),FAge NUMBER (10),FISMaster VARCHAR2(5) DEFAULT 'NO')

可以看到，与普通字段定义不同的地方是，非空字段的定义在类型定义后增加了“DEFAULT 默认值表达式”，其他定义方式与普通字段相同。

2.2.4 定义主键。

通过主键能够唯一定位一条数据记录，而且在进行外键关联的时候也需要被关联的数据表具有主键，所以为数据表定义主键是非常好的习惯。在 CREATE TABLE 中定义主键是通过 PRIMARY KEY 关键字来进行的，定义的位置是在所有字段定义之后。比如我们为公交车建立一张数据表，这张表中有公交车编号 FNumber、驾驶员姓名 FDriverName、投入使用年数 FUsedYears 等字段，其中公交车编号 FNumber 字段要定义为主键，那么只要如下设计建表 SQL：

MYSQL,MSSQLServer:

CREATE TABLE T_Bus (FNumber VARCHAR(20),FDriverName VARCHAR(20),
FUsedYears INT,PRIMARY KEY (FNumber))

Oracle:

CREATE TABLE T_Bus (FNumber VARCHAR2(20),FDriverName VARCHAR2(20),
FUsedYears NUMBER (10),PRIMARY KEY (FNumber))

DB2:

CREATE TABLE T_Bus (FNumber VARCHAR(20) NOT NULL,FDriverName VARCHAR(20),
FUsedYears INT,PRIMARY KEY (FNumber))

可以看到，主键定义是在所有字段后的“约束定义段”中定义的，格式为 PRIMARY KEY (主键字段名)，在有的数据库系统中主键字段名两侧的括号是可以省略的，也就是可以写成 PRIMARY KEY FNumber，不过为了能够更好的跨数据库，建议不要采用这种不通用的写法。

需要注意的是，在上边列出的 DB2 数据库的 CREATE TABLE 语句中，我们为 FNumber 字段设置了非空约束。因为在 DB2 中，主键字段必须被添加非空约束，否则会报出类似“"FNUMBER" 不能是一列主键或唯一键，因为它可包含空值。”的错误。

有的时候数据表中是不存在一个唯一的主键的，比如某个行业协会需要创建一个保存个人会员信息的表，表中记录了所属公司名称 FCompanyName、公司内部工号 FInternalNumber、姓名 FName 等，由于存在同名的情况，所以不能够使用姓名做为主键，同样由于各个公司

之间的内部工号也有可能重复，所以也不能使用公司内部工号做为主键。不过如果确定了公司名称，那么公司内部工号也就唯一了，也就是说通过公司名称 FCompanyName 和公司内部工号 FInternalNumber 两个字段一起就可以唯一确定一个个人会员了，我们可以让 FCompanyName、FInternalNumber 两个字段联合起来做为主键，这样的主键被称为联合主键（或者称为复合主键）。可以有两个甚至多个字段来做为联合主键，这就可以解决一张表中没有唯一主键字段的问题了。定义联合主键的方式和唯一主键类似，只要在 PRIMARY KEY 后的括号中列出做为联合主键的各个字段就可以了。上面的例子的建表 SQL 如下：

MYSQL,MSSQLServer,DB2:

```
CREATE TABLE T_PersonalMember (FCompanyName VARCHAR(20),
FInternalNumber VARCHAR(20),FName VARCHAR(20),
PRIMARY KEY (FCompanyName,FInternalNumber))
```

Oracle:

```
CREATE TABLE T_PersonalMember (FCompanyName VARCHAR2(20),
FInternalNumber VARCHAR2(20),FName VARCHAR2(20),
PRIMARY KEY (FCompanyName,FInternalNumber))
```

DB2:

```
CREATE TABLE T_PersonalMember (FCompanyName VARCHAR(20) NOT NULL,
FInternalNumber VARCHAR(20) NOT NULL,FName VARCHAR(20),
PRIMARY KEY (FCompanyName,FInternalNumber))
```

同样需要注意的是，在 DB2 中组成联合主键的每一个字段也都必须被添加非空约束。

采用联合主键可以解决表中没有唯一主键字段的问题，不过联合主键有如下的缺点：

- 效率低。在进行数据的添加、删除、查找以及更新的时候数据库系统必须处理两个字段，这样大大降低了数据处理的速度。
- 使得数据库结构设计变得糟糕。组成联合主键的字段通常都是有业务含义的字段，这与“使用逻辑主键而不是业务主键”的最佳实践相冲突，容易造成系统开发以及维护上的麻烦。
- 使得创建指向此表的外键关联关系变得非常麻烦甚至无法创建指向此表的外键关联关系。
- 加大开发难度。很多开发工具以及框架只对单主键有良好的支持，对于联合主键经常需要进行非常复杂的特殊处理。

考虑到这些缺点，我们应该只在兼容遗留系统等特殊场合才使用联合主键，而在其他场合则应该使用唯一主键。

2.2.5 定义外键

外键是非常重要的概念，也是体现关系数据库中“关系”二字的体现，通过使用外键，我们才能把互相独立的表关联起来，从而表达丰富的业务语义。

外键是定义在源表中的，定义位置同样为所有字段定义的后面，使用 FOREIGN KEY 关键字来定义外键字段，并且使用 REFERENCES 关键字来定义目标表名以及目标表中被关联的字段，格式为：

```
FOREIGN KEY 外键字段名称 REFERENCES 目标表名(被关联的字段名称)
```

比如我们创建一张部门信息表，表中记录了部门主键 FId、部门名称 FName、部门级别 FLevel 等字段，建表 SQL 如下：

MYSQL,MSSQLServer:

```
CREATE TABLE T_Department (FId VARCHAR(20),FName VARCHAR(20),
```

FLevel INT,PRIMARY KEY (FId))

Oracle:

CREATE TABLE T_Department (FId VARCHAR2(20),FName VARCHAR2(20),
FLevel NUMBER (10) ,PRIMARY KEY (FId))

DB2:

CREATE TABLE T_Department (FId VARCHAR(20) NOT NULL,FName VARCHAR(20),
FLevel INT,PRIMARY KEY (FId))

接着创建员工信息表，表中记录工号、姓名以及所属部门等信息，为了能够建立同部门信息表之间的关联关系，我们在员工信息表中保存部门信息表中的主键，保存这个主键的字段就被称为员工信息表中指向部门信息表的外键。建表 SQL 如下：

MYSQL,MSSQLServer,DB2:

CREATE TABLE T_Employee (FNumber VARCHAR(20),FName VARCHAR(20),
FDepartmentId VARCHAR(20),
FOREIGN KEY (FDepartmentId) REFERENCES T_Department(FId))

Oracle:

CREATE TABLE T_Employee (FNumber VARCHAR2(20),FName VARCHAR2(20),
FDepartmentId VARCHAR2(20),
FOREIGN KEY (FDepartmentId) REFERENCES T_Department(FId))

2.2.6 修改已有数据表

通过 CREATE TABLE 语句创建的数据表的结构并不是永远不变的，很多因素决定我们需要对数据表的结构进行修改，比如我们需要在 T_Person 表中记录一个人的个人爱好信息，那么就需要在 T_Person 中增加一个记录个人爱好的字段，再如我们不再需要记录一个人的年龄，那么我们就可以将 FAge 字段删除。这些操作都可以使用 ALTER TABLE 语句来完成。ANSI-SQL 中为 ALTER TABLE 语句规定了两种修改方式：添加字段和删除字段，有的数据库系统中还提供了修改表名、修改字段类型、修改字段名称的语法。

首先来看添加字段的语法：

ALTER TABLE 待修改的表名 ADD 字段名 字段类型

在语句中需要指定要修改的表的表名、要增加的字段名以及字段的数据类型，其使用方式和创建表的非常类似。下面是为 T_Person 表增加个人爱好字段的 SQL 语句：

ALTER TABLE T_PERSON ADD FFavorite VARCHAR(20)

注意：上边的 SQL 在 MYSQL、MSSQLServer 以及 DB2 下可以正常运行，不过由于各个主流数据库系统中数据类型的差异，所以在其他数据库中可能需要改写。

下面是此 SQL 在 Oracle 下的写法：

ALTER TABLE T_PERSON ADD FFavorite VARCHAR2(20)

接下来看删除字段的语法：

ALTER TABLE 待修改的表名 DROP 待删除的字段名

在语句中需要指定要修改的表的表名以及要删除字段的名称。下面是删除 T_Person 表中年龄字段的 SQL 语句：

ALTER TABLET_Person DROP FAge

注意：DB2 中不能删除字段，所以这个 SQL 语句在 DB2 中是无法正确执行的。

2.2.7 删除数据表

当一个数据表不再有用的时候我们就可以将其删除，使用 DROP TABLE 语句就可以完成这个功能，DROP TABLE 语句的语法如下：

DROP TABLE 要删除的表名

可以看到 DROP TABLE 语句语法非常简单，只要指定要删除的表名就可以了。执行下面的 SQL 就可以将 T_Person 表删除了：

DROP TABLE T_Person

需要注意的是，如果在表之间创建了外键关联关系，那么在删除被引用数据表的时候会删除失败，因为这样会导致关联关系被破坏，所以必须首先删除引用表，然后才能删除被引用表。比如 A 表创建了指向 B 表的外键关联关系，那么必须首先删除 A 表后才能删除 B 表。

2.2.8 受限操作的变通解决方案

各个数据库系统中提供的修改表结构的方法是不同的，有的提供了修改表名、修改字段类型、修改字段名称等操作的 SQL 语句，而有的则没有提供这些功能，甚至有的数据库系统连删除字段的功能都不支持。但是这些操作有的时候又是必要的，那么有没有变通的手段来实现这些功能呢？答案是有！

在 DB2 中如果要在表 T 中删除一个字段 F1，那么可以首先创建一个表 T1，这个表 T1 的结构和表 T 结构一致，唯一区别就是缺少字段 F1；接着将表 T 中的数据导出到 T1 中，然后将表 T 删除；最后将表 T1 重命名为 T 就可以了。这样就可以达到修改表名的效果了。

在不支持修改字段名称操作的数据库系统上同样可以采用类似策略来解决。比如我们要将表 T 的 F1 字段重命名为 F2，那么首先在表 T 上创建新字段 F2，类型和 F1 一致，然后将 F1 的数据复制到 F2 上，最后将字段 F1 删除就可以了。这样就可以达到修改字段名称的效果了。

第三章 数据的增删改

上一章中介绍了创建和管理数据表的方法，数据表只是数据的容器，没有任何数据的表是没有任何意义的。主流的数据库系统都提供了管理数据库的工具，使用这些工具可以查看表中的数据，还可以添加、修改和删除表中的数据，但是使用工具进行数据的增删改通常只限于测试数据库时使用，更常见的方式时通过程序或者 Web 页面来向数据库发出 SQL 语句指令来进行这些操作，因此本章将介绍通过 SQL 语句增删改表中数据的方法。

本章中我们将使用一些数据表，为了更容易的运行本章中的例子，必须首先创建所需要的数据表，因此下面列出本章中要用到数据表的创建 SQL 语句：

MYSQL：

CREATE TABLE T_Person (FName VARCHAR(20),FAge INT,FRemark VARCHAR(20),PRIMARY KEY (FName));

CREATE TABLE T_Debt (FNumber VARCHAR(20),FAmount DECIMAL(10,2) NOT NULL,

FPerson VARCHAR(20),PRIMARY KEY (FNumber),

FOREIGN KEY (FPerson) REFERENCES T_Person(FName)) ;

MSSQLServer：

CREATE TABLE T_Person (FName VARCHAR(20),FAge INT,FRemark VARCHAR(20),PRIMARY KEY (FName));

CREATE TABLE T_Debt (FNumber VARCHAR(20),FAmount NUMERIC(10,2) NOT NULL,

FPerson VARCHAR(20),PRIMARY KEY (FNumber),

FOREIGN KEY (FPerson) REFERENCES T_Person(FName)) ;

Oracle：

CREATE TABLE T_Person (FName VARCHAR2(20),FAge NUMBER (10) ,FRemark VARCHAR2(20),PRIMARY KEY (FName)) ;

CREATE TABLE T_Debt (FNumber VARCHAR2(20),FAmount NUMERIC(10,2) NOT NULL,

FPerson VARCHAR2(20),PRIMARY KEY (FNumber),

FOREIGN KEY (FPerson) REFERENCES T_Person(FName)) ;

DB2：

CREATE TABLE T_Person (FName VARCHAR(20) NOT NULL,FAge INT,FRemark VARCHAR(20),PRIMARY KEY (FName));

CREATE TABLE T_Debt (FNumber VARCHAR(20) NOT NULL,FAmount DECIMAL(10,2) NOT NULL,

FPerson VARCHAR(20),PRIMARY KEY (FNumber),

FOREIGN KEY (FPerson) REFERENCES T_Person(FName)) ;

请在不同的数据库系统中运行相应的 SQL 语句。T_Person 为记录人员信息的数据表，其中主键字段 FName 为人员姓名，FAge 为年龄，而 FRemark 则为备注信息；T_Debt 记录了债务信息，其中主键字段 FNumber 为债务编号，FAmount 为欠债金额，FPerson 字段为欠债人姓名，FPerson 字段与 T_Person 中的 FName 字段建立了外键关联关系。

3.1 数据的插入

数据表是数据的容器，没有任何数据的数据表是没有意义的，数据表创建完成以后比如向其中插入有用的数据才能使得系统运转起来。

3.1.1 简单的 INSERT 语句

INSERT INTO 语句用来向数据表中插入数据，比如执行下面的语句就可以向 T_Person 表中插入一条数据：

INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Tom',18,'USA')[3]

这句 SQL 向 T_Person 表中插入了一条数据，其中 FName 字段的值为'Tom'，FAge 字段的值为 18，而 FRemark 字段的值为'USA'。VALUES 前边的括号中列出的是要设置字段的字段名，字段名之间用逗号隔开；VALUES 后边的括号中列出的是要设置字段的值，各个值同样用逗号隔开。需要注意的是 VALUES 前列出的字段名和 VALUES 后边列出的字段值是按顺序一一对应的，也就是第一个值'Tom'设置的是字段 FName 的值，第二个值 18 设置的是字段 FAge 的值，第三个值'USA'设置的是字段 FRemark 的值，不能打乱它们之间的对应关系，而且要保证两边的条数是一致的。由于 FName 和 FRemark 字段是字符串类型的，所

[3] 需要注意，这里的单引号是半角字符，如果使用全角字符将会导致执行错误。

以需要用单引号[4]将值包围起来，而整数类型的 FAge 字段的值则不需要用单引号包围起来。

我们来检验一下数据是否真的插入数据表中了，执行下面的 SQL 语句：

SELECT * FROM T_Person[5]

执行完毕我们将会看到如下的输出结果（在不同的数据库系统以及管理工具下的显示效果会略有不同）：

FName	FAge	FRemark
Tom	18	USA

可以看到插入的数据已经保存在 T_Person 表中了，我们还可以运行多条 SQL 语句来插入多条数据：

```
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Jim',20,'USA');
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Lili',22,'China') ;
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('XiaoWang',17,' China ') ;
```

再次执行 SELECT * FROM T_Person 来查看表中的数据：

FNAME	FAGE	FREMARK
Tom	18	USA
Jim	20	USA
Lili	22	China
XiaoWang	17	China

INSERT 语句中列的顺序可以是任意的，比如我们也可以用下面的 SQL 来插入数据：

```
INSERT INTO T_Person(FAge,FName,FRemark) VALUES(21,'Kimisushi','Korea')
```

执行 SELECT * FROM T_Person 来查看表中的数据：

FNAME	FAGE	FREMARK
Tom	18	USA
Jim	20	USA
Lili	22	China
XiaoWang	17	China
Kimisushi	21	Korea

可见 INSET 语句中列的顺序不会影响数据插入的结果。

3.1.2 简化的 INSERT 语句

INSERT 语句中也并不需要我们指定表中的所有列，比如在插入数据的时候某些字段没有值，我们可以忽略这些字段。下面我们插入一条没有备注信息的数据：

```
INSERT INTO T_Person(FAge,FName) VALUES(22,'LXF')
```

执行 SELECT * FROM T_Person 来查看表中的数据：

FName	FAge	FRemark
Tom	18	USA
Jim	20	USA
Lili	22	China
XiaoWang	17	China
Kimisushi	21	Korea
LXF	22	<NULL>

INSERT 语句还有另一种用法，可以不用指定要插入的表列，这种情况下将按照定义表

[4]有的数据库系统也支持用双引号来包围，不过为了使得我们编写的 SQL 更容易的在主流数据库系统中运行，本书将一律采用单引号来包围字符串类型数据。

[5]先不用管这句 SQL 语句的具体语法，只要知道它是用来查看表 T_Person 中的数据即可。

中字段顺序来进行插入，我们执行下面的 SQL：

```
INSERT INTO T_Person VALUES('luren1',23,'China')
```

这里省略了 VALUES 前面的字段定义，VALUES 后面的值列表中按照 CREATE TABLE 语句中的顺序排列。执行 SELECT * FROM T_Person 来查看表中的数据：

FNAME	FAGE	FREMARK
Tom	18	USA
Jim	20	USA
Lili	22	China
XiaoWang	17	China
Kimisushi	21	Korea
LXF	22	<NULL>
luren1	23	China

这种省略字段列表的方法可以简化输入，不过我们推荐这种用法，因为省略字段列表之后就无法很容易的弄清楚值列表中各个值到底对应哪个字段了，非常容易导致程序出现 BUG 并且给程序的调试带来非常大的麻烦。

3.1.3 非空约束对数据插入的影响

正如“非空约束”表达的意思，如果对一个字段添加了非空约束，那么我们是不能向这个字段中插入 NULL 值的。T_Debt 表的 FAmount 字段是有非空约束的，如果我们执行下面 SQL：

```
INSERT INTO T_Debt (FNumber, FPerson) VALUES ('1', 'Jim')
```

这句 SQL 中没有为字段 FAmount 赋值，也就是说 FAmount 为空值。我们执行这句 SQL 以后数据库系统会报出类似如下的错误信息：

```
不能将值 NULL 插入列 'FAmount'，表 'demo.dbo.T_Debt'；列不允许有空值。INSERT 失败。
```

如果我们为 FAmount 设置非空值的话，则会插入成功，执行下面的 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('1',200, 'Jim')
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim

可以看到数据已经被正确的插入到表中了。

3.1.3 主键对数据插入的影响

主键是在同一张表中必须是唯一的，如果在进行数据插入的时候指定的主键与表中已有的数据重复的话则会导致违反主键约束的异常。T_Debt 表中 FNumber 字段是主键，如果我们执行下面 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('1',300, 'Jim')
```

由于在上一节中我们已经向表中插入了一条 FNumber 字段为 1 的记录，所以运行这句 SQL 的时候会报出类似如下的错误信息：

```
不能在对象 'dbo.T_Debt' 中插入重复键。
```

而如果我们为 FNumber 设置一个不重复值的话，则会插入成功，执行下面的 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('2',300, 'Jim')
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim
2	300.00	Jim

可以看到数据已经被正确的插入到表中了。

3.1.4 外键对数据插入的影响

外键是指向另一个表中已有数据的约束，因此外键值必须是在目标表中存在的。如果插入的数据在目标表中不存在的话则会导致违反外键约束异常。T_Debt 表中 FPerson 字段是指向表 T_Person 的 FName 字段的外键，如果我们执行下面 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('3',100, 'Jerry')
```

由于在 T_Person 表中不存在 FName 字段等于“Jerry”的数据行，所以会数据库系统会报出类似如下的错误信息：

INSERT 语句与 FOREIGN KEY 约束"FK__T_Debt__FPerson__1A14E395"冲突。该冲突发生于数据库"demo"，表"dbo.T_Person", column 'FName'。

而如果我们为 FPerson 字段设置已经在 T_Person 表中存在的 FName 字段值的话则会插入成功，执行下面的 SQL：

```
INSERT INTO T_Debt (FNumber,FAmount, FPerson) VALUES ('3',100, 'Tom')
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim
2	300.00	Jim
3	100.00	Tom

可以看到数据已经被正确的插入到表中了。

3.2 数据的更新

录入到数据表中的数据很少有一成不变的，随着系统的运行经常需要更新表中的某些数据，比如 Tom 的家庭住址变化了我们就要在数据库中将他的家庭住址更新、新年度到来的时候我们就要将所有人员的年龄增加一岁，类似需求都要求对数据库中现有的数据进行更新。

3.2.1 简单的数据更新

UPDATE 语句用来对数据表中的数据进行更新。下边的语句用来将表 T_Person 中所有人员的 FREMARK 字段值更新为“SuperMan”：

```
UPDATE T_Person
SET FRemark = 'SuperMan'
```

执行 SELECT * FROM T_Person 来查看表中的数据：

FName	FAge	FRemark
Jim	20	SuperMan
Kimisushi	21	SuperMan
Lili	22	SuperMan
luren1	23	SuperMan
LXF	22	SuperMan
Tom	18	SuperMan
XiaoWang	17	SuperMan

可以看到所有行的 FRemark 字段值都被设置成了“SuperMan”。

来看一下刚才执行的 SQL 语句，首先它声明了要更新的表为 T_Person：

```
UPDATE T_Person
```

在 SET 子句中，我们指定将 FRemark 字段更新为新值'SuperMan'：

```
SET FRemark = 'SuperMan'
```

我们还可以在 SET 语句中定义多个列，这样就可以实现多列同时更新了，比如下面的

UPDATE 语句用来将所有人员的 FRemark 字段更新为“Sonic”，并且将年龄更新为 25：

```
UPDATE T_Person
SET FRemark = 'Sonic',
FAge=25
```

多个列之间需要使用逗号分隔开。执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FName	FAge	FRemark
Jim	25	Sonic
Kimisushi	25	Sonic
Lili	25	Sonic
luren1	25	Sonic
LXF	25	Sonic
Tom	25	Sonic
XiaoWang	25	Sonic

3.2.2 带 WHERE 子句的 UPDATE 语句

目前演示的几个 UPDATE 语句都是一次性更新所有行的数据，这无法满足只更新符合特定条件的行的需求，比如“将 Tom 的年龄修改为 12 岁”。要实现这样的功能只要使用 WHERE 子句就可以了，在 WHERE 语句中我们设定适当的过滤条件，这样 UPDATE 语句只会更新符合 WHERE 子句中过滤条件的行，而其他行的数据则不被修改。

执行下边的 UPDATE 语句：

```
UPDATE T_Person
SET FAge = 12
WHERE FNAME='Tom'
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FName	FAge	FRemark
Jim	25	Sonic
Kimisushi	25	Sonic
Lili	25	Sonic
luren1	25	Sonic
LXF	25	Sonic
Tom	12	Sonic
XiaoWang	25	Sonic

可以看到只有第一行中的 FAGE 被更新了。WHERE 子句“WHERE FNAME='Tom'”表示我们只更新 FNAME 字段等于'Tom'的行。由于 FNAME 字段等于'Tom'的只有一行，所以仅有一行记录被更新，但是如果有多个符合条件的行的话将会有多行被更新，比如下面 UPDATE 语句将所有年龄为 25 的人员的备注信息修改为“BlaBla”：

```
UPDATE T_Person
SET FRemark = 'BlaBla'
WHERE FAge =25
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FName	FAge	FRemark
Jim	25	BlaBla
Kimisushi	25	BlaBla
Lili	25	BlaBla

luren1	25	BlaBla
LXF	25	BlaBla
Tom	12	Sonic
XiaoWang	25	BlaBla

目前为止我们演示的都是非常简单的 WHERE 子句，我们可以使用复杂的 WHERE 语句来满足更加复杂的需求，比如下面的 UPDATE 语句就用来将 FNAME 等于'Jim'或者'LXF'的行的 FAge 字段更新为 22：

```
UPDATE T_Person
SET FAge = 22
WHERE FName ='jim' OR FName='LXF'
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FName	FAge	FRemark
Jim	22	BlaBla
Kimisushi	25	BlaBla
Lili	25	BlaBla
luren1	25	BlaBla
LXF	22	BlaBla
Tom	12	Sonic
XiaoWang	25	BlaBla

这里我们使用 OR 逻辑运算符来组合两个条件来实现复杂的过滤逻辑，我们还可以使用 OR、NOT 等运算符实现更加复杂的逻辑，甚至能够使用模糊查询、子查询等实现高级的数据过滤，关于这些知识我们将在后面的章节专门介绍。

3.2.3 非空约束对数据更新的影响

正如“非空约束”表达的意思，如果对一个字段添加了非空约束，那么我们是不能将这个字段中的值更新为 NULL 的。T_Debt 表的 FAmount 字段是有非空约束的，如果我们执行下面 SQL：

UPDATE T_Debt set FAmount = NULL WHERE FPerson='Tom'

这句 SQL 为 FAmount 设置空值。我们执行这句 SQL 以后数据库系统会报出类似如下的错误信息：

不能将值 NULL 插入列 'FAmount'，表 'demo.dbo.T_Debt'；列不允许有空值。UPDATE 失败。

如果我们为 FAmount 设置非空值的话，则会插入成功，执行下面的 SQL：

UPDATE T_Debt set FAmount =123 WHERE FPerson='Tom'

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim
2	300.00	Jim
3	123.00	Tom

可以看到数据已经被正确的更新到表中了。

3.2.3 主键对数据更新的影响

主键是在同一张表中必须是唯一的，如果在进行数据更新的时候指定的主键与表中已有的数据重复的话则会导致违反主键约束的异常。T_Debt 表中 FNumber 字段是主键，如果我们执行下面 SQL：

```
UPDATE T_Debt set FNumber = '2' WHERE FPerson='Tom'
```

由于表中已经存在一条 FNumber 字段为 2 的记录，所以运行这句 SQL 的时候会报出类似如下的错误信息：

违反了 PRIMARY KEY 约束 'PK__T_Debt__1920BF5C'。不能在对象 'dbo.T_Debt' 中插入重复键。

而如果我们为 FNumber 设置一个不重复值的话，则会插入成功，执行下面的 SQL：

```
UPDATE T_Debt set FNumber = '8' WHERE FPerson='Tom'
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Jim
2	300.00	Jim
8	123.00	Tom

可以看到数据已经被正确的更新到表中了。

3.2.4 外键对数据更新的影响

外键是指向另一个表中已有数据的约束，因此外键值必须是在目标表中存在的。如果更新后的数据在目标表中不存在的话则会导致违反外键约束异常。T_Debt 表中 FPerson 字段是指向表 T_Person 的 FName 字段的外键，如果我们执行下面 SQL：

```
UPDATE T_Debt set FPerson = 'Merry' WHERE FNumber='1'
```

由于在 T_Person 表中不存在 FName 字段等于“Merry”的数据行，所以会数据库系统会报出类似如下的错误信息：

UPDATE 语句与 FOREIGN KEY 约束"FK__T_Debt__FPerson__1A14E395"冲突。该冲突发生于数据库"demo"，表"dbo.T_Person", column 'FName'。

而如果我们为 FPerson 字段设置已经在 T_Person 表中存在的 FName 字段值的话则会插入成功，执行下面的 SQL：

```
UPDATE T_Debt set FPerson = 'Lili' WHERE FNumber='1'
```

此句 SQL 则可以正常的执行成功。执行 SELECT * FROM T_Debt 来查看表中的数据：

FNumber	FAmount	FPerson
1	200.00	Lili
2	300.00	Jim
8	123.00	Tom

可以看到数据已经被正确的更新到表中了。

3.3 数据的删除

数据库中的数据一般都有一定的生命周期，当数据不再需要的时候我们就要将其删除，执行 DELETE 语句就可以将数据从表中删除。不过需要注意的就是如果被删除的数据行是某个外键关联关系中的被引用数据的话，则进行删除的时候会失败，如果要删除成功则必须首先删除引用者才可以。

3.3.1 简单的数据删除

删除数据的 SQL 语句非常简单，我们只要指定要删除的表就可以了，比如我们要将 T_Debt 和 T_Person 表中的数据删除，那么执行下面的 SQL 语句即可：

```
DELETE FROM T_Debt;
DELETE FROM T_Person;
```

由于 T_Debt 表中 FPerson 字段是指向表 T_Person 的 FName 字段的外键，所以必须首先删除 T_Debt 表中的数据后才能删除 T_Person 中的数据。

执行 SELECT * FROM T_Debt 查看 T_Debt 表中的数据变化：

FNumber	FAmount	FPerson

执行完此SQL语句后执行SELECT * FROM T_Person来查看T_Person表中的数据变化：

FName	FAge	FRemark

可以见表中所有的数据行都被删除了，T_Debt 和 T_Person 中没有任何数据。

初学者往往容易把 DROP TABLE 语句和 DELETE 混淆，虽然二者名字中都有“删除”两个字，不过 DELETE 语句仅仅是删除表中的数据行，而表的结构还存在，而 DROP TABLE 语句则不仅将表中的数据行全部删除，而且还将表的结构也删除。可以形象的比喻成 DELETE 语句仅仅是“吃光碗里的饭”，而 DROP TABLE 语句则是“吃光碗里的饭还将碗砸碎”。如果我们执行“DROP TABLE T_Person”的话，那么再次执行“SELECT * FROM T_Person”的时候数据库系统就会报告“数据表 T_Person 不存在”。

上边介绍的 DELETE 语句将表中的所有数据都删除了，如果我们只想删除我们指定的数据行怎么办呢？和 UPDATE 语句类似，DELETE 语句也提供了 WHERE 语句进行数据的过滤，这样只有符合过滤条件的数据行才会被删除。

3.3.2 带 WHERE 子句的 DELETE 语句

由于前面我们执行“DELETE FROM T_Person”语句将数据表 T_Person 中的数据全部删除了，为了演示带 WHERE 子句的 DELETE 语句，我们需要重新插入一些数据到 T_Person 中。请执行下面的 SQL 语句：

```
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Jim',20,'USA');
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Lili',22,'China') ;
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('XiaoWang',17,' China ') ;
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('Sam',16,'China') ;
INSERT INTO T_Person(FName,FAge,FRemark) VALUES('BlueFin',12,'Mars') ;
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看 T_Person 表中新插入的数据：

FNAME	FAGE	FREMARK
Jim	20	USA
Lili	22	China
XiaoWang	17	China
Sam	16	China
BlueFin	12	Mars

我们要删除年龄大于 20 岁或者来自火星（Mars）的人员，因此使用带复合逻辑 WHERE 子句，如下：

```
DELETE FROM T_Person WHERE FAge > 20 or FRemark = 'Mars'
```

执行完此 SQL 语句后执行 SELECT * FROM T_Person 来查看表中的数据的变化：

FNAME	FAGE	FREMARK
Jim	20	USA
XiaoWang	17	China
Sam	16	China

可以看到年龄为 22 岁的 Lili 和来自火星的 BlueFin 被删除了。

本章已经结束，我们不再需要 T_Person、T_Debt 这两张表，因此需要将它们删除，执行下面的 SQL 即可：

```
DROP TABLE T_Debt;
DROP TABLE T_Person;
```

第四章 数据的检索

4.1 SELECT 基本用法

4.1.1 检索所有列

4.1.2 检索指定列

列别名

4.1.3 按条件过滤

4.1.4 数据汇总（总个数，最高、最低工资、平均年龄，每个人和平均工资之间的差额），COUNT（*）和 COUNT（FID）的区别。

4.1.5 数据排序

4.1.5 案例分析：不稳定的排序结果

4.2 高级数据过滤

4.2.1 模糊匹配 Like。用 OR 模拟实现[]匹配

4.2.2 空值检测

4.2.3 多值检测

4.2.4 低效的 where 1=1 及解决方案

4.2.5 !=和<>

4.2.6 Between and

复杂过滤同样适用于 DELETE 和 UPDATE 语句

4.3 分组

4.3.1 分组的作用

4.3.2 数据分组

4.3.3 分组的陷阱

HAVING

GROUPBY 与 SUM。SELECT SUM(A),A from t

4.4 限制数据条数

4.4.1 不同数据库的实现

4.4.2 分页机制的实现

4.5 DISTINCT

4.5.1 同去存异

4.5.2 DISTINCT 的陷阱

4.5.3 案例分析：除去所有相同姓名的人

4.6 计算字段

4.6.1 字段间计算

常量间的计算、字段与常量的计算

不同数据库中连接字符串类型的差异（参考 SQL 入门的 68 页）

为什么不用+表示字符串连接

4.6.2 为字段增加注释：FName+’的工资为：’+FSalary

4.6.2 使用函数 用 NULLIF 将 null 的姓名转化为“佚名”

在 Beween And 中使用计算字段，来实现取得所有处于合理工资范围内的员工（Fsalary Between Fage*1.5+2000 And Fage*1.8+5000）

在 WHERE 语句和函数中使用计算字段

使用 Update t set fgroup=id/10 的例子来演示通过 SQL 而不是代码的方式来计算的好处

4.7 组合查询

与临时工表进行 Union。入库单、出库单进行 Union

4.7.1 何为 UNION

4.7.2 UNION 与 UNION ALL

到目前为止，我们已经学习了如何创建数据表、如何修改数据表以及如何删除数据表，我们还学习了如何将数据插入数据表、如何更新数据表中的数据以及如何数据删除。创建数据表是在创建存放数据的容器，修改和删除数据表是在维护数据模型的正确性，将数据插入数据表、更新数据表以及删除数据表中的数据则是在维护数据库中数据与真实业务数据之间的同步，这些操作都不是经常发生的，它们只占据数据库操作中很小的一部分，我们大部分时间都是在对数据库中的数据进行检索，并且基于检索结果进行响应的分析，可以说数据的检索是数据库中最重要的功能。

与数据表结构的管理以及数据表中数据的管理不同，数据检索所需要面对的问题是非常复杂的，不仅要求能够完成“检索出所有年龄小于 12 岁的学生”、“检索出所有旷工时间超过 3 天的职工”等简单的检索任务，而且还要完成“检索出本季度每种商品的出库入库详细情况”、“检索出所有学生家长的工作单位信息”等复杂的任务，甚至还需要完成其他更加复杂的检索任务。数据检索面对的场景是异常复杂的，因此数据检索的语法也是其他功能所不能比的，不仅语法规则非常复杂，而且使用方式也非常灵活。本书中大部分内容都是讲解数据检索相关知识的，为了降低学习的梯度，本章我们将讲解基本的数据检索语法，这些语法是数据检索功能中最基础也是最核心的部分，因此只有掌握我们才能继续学习更加复杂的应用。

本章中我们将使用一些数据表，为了更容易的运行本章中的例子，必须首先创建所需要的数据表，因此下面列出本章中要用到数据表的创建 SQL 语句：

MYSQL:

```
CREATE TABLE T_Employee (FNumber VARCHAR(20),FName VARCHAR(20),FAge INT,FSalary DECIMAL(10,2),PRIMARY KEY (FNumber))
```

MSSQLServer:

```
CREATE TABLE T_Employee (FNumber VARCHAR(20),FName VARCHAR(20),FAge INT,FSalary NUMERIC(10,2),PRIMARY KEY (FNumber))
```

Oracle:

```
CREATE TABLE T_Employee (FNumber VARCHAR2(20),FName VARCHAR2(20),FAge NUMBER (10),FSalary NUMERIC(10,2),PRIMARY KEY (FNumber))
```

DB2:

```
CREATE TABLE T_Employee (FNumber VARCHAR(20) NOT NULL,FName VARCHAR(20),FAge INT,FSalary DECIMAL(10,2),PRIMARY KEY (FNumber))
```

请在不同的数据库系统中运行相应的 SQL 语句。T_Employee 为记录员工信息的数据表，其中主键字段 FNumber 为员工工号，FName 为人员姓名，FAge 为年龄，FSalary 为员工月工资。

为了更加直观的验证本章中检索语句的正确性，我们需要在 T_Employee 表中预置一些初始数据，请在数据库中执行下面的数据插入 SQL 语句：

```
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('DEV001','Tom',25,8300);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('DEV002','Jerry',28,2300.80);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('SALES001','John',23,5000);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary)
```

VALUES('SALES002','Kerry',28,6200);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('SALES003','Stone',22,1200);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('HR001','Jane',23,2200.88);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('HR002','Tina',25,5200.36);
INSERT INTO T_Employee(FNumber,FName,FAge,FSalary) VALUES('IT001','Smith',28,3900);

4.1 SELECT 基本用法

SELECT 是实现数据检索的 SQL 语句，本节我们学习 SELECT 语句最基本的用法。

4.1.1 简单的数据检索

“取出一张表中所有的数据”是最简单的数据检索任务，完成这个最简单任务的 SQL 语句也是最简单的，我们只要执行“SELECT * FROM 表名”即可。比如我们执行下面的 SQL 语句：

```
SELECT * FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

执行结果中列出了表中的所有行，而且包含了表中每一列的数据。

4.1.2 检索出需要的列

上面的 SQL 语句执行的结果中包含了表中每一列的数据，有的时候并不需要所有列的数据。比如我们只需要检索所有员工的工号，如果我们采用“`SELECT * FROM T_Employee`”进行检索的话，数据库系统会将所有列的数据从数据库中取出来，然后通过网络发送给我们，这不仅会占用不必要的 CPU 资源和内存资源，而且会占用一定的网络带宽，这在我们这种测试模式下不会有影响，但是如果是在真实的生产环境中的话就会大大降低系统的吞吐量，因此最好在检索的之后只检索需要的列。那么如何只检索出需要的列呢？

检索出所有的列的 SQL 语句为“`SELECT * FROM T_Employee`”，其中的星号“*”就意味着“所有列”，那么我们只要将星号“*”替换成我们要检索的列名就可以了。比如我们执行下面的 SQL 语句：

```
SELECT FNumber FROM T_Employee
```

这就表示我们要检索出表 `T_Employee` 中的所有数据，并且只取出 FNumber 列。执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber
DEV001
DEV002
HR001

HR002
IT001
SALES001
SALES002
SALES003

可以看到只有 FNumber 列中的数据被检索出来了。

上面的 SQL 语句列出了 FNumber 列中的数据，那么如果想列出不止一个列中的数据呢？非常简单，只要在 SELECT 语句后列出各个列的列名就可以了，需要注意的就是各个列之间要用半角的逗号“,”分隔开。比如我们执行下面的 SQL 语句：

```
SELECT FName,FAge FROM T_Employee
```

这就表示我们要检索出表 `T_Employee` 中的所有数据，并且只取出 FName 和 FAge 两列的内容。执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FAge
Tom	25
Jerry	28
Jane	23
Tina	25
Smith	28
John	23
Kerry	28
Stone	22

可以看到，执行结果中列出了所有员工的姓名和他们的年龄。

如果要用这种显式指定数据列的方式取出所有列，我们就可以编写下面的 SQL：

```
SELECT FNumber,FName,FAge,FSalary FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

这和“`SELECT * FROM T_Employee`”的执行结果是一致的，也就是说“`SELECT FNumber,FName,FAge,FSalary FROM T_Employee`”和“`SELECT * FROM T_Employee`”是等价的。

4.1.3 列别名

由于编码命名规范、编程框架要求等的限制，数据表的列名有的时候意思并不是非常易读，比如 `T_Employee` 中的姓名字段名称为 `FName`，而如果我们能用 `Name` 甚至“姓名”来代表这个字段就更清晰易懂了，可是字段名已经不能更改了，那么难道就不能用别的名字来使用已有字段了吗？

当然不是！就像可以为每个人取一个外号一样，我们可以为字段取一个别名，这样就可以

使用这个别名来引用这个列了。别名的定义格式为“列名 AS 别名”，比如我们要为 FNumber 字段取别名为 Number1[6]，FName 字段取别名为 Name、FAge 字段取别名为 Age、为 FSalary 字段取别名为 Salary，那么编写下面的 SQL 即可：

```
SELECT FNumber AS Number1,FName AS Name,FAge AS Age,FSalary AS Salary FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

Number1	Name	Age	Salary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

这里的执行结果和“`SELECT FNumber,FName,FAge,FSalary FROM T_Employee`”执行结果一样，唯一不同的地方就是表头中的列名，这里的表头的列名就是我们为各列设定的别名。

定义别名的时候“AS”不是必须的，是可以省略的，比如下面的 SQL 也是正确的：

```
SELECT FNumber Number1,FName Name,FAge Age,FSalary Salary FROM T_Employee
```

如果数据库系统支持中文列名，那么还可以用中文来为列设定别名，这样可读性就更好了，比如在 MSSQLServer 中文版上执行下面的 SQL：

```
SELECT FNumber 工号,FName 姓名,FAge 年龄,FSalary 工资 FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

工号	姓名	年龄	工资
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

4.1.4 按条件过滤

前面演示的例子都是检索出表中所有的数据，不过在很多情况下我们需要按照一定的过滤条件来检索表中的部分数据，这个时候可以先检索出表中所有的数据，然后检查每一行看是否符合指定的过滤条件。比如我们要检索出所有工资少于 5000 元的员工的姓名，那么可以编写下面的代码来处理[7]：

```
result = executeQuery(“SELECT FName, FSalary FROM T_Employee”);
for(i=0;i<result.count;i++)
```

---

[6]由于 Number 在 Oracle 中为关键字，所以如果在为 FNumber 字段取别名为 Number，那么将会在 Oracle 中运行失败，所以这里取别名为 Number1。

[7]为了不涉及具体宿主语言的细节，这里采用的是实例性的类 C 伪代码，如果需要您可以将其翻译成对应宿主语言的代码。本书其他部分也将采用相同的伪代码来表示宿主语言无关的一些算法。

```
{
    salary = result[i].get("FSalary");
    if(salary<5000)
    {
        name = result[i].get("FName");
        print(name+"的工资少于 5000 元，为:"+salary);
    }
}
```

这种处理方式非常清晰简单，在处理小数据量以及简单的过滤条件的时候没有什么不妥的地方，但是如果数据表中有大量的数据（数以万计甚至百万、千万数量级）或者过滤条件非常复杂的话就会带来很多问题：

- 由于将表中所有的数据都从数据库中检索出来，所以会有非常大的内存消耗以及网络资源消耗。
- 需要逐条检索每条数据是否符合过滤条件，所以检索速度非常慢，当数据量大的时候这种速度是让人无法忍受的。
- 无法实现复杂的过滤条件。如果要实现“检索工资小于 5000 或者年龄介于 23 岁与 28 岁之间的员工姓名”这样的逻辑的话就要编写复杂的判断语句，而如果要关联其他表进行查询的话则会更加复杂。

数据检索是数据库系统的一个非常重要的任务，它内置了对按条件过滤数据的支持，只要为 SELECT 语句指定 WHERE 语句即可，其语法与上一章中讲的数据更新、数据删除的 WHERE 语句非常类似，比如完成“检索出所有工资少于 5000 元的员工的姓名”这样的功能可以使用下面的 SQL 语句：

```
SELECT FName FROM T_Employee
WHERE FSalary<5000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName
Jerry
Jane
Smith
Stone

WHERE 子句还支持复杂的过滤条件，下面的 SQL 语句用来检索出所有工资少于 5000 元或者年龄大于 25 岁的员工的所有信息：

```
SELECT * FROM T_Employee
WHERE FSalary<5000 OR FAge>25
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
IT001	Smith	28	3900.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

使用 WHERE 子句只需指定过滤条件就可以，我们无需关心数据库系统是如果进行查找的，数据库会采用适当的优化算法进行查询，大大降低了 CPU 资源的占用。

4.1.5 数据汇总

有时需要对数据库中的数据进行一些统计，比如统计员工总数、统计年龄大于 25 岁的员工中的最低工资、统计工资大于 3800 元的员工的平均年龄。SQL 中提供了聚合函数来完成计算统计结果集条数、某个字段的最大值、某个字段的最小值、某个字段的平均值以及某个字段的合计值等数据统计的功能，SQL 标准中规定了下面几种聚合函数：

函数名	说明
MAX	计算字段最大值
MIN	计算字段最小值
AVG	计算字段平均值
SUM	计算字段合计值
COUNT	统计数据条数

这几个聚合函数都有一个参数，这个参数表示要统计的字段名，比如要统计工资总额，那么就需要把 FSalary 做为 SUM 函数的参数。通过例子来看一下聚合函数的用法。第一个例子是查询年龄大于 25 岁的员工的最高工资，执行下面的 SQL：

```
SELECT MAX(FSalary) FROM T_Employee
WHERE FAge>25
```

执行完毕我们就能在输出结果中看到下面的执行结果：

6200.00

为了方面的引用查询的结果，也可以为聚合函数的计算结果指定一个别名，执行下面的 SQL：

```
SELECT MAX(FSalary) as MAX_SALARY FROM T_Employee
WHERE FAge>25
```

执行完毕我们就能在输出结果中看到下面的执行结果：

MAX_SALARY
6200.00

第二个例子我们来统计一下工资大于 3800 元的员工的平均年龄，执行下面的 SQL：

```
SELECT AVG(FAge) FROM  T_Employee
WHERE FSalary>3800
```

执行完毕我们就能在输出结果中看到下面的执行结果：

25

第三个例子我们来统计一下公司每个月应支出工资总额，执行下面的 SQL：

```
SELECT SUM(FSalary) FROM  T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

34302.04

我们还可以多次使用聚合函数，比如下面的 SQL 用来统计公司的最低工资和最高工资：

```
SELECT MIN(FSalary),MAX(FSalary) FROM  T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1200.00	8300.00

最后一个介绍的函数就是统计记录数量的 COUNT，这个函数有一点特别，因为它的即可以像其他聚合函数一样使用字段名做参数，也可以使用星号“*”做为参数。我们执行下面的 SQL：

```
SELECT COUNT(*),COUNT(FNumber) FROM  T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

8	8

可以看到 **COUNT**(*)、**COUNT**(FNumber)两种方式都能统计出记录的条数，据此为数不少的开发人员都认为 **COUNT**(*)、**COUNT**(字段名)这两种使用方式是等价的。下面通过例子来说

明，为了看到两种使用方式的区别需要首先向表 T_Employee 中插入一条新的数据，执行下面的 SQL：

```
INSERT INTO T_Employee(FNumber,FAge,FSalary) VALUES('IT002',27,2800)
```

需要注意的就是这句 INSERT 语句没有为 FName 字段赋值，也就是说新插入的这条数据的 FName 字段值为空，可以执行 SELECT * FROM T_Employee 来查看表 T_Employee 中的内容：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

可以看到 FNumber 为 IT002 的行的 FName 字段是空值。接着执行下面的 SQL：

```
SELECT COUNT(*),COUNT(FNumber),COUNT(FName) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

9	9	8

可以看到COUNT(*)、COUNT(FNumber)两个表达式的计算结果都是9，而COUNT(FName)的计算结果是8。也就反应出了两种使用方式的区别：COUNT(*)统计的是结果集的总条数，而COUNT(FName)统计的则是除了结果集中 FName 不为空值（也就是不等于 NULL）的记录的总条数。由于 FNumber 为 IT002 的行的 FName 字段是空值，所以 COUNT(FName)的计算结果是 8。因此在使用聚合函数 COUNT 的时候一定要区分两种使用方式的区别，以防止出现数据错误。

4.1.6 排序

到目前为止，数据检索结果的排列顺序取决于数据库系统所决定的排序机制，这种排序机制可能是按照数据的输入顺序决定的，也有可能是按照其他的算法来决定的。在有的情况下我们需要按照某种排序规则来排列检索结果，比如按照工资从高到低的顺序排列或者按照姓名的字符顺序排列等。SELECT 语句允许使用 ORDER BY 子句来执行结果集的排序方式。

ORDER BY 子句位于 SELECT 语句的末尾，它允许指定按照一个列或者多个列进行排序，还可以指定排序方式是升序（从小到大排列）还是降序（从大到小排列）。比如下面的 SQL 语句演示了按照年龄排序所有员工信息的列表：

```
SELECT * FROM T_Employee
ORDER BY FAge ASC
```

执行完毕我们就能在输出结果中看到下面的执行结果，可以看到输出结果已经按照 FAge 字段进行升序排列了：

FNumber	FName	FAge	FSalary
SALES003	Stone	22	1200.00
SALES001	John	23	5000.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
DEV001	Tom	25	8300.00

IT002	<NULL>	27	2800.00
SALES002	Kerry	28	6200.00
DEV002	Jerry	28	2300.80
IT001	Smith	28	3900.00

这句 SQL 中的“**ORDER BY** FAge ASC”指定了按照 FAge 字段的顺序进行升序排列，其中 ASC 代表升序。因为对于 ORDER BY 子句来说，升序是默认的排序方式，所以如果要采用升序的话可以不指定排序方式，也就是“ASC”是可以省略的，比如下面的 SQL 语句具有和上面的 SQL 语句等效的执行效果：

```
SELECT * FROM  T_Employee
ORDER BY FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果，可以看到输出结果同样按照 FAge 字段进行升序排列了：

FNumber	FName	FAge	FSalary
SALES003	Stone	22	1200.00
SALES001	John	23	5000.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
DEV001	Tom	25	8300.00
IT002	<NULL>	27	2800.00
SALES002	Kerry	28	6200.00
DEV002	Jerry	28	2300.80
IT001	Smith	28	3900.00

如果需要按照降序排列，那么只要将 ASC 替换为 DESC 即可，其中 DESC 代表降序。执行下面的 SQL 语句：

```
SELECT * FROM  T_Employee
ORDER BY FAge DESC
```

执行完毕我们就能在输出结果中看到下面的执行结果，可以看到输出结果已经按照 FAge 字段进行降序排序了：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
IT001	Smith	28	3900.00
SALES002	Kerry	28	6200.00
IT002	<NULL>	27	2800.00
DEV001	Tom	25	8300.00
HR002	Tina	25	5200.36
HR001	Jane	23	2200.88
SALES001	John	23	5000.00
SALES003	Stone	22	1200.00

可以看到上面的检索结果中有几组年龄相同的记录，这些年龄相同的记录之间的顺序是由数据库系统决定的，但是有时可能需要需要完成“按照年龄从大到小排序，如果年龄相同则按照工资从大到小排序”之类的排序功能。这可以通过指定多个排序规则来完成，因为 ORDER BY 语句允许指定多个排序列，各个列之间使用逗号隔开即可。执行下面的 SQL 语句：

```
SELECT * FROM  T_Employee
ORDER BY FAge DESC,FSalary DESC
```

FNumber	FName	FAge	FSalary
SALES002	Kerry	28	6200.00
IT001	Smith	28	3900.00
DEV002	Jerry	28	2300.80
IT002	<NULL>	27	2800.00
DEV001	Tom	25	8300.00
HR002	Tina	25	5200.36
SALES001	John	23	5000.00
HR001	Jane	23	2200.88
SALES003	Stone	22	1200.00

可以看到年龄相同的记录按照工资从高到低的顺序排列了。

对于多个排序规则，数据库系统会按照优先级进行处理。数据库系统首先按照第一个排序规则进行排序；如果按照第一个排序规则无法区分两条记录的顺序，则按照第二个排序规则进行排序；如果按照第二个排序规则无法区分两条记录的顺序，则按照第三个排序规则进行排序；……以此类推。以上面的 SQL 语句为例，数据库系统首先按照 FAge 字段的降序进行排列，如果按照个排序规则无法区分两条记录的顺序，则按照 FSalary 字段的降序进行排列。

ORDER BY 子句完全可以与 WHERE 子句一起使用，唯一需要注意的就是 ORDER BY 子句要放到 WHERE 子句之后，不能颠倒它们的顺序。比如我们尝试执行下面的 SQL 语句：

```
SELECT * FROM T_Employee
ORDER BY FAge DESC,FSalary DESC
WHERE FAge>23
```

执行以后数据库系统会报错提示此语句有语法错误，如果我们颠倒 ORDER BY 和 WHERE 子句的位置则可以执行通过：

```
SELECT * FROM T_Employee
WHERE FAge>23
ORDER BY FAge DESC,FSalary DESC
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
SALES002	Kerry	28	6200.00
IT001	Smith	28	3900.00
DEV002	Jerry	28	2300.80
IT002	<NULL>	27	2800.00
DEV001	Tom	25	8300.00
HR002	Tina	25	5200.36

前面我们提到，如果没有为 SELECT 语句指定 ORDER BY 子句，数据库系统会按照某种内置的规则对检索结果进行排序。如果您对检索结果的前后排列顺序有要求，那么即使数据库系统返回的检索结果符合要求也最好显式的指定 ORDER BY 子句，因为这种系统提供的排序方式是不稳定的，不仅在不同数据库系统之间存在差异，而且即使对同一种数据库系统来说在不同的条件下这种排序方式也是有可能发生改变的。

4.2 高级数据过滤

数据检索是数据库系统中最复杂的功能，而数据过滤则是数据检索中最核心的部分，到目前为止我们讲解的数据过滤都是“过滤某字段等于某个值的所有记录”、“过滤某字段小于某个值或者大于某个值的所有记录”等简单的数据过滤方式，这显然是无法满足真实业务系统中的各种数据过滤条件的，因此本节我们将介绍一些单表查询时的高级数据过滤技术。需

要注意的是，本节讲解的高级数据过滤技巧几乎同样适用于 Update 语句和 Delete 语句中的 Where 子句。

4.2.1 通配符过滤

到目前为止，我们讲解的数据过滤方式都是针对特定值的过滤，比如“检索所有年龄为 25 的所有员工信息”、“检索所有工资介于 2500 元至 3800 元之间的所有记录”，但是这种过滤方式并不能满足一些模糊的过滤方式。比如，检索所有姓名中含有“th”的员工或者检索所有姓“王”的员工，实现这样的检索操作必须使用通配符进行过滤。

SQL 中的通配符过滤使用 LIKE 关键字，可以像使用 OR、AND 等操作符一样使用它，它是一个二元操作符，左表达式为待匹配的字段，而右表达式为待匹配的通配符表达式。通配符表达式由通配符和普通字符组成，主流数据库系统支持的通配符有单字符匹配和多字符匹配，有的数据库系统还支持集合匹配。

4.2.1.1 单字符匹配

进行单字符匹配的通配符为半角下划线“_”，它匹配单个出现的字符。比如通配符表达式“b_d”匹配第一个字符为 b、第二个字符为任意字符、第三个字符为 d 的字符串，“bed”、“bad”都能匹配这个表达式，而“bd”、“abc”、“build”等则不能匹配这个表达式；通配符表达式“_oo_”匹配第一个字符为任意字符、第二个字符为 o、第三个字符为 o、第四个字符为任意字符的字符串，“look”、“took”、“cool”都能匹配这个表达式，而“rom”、“todo”等则不能匹配这个表达式。

下面来演示一下单字符匹配的用法。我们来检索 `T_Employee` 表中 `FName` 字段匹配如下规则的数据行：以任意字符开头，剩余部分为“`erry`”。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“`_erry`”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '_erry'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
SALES002	Kerry	28	6200.00

“Jerry”、“Kerry”两个字符串能够匹配通配符表达式“`_erry`”，所以被显示到了结果集中，而其他数据行则由于不匹配此通配符表达式，所以被过滤掉了。

单字符匹配在通配符表达式中可以出现多次，比如我们要检索长度为 4、第三个字符为“n”、其它字符为任意字符的姓名。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“`__n_`”（注意前两个字符为连续的两个下划线），那么需要编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '__n_'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36

4.2.1.2 多字符匹配

使用下划线可以实现“匹配长度为 5、以 ab 开头、剩余字符任意”的功能，而对于“匹配以 k 开头，长度不限，剩余字符任意”这样的需求则无法满足，这时就需要使用多字符匹配了。进行多字符匹配的通配符为半角百分号“%”，它匹配任意次数（零或多个）出现的任意字符。比如通配符表达式“k%”匹配以“k”开头、任意长度的字符串，“k”、“kerry”、“kb” 都能匹配这个表达式，而“ark”、“luck”、“3kd”等则不能匹配这个表达式；配符表

达式“b%t”匹配以“b”开头、以“t”结尾、任意长度的字符串，“but”、“bt”、“belt” 都能匹配这个表达式，而“turbo”、“tube”、“tb”等则不能匹配这个表达式。

下面来演示一下多字符匹配的用法。我们来检索 T_Employee 表中 FName 字段匹配如下规则的数据行：以“T”开头长度，长度任意。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“T%”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE 'T%'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
HR002	Tina	25	5200.36

接着我们来检索姓名中包含字母“n”的员工信息，编写如下 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '%n%'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
SALES001	John	23	5000.00
SALES003	Stone	22	1200.00

单字符匹配和多字符匹配还可以一起使用。我们来检索 T_Employee 表中 FName 字段匹配如下规则的数据行：最后一个字符为任意字符、倒数第二个字符为“n”、长度任意的字符串。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“%n_”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '%n_'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
SALES003	Stone	22	1200.00

4.2.1.3 集合匹配

集合匹配只在 MSSQLServer 上提供支持，在 MYSQL、Oracle、DB2 等数据库中不支持，必须采用变通的手段来实现。

进行集合匹配的通配符为“[]”，方括号中包含一个字符集，它匹配与字符集中任意一个字符相匹配的字符。比如通配符表达式“[bt]%”匹配第一个字符为 b 或者 t、长度不限的字符串，“bed”、“token”、“t”都能匹配这个表达式，而“at”、“lab”、“lot”等则不能匹配这个表达式。

下面来演示一下多字符匹配的用法。我们来检索 T_Employee 表中 FName 字段匹配如下规则的数据行：以“S”或者“J“开头长度，长度任意。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“[SJ]%”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '[SJ]%'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES003	Stone	22	1200.00

还可以使用否定符“^”来对集合取反，它匹配不与字符集中任意一个字符相匹配的字符。比如通配符表达式“[^bt]%”匹配第一个字符不为 b 或者 t、长度不限的字符串，“at”、“lab”、“lot”都能匹配这个表达式，而 “bed”、“token”、“t”等则不能匹配这个表达式。

我们来检索 T_Employee 表中 FName 字段匹配如下规则的数据行：不以“S”或者“J“开头长度，长度任意。根据通配符表达式语法，我们得知这个匹配规则对应的通配符表达式为“[^SJ]%”，因此编写如下的 SQL：

```
SELECT * FROM T_Employee
WHERE FName LIKE '[^SJ]%'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
HR002	Tina	25	5200.36
SALES002	Kerry	28	6200.00

集合匹配只在 MSSQLServer 上提供支持，不过在其他数据库中我们可以通过变通手段来实现相同的效果。比如下面的 SQL 可以实现和本节第一个例子相同的效果：

```
SELECT * FROM T_Employee
WHERE FName LIKE 'S%' OR FName LIKE 'J%'
```

而下面的 SQL 可以实现和本节第二个例子相同的效果：

```
SELECT * FROM T_Employee
WHERE NOT(FName LIKE 'S%') AND NOT(FName LIKE 'J%')
```

通配符过滤一个非常强大的功能，不过在使用通配符过滤进行检索的时候，数据库系统会对全表进行扫描，所以执行速度非常慢。因此不要过分使用通配符过滤，在使用其他方式可以实现的效果的时候就应该避免使用通配符过滤。

4.2.2 空值检测

没有添加非空约束列是可以为空值的（也就是 NULL），有时我们需要对空值进行检测，比如要查询所有姓名未知的员工信息。既然 NULL 代表空值，有的开发人员试图通过下面的 SQL 语句来实现：

```
SELECT * FROM T_Employee
WHERE FNAME=null
```

这个语句是可以执行的，不过执行以后我们看不到任何的执行结果，那个Fnumber为“IT002”的数据行中Fname字段为空，但是没有被查询出来。这是因为在SQL语句中对空值的处理有些特别，不能使用普通的等于运算符进行判断，而要使用IS NULL关键字，使用方法为“待检测字段名IS NULL”，比如要查询所有姓名未知的员工信息，则运行下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNAME IS NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
IT002	<NULL>	27	2800.00

如果要检测“字段不为空”，则要使用IS NOT NULL，使用方法为“待检测字段名IS NOT NULL”，比如要查询所有姓名已知的员工信息，则运行下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNAME IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
SALES003	Stone	22	1200.00

IS NULL/IS NOT NULL可以和其他的过滤条件一起使用。比如要查询所有姓名已知且工资小于5000的员工信息，则运行下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNAME IS NOT NULL AND FSalary <5000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
IT001	Smith	28	3900.00
SALES003	Stone	22	1200.00

4.2.3 反义运算符

“=”、“<”、“>”等运算符都是用来进行数值判断的，有的时候则会想使用这些运算符的反义，比如“不等于”、“不小于”或者“不大于”，MSSQLServer、DB2提供了“!”运算符来对运算符求反义，也就是“!=”表示“不等于”、“!<”表示“不小于”，而“!>”表示“不大于”。

比如要完成下面的功能“检索所有年龄不等于22岁并且工资不小于2000元”，我们可以编写下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FAge!=22 AND FSALARY!<2000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FNAME	FAGE	FSALARY
DEV001	Tom	25	8300.00
DEV002	Jerry	28	2300.80
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT001	Smith	28	3900.00

“!”运算符能够把“不等于”、“不大于”、“不小于”这样的语义直接翻译成SQL运算符，不过这个运算符只在MSSQLServer和DB2两种数据库系统上提供支持，如果在其他数据库

系统上则可以用其他的变通的方式实现，最常用的变通实现方式有两种：使用同义运算符、使用NOT运算符。

否定的语义都有对应的同义运算符，比如“不大于”的同义词是“小于等于”、而“不小于”的同义词是“大于等于”，同时SQL提供了通用的表示“不等于”的运算符“<>”，这样“不等于”、“不大于”和“不小于”就分别可以表示成“<>”、“<=”和“>=”。 因此要完成下面的功能“检索所有年龄不等于22岁并且工资不小于2000元”，我们可以编写下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FAge<>22 AND FSALARY>=2000
```

NOT运算符用来将一个表达式的值取反，也就是将值为“真”的表达式结果变为“假”、 将值为“假”的表达式结果变为“真”，使用方式也非常简单“NOT (表达式)”，比如要表达“年龄不小于20”，那么可以如下使用“NOT(Fage<20)”。因此要完成下面的功能“检索所有年龄不等于22岁并且工资不小于2000元”，我们可以编写下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE NOT(FAge=22) AND NOT(FSALARY<2000)
```

使用“!”运算符的方式由于只能运行在MSSQLServer和DB2两种数据库系统上，所以如果应用程序有移植到其他数据库系统上的需求的话，就应该避免使用这种方式；使用同义运算符的方式能够运行在所有主流数据库系统上，不过由于粗心等原因，很容易将“不大于”表示成“<”，而忘记了“不大于”是包含“小于”和“等于”这两个意思的，这样就会造成检索数据的错误，造成应用程序的Bug；而采用NOT运算符的方式能比较容易的表达要实现的需求，而且能够实现复杂的嵌套，最重要的是避免了潜在的应用程序的Bug，所以除了“<>”这种方式之外，我们推荐使用NOT运算符的方式来表示“非”的语义。

4.2.4 多值检测

“公司要为年龄为23岁、25岁和28岁的员工发福利，请将他们的年龄、工号和姓名检索出来”，要完成这样的功能，我们可以使用OR语句来连接多个等于判断。SQL语句如下：

```
SELECT FAge,FNumber,FName FROM T_Employee
WHERE FAge=23 OR FAge=25 OR FAge=28
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	FNumber	FName
25	DEV001	Tom
28	DEV002	Jerry
23	HR001	Jane
25	HR002	Tina
28	IT001	Smith
23	SALES001	John
28	SALES002	Kerry

这里要检索的年龄值是很少的，只有3个，如果要求我们“检索年龄为21岁、22岁、25岁、28岁、30岁、33岁、35岁、38岁、46岁的员工信息”，那么我们就要用OR连接九个等于判断：

```
SELECT FAge,FNumber,FName FROM T_Employee
WHERE FAge=21 OR FAge=22 OR FAge=25
OR FAge=28 OR FAge=30 OR FAge=33
OR FAge=35 OR FAge=38 OR FAge=46
```

这不仅写起来是非常麻烦的，而且维护的难度也相当大，一不小心就会造成数据错误。为了解决进行多个离散值的匹配问题，SQL提供了IN语句，使用IN我们只要指定要匹配的数据集

合就可以了，使用方法为“IN (值1,值2,值3……)”。要完成“公司要为年龄为23岁、25岁和28岁的员工发福利，请将他们的年龄、工号和姓名检索出来”这样功能的话，可以使用下面的SQL语句：

```
SELECT FAge,FNumber,FName FROM T_Employee
WHERE FAge IN (23,25,28)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	FNumber	FName
25	DEV001	Tom
28	DEV002	Jerry
23	HR001	Jane
25	HR002	Tina
28	IT001	Smith
23	SALES001	John
28	SALES002	Kerry

可以看到执行结果和使用OR语句来连接多个等于判断的方式是一样的。

使用IN我们还可以让字段与其他表中的值进行匹配，比如“查找所有姓名在迟到记录表中的员工信息”，要实现这样的功能就需要IN来搭配子查询来使用，关于这一点我们将在后面的章节介绍。

4.2.5 范围值检测

使用IN语句只能进行多个离散值的检测，如果要实现范围值的检测就非常麻烦甚至不可能了。比如我们要完成下面的功能“检索所有年龄介于23岁到27岁之间的员工信息”，如果用IN语句来实现的话就必须列出此范围内的所有可能的值，SQL如下：

```
SELECT * FROM T_Employee
WHERE FAGE IN(23,24,25,26,27)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00

当范围内的值比较多的时候使用这种方式非常麻烦，比如“检索所有年龄介于20岁到60岁之间的员工信息”就要列出20到60之间的每一个值，这个工作量是非常大的。而且这种方式也无法表达非离散的范围值，比如要实现“检索所有工资介于3000元到5000元之间的员工信息”的话就是不可能的，因为介于3000到5000之间的值是无数的。

这种情况下我们可以使用普通的“大于等于”和“小于等于”来实现范围值检测，比如完成下面的功能“检索所有年龄介于23岁到27岁之间的员工信息”，可以使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FAGE>=23 AND FAGE <=27
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
HR001	Jane	23	2200.88

HR002	Tina	25	5200.36
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00

这种方式能够实现几乎所有的范围值检测的功能，不过SQL提供了一个专门用语范围值检测的语句“BETTWEEN AND”，它可以用来检测一个值是否处于某个范围中（包括范围的边界值，也就是闭区间）。使用方法如下“字段名 BETTWEEN 左范围值 AND 右范围值”，其等价于“字段名>=左范围值 AND 字段名<=右范围值”。 比如完成下面的功能“检索所有年龄介于23岁到27岁之间的员工信息”，可以使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FAGE BETWEEN 23 AND 27
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV001	Tom	25	8300.00
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00

使用“BETTWEEN AND”我们还能够进行多个不连续范围值的检测，比如要实现“检索所有工资介于2000元到3000元之间以及5000元到8000元的员工信息”，可以使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE (FSalary BETWEEN 2000 AND 3000)
OR (FSalary BETWEEN 5000 AND 8000)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary
DEV002	Jerry	28	2300.80
HR001	Jane	23	2200.88
HR002	Tina	25	5200.36
IT002	<NULL>	27	2800.00
SALES001	John	23	5000.00
SALES002	Kerry	28	6200.00

数据库系统对“BETTWEEN AND”进行了查询优化，使用它进行范围值检测将会得到比其他方式更好的性能，因此在进行范围值检测的时候应该优先使用“BETTWEEN AND”。需要注意的就是“BETTWEEN AND”在进行检测的时候是包括了范围的边界值的（也就是闭区间），如果需要进行开区间或者半开半闭区间的范围值检测的话就必须使用其他的解决方案了。

4.2.6 低效的“WHERE 1=1”

网上有不少人提出过类似的问题：“看到有人写了WHERE 1=1这样的SQL，到底是什么意思？”。其实使用这种用法的开发人员一般都是在使用动态组装的SQL。

让我们想像如下的场景：用户要求提供一个灵活的查询界面来根据各种复杂的条件来查询员工信息，界面如下图：

界面中列出了四个查询条件，包括按工号查询、按姓名查询、按年龄查询以及按工资查询，每个查询条件前都有一个复选框，如果复选框被选中，则表示将其做为一个过滤条件。比如上图就表示“检索工号介于DEV001和DEV008之间、姓名中含有J并且工资介于3000元到6000元的员工信息”。如果不选中姓名前的复选框，比如下图表示“检索工号介于DEV001和DEV008之间并且工资介于3000元到6000元的员工信息”：

如果将所有的复选框都不选中，则表示表示“检索所有员工信息”，比如下图：

这里的数据检索与前面的数据检索都不一样，因为前边例子中的数据检索的过滤条件都是确定的，而这里的过滤条件则随着用户设置的不同而有变化，这时就要根据用户的设置来动态组装SQL了。当不选中年龄前的复选框的时候要使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNumber BETWEEN 'DEV001' AND 'DEV008'
AND FName LIKE '%J%'
AND FSalary BETWEEN 3000 AND 6000
```

而如果不选中姓名和年龄前的复选框的时候就要使用下面的SQL语句：

```
SELECT * FROM T_Employee
WHERE FNumber BETWEEN 'DEV001' AND 'DEV008'
AND FSalary BETWEEN 3000 AND 6000
```

而如果将所有的复选框都不选中的时候就要使用下面的SQL语句：

```
SELECT * FROM T_Employee
```

要实现这种动态的SQL语句拼装，我们可以在宿主语言中建立一个字符串，然后逐个判断各个复选框是否选中来向这个字符串中添加SQL语句片段。这里有一个问题就是当有复选框被选中的时候SQL语句是含有WHERE子句的，而当所有的复选框都没有被选中的时候就没有WHERE子句了，因此在添加每一个过滤条件判断的时候都要判断是否已经存在WHERE语句了，如果没有WHERE语句则添加WHERE语句。在判断每一个复选框的时候都要去判断，这使得用起来非常麻烦，“聪明的程序员是会偷懒的程序员”，因此开发人员想到了一个捷径：为SQL语句指定一个永远为真的条件语句（比如“1=1”），这样就不用考虑WHERE语句是否存在的问题了。伪代码如下[8]：

```
String sql = " SELECT * FROM T_Employee WHERE 1=1";
if(工号复选框选中)
{
    sql.appendLine("AND FNumber BETWEEN '"+工号文本框1内容+"' AND '"+工号文本框2内容+"'");
}
if(姓名复选框选中)
{
```

[8] 这里演示的将检索参数值直接拼接到 SQL 中的做法是有一定的问题的，会造成性能问题以及注入漏洞攻击。为了降低问题的复杂度，这里规避了这个问题，在本书的后续章节将会详细讲解。

```
    sql.appendLine("AND FName LIKE '%"+姓名文本框内容+"%'");
}
if(年龄复选框选中)
{
    sql.appendLine("AND FAge BETWEEN "+年龄文本框1内容+" AND "+年龄文本框2
内容);
}
executeSQL(sql);
```

这样如果不选中姓名和年龄前的复选框的时候就会执行下面的SQL语句：

```
SELECT * FROM T_Employee WHERE 1=1
AND FNumber BETWEEN 'DEV001' AND 'DEV008'
AND FSalary BETWEEN 3000 AND 6000
```

而如果将所有的复选框都不选中的时候就会执行下面的SQL语句：

```
SELECT * FROM T_Employee WHERE 1=1
```

这看似非常优美的解决了问题，殊不知这样很可能会造成非常大的性能损失，因为使用添加了“1=1”的过滤条件以后数据库系统就无法使用索引等查询优化策略，数据库系统将会被迫对每行数据进行扫描（也就是全表扫描）以比较此行是否满足过滤条件，当表中数据量比较大的时候查询速度会非常慢。因此如果数据检索对性能有比较高的要求就不要使用这种“简便”的方式。下面给出一种参考实现，伪代码如下：

```
private void doQuery()
{
    Bool hasWhere = false;
    StringBuilder sql = new StringBuilder(" SELECT * FROM T_Employee");
    if(工号复选框选中)
    {
        hasWhere = appendWhereIfNeed(sql, hasWhere);
        sql.appendLine("FNumber BETWEEN '"+工号文本框1内容+"' AND '"+工号
        文本框2内容+"'");
    }
    if(姓名复选框选中)
    {
        hasWhere = appendWhereIfNeed(sql, hasWhere);
        sql.appendLine("FName LIKE '%"+姓名文本框内容+"%'");
    }
    if(年龄复选框选中)
    {
        hasWhere = appendWhereIfNeed(sql, hasWhere);
        sql.appendLine("FAge BETWEEN "+年龄文本框1内容+" AND "+年龄文本框2
    内容);
    }
    executeSQL(sql);
}
private Bool appendWhereIfNeed(StringBuilder sql,Bool hasWhere)
{
```

```
    if(hasWhere==false)
    {
        sql. appendLine("WHERE");
    }
    else
    {
        sql. appendLine("AND");
    }
}
```

4.3 数据分组

前面我们讲解了聚合函数的使用，比如要查看年龄为23岁员工的人数，只要执行下面的SQL就可以：

```
SELECT COUNT(*) FROM T_Employee
WHERE FAge=23
```

可是如果我们想查看每个年龄段的员工的人数怎么办呢？一个办法是先得到所有员工的年龄段信息，然后分别查询每个年龄段的人数，显然这样是非常低效且烦琐的。这时候就是数组分组开始显现威力的时候了。

为了更好的演示本节中的例子，我们为T_Employee表增加两列，分别为表示其所属分公司的FSubCompany字段和表示其所属部门的FDepartment，在不同的数据库下执行相应的SQL语句：

```
MYSQL,MSSQLServer,DB2:
ALTER TABLE T_Employee ADD FSubCompany VARCHAR(20);
ALTER TABLE T_Employee ADD FDepartment VARCHAR(20);
Oracle:
ALTER TABLE T_Employee ADD FSubCompany VARCHAR2(20);
ALTER TABLE T_Employee ADD FDepartment VARCHAR2(20);
```

两个字段添加完毕后还需要将表中原有数据行的这两个字段值更新，执行下面的SQL语句：

```
UPDATE T_Employee SET FSubCompany='Beijing',FDepartment='Development'
WHERE FNumber='DEV001';
UPDATE T_Employee SET FSubCompany='ShenZhen',FDepartment='Development'
WHERE FNumber='DEV002';
UPDATE                          T_Employee                          SET
FSubCompany='Beijing',FDepartment='HumanResource'
WHERE FNumber='HR001';
UPDATE                          T_Employee                          SET
FSubCompany='Beijing',FDepartment='HumanResource'
WHERE FNumber='HR002';
UPDATE T_Employee SET FSubCompany='Beijing',FDepartment='InfoTech'
WHERE FNumber='IT001';
UPDATE T_Employee SET FSubCompany='ShenZhen',FDepartment='InfoTech'
WHERE FNumber='IT002';
UPDATE T_Employee SET FSubCompany='Beijing',FDepartment='Sales'
WHERE FNumber='SALES001';
UPDATE T_Employee SET FSubCompany='Beijing',FDepartment='Sales'
```

```
WHERE FNumber='SALES002';
UPDATE T_Employee SET FSubCompany='ShenZhen',FDepartment='Sales'
WHERE FNumber='SALES003';
```

4.3.1 数据分组入门

数据分组用来将数据分为多个逻辑组，从而可以对每个组进行聚合运算。SQL语句中使用GROUP BY子句进行分组，使用方式为“GROUP BY 分组字段”。分组语句必须和聚合函数一起使用，GROUP BY子句负责将数据分成逻辑组，而聚合函数则对每一个组进行统计计算。

虽然GROUP BY子句常常和聚合函数一起使用，不过GROUP BY子句并不是不能离开聚合函数而单独使用的，虽然不使用聚合函数的GROUP BY子句看起来用处不大，不过它能够帮助我们更好的理解数据分组的原理，所以本小节我们将演示GROUP BY子句的分组能力。

我们首先来看一下如果通过SQL语句实现“查看公司员工有哪些年龄段的”，因为这里只需要列出员工的年龄段，所以使用GROUP BY子句就完全可以实现：

```
SELECT FAge FROM T_Employee
GROUP BY FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge
22
23
25
27
28

这个SQL语句处理表中的所有记录，并且将FAge相同的数据行放到一组，分组后的数据可以看作一个临时的结果集，而SELECT FAge语句则取出每组的FAge字段的值，这样我们就得到上表的员工年龄段表了。

GROUP BY子句将检索结果划分为多个组，每个组是所有记录的一个子集。上面的SQL例子在执行的时候数据库系统将数据分成了下面的分组：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment	分组
SALES003	Stone	22	1200.00	ShenZhen	Sales	22 岁组
SALES001	John	23	5000.00	Beijing	Sales	23 岁组
HR001	Jane	23	2200.88	Beijing	HumanResource	
HR002	Tina	25	5200.36	Beijing	HumanResource	25 岁组
DEV001	Tom	25	8300.00	Beijing	Development	
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech	27 岁组
SALES002	Kerry	28	6200.00	Beijing	Sales	28 岁组
DEV002	Jerry	28	2300.80	ShenZhen	Development	
IT001	Smith	28	3900.00	Beijing	InfoTech	

需要注意的是GROUP BY子句的位置，GROUP BY子句必须放到SELECT语句的之后，如果SELECT语句有WHERE子句，则GROUP BY子句必须放到WHERE语句的之后。比如下面的SQL语句是错误的：

```
SELECT FAge FROM T_Employee
GROUP BY FAge
WHERE FSubCompany = 'Beijing'
```

而下面的SQL语句则是正确的：

```
SELECT FAge FROM T_Employee
WHERE FSubCompany = 'Beijing'
GROUP BY FAge
```

需要分组的所有列都必须位于GROUP BY子句的列名列表中，也就是没有出现在GROUP BY子句中的列（聚合函数除外）是不能放到SELECT语句后的列名列表中的。比如下面的SQL语句是错误的：

```
SELECT FAge,FSalary FROM T_Employee
GROUP BY FAge
```

道理非常简单，因为采用分组以后的查询结果集是以分组形式提供的，由于每组中人员的员工工资都不一样，所以就不存在能够统一代表本组工资水平的FSalary字段了，所以上面的SQL语句是错误的。不过每组中员工的平均工资却是能够代表本组统一工资水平的，所以可以对FSalary使用聚合函数，下面的SQL语句是正确的：

```
SELECT FAge,AVG(FSalary) FROM T_Employee
GROUP BY FAge
```

GROUP BY子句中可以指定多个列，只需要将多个列的列名用逗号隔开即可。指定多个分组规则以后，数据库系统将按照定义的分组顺序来对数据进行逐层分组，首先按照第一个分组列进行分组，然后在每个小组内按照第二个分组列进行再次分组……逐层分组，从而实现“组中组”的效果，而查询的结果集是以最末一级分组来进行输出的。比如下面的SQL语句将会列出所有分公司的所有部门情况：

```
SELECT FSubCompany,FDepartment FROM T_Employee
GROUP BY FSubCompany,FDepartment
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FSubCompany	FDepartment
Beijing	Development
Beijing	HumanResource
Beijing	InfoTech
Beijing	Sales
ShenZhen	Development
ShenZhen	InfoTech
ShenZhen	Sales

上面的SQL例子在执行的时候数据库系统将数据分成了下面的分组：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment	FSubCompany 分组	FDepartment 分组
DEV001	Tom	25	8300.00	Beijing	Development	Beijing 组	Development 组
HR001	Jane	23	2200.88	Beijing	HumanResource		HumanResource 组
HR002	Tina	25	5200.36	Beijing	HumanResource		
IT001	Smith	28	3900.00	Beijing	InfoTech		InfoTech 组
SALES001	John	23	5000.00	Beijing	Sales		Sales 组
SALES002	Kerry	28	6200.00	Beijing	Sales		
DEV002	Jerry	28	2300.80	ShenZhen	Development	ShenZhen 组	Development 组
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech		InfoTech 组

SALES003	Stone	22	1200.00	ShenZhen	Sales		Sales 组

4.3.2 数据分组与聚合函数

到目前为止我们使用的聚合函数都是对普通结果集进行统计的，我们同样可以使用聚合函数来对分组后的数据进行统计，也就是统计每一个分组的数据。我们甚至可以认为在没有使用GROUP BY语句中使用聚合函数不过是在一个整个结果集是一个组的分组数据中进行数据统计分析罢了。

让我们来看一下“查看每个年龄段的员工的人数”如何用数据分组来实现，下面是实现此功能的SQL语句：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
22	1
23	2
25	2
27	1
28	3

GROUP BY子句将检索结果按照年龄划分为多个组，每个组是所有记录的一个子集。上面的SQL例子在执行的时候数据库系统将数据分成了下面的分组：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment	分组
SALES003	Stone	22	1200.00	ShenZhen	Sales	22 岁组
SALES001	John	23	5000.00	Beijing	Sales	23 岁组
HR001	Jane	23	2200.88	Beijing	HumanResource	
HR002	Tina	25	5200.36	Beijing	HumanResource	25 岁组
DEV001	Tom	25	8300.00	Beijing	Development	
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech	27 岁组
SALES002	Kerry	28	6200.00	Beijing	Sales	28 岁组
DEV002	Jerry	28	2300.80	ShenZhen	Development	
IT001	Smith	28	3900.00	Beijing	InfoTech	

可以看到年龄相同的员工被分到了一组，接着使用“COUNT(*)”来统计每一组中的条数，这样就得到了每个年龄段的员工的个数了。

可以使用多个分组来实现更精细的数据统计，比如下面的SQL语句就可以统计每个分公司的年龄段的人数：

```
SELECT FSubCompany,FAge,COUNT(*) AS CountOfThisSubCompAge FROM
T_Employee
GROUP BY FSubCompany,FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FSubCompany	FAge	CountOfThisAge
ShenZhen	22	1
Beijing	23	2
Beijing	25	2

ShenZhen	27	1
Beijing	28	2
ShenZhen	28	1

上面的执行结果是按照数据库系统默认的年龄进行排序的，为了更容易的按照每个分公司进行查看，我们可以指定按照FSubCompany字段进行排序，带ORDER BY的SQL语句如下：

```
SELECT FSubCompany,FAge,COUNT(*) AS CountOfThisSubCompAge FROM
T_Employee
GROUP BY FSubCompany,FAge
ORDER BY FSubCompany
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FSubCompany	FAge	CountOfThisSubCompAge
Beijing	23	2
Beijing	25	2
Beijing	28	2
ShenZhen	22	1
ShenZhen	27	1
ShenZhen	28	1

上面的SQL语句中，GROUP BY子句将检索结果首先按照FSubCompany进行分组，然后在每一个分组内又按照FAge进行分组，数据库系统将数据分成了下面的分组：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment	FSubCompany 分组	FAge 分组
HR001	Jane	23	2200.88	Beijing	HumanResource	Beijing 组	23 岁组
SALES001	John	23	5000.00	Beijing	Sales		
DEV001	Tom	25	8300.00	Beijing	Development		25 岁组
HR002	Tina	25	5200.36	Beijing	HumanResource		
IT001	Smith	28	3900.00	Beijing	InfoTech		28 岁组
SALES002	Kerry	28	6200.00	Beijing	Sales		
SALES003	Stone	22	1200.00	ShenZhen	Sales	ShenZhen 组	22 岁组
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech		27 岁组
DEV002	Jerry	28	2300.80	ShenZhen	Development		28 岁组

“COUNT(*)”对每一个分组统计总数，这样就可以统计出每个公司每个年龄段的员工的人数了。

SUM、AVG、MIN、MAX也可以在分组中使用。比如下面的SQL可以统计每个公司中的工资的总值：

```
SELECT FSubCompany,SUM(FSalary) AS FSalarySUM FROM T_Employee
GROUP BY FSubCompany
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FSubCompany	FSalarySUM
Beijing	30801.24
ShenZhen	6300.80

下面的SQL可以统计每个垂直部门中的工资的平均值：

```
SELECT FDepartment,SUM(FSalary) AS FSalarySUM FROM T_Employee
GROUP BY FDepartment
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment	FSalarySUM
Development	10600.80
HumanResource	7401.24
InfoTech	6700.00
Sales	12400.00

下面的SQL可以统计每个垂直部门中员工年龄的最大值和最小值：

```
SELECT FDepartment,MIN(FAge) AS FAgeMIN,MAX(FAge) AS FAgeMAX FROM
T_Employee
GROUP BY FDepartment
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment	FAgeMIN	FAgeMAX
Development	25	28
HumanResource	23	25
InfoTech	27	28
Sales	22	28

4.3.3 HAVING 语句

有的时候需要对部分分组进行过滤，比如只检索人数多余1个的年龄段，有的开发人员会使用下面的SQL语句：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
WHERE COUNT(*)>1
```

可以在数据库系统中执行下面的SQL的时候，数据库系统会提示语法错误，这是因为聚合函数不能在WHERE语句中使用，必须使用HAVING子句来代替，比如：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
HAVING COUNT(*)>1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
23	2
25	2
28	3

HAVING语句中也可以像WHERE语句一样使用复杂的过滤条件，比如下面的SQL用来检索人数为1个或者3个的年龄段，可以使用下面的SQL：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
HAVING COUNT(*) =1 OR COUNT(*) =3
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
22	1
27	1

28	3

也可以使用IN操作符来实现上面的功能，SQL语句如下：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
HAVING COUNT(*) IN (1,3)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
22	1
27	1
28	3

HAVING语句能够使用的语法和WHERE几乎是一样的，不过使用WHERE的时候GROUP BY子句要位于WHERE子句之后，而使用HAVING子句的时候GROUP BY子句要位于HAVING子句之后，比如下面的SQL是错误的：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
HAVING COUNT(*) IN (1,3)
GROUP BY FAge
```

需要特别注意，在HAVING语句中不能包含未分组的列名，比如下面的SQL语句是错误的：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
GROUP BY FAge
HAVING FName IS NOT NULL
```

执行的时候数据库系统会提示类似如下的错误信息：

```
HAVING 子句中的列 'T_Employee.FName' 无效，因为该列没有包含在聚合函数或 GROUP BY 子句中。
```

需要用WHERE语句来代替HAVING，修改后的SQL语句如下：

```
SELECT FAge,COUNT(*) AS CountOfThisAge FROM T_Employee
WHERE FName IS NOT NULL
GROUP BY FAge
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FAge	CountOfThisAge
22	1
23	2
25	2
28	3

4.4 限制结果集行数

在进行数据检索的时候有时候需要只检索结果集中的部分行，比如说“检索成绩排前三名的学生”、“检索工资水平排在第3位到第7位的员工信息”，这种功能被称为“限制结果集行数”。在虽然主流的数据库系统中都提供了限制结果集行数的方法，但是无论是语法还是使用方式都存在着很大的差异，即使是同一个数据库系统的不同版本（比如MSSQLServer2000和MSSQLServer2005）也存在着一定的差异。因此本节将按照数据库系统来讲解每种数据库系统对限制结果集行数的特性支持。

4.4.1 MYSQL

MYSQL中提供了LIMIT关键字用来限制返回的结果集，LIMIT放在SELECT语句的最后位置，语法为“LIMIT 首行行号，要返回的结果集的最大数目”。比如下面的SQL语句将返回按照工资降序排列的从第二行开始（行号从0开始）的最多五条记录：

```
SELECT * FROM T_Employee ORDER BY FSalary DESC LIMIT 2,5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
IT001	Smith	28	3900.00	Beijing	InfoTech
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech
DEV002	Jerry	28	2300.80	ShenZhen	Development

很显然，下面的SQL语句将返回按照工资降序排列的前五条记录：

```
SELECT * FROM T_Employee ORDER BY FSalary DESC LIMIT 0,5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV001	Tom	25	8300.00	Beijing	Development
SALES002	Kerry	28	6200.00	Beijing	Sales
HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
IT001	Smith	28	3900.00	Beijing	InfoTech

4.4.2 MSSQLServer2000

MSSQLServer2000中提供了TOP关键字用来返回结果集中的前N条记录，其语法为“SELECT TOP 限制结果集数目 字段列表 SELECT语句其余部分”，比如下面的SQL语句用来检索工资水平排在前五位（按照工资从高到低）的员工信息：

```
select top 5 * from T_Employee order by FSalary Desc
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV001	Tom	25	8300.00	Beijing	Development
SALES002	Kerry	28	6200.00	Beijing	Sales
HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
IT001	Smith	28	3900.00	Beijing	InfoTech

MSSQLServer2000没有直接提供返回提供“检索从第5行开始的10条数据”、“检索第五行至第十二行的数据”等这样的取区间范围的功能，不过可以采用其他方法来变通实现，最常使用的方法就是用子查询[9]，比如要实现检索按照工资从高到低排序检索从第六名开始一共三个人的信息，那么就可以首先将前五名的主键取出来，在检索的时候检索排除了这五名员工的前三个人，SQL如下：

```
SELECT top 3 * FROM T_Employee
WHERE FNumber NOT IN
(SELECT TOP 5 FNumber FROM T_Employee ORDER BY FSalary DESC)
 ORDER BY FSalary DESC
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech

[9] 在本书的后面章节将会详细介绍子查询。对于子查询不熟悉的读者可以暂时认为子查询就是将另外一个查询结果当作一个新的表进行检索。

DEV002	Jerry	28	2300.80	ShenZhen	Development
HR001	Jane	23	2200.88	Beijing	HumanResource

4.4.3 MSSQLServer2005

MSSQLServer2005兼容几乎所有的MSSQLServer2000的语法，所以可以使用上个小节提到的方式来在MSSQLServer2005中实现限制结果集行数，不过MSSQLServer2005提供了新的特性来帮助更好的限制结果集行数的功能，这个新特性就是窗口函数ROW_NUMBER()。

ROW_NUMBER()函数可以计算每一行数据在结果集中的行号（从1开始计数），其使用语法如下：

```
ROW_NUMBER OVER(排序规则)
```

比如我们执行下面的SQL语句：

```
SELECT ROW_NUMBER() OVER(ORDER BY FSalary),FNumber,FName,FSalary,FAge
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

	FNumber	FName	FSalary	FAge
1	DEV001	Tom	8300.00	25
2	SALES002	Kerry	6200.00	28
3	HR002	Tina	5200.36	25
4	SALES001	John	5000.00	23
5	IT001	Smith	3900.00	28
6	IT002	<NULL>	2800.00	27
7	DEV002	Jerry	2300.80	28
8	HR001	Jane	2200.88	23
9	SALES003	Stone	1200.00	22

可以看到第一列中的数据就是通过ROW_NUMBER()计算出来的行号。有的开发人员想使用如下的方式来实现返回第3行到第5行的数据（按照工资降序）：

```
SELECT ROW_NUMBER() OVER(ORDER BY FSalary DESC),FNumber,FName,FSalary,FAge
FROM T_Employee
WHERE (ROW_NUMBER() OVER(ORDER BY FSalary DESC))>=3
AND (ROW_NUMBER() OVER(ORDER BY FSalary DESC))<=5
```

但是在运行的时候数据库系统会报出下面的错误信息：

```
开窗函数只能出现在 SELECT 或 ORDER BY 子句中。
```

也就是说ROW_NUMBER()不能用在WHERE语句中。我们可以用子查询来解决这个问题，下面的SQL语句用来返回第3行到第5行的数据：

```
SELECT * FROM
(
SELECT ROW_NUMBER() OVER(ORDER BY FSalary DESC) AS rownum,
FNumber,FName,FSalary,FAge FROM T_Employee
) AS a
WHERE a.rownum>=3 AND a.rownum<=5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

rownum	FNumber	FName	FSalary	FAge
3	HR002	Tina	5200.36	25
4	SALES001	John	5000.00	23

5	IT001	Smith	3900.00	28

4.4.4 Oracle

Oracle中支持窗口函数ROW_NUMBER()，其用法和MSSQLServer2005中相同，比如我们执行下面的SQL语句：

```
SELECT * FROM
(
SELECT ROW_NUMBER() OVER(ORDER BY FSalary DESC) row_num,
FNumber,FName,FSalary,FAge FROM T_Employee
) a
WHERE a.row_num>=3 AND a.row_num<=5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROW_NUM	FNUMBER	FNAME	FSALARY	FAGE
3	HR002	Tina	5200.36	25
4	SALES001	John	5000	23
5	IT001	Smith	3900	28

注意：rownum在Oracle中为保留字，所以这里将MSSQLServer2005中用到的rownum替换为row_num；Oracle中定义表别名的时候不能使用AS关键字，所以这里也去掉了AS。

Oracle支持标准的函数ROW_NUMBER()，不过Oracle中提供了更方便的特性用来计算行号，也就在Oracle中可以无需自行计算行号，Oracle为每个结果集都增加了一个默认的表示行号的列，这个列的名称为rownum。比如我们执行下面的SQL语句：

```
SELECT rownum,FNumber,FName,FSalary,FAge FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROWNUM	FNUMBER	FNAME	FSALARY	FAGE
1	DEV001	Tom	8300	25
2	DEV002	Jerry	2300.8	28
3	SALES001	John	5000	23
4	SALES002	Kerry	6200	28
5	SALES003	Stone	1200	22
6	HR001	Jane	2200.88	23
7	HR002	Tina	5200.36	25
8	IT001	Smith	3900	28
9	IT002	<NULL>	2800	27

使用rownum我们可以很轻松的取得结果集中前N条的数据行，比如我们执行下面的SQL语句可以得到按工资从高到底排序的前6名员工的信息：

```
SELECT * FROM T_Employee
WHERE rownum<=6
ORDER BY FSalary Desc
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FNAME	FAGE	FSALARY	FSUBCOMPANY	FDEPARTMENT
DEV001	Tom	25	8300	Beijing	Development
SALES002	Kerry	28	6200	Beijing	Sales
SALES001	John	23	5000	Beijing	Sales
DEV002	Jerry	28	2300.8	ShenZhen	Development

HR001	Jane	23	2200.88	Beijing	HumanResource
SALES003	Stone	22	1200	ShenZhen	Sales

看到这里，您可能认为下面的SQL就可以非常容易的实现“按照工资从高到低的顺序取出第三个到第五个员工信息”的功能了：

```
SELECT rownum,FNumber,FName,FSalary,FAge FROM T_Employee
WHERE rownum BETWEEN 3 AND 5
ORDER BY FSalary DESC
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROWNUM	FNUMBER	FNAME	FSALARY	FAGE

检索结果为空！！！这非常出乎我们的意料。让我们来回顾一下rownum的含义：rownum为结果集中每一行的行号（从1开始计数）。对于下面的SQL：

```
SELECT * FROM T_Employee
WHERE rownum<=6
ORDER BY FSalary Desc
```

当进行检索的时候，对于第一条数据，其rownum为1，因为符合“WHERE rownum<=6”所以被放到了检索结果中；当检索到第二条数据的时候，其rownum为2，因为符合“WHERE rownum<=6”所以被放到了检索结果中……依次类推，直到第七行。所以这句SQL语句能够实现“按照工资从高到低的顺序取出第三个到第五个员工信息”的功能。

而对于这句SQL语句：

```
SELECT rownum,FNumber,FName,FSalary,FAge FROM T_Employee
WHERE rownum BETWEEN 3 AND 5
ORDER BY FSalary DESC
```

当进行检索的时候，对于第一条数据，其rownum为1，因为不符合“WHERE rownum BETWEEN 3 AND 5”，所以没有被放到了检索结果中；当检索到第二条数据的时候，因为第一条数据没有放到结果集中，所以第二条数据的rownum仍然为1，而不是我们想像的2，所以因为不符合“WHERE rownum<=6”，没有被放到了检索结果中；当检索到第三条数据的时候，因为第一、二条数据没有放到结果集中，所以第三条数据的rownum仍然为1，而不是我们想像的3，所以因为不符合“WHERE rownum<=6”，没有被放到了检索结果中……依此类推，这样所有的数据行都没有被放到结果集中。

因此如果要使用rownum来实现“按照工资从高到低的顺序取出第三个到第五个员工信息”的功能，就必须借助于窗口函数ROW_NUMBER()。

4.4.5 DB2

DB2中支持窗口函数ROW_NUMBER()，其用法和MSSQLServer2005以及Oracle中相同，比如我们执行下面的SQL语句：

```
SELECT * FROM
(
SELECT ROW_NUMBER() OVER(ORDER BY FSalary DESC) row_num,
FNumber,FName,FSalary,FAge FROM T_Employee
) a
WHERE a.row_num>=3 AND a.row_num<=5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROW_NUM	FNUMBER	FNAME	FSALARY	FAGE
3	HR002	Tina	5200.36	25
4	SALES001	John	5000.00	23

5	IT001	Smith	3900.00	28

除此之外，DB2还提供了FETCH关键字用来提取结果集的前N行，其语法为“FETCH FIRST 条数 ROWS ONLY”，比如我们执行下面的SQL语句可以得到按工资从高到底排序的前6名员工的信息：

```
SELECT * FROM T_Employee
ORDER BY FSalary Desc
FETCH FIRST 6 ROWS ONLY
```

需要注意的是FETCH子句要放到ORDER BY语句的后面，执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FNAME	FAGE	FSALARY	FSUBCOMPANY	FDEPARTMENT
DEV001	Tom	25	8300.00	Beijing	Development
SALES002	Kerry	28	6200.00	Beijing	Sales
HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
IT001	Smith	28	3900.00	Beijing	InfoTech
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech

DB2没有直接提供返回提供“检索从第5行开始的10条数据”、“检索第五行至第十二行的数据”等这样的取区间范围的功能，不过可以采用其他方法来变通实现，最常使用的方法就是用子查询，比如要实现检索按照工资从高到低排序检索从第六名开始一共三个人的信息，那么就可以首先将前五名的主键取出来，在检索的时候检索排除了这五名员工的前三个人，SQL如下：

```
SELECT * FROM T_Employee
WHERE FNumber NOT IN
(
SELECT FNumber FROM T_Employee
ORDER BY FSalary DESC
FETCH FIRST 5 ROWS ONLY
)
ORDER BY FSalary DESC
FETCH FIRST 3 ROWS ONLY
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FNAME	FAGE	FSALARY	FSUBCOMPANY	FDEPARTMENT
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech
DEV002	Jerry	28	2300.80	ShenZhen	Development
HR001	Jane	23	2200.88	Beijing	HumanResource

4.4.6 数据库分页

在进行信息检索的时候，检索结果的数量通常会非常多，达到成百上千条，甚至更多，这么多的检索结果同时显示在同一个界面中，不仅查看起来非常麻烦，而且过多的数据显示在界面上也会造成占用过多的系统资源。解决这个问题的最常用方案就是数据库分页，就像我们在论坛上查看帖子的时候，一个网页中只显示50条帖子，论坛会提供【上一页】、【下一页】、【首页】以及【尾页】等按钮用来显示不同的页。实现数据库分页的核心技术就是“限制结果集行数”。

假设每一页显示的数据条数为PageSize，当前页数（从0开始技术）为CurrentIndex，那么我们只要查询从第PageSize*CurrentIndex开始的PageSize条数据得到的结果就是当前页中

的数据；当用户点击【上一页】按钮的时候，将CurrentIndex设置为CurrentIndex-1，然后重新检索；当用户点击【下一页】按钮的时候，将CurrentIndex设置为CurrentIndex+1，然后重新检索；当用户点击【首页】按钮的时候，将CurrentIndex设置为0，然后重新检索；当用户点击【首页】按钮的时候，将CurrentIndex设置为“总条数/PageSize”，然后重新检索。

下面我们将要用伪代码来演示数据库分页功能，其中的数据库系统使用的MYSQL：

```
int CurrentIndex=0;
PageSize=10;
//按钮【首页】被点击
private void btnFirstButtonClick()
{
    CurrentIndex=0;
    DoSearch();
}

//按钮【尾页】被点击
private void btnLastButtonClick()
{
    CurrentIndex=GetTotalCount()/PageSize;
    DoSearch();
}

//按钮【下一页】被点击
private void btnNextButtonClick()
{
    CurrentIndex= CurrentIndex+1;
    DoSearch();
}

//按钮【上一页】被点击
private void btnNextButtonClick()
{
    CurrentIndex= CurrentIndex-1;
    DoSearch();
}

//计算表中的总数据条数
private int GetTotalCount()
{
    ResultSet rs = ExecuteSQL("SELECT COUNT(*) AS TOTALCOUNT FROM T_Employee ");
    return rs.getInt("TOTALCOUNT");
}

//查询当前页中的数据
```

```
private void DoSearch()
{
    //计算当前页的起始行数
    String startIndex = (CurrentIndex* PageSize).ToString();
    String size = PageSize.ToString()
    ResultSet rs = ExecuteSQL("SELECT * FROM T_Employee   LIMIT "
                                + startIndex +","+ size);
    //显示查询结果
    DisplayResult(rs);
}
```

4.5 抑制数据重复

如果要检索公司里有哪些垂直部门，那么可以执行下面的SQL语句：

```
SELECT FDepartment FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment
Development
Development
HumanResource
HumanResource
InfoTech
InfoTech
Sales
Sales
Sales

这里列出了公司所有的垂直部门，不过很多部门名称是重复的，我们必须去掉这些重复的部门名称，每个重复部门只保留一个名称。**DISTINCT关键字是用来进行重复数据抑制的最简单的功能，而且所有的数据库系统都支持DISTINCT，DISTINCT的使用也非常简单，只要在SELECT之后增加DISTINCT即可。比如下面的SQL语句用于**检索公司里有哪些垂直部门，并且抑制了重复数据的产生：

```
SELECT DISTINCT FDepartment FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment
Development
HumanResource
InfoTech
Sales

DISTINCT是对整个结果集进行数据重复抑制的，而不是针对每一个列，执行下面的SQL语句：

```
SELECT DISTINCT FDepartment,FSubCompany FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FDepartment	FSubCompany
Development	Beijing
Development	ShenZhen
HumanResource	Beijing

InfoTech	Beijing
InfoTech	ShenZhen
Sales	Beijing
Sales	ShenZhen

检索结果中不存在FDepartment和FSubCompany列都重复的数据行，但是却存在FDepartment列重复的数据行，这就验证了“**DISTINCT是对整个结果集进行数据重复抑制的**”这句话。

4.6 计算字段

存在数据库系统中的数据的默认展现方式不一定完全符合应用的要求，比如：

- 数据库系统中姓名、工号是单独存储在两个字段的，但是在显示的时候想显示成“姓名+工号”的形式。
- 数据库系统中金额的显示格式是普通的数字显示方式（比如668186.99），但是显示的时候想以千分位的形式显示（比如668,186.99）。
- 数据库系统中基本工资、奖金是单独存储在两个字段的，但是希望显示员工的工资总额。
- 要检索工资总额的80%超过5000元的员工信息。
- 要升级员工工号，需要将所有员工的工号前增加两位0。

所有这些功能都不能通过简单的SQL语句来完成的，因为需要的数据不是数据表中本来就有的，必须经过一定的计算、转换或者格式化，这种情况下我们可以在宿主语言中通过编写代码的方式来进行这些计算、转换或者格式化的工作，但是可以想象当数据量比较大的时候这样处理的速度是非常慢的。计算字段是数据库系统提供的对数据进行计算、转换或者格式化的功能，由于是在数据库系统内部进行的这些工作，而且数据库系统都这些工作进行了优化，所以其处理效率比在宿主语言中通过编写代码的方式进行处理要高效的多。本节将介绍计算字段的基本使用以及在SELECT、Update、Delete等语句中的应用。

4.6.1 常量字段

软件协会要求各个公司提供所有员工的资料信息，其中包括公司名称、注册资本、员工姓名、年龄、所在子公司，而且出于特殊考虑，要求每个员工都列出这些资料信息。对于单个公司而言，公司名称、注册资本这两部分信息不是能从现有的T_Employee，但是它们是确定的值，因此我们编写下面的SQL语句：

```
SELECT 'CowNew集团',918000000,FName,FAge,FSubCompany FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

		FName	FAge	FSubCompany
CowNew 集团	918000000	Tom	25	Beijing
CowNew 集团	918000000	Jerry	28	ShenZhen
CowNew 集团	918000000	Jane	23	Beijing
CowNew 集团	918000000	Tina	25	Beijing
CowNew 集团	918000000	Smith	28	Beijing
CowNew 集团	918000000	<NULL>	27	ShenZhen
CowNew 集团	918000000	John	23	Beijing
CowNew 集团	918000000	Kerry	28	Beijing
CowNew 集团	918000000	Stone	22	ShenZhen

这里的'CowNew集团'和918000000并不是一个实际的存在的列，但是在查询出来的数据中它们看起来是一个实际存在的字段，这样的字段被称为“常量字段”（也称为“常量值”），它们完全可以被看成一个值确定的字段，比如可以为常量字段指定别名，执行下面的SQL

语句：

SELECT 'CowNew 集 团 ' AS CompanyName,918000000 AS RegAmount,FName,FAge,FSubCompany
FROM T_Employee

执行完毕我们就能在输出结果中看到下面的执行结果：

CompanyName	RegAmount	FName	FAge	FSubCompany
CowNew 集团	918000000	Tom	25	Beijing
CowNew 集团	918000000	Jerry	28	ShenZhen
CowNew 集团	918000000	Jane	23	Beijing
CowNew 集团	918000000	Tina	25	Beijing
CowNew 集团	918000000	Smith	28	Beijing
CowNew 集团	918000000	<NULL>	27	ShenZhen
CowNew 集团	918000000	John	23	Beijing
CowNew 集团	918000000	Kerry	28	Beijing
CowNew 集团	918000000	Stone	22	ShenZhen

4.6.2 字段间计算

人力资源部要求统计全体员工的工资指数，工资指数的计算公式为年龄与工资的乘积，这就需要计算将FAge和FSalary的乘积做为一个工资指数列体现到检索结果中，执行下面的SQL语句：

```
SELECT FNumber,FName,FAge * FSalary FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	
DEV001	Tom	207500.00
DEV002	Jerry	64422.40
HR001	Jane	50620.24
HR002	Tina	130009.00
IT001	Smith	109200.00
IT002	<NULL>	75600.00
SALES001	John	115000.00
SALES002	Kerry	173600.00
SALES003	Stone	26400.00

同样，这里的`FAge * FSalary`并不是一个实际的存在的列，但是在查询出来的数据中它们看起来是一个实际存在的字段，它们完全可以被看成一个普通字段，比如可以为此字段指定别名，执行下面的SQL语句：

```
SELECT FNumber,FName,FAge * FSalary AS FSalaryIndex FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FSalaryIndex
DEV001	Tom	207500.00
DEV002	Jerry	64422.40
HR001	Jane	50620.24
HR002	Tina	130009.00
IT001	Smith	109200.00
IT002	<NULL>	75600.00

SALES001	John	115000.00
SALES002	Kerry	173600.00
SALES003	Stone	26400.00

前面提到常量字段完全可以当作普通字段来看待，那么普通字段也可以和常量字段进行计算，甚至常量字段之间也可以进行计算。比如人力资源部要求统计每个员工的工资幸福指数，工资幸福指数的计算公式为工资/（年龄-21），而且要求在每行数据前添加一列，这列的值等于125与521的和。我们编写下面的SQL：

```
SELECT 125+521,FNumber,FName,FSalary/(FAge-21) AS FHappyIndex
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果[10]：

	FNumber	FName	FHappyIndex
646	DEV001	Tom	2075.0000000000000
646	DEV002	Jerry	328.6857142857142
646	HR001	Jane	1100.4400000000000
646	HR002	Tina	1300.0900000000000
646	IT001	Smith	557.1428571428571
646	IT002	<NULL>	466.6666666666666
646	SALES001	John	2500.0000000000000
646	SALES002	Kerry	885.7142857142857
646	SALES003	Stone	1200.0000000000000

计算字段也可以在WHERE语句等子句或者UPDATE、DELETE中使用。比如下面的SQL用来检索所有资幸福指数大于1000的员工信息：

```
SELECT * FROM T_Employee
WHERE FSalary/(FAge-21)>1000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV001	Tom	25	8300.00	Beijing	Development
HR001	Jane	23	2200.88	Beijing	HumanResource
HR002	Tina	25	5200.36	Beijing	HumanResource
SALES001	John	23	5000.00	Beijing	Sales
SALES003	Stone	22	1200.00	ShenZhen	Sales

4.6.3 数据处理函数

像普通编程语言一样，SQL也支持使用函数处理数据，函数使用若干字段名或者常量值做为参数；参数的数量是不固定的，有的函数的参数为空，甚至有的函数的参数个数可变；几乎所有函数都有返回值，返回值即为函数的数据处理结果。

其实在前面的章节中我们已经用到函数了，最典型的就是“聚合函数”，“聚合函数”是函数的一种，它们可以对一组数据进行统计计算。除了“聚合函数”，SQL中还有其他类型的函数，比如进行数值处理的数学函数、进行日期处理的日期函数、进行字符串处理的字符串函数等。

我们来演示几个函数使用的典型场景。

主流数据库系统都提供了计算字符串长度的函数，在MYSQL、Oracle、DB2中这个函数名称为LENGTH，而在MSSQLServer中这个函数的名称则为LEN。这个函数接受一个字符

[10] 由于不同的数据库系统对于除法的精度处理不同，所以 FHappyIndex 的显示精度也不一样。

串类型的字段值做为参数，返回值为这个字符串的长度。下面的SQL语句计算每一个名称不为空的员工的名字以及名字的长度：

MYSQL、Oracle、DB2：

SELECT FName, LENGTH(FName)　AS namelength FROM T_Employee
WHERE FName IS NOT　NULL

MSSQLServer：

SELECT FName, LEN(FName)　AS namelength FROM T_Employee
WHERE FName IS NOT　NULL

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	namelength
Tom	3
Jerry	5
Jane	4
Tina	4
Smith	5
John	4
Kerry	5
Stone	5

主流系统都提供了取得字符串的子串的函数，在MYSQL、MSSQLServer中这个函数名称为SUBSTRING，而在Oracle、DB2这个函数名称为SUBSTR。这个函数接受三个参数，第一个参数为要取的主字符串，第二个参数为字串的起始位置（从1开始计数），第三个参数为字串的长度。下面的SQL语句取得每一个名称不为空的员工的名字以及名字中从第二个字符开始、长度为3的字串：

MYSQL、MSSQLServer：

SELECT FName, SUBSTRING(FName,2,3)　FROM T_Employee
WHERE FName IS NOT NULL

Oracle、DB2：

SELECT FName, SUBSTR(FName,2,3)　FROM T_Employee
WHERE FName IS NOT NULL

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	namelength
Tom	om
Jerry	er
Jane	an
Tina	in
Smith	mi
John	oh
Kerry	er
Stone	to

多个函数还可以嵌套使用。主流系统都提供了计算正弦函数值的函数SIN和计算绝对值的函数ABS，它们都接受一个数值类型的参数。下面的SQL语句取得每个员工的姓名、年龄、年龄的正弦函数值以及年龄的正弦函数值的绝对值，其中计算“年龄的正弦函数值的绝对值”时就要使用嵌套函数，SQL语句如下：

```
SELECT FName,FAge, SIN(FAge) , ABS(SIN(FAge)) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FAge		
Tom	25	-0.13235175009777303	0.13235175009777303
Jerry	28	0.27090578830786904	0.27090578830786904
Jane	23	-0.8462204041751706	0.8462204041751706
Tina	25	-0.13235175009777303	0.13235175009777303
Smith	28	0.27090578830786904	0.27090578830786904
<NULL>	27	0.956375928404503	0.956375928404503
John	23	-0.8462204041751706	0.8462204041751706
Kerry	28	0.27090578830786904	0.27090578830786904
Stone	22	-0.008851309290403876	0.008851309290403876

数据库系统提供的函数是非常丰富的，而且不同的数据库系统提供的函数差异也非常大，本书后面章节将对这些函数进行详细讲解。

4.6.4 字符串的拼接

SQL允许两个或者多个字段之间进行计算，字符串类型的字段也不例外。比如我们需要以“工号+姓名”的方式在报表中显示一个员工的信息，那么就需要把工号和姓名两个字符串类型的字段拼接计算；再如我们需要在报表中在每个员工的工号前增加“Old”这个文本。这时候就需要我们对字符串类型的字段（包括字符串类型的常量字段）进行拼接。在不同的数据库系统下的字符串拼接是有很大差异的，因此这里我们将讲解主流数据库下的字符串拼接的差异。

需要注意的是，在Java、C#等编程语言中字符串是用半角的双引号来包围的，但是在有的数据库系统的SQL语法中双引号有其他的含义（比如列的别名），而所有的数据库系统都支持用单引号包围的形式定义的字符串，所以建议读者使用以单引号包围的形式定义的字符串，而且本书也将使用这种方式。

4.6.4.1 MYSQL

在Java、C#等编程语言中字符串的拼接可以通过加号“+”来实现，比如："1"+"3"、"a"+"b"。在MYSQL中也可以使用加号“+”来连接两个字符串，比如下面的SQL：

```
SELECT '12'+'33',FAge+'1' FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

'12'+'33'	FAge+'1'
45	26
45	29
45	24
45	26
45	29
45	28
45	24
45	29
45	23

仔细观察第一列，惊讶吗？这个列的显示结果并不是我们希望的“1233”，而是把“12”和“33”两个字符串当成数字来求两个数的和了；同样将一个数字与一个字符串用加号“+”连接也是同样的效果，比如这里的第二列。

在MYSQL中，当用加号“+”连接两个字段（或者多个字段）的时候，MYSQL会尝试将字段值转换为数字类型（如果转换失败则认为字段值为0），然后进行字段的加法运算。因

此，当计算的'12'+'33'的时候，MYSQL会将“12”和“33”两个字符串尝试转换为数字类型的12和33，然后计算12+33的值，这就是为什么我们会得到45的结果了。同样道理，在计算FAge+'1'的时候，由于FAge为数字类型，所以不需要进行转换，而'1'为字符串类型，所以MYSQL将'1'尝试转换为数字1，然后计算FAge+1做为计算列的值。

MYSQL会尝试将加号两端的字段值尝试转换为数字类型，如果转换失败则认为字段值为0，比如我们执行下面的SQL语句：

```
SELECT 'abc'+'123',FAge+'a' FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

'abc'+'123'	FAge+'a'
123	25
123	28
123	23
123	25
123	28
123	27
123	23
123	28
123	22

在MYSQL中进行字符串的拼接要使用CONCAT函数，CONCAT函数支持一个或者多个参数，参数类型可以为字符串类型也可以是非字符串类型，对于非字符串类型的参数MYSQL将尝试将其转化为字符串类型，CONCAT函数会将所有参数按照参数的顺序拼接成一个字符串做为返回值。比如下面的SQL语句用于将用户的多个字段信息以一个计算字段的形式查询出来：

```
SELECT CONCAT('工号为:',FNumber,'的员工的幸福指数:',FSalary/(FAge-21))
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT('工号为:',FNumber,'的员工的幸福指数:',FSalary/(FAge-21))
工号为:DEV001 的员工的幸福指数:2075.000000
工号为:DEV002 的员工的幸福指数:328.685714
工号为:HR001 的员工的幸福指数:1100.440000
工号为:HR002 的员工的幸福指数:1300.090000
工号为:IT001 的员工的幸福指数:557.142857
工号为:IT002 的员工的幸福指数:466.666667
工号为:SALES001 的员工的幸福指数:2500.000000
工号为:SALES002 的员工的幸福指数:885.714286
工号为:SALES003 的员工的幸福指数:1200.000000

CONCAT支持只有一个参数的用法，这时的CONCAT可以看作是一个将这个参数值尝试转化为字符串类型值的函数。MYSQL中还提供了另外一个进行字符串拼接的函数CONCAT_WS，CONCAT_WS可以在待拼接的字符串之间加入指定的分隔符，它的第一个参数值为采用的分隔符，而剩下的参数则为待拼接的字符串值，比如执行下面的SQL：

```
SELECT CONCAT_WS(',',FNumber,FAge,FDepartment,FSalary) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT_WS(',',FNumber,FAge,FDepartment,FSalary)
DEV001,25,Development,8300.00

DEV002,28,Development,2300.80
HR001,23,HumanResource,2200.88
HR002,25,HumanResource,5200.36
IT001,28,InfoTech,3900.00
IT002,27,InfoTech,2800.00
SALES001,23,Sales,5000.00
SALES002,28,Sales,6200.00
SALES003,22,Sales,1200.00

4.6.4.2 MSSQLServer

与MYSQL不同，MSSQLServer中可以直接使用加号“+”来拼接字符串。比如执行下面的SQL语句：

```
SELECT '工号为'+FNumber+'的员工姓名为'+Fname FROM T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina
工号为 IT001 的员工姓名为 Smith
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone

4.6.4.3 Oracle

Oracle中使用“||”进行字符串拼接，其使用方式和MSSQLServer中的加号“+”一样。比如执行下面的SQL语句：

```
SELECT '工号为'||FNumber||'的员工姓名为'||FName FROM T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

工号为||FNUMBER||的员工姓名为||FNAME
工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina
工号为 IT001 的员工姓名为 Smith

除了“||”，Oracle还支持使用CONCAT()函数进行字符串拼接，比如执行下面的SQL语句：

```
SELECT CONCAT('工号:',FNumber) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT(工号:,FNUMBER)

工号:DEV001
工号:DEV002
工号:HR001
工号:HR002
工号:IT001
工号:IT002
工号:SALES001
工号:SALES002
工号:SALES003

如果CONCAT中连接的值不是字符串，Oracle会尝试将其转换为字符串，比如执行下面的SQL语句：

```
SELECT CONCAT('年龄:',FAge) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT(年龄:,FAGE)
年龄:25
年龄:28
年龄:23
年龄:28
年龄:22
年龄:23
年龄:25
年龄:28
年龄:27

与MYSQL的CONCAT()函数不同，Oracle的CONCAT()函数只支持两个参数，不支持两个以上字符串的拼接，比如下面的SQL语句在Oracle中是错误的：

```
SELECT CONCAT('工号为',FNumber,'的员工姓名为',FName) FROM T_Employee
WHERE FName IS NOT NULL
```

运行以后Oracle会报出下面的错误信息：

参数个数无效

如果要进行多个字符串的拼接的话，可以使用多个CONCAT()函数嵌套使用，上面的SQL可以如下改写：

```
SELECT CONCAT(CONCAT(CONCAT('工号为',FNumber),'的员工姓名为'),FName) FROM
T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CONCAT(CONCAT(CONCAT(工号为,FNUMBER),的员工姓名为),FNAME)
工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina

工号为 IT001 的员工姓名为 Smith

4.6.4.4 DB2

DB2中使用“||”进行字符串拼接，其使用方式和MSSQLServer中的加号“+”一样。比如执行下面的SQL语句：

```
SELECT '工号为'||FNumber||'的员工姓名为'||FName FROM T_Employee
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina
工号为 IT001 的员工姓名为 Smith

除了“||”，DB2还支持使用CONCAT()函数进行字符串拼接，比如执行下面的SQL语句：

```
SELECT CONCAT('工号:',FNumber) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
工号:DEV001
工号:DEV002
工号:HR001
工号:HR002
工号:IT001
工号:IT002
工号:SALES001
工号:SALES002
工号:SALES003

与Oracle不同，如果CONCAT中连接的值不是字符串，则DB2不会尝试进行类型转换而是报出错误信息，比如执行下面的SQL语句是错误的：

```
SELECT CONCAT('年龄:',FAge) FROM T_Employee
```

运行以后DB2会报出下面的错误信息：

```
未找到类型为 "FUNCTION" 命名为 "CONCAT" 且具有兼容自变量的已授权例程
```

与MYSQL的CONCAT()函数不同，DB2的CONCAT()函数只支持两个参数，不支持两个以上字符串的拼接，比如下面的SQL语句在Oracle中是错误的：

```
SELECT CONCAT('工号为',FNumber,'的员工姓名为',FName) FROM T_Employee
WHERE FName IS NOT NULL
```

运行以后Oracle会报出下面的错误信息：

```
未找到类型为 "FUNCTION" 命名为 "CONCAT" 且具有兼容自变量的已授权例程
```

如果要进行多个字符串的拼接的话，可以使用多个CONCAT()函数嵌套使用，上面的SQL可以如下改写：

```
SELECT CONCAT(CONCAT(CONCAT('工号为',FNumber),'的员工姓名为'),FName) FROM
T_Employee
```

```
WHERE FName IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
工号为 DEV001 的员工姓名为 Tom
工号为 DEV002 的员工姓名为 Jerry
工号为 SALES001 的员工姓名为 John
工号为 SALES002 的员工姓名为 Kerry
工号为 SALES003 的员工姓名为 Stone
工号为 HR001 的员工姓名为 Jane
工号为 HR002 的员工姓名为 Tina
工号为 IT001 的员工姓名为 Smith

4.6.5 计算字段的其他用途

我们不仅能在SELECT语句中使用计算字段，我们同样可以在进行数据过滤、数据删除以及数据更新的时候使用计算字段，下面我们举几个例子。

4.6.5.1 计算处于合理工资范围内的员工

我们规定一个合理工资范围：上限为年龄的1.8倍加上5000元，下限为年龄的1.5倍加上2000元，介于这两者之间的即为合理工资。我们需要查询所有处于合理工资范围内的员工信息。因此编写如下的SQL语句：

```
SELECT * FROM T_Employee
WHERE Fsalary BETWEEN Fage*1.5+2000 AND Fage*1.8+5000
```

这里我们在**BETWEEN……AND……语句中使用了计算表达式。**执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV002	Jerry	28	2300.80	ShenZhen	Development
HR001	Jane	23	2200.88	Beijing	HumanResource
IT001	Smith	28	3900.00	Beijing	InfoTech
IT002	<NULL>	27	2800.00	ShenZhen	InfoTech
SALES001	John	23	5000.00	Beijing	Sales

4.6.5.2 查询“工资年龄指数”

我们定义“工资年龄指数”为“工资除以年龄”。我们需要查询“工资年龄指数”的最高值和最低值。因此编写如下的SQL语句：

```
SELECT MAX(FSalary/FAge) AS MAXVALUE,MIN(FSalary/FAge) AS MINVALUE
FROM T_Employee
```

这里我们在MAX、MIN函数中使用了计算字段。执行完毕我们就能在输出结果中看到下面的执行结果：

MAXVALUE	MINVALUE
332.0000000000000	54.5454545454545

4.6.5.3 年龄全部加1

新的一年到来了，系统需要自动将员工的年龄全部加1。这个工作如果使用代码来完成的话会是这样：

```
result = executeQuery(“SELECT * FROM T_Employee”);
for(i=0;i<result.count;i++)
{
    age = result[i].get(“FAge”);
```

number = result[i].get(“FNumber”);

age=age+1;

executeUpdate(“UPDATE T_Employee SET FAge=”+age+” WHERE FNumber=”+number);

}

这种方式在数据量比较大的时候速度是非常慢的，而在UPDATE中使用计算字段则可以非常简单快速的完成任务，编写下面的SQL语句：

UPDATE T_Employee SET FAge=FAge+1

这里在SET子句中采用计算字段的方式为FAge字段设定了新值。执行完毕后执行**SELECT** * **FROM** T_Employee来查看修改后的数据：

FNumber	FName	FAge	FSalary	FSubCompany	FDepartment
DEV001	Tom	26	8300.00	Beijing	Development
DEV002	Jerry	29	2300.80	ShenZhen	Development
HR001	Jane	24	2200.88	Beijing	HumanResource
HR002	Tina	26	5200.36	Beijing	HumanResource
IT001	Smith	29	3900.00	Beijing	InfoTech
IT002	<NULL>	28	2800.00	ShenZhen	InfoTech
SALES001	John	24	5000.00	Beijing	Sales
SALES002	Kerry	29	6200.00	Beijing	Sales
SALES003	Stone	23	1200.00	ShenZhen	Sales

4.7 不从实体表中取的数据

有的时候我们需要查询一些不能从任何实体表中能够取得的数据，比如将数字1作为结果集或者计算字符串“abc”的长度。

有的开发人员尝试使用下面的SQL来完成类似的功能：

SELECT 1 FROM T_Employee

可是执行以后却得到了下面的执行结果集

1
1
1
1
1
1
1
1
1

结果集中出现了不止一个1，这时因为通过这种方式得到的结果集数量是取决于T_Employee表中的数据条目数的，必须要借助于DISTINCT关键字来将结果集条数限定为一条，SQL语句如下：

SELECT DISTINCT 1 FROM T_Employee

执行完毕我们就能在输出结果中看到下面的执行结果：

1

这种实现方式非常麻烦，而且如果数据库中一张表都没有的时候这样就不凑效了。主流数据库系统对这种需求提供了比较好的支持，这一节我们就来看一下主流数据库系统对此的支持。

MYSQL和MSSQLServer允许使用不带FROM子句的SELECT语句来查询这些不属于任何实体表的数据，比如下面的SQL将1作为结果集：

SELECT 1

执行完毕我们就能在输出结果中看到下面的执行结果：

1

还可以在不带FROM子句的SELECT语句中使用函数，比如下面的SQL将字符串“abc”的长度作为结果集：

MYSQL:

SELECT LENGTH('abc')

MSSQLServer:

SELECT LEN('abc')

执行完毕我们就能在输出结果中看到下面的执行结果：

3

还可以在SELECT语句中同时计算多个表达式，比如下面的SQL语句将1、2、3、’a’、’b’、’c’作为结果集：

```
SELECT 1,2,3,'a','b','c'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	2	3	a	b	c

在Oracle中是不允许使用这种不带FROM子句的SELECT语句，不过我们可以使用Oracle的系统表来作为FROM子句中的表名，系统表是Oracle内置的特殊表，最常用的系统表为DUAL。比如下面的SQL将1以及字符串'abc'的长度作为结果集：

SELECT 1, LENGTH('abc') FROM DUAL

执行完毕我们就能在输出结果中看到下面的执行结果：

1	LENGTH(ABC)
1	3

在DB2中也同样不支持不带FROM子句的SELECT语句，它也是采用和Oracle类似的系统表，最常用的系统表为SYSIBM.SYSDUMMY1。比如下面的SQL将1以及字符串'abc'的长度作为结果集：

SELECT 1, LENGTH('abc') FROM SYSIBM.SYSDUMMY1

执行完毕我们就能在输出结果中看到下面的执行结果：

1	2
1	3

4.8 联合结果集

有的时候我们需要组合两个完全不同的查询结果集，而这两个查询结果之间没有必然的联系，只是我们需要将他们显示在一个结果集中而已。在SQL中可以使用UNION运算符来将两个或者多个查询结果集联合为一个结果集中。

为了更好的讲解本节的内容，需要首先创建一张用来存储临时工信息的新表，在数据库系统下执行下面的SQL语句：

MYSQL:

CREATE TABLE T_TempEmployee (FIdCardNumber VARCHAR(20),FName VARCHAR(20),FAge INT ,PRIMARY KEY (FIdCardNumber))

MSSQLServer:

CREATE TABLE T_TempEmployee (FIdCardNumber VARCHAR(20),FName VARCHAR(20),FAge INT, PRIMARY KEY (FIdCardNumber))

Oracle:

CREATE TABLE T_TempEmployee (FIdCardNumber VARCHAR2(20),FName VARCHAR2(20),FAge NUMBER (10), PRIMARY KEY (FIdCardNumber))

DB2：

```
CREATE TABLE T_TempEmployee (FIdCardNumber VARCHAR(20) Not NULL,FName VARCHAR(20),FAge INT, PRIMARY KEY (FIdCardNumber))
```

由于临时工没有分配工号，所以使用身份证号码FIdCardNumber来标识一个临时工，同时由于临时工不是实行月薪制，所以这里也没有记录月薪信息。我们还需要一些初始数据，执行下面的SQL语句以插入初始数据：

INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890121','Sarani',33);

INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890122','Tom',26);

INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890123','Yalaha',38);

INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890124','Tina',26);

INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890125','Konkaya',29);

INSERT INTO T_TempEmployee(FIdCardNumber,FName,FAge) VALUES('1234567890126','Fotifa',46);

插入初始数据完毕以后执行SELECT * FROM T_TempEmployee查看表中的数据：

FIdCardNumber	FName	FAge
1234567890121	Sarani	33
1234567890122	Tom	26
1234567890123	Yalaha	38
1234567890124	Tina	26
1234567890125	Konkaya	29
1234567890126	Fotifa	46

4.8.1 简单的结果集联合

UNION运算符要放置在两个查询语句之间。比如我们要查询公司所有员工（包括临时工）的标识号码、姓名、年龄信息。

查询正式员工信息的SQL语句如下：

SELECT FNumber,FName,FAge FROM T_Employee

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge
DEV001	Tom	26
DEV002	Jerry	29

HR001	Jane	24
HR002	Tina	26
IT001	Smith	29
IT002	<NULL>	28
SALES001	John	24
SALES002	Kerry	29
SALES003	Stone	23

而查询临时工信息的SQL语句如下：

```
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FIdCardNumber	FName	FAge
1234567890121	Sarani	33
1234567890122	Tom	26
1234567890123	Yalaha	38
1234567890124	Tina	26
1234567890125	Konkaya	29
1234567890126	Fotifa	46

只要用UNION操作符连接这两个查询语句就可以将两个查询结果集联合为一个结果集，SQL语句如下：

```
SELECT FNumber,FName,FAge FROM T_Employee
UNION
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge
1234567890121	Sarani	33
1234567890122	Tom	26
1234567890123	Yalaha	38
1234567890124	Tina	26
1234567890125	Konkaya	29
1234567890126	Fotifa	46
DEV001	Tom	26
DEV002	Jerry	29
HR001	Jane	24
HR002	Tina	26
IT001	Smith	29
IT002	<NULL>	28
SALES001	John	24
SALES002	Kerry	29
SALES003	Stone	23

可以看到UNION操作符将两个独立的结果集联合成为了一个结果集。

UNION可以连接多个结果集，就像“+”可以连接多个数字一样简单，只要在每个结果集之间加入UNION即可，比如下面的SQL语句就连接了三个结果集：

```
SELECT FNumber,FName,FAge FROM T_Employee
```

```
WHERE FAge<30
UNION
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
WHERE FAge>40
UNION
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
WHERE FAge<30
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge
1234567890122	Tom	26
1234567890124	Tina	26
1234567890125	Konkaya	29
1234567890126	Fotifa	46
DEV001	Tom	26
DEV002	Jerry	29
HR001	Jane	24
HR002	Tina	26
IT001	Smith	29
IT002	<NULL>	28
SALES001	John	24
SALES002	Kerry	29
SALES003	Stone	23

4.8.2 联合结果集的原则

联合结果集不必受被联合的多个结果集之间的关系限制，不过使用UNION仍然有两个基本的原则需要遵守：一是每个结果集必须有相同的列数；二是每个结果集的列必须类型相容。

首先看第一个原则，每个结果集必须有相同的列数，两个不同列数的结果集是不能联合在一起的。比如下面的SQL语句是错误的：

```
SELECT FNumber,FName,FAge,FDepartment FROM T_Employee
UNION
SELECT FIdCardNumber,FName,FAge FROM T_TempEmployee
```

执行以后数据库系统会报出如下的错误信息：

使用 UNION、INTERSECT 或 EXCEPT 运算符合并的所有查询必须在其目标列表中有相同数目的表达式。

因为第一个结果集返回了4列数据，而第二个结果集则返回了3列数据，数据库系统并不会用空值将第二个结果集补足为4列。如果需要将未知列补足为一个默认值，那么可以使用常量字段，比如下面的SQL语句就将第二个结果集的与FDepartment对应的字段值设定为“临时工，不属于任何一个部门”：

```
SELECT FNumber,FName,FAge,FDepartment FROM T_Employee
UNION
SELECT FIdCardNumber,FName,FAge,'临时工，不属于任何一个部门' FROM
T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FAge	FDepartment
1234567890121	Sarani	33	临时工，不属于任何一个部门
1234567890122	Tom	26	临时工，不属于任何一个部门

1234567890123	Yalaha	38	临时工，不属于任何一个部门
1234567890124	Tina	26	临时工，不属于任何一个部门
1234567890125	Konkaya	29	临时工，不属于任何一个部门
1234567890126	Fotifa	46	临时工，不属于任何一个部门
DEV001	Tom	26	Development
DEV002	Jerry	29	Development
HR001	Jane	24	HumanResource
HR002	Tina	26	HumanResource
IT001	Smith	29	InfoTech
IT002	<NULL>	28	InfoTech
SALES001	John	24	Sales
SALES002	Kerry	29	Sales

联合结果集的第二个原则是：每个结果集的列必须类型相容，也就是说结果集的每个对应列的数据类型必须相同或者能够转换为同一种数据类型。比如下面的SQL语句在MYSQL中可以正确的执行：

```
SELECT FIdCardNumber,FAge,FName FROM T_TempEmployee
UNION
SELECT FNumber,FName,FAge FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FIdCardNumber	FAge	FName
1234567890121	33	Sarani
1234567890122	26	Tom
1234567890123	38	Yalaha
1234567890124	26	Tina
1234567890125	29	Konkaya
1234567890126	46	Fotifa
DEV001	Tom	26
DEV002	Jerry	29
HR001	Jane	24
HR002	Tina	26
IT001	Smith	29
IT002	<NULL>	28
SALES001	John	24
SALES002	Kerry	29
SALES003	Stone	23

可以看到MYSQL将FAge转换为了文本类型，以便于与FName字段值匹配。不过这句SQL语句在MSSQLServer、Oracle、DB2中执行则会报出类似如下的错误信息：

表达式必须具有与对应表达式相同的数据类型。

因为这些数据库系统不会像MYSQL那样进行默认的数据类型转换。由于不同数据库系统中数据类型转换规则是各不相同的，因此如果开发的应用程序要考虑跨多数据库移植的话最好保证结果集的每个对应列的数据类型完全相同。

4.8.3 UNION ALL

我们想列出公司中所有员工（包括临时工）的姓名和年龄信息，那么我们可以执行下面

的SQL语句：

```
SELECT FName,FAge FROM T_Employee
UNION
SELECT FName,FAge FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FAge
<NULL>	28
Fotifa	46
Jane	24
Jerry	29
John	24
Kerry	29
Konkaya	29
Sarani	33
Smith	29
Stone	23
Tina	26
Tom	26
Yalaha	38

仔细观察结果集，我们发现输出的结果和预想的是不一致的，在正式员工中有姓名为Tom、年龄为26以及姓名为Tina、年龄为26的两名员工，而临时工中也有姓名为Tom、年龄为26以及姓名为Tina、年龄为26的两名员工，也就是说正式员工的临时工中存在重名和年龄重复的现象，但是在查询结果中却将重复的信息只保留了一条，也就是只有一个姓名为Tom、年龄为26的员工和一个姓名为Tina、年龄为26的员工。

这时因为默认情况下，UNION运算符合并了两个查询结果集，其中完全重复的数据行被合并为了一条。如果需要在联合结果集中返回所有的记录而不管它们是否唯一，则需要在UNION运算符后使用ALL操作符，比如下面的SQL语句：

```
SELECT FName,FAge FROM T_Employee
UNION ALL
SELECT FName,FAge FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FAge
Tom	26
Jerry	29
Jane	24
Tina	26
Smith	29
<NULL>	28
John	24
Kerry	29
Stone	23
Sarani	33
Tom	26

Yalaha	38
Tina	26
Konkaya	29
Fotifa	46

4.8.4 联合结果集应用举例

联合结果集在制作报表的时候经常被用到，使用联合结果集我们可以将没有直接关系的数据显示到同一张报表中。使用UNION运算符，只要被联合的结果集符合联合结果集的原则，那么被连接的两个SQL语句可以是非常复杂，也可以是非常简单的。本小节将展示几个实用的例子来看一下联合结果集在实际开发中的应用。

4.8.4.1 员工年龄报表

要求查询员工的最低年龄和最高年龄，临时工和正式员工要分别查询。

实现SQL语句如下：

```
SELECT '正式员工最高年龄',MAX(FAge) FROM T_Employee
UNION
SELECT '正式员工最低年龄',MIN(FAge) FROM T_Employee
UNION
SELECT '临时工最高年龄',MAX(FAge) FROM T_TempEmployee
UNION
SELECT '临时工最低年龄',MIN(FAge) FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

临时工最低年龄	26
临时工最高年龄	46
正式员工最低年龄	23
正式员工最高年龄	29

4.8.4.2 正式员工工资报表

要求查询每位正式员工的信息，包括工号、工资，并且在最后一行加上所有员工工资额合计。

实现SQL语句如下：

```
SELECT FNumber,FSalary FROM T_Employee
UNION
SELECT '工资合计',SUM(FSalary) FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FSalary
DEV001	8300.00
DEV002	2300.80
HR001	2200.88
HR002	5200.36
IT001	3900.00
IT002	2800.00
SALES001	5000.00
SALES002	6200.00
SALES003	1200.00
工资合计	37102.04

4.8.4.3 打印5以内自然数的平方

要求打印出打印5以内自然数以及它们的平方数。

实现SQL语句如下：

MYSQL、MSSQLServer：

```
SELECT 1,1 * 1
UNION
SELECT 2,2 * 2
UNION
SELECT 3,3 * 3
UNION
SELECT 4,4 * 4
UNION
SELECT 5,5 * 5
```

Oracle：

```
SELECT 1,1 * 1 FROM DUAL
UNION
SELECT 2,2 * 2 FROM DUAL
UNION
SELECT 3,3 * 3 FROM DUAL
UNION
SELECT 4,4 * 4 FROM DUAL
UNION
SELECT 5,5 * 5 FROM DUAL
```

DB2:

```
SELECT 1,1 * 1 FROM SYSIBM.SYSDUMMY1
UNION
SELECT 2,2 * 2 FROM SYSIBM.SYSDUMMY1
UNION
SELECT 3,3 * 3 FROM SYSIBM.SYSDUMMY1
UNION
SELECT 4,4 * 4 FROM SYSIBM.SYSDUMMY1
UNION
SELECT 5,5 * 5 FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	1
2	4
3	9
4	16
5	25

4.8.4.4 列出员工姓名

要求列出公司中所有员工（包括临时工）的姓名，将重复的姓名过滤掉。

实现SQL语句如下：

```
SELECT FName FROM T_Employee
UNION
SELECT FName FROM T_TempEmployee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName
<NULL>
Fotifa
Jane
Jerry
John
Kerry
Konkaya
Sarani
Smith
Stone
Tina
Tom
Yalaha

4.8.4.5 分别列出正式员工和临时工的姓名

要求分别列出正式员工和临时工的姓名，要保留重复的姓名。

实现SQL语句如下：

MYSQL、MSSQLServer：

```
SELECT '以下是正式员工的姓名'
UNION ALL
SELECT FName FROM T_Employee
UNION ALL
SELECT '以下是临时工的姓名'
UNION ALL
SELECT FName FROM T_TempEmployee
```

Oracle：

```
SELECT '以下是正式员工的姓名' FROM DUAL
UNION ALL
SELECT FName FROM T_Employee
UNION ALL
SELECT '以下是临时工的姓名' FROM DUAL
UNION ALL
SELECT FName FROM T_TempEmployee
```

DB2：

```
SELECT '以下是正式员工的姓名' FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT FName FROM T_Employee
```

UNION ALL
SELECT '以下是临时工的姓名' FROM SYSIBM.SYSDUMMY1
UNION ALL
SELECT FName FROM T_TempEmployee

执行完毕我们就能在输出结果中看到下面的执行结果：

以下是正式员工的姓名
Tom
Jerry
Jane
Tina
Smith
<NULL>
John
Kerry
Stone
以下是临时工的姓名
Sarani
Tom
Yalaha
Tina
Konkaya

本章即将结束，请执行下面的SQL语句将本章用到的数据表删除：

```
Drop TABLE T_Employee;
Drop TABLE T_TempEmployee;
```

章 函数

5.1 数学函数

5.1.1 Oracle 中的数学函数

5.1.2 DB2 中数学函数

5.1.3 MSSQLServer 中数学函数

5.1.4 MySQL 中数学函数

5.1.5 Access 中数学函数

5.1.6 主流数据库的数学函数对照表

5.2 字符串函数

5.2.1 Oracle 中的字符串函数

5.2.2 DB2 中字符串函数

5.2.3 MSSQLServer 中字符串函数

5.2.4 MySQL 中字符串函数

5.2.5 Access 中字符串函数

5.2.6 主流数据库的字符串函数对照表

5.3 日期函数

5.3.1 Oracle 中的日期函数

第四章中我们讲解了 SQL 中函数的用法。SQL 中可供使用的函数是非常多的，这些函数的功能包括转换字符串大小写、求一个数的对数、计算两个日期之间的天数间隔等，数量的掌握这些函数将能够帮助我们更快的完成业务功能。本章将讲解这些函数的使用，并且对它们在不同数据库系统中的差异性进行比较。

为了更容易的运行本章中的例子，必须首先创建所需要的数据表，因此下面列出本章中要用到数据表的创建 SQL 语句：

MYSQL:

```
CREATE TABLE T_Person (FIdNumber VARCHAR(20),
FName VARCHAR(20),FBirthDay DATETIME,
FRegDay DATETIME,FWeight DECIMAL(10,2))
```

MSSQLServer:

```
CREATE TABLE T_Person (FIdNumber VARCHAR(20),
FName VARCHAR(20),FBirthDay DATETIME,
FRegDay DATETIME,FWeight NUMERIC(10,2))
```

Oracle:

```
CREATE TABLE T_Person (FIdNumber VARCHAR2(20),
FName VARCHAR2(20),FBirthDay DATE,
FRegDay DATE,FWeight NUMERIC(10,2))
```

DB2:

```
CREATE TABLE T_Person (FIdNumber VARCHAR(20),
FName VARCHAR(20),FBirthDay DATE,
FRegDay DATE,FWeight DECIMAL(10,2))
```

请在不同的数据库系统中运行相应的 SQL 语句。T_Person 为记录人员信息的数据表，其中字段 FIdNumber 为人员的身份证号码，FName 为人员姓名，FBirthDay 为出生日期，FRegDay 为注册日期，FWeight 为体重。

为了更加直观的验证本章中函数使用方法的正确性，我们需要在 T_Person 表中预置一些初始数据，请在数据库中执行下面的数据插入 SQL 语句：

MYSQL、MSSQLServer、DB2：

```
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
```

```
VALUES ('123456789120','Tom','1981-03-22','1998-05-01',56.67);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789121','Jim','1987-01-18','1999-08-21',36.17);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789122','Lily','1987-11-08','2001-09-18',40.33);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789123','Kelly','1982-07-12','2000-03-01',46.23);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789124','Sam','1983-02-16','1998-05-01',48.68);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789125','Kerry','1984-08-07','1999-03-01',66.67);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789126','Smith','1980-01-09','2002-09-23',51.28);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789127','BillGates','1972-07-18','1995-06-19',60.32);
```

Oracle：

```
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789120','Tom',TO_DATE('1981-03-22', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('1998-05-01', 'YYYY-MM-DD HH24:MI:SS'),56.67);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789121','Jim',TO_DATE('1987-01-18', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('1999-08-21', 'YYYY-MM-DD HH24:MI:SS'),36.17);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789122','Lily',TO_DATE('1987-11-08', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2001-09-18', 'YYYY-MM-DD HH24:MI:SS'),40.33);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789123','Kelly',TO_DATE('1982-07-12', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2000-03-01', 'YYYY-MM-DD HH24:MI:SS'),46.23);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789124','Sam',TO_DATE('1983-02-16', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('1998-05-01', 'YYYY-MM-DD HH24:MI:SS'),48.68);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789125','Kerry',TO_DATE('1984-08-07', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('1999-03-01', 'YYYY-MM-DD HH24:MI:SS'),66.67);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789126','Smith',TO_DATE('1980-01-09', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2002-09-23', 'YYYY-MM-DD HH24:MI:SS'),51.28);
INSERT INTO T_Person(FIdNumber,FName,FBirthDay,FRegDay,FWeight)
VALUES ('123456789127','BillGates',TO_DATE('1972-07-18', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('1995-06-19', 'YYYY-MM-DD HH24:MI:SS'),60.32);
```

初始数据预置完毕以后执行 `SELECT * FROM T_Person` 来查看表中的数据，内容如下：

FIdNumber	FName	FBirthDay	FRegDay	FWeight
123456789120	Tom	1981-03-22	1998-05-01	56.67

		00:00:00.0	00:00:00.0	
123456789121	Jim	1987-01-18 00:00:00.0	1999-08-21 00:00:00.0	36.17
123456789122	Lily	1987-11-08 00:00:00.0	2001-09-18 00:00:00.0	40.33
123456789123	Kelly	1982-07-12 00:00:00.0	2000-03-01 00:00:00.0	46.23
123456789124	Sam	1983-02-16 00:00:00.0	1998-05-01 00:00:00.0	48.68
123456789125	Kerry	1984-08-07 00:00:00.0	1999-03-01 00:00:00.0	66.67
123456789126	Smith	1980-01-09 00:00:00.0	2002-09-23 00:00:00.0	51.28
123456789127	BillGates	1972-07-18 00:00:00.0	1995-06-19 00:00:00.0	60.32

5.1 数学函数

SQL 标准中只规定了 4 个数学函数，不过很多主流的数据库系统都提供了大量常用的数学函数，而且几乎所有的数据库系统都对它们提供了支持，因此这里我们有必要对这些函数进行详细的介绍。

5.1.1 求绝对值

ABS()函数用来返回一个数值的绝对值。该函数接受一个参数，这个参数为待求绝对值的表达式。执行下面的 SQL 语句：

```
SELECT  FWeight - 50,ABS(FWeight - 50) , ABS(-5.38)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

6.67	6.67	5.38
-13.83	13.83	5.38
-9.67	9.67	5.38
-3.77	3.77	5.38
-1.32	1.32	5.38
16.67	16.67	5.38
1.28	1.28	5.38
10.32	10.32	5.38

5.1.2 求指数

POWER()函数是用来计算指数的函数。该函数接受两个参数，第一个参数为待求幂的表达式，第二个参数为幂。执行下面的 SQL 语句：

```
SELECT FWeight,POWER(FWeight,-0.5),POWER(FWeight,2),
POWER(FWeight,3),POWER(FWeight,4) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FWeight				
56.67	0.13	3211.49	181995.08	10313660.95
36.17	0.17	1308.27	47320.09	1711567.51
40.33	0.16	1626.51	65597.10	2645531.20

46.23	0.15	2137.21	98803.35	4567678.98
48.68	0.14	2369.74	115359.06	5615679.04
66.67	0.12	4444.89	296340.74	19757037.33
51.28	0.14	2629.64	134847.86	6914998.11
60.32	0.13	3638.50	219474.46	13238699.71

5.1.3 求平方根

SQRT()函数是用来计算平方根的函数。该函数接受一个参数，这个参数为待计算平方根的表达式。执行下面的 SQL 语句：

SELECT FWeight,SQRT(FWeight) FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FWeight	
56.67	7.527947927556354
36.17	6.014149981501958
40.33	6.350590523722971
46.23	6.79926466612383
48.68	6.977105417004963
66.67	8.165169930871004
51.28	7.161005515987263
60.32	7.766595135579039

5.1.4 求随机数

在生成随机密码、随机问题等的时候需要使用随机算法生成随机数。不同的数据库提供的生成随机数的方法不同。

5.1.4.1 MYSQL

在 MYSQL 中提供了 RAND()函数用来生成随机算法，执行下面的 SQL 语句：

SELECT RAND()

执行完毕我们就能在输出结果中看到下面的执行结果：

0.4614449609115853

因为 RAND()函数的返回值是随机的，所以这里看到的执行结果很可能与您的执行结果不同。

5.1.4.2 MSSQLServer

在 MSSQLServer 中也提供了 RAND()函数用来生成随机算法，其使用方法和 MYSQL 中的类似。除此之外 RAND()函数还支持一个参数，这个参数为随机数种子，执行下面的 SQL 语句：

SELECT RAND(9527)

执行完毕我们就能在输出结果中看到下面的执行结果：

0.8910896774185367

5.1.4.3 Oracle

Oracle 中没有内置的生成随机数的函数，不过 Oracle 提供了包 dbms_random 用来生成随机数，使用方法如下：

SELECT dbms_random.value FROM dual

执行完毕我们就能在输出结果中看到下面的执行结果：

VALUE
0.14190212623840265315906642197785380122

除了上面这种使用方式，dbms_random 包中还提供了其他几种方法用来完成其他的随机处理。

dbms_random. value (low, high)用来返回一个大于或等于 low，小于 high 的随机数，执行下面的 SQL 语句：

```
SELECT dbms_random.value(60,100) FROM dual
```

执行完毕我们就能在输出结果中看到下面的执行结果：

DBMS_RANDOM.VALUE(60,100)
89.120578470938982070819016925263396263 6

dbms_random. normal 用来返回服从正态分布的一组数。此正态分布标准偏差为 1，期望值为 0。这个函数返回的数值中有 68%是介于-1 与+1 之间，95%介于-2 与+2 之间，99%介于-3 与+3 之间。执行下面的 SQL 语句：

```
SELECT dbms_random.normal FROM dual
```

执行完毕我们就能在输出结果中看到下面的执行结果：

NORMAL
-0.837608509826025240612747781704228255221 7

dbms_random.string(opt, len)用来返回一个随机字符串，opt 为选项参数，len 表示返回的字符串长度，最大值为 60。参数 opt 可选值如下：

'U'：返回全是大写的字符串。

'L'：返回全是小写的字符串。

'A'：返回大小写结合的字符串。

'X'：返回全是大写和数字的字符串。

'P'：返回键盘上出现字符的随机组合。

可以使用 dbms_random.string 来产生随机的用户密码或者验证码。执行下面的 SQL 语句：

```
SELECT dbms_random.string('U',8) as UP,
dbms_random.string('L',5) as LP,
dbms_random.string('A',6) as AP,
dbms_random.string('X',6) as XP,
dbms_random.string('P',8) as PP FROM dual
```

执行完毕我们就能在输出结果中看到下面的执行结果：

UP	LP	AP	XP	PP
RXSINVJD	swokb	blsYor	M63XKZ	4H{lUKMs

5.1.4.4 DB2

和 Oracle 一样，DB2 同样没有内置的生成随机数的函数，不过 DB2 的 SYSFUN 包提供了 rand 函数用来生成随机数，在使用之前要确保已经被正确安装了，其使用方法非常简单，执行下面的 SQL 语句：

```
select SYSFUN.rand() from SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
0.1216772972808008

5.1.5 舍入到最大整数

在 MYSQL、MSSQLServer 和 DB2 中提供了名称为 CEILING()函数，Oracle 也提供了

类似的函数 CEIL()。这个函数用来舍掉一个数的小数点后的部分，并且向上舍入到邻近的最大的整数。比如 3.33 将被舍入为 4、2.89 将被舍入为 3、-3.61 将被舍入为-3。如果感觉-3.61 将被舍入为-3 而不是-4 感到奇怪，那么请记住这个函数是用来向上舍入到最大的整数，在英语中 CEILING 可以理解为“天花板”，舍入到“高高在上的天花板”当然也就是尽可能大的舍入了。

这个函数有一个参数，参数为待舍入的数值，执行下面的 SQL 语句：

MYSQL、MSSQLServer、DB2：

```
SELECT FName,FWeight, CEILING(FWeight), CEILING(FWeight*-1)
FROM T_Person
```

Oracle：

```
SELECT FName,FWeight, CEIL(FWeight) , CEIL (FWeight*-1)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight		
Tom	56.67	57	-56
Jim	36.17	37	-36
Lily	40.33	41	-40
Kelly	46.23	47	-46
Sam	48.68	49	-48
Kerry	66.67	67	-66
Smith	51.28	52	-51
BillGates	60.32	61	-60

5.1.6 舍入到最小整数

SQL 中提供了 FLOOR()函数，和 CEILING()函数正好相反，FLOOR()函数用来舍掉一个数的小数点后的部分，并且向下舍入到邻近的最小的整数。比如 3.33 将被舍入为 3、2.89 将被舍入为 2、-3.61 将被舍入为-4。如果感觉-3.61 将被舍入为-4 而不是-3 感到奇怪，那么请记住这个函数是用来向下舍入到最小的整数，在英语中 FLOOR 可以理解为“地板”，舍入到“低低在下的地板”当然也就是尽可能小的舍入了。

这个函数有一个参数，参数为待舍入的数值，执行下面的 SQL 语句：

```
SELECT FName,FWeight,FLOOR(FWeight),FLOOR(FWeight*-1)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight		
Tom	56.67	56	-57
Jim	36.17	36	-37
Lily	40.33	40	-41
Kelly	46.23	46	-47
Sam	48.68	48	-49
Kerry	66.67	66	-67
Smith	51.28	51	-52
BillGates	60.32	60	-61

5.1.7 四舍五入

ROUND()函数也是用来进行数值四舍五入的，不过不像 CEILING()函数那样总是向最大值舍入或者像 FLOOR()函数那样总是向最小值舍入，ROUND()函数将数值向最近的数值

舍入。在英语中 ROUND 可以理解为“半径”，舍入到“离我半径最近的数”当然也就是四舍五入了。

ROUND()函数有两个参数和单一参数两种用法，下面分别进行介绍。

5.1.7.1 两个参数

两个参数的 ROUND()函数用法为：ROUND(m,d)，其中 m 为待进行四舍五入的数值，而 d 则为计算精度，也就是进行四舍五入时保留的小数位数。比如 3.663 进行精度为 2 的四舍五入得到 3.66、-2.337 进行精度为 2 的四舍五入得到-2.34、3.32122 进行精度为 3 的四舍五入得到 3.321。当 d 为 0 的时候则表示不保留小数位进行四舍五入，比如 3.663 进行精度为 0 的四舍五入得到 4、-2.337 进行精度为 0 的四舍五入得到-2、3.32122 进行精度为 0 的四舍五入得到 3。

特别值得一提的是，d 还可以取负值，这时表示在整数部分进行四舍五入。比如 36.63 进行精度为-1 的四舍五入得到 40、233.7 进行精度为-2 的四舍五入得到 200、3321.22 进行精度为-2 的四舍五入得到 3300。

执行下面的 SQL 语句：

```
SELECT FName,FWeight, ROUND(FWeight,1),
ROUND(FWeight*-1,0) , ROUND(FWeight,-1)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	ROUND(FWeight,1)	ROUND(FWeight*-1,0)	ROUND(FWeight,-1)
Tom	56.67	56.7	-57	60
Jim	36.17	36.2	-36	40
Lily	40.33	40.3	-40	40
Kelly	46.23	46.2	-46	50
Sam	48.68	48.7	-49	50
Kerry	66.67	66.7	-67	70
Smith	51.28	51.3	-51	50
BillGates	60.32	60.3	-60	60

5.1.7.2 单一参数

单一参数的 ROUND()函数用法为：ROUND(m)，其中 m 为待进行四舍五入的数值，它可以看做精度为 0 的四舍五入运算，也就是 ROUND(m,0)。比如 3.663 进行四舍五入得到 4、-2.337 进行四舍五入得到-2、3.32122 进行四舍五入得到 3。

执行下面的 SQL 语句：

```
SELECT FName,FWeight, ROUND(FWeight), ROUND(FWeight*-1)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	ROUND(FWeight)	ROUND(FWeight*-1)
Tom	56.67	57	-57
Jim	36.17	36	-36
Lily	40.33	40	-40
Kelly	46.23	46	-46
Sam	48.68	49	-49
Kerry	66.67	67	-67
Smith	51.28	51	-51
BillGates	60.32	60	-60

注意这种单一函数的用法在 MSSQLServer 上以及 DB2 上不被支持，必须显示的指明精度为 0。

5.1.8 求正弦值

用来计算一个数值的正弦值的函数为 SIN()，它接受一个参数，这个参数为待计算正弦值的表达式。执行下面的 SQL 语句：

```
SELECT FName,FWeight,SIN(FWeight)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	SIN(FWeight)
Tom	56.67	0.12103475545596
Jim	36.17	-0.99913132770301
Lily	40.33	0.48879197128413
Kelly	46.23	0.77951415191171
Sam	48.68	-0.99989216072166
Kerry	66.67	-0.64157843190998
Smith	51.28	0.84922589875387
BillGates	60.32	-0.58893431289327

5.1.9 求余弦值

用来计算一个数值的余弦值的函数为 COS ()，它接受一个参数，这个参数为待计算余弦值的表达式。执行下面的 SQL 语句：

```
SELECT FName,FWeight, COS(FWeight) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	COS(FWeight)
Tom	56.67	0.992648270019
Jim	36.17	0.041672412966167
Lily	40.33	-0.87240037185238
Kelly	46.23	-0.62638461584666
Sam	48.68	-0.01468560272409
Kerry	66.67	-0.76705743964056
Smith	51.28	0.52802970833626
BillGates	60.32	-0.80818090493214

5.1.10 求反正弦值

用来计算一个数值的反正弦值的函数为 ASIN()，它接受一个参数，这个参数为待计算反正弦值的表达式。执行下面的 SQL 语句：

```
SELECT FName,FWeight, ASIN(1/FWeight) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	ASIN(1/FWeight)
Tom	56.67	0.017646935903908
Jim	36.17	0.027650744324362
Lily	40.33	0.024797978465578
Kelly	46.23	0.021632662207488
Sam	48.68	0.020543762038815
Kerry	66.67	0.014999812472576
Smith	51.28	0.019502016172357

BillGates	60.32	0.016579008483674

5.1.11 求反余弦值

用来计算一个数值的反余弦值的函数为 ACOS()，它接受一个参数，这个参数为待计算反余弦值的表达式。执行下面的 SQL 语句：

```
SELECT FName,FWeight, ACOS(1/FWeight) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	ACOS(1/FWeight)
Tom	56.67	1.553149390891
Jim	36.17	1.5431455824705
Lily	40.33	1.5459983483293
Kelly	46.23	1.5491636645874
Sam	48.68	1.5502525647561
Kerry	66.67	1.5557965143223
Smith	51.28	1.5512943106225
BillGates	60.32	1.5542173183112

5.1.12 求正切值

用来计算一个数值的正切值的函数为 TAN()，它接受一个参数，这个参数为待计算正切值的表达式。执行下面的 SQL 语句：

```
SELECT FName,FWeight, TAN(FWeight) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	TAN(FWeight)
Tom	56.67	0.12193116042367
Jim	36.17	-23.97584532756
Lily	40.33	-0.56028400153735
Kelly	46.23	-1.2444656720346
Sam	48.68	68.086559299432
Kerry	66.67	0.8364151089006
Smith	51.28	1.6082918921923
BillGates	60.32	0.7287159462679

5.1.13 求反正切值

用来计算一个数值的反正切值的函数为 ATAN()，它接受一个参数，这个参数为待计算反正切值的表达式。执行下面的 SQL 语句：

```
SELECT FName,FWeight, ATAN(FWeight) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	ATAN(FWeight)
Tom	56.67	1.5531521371819
Jim	36.17	1.5431561463368
Lily	40.33	1.5460059688069
Kelly	46.23	1.5491687239958
Sam	48.68	1.550256898419
Kerry	66.67	1.5557982014369
Smith	51.28	1.5512980181215
BillGates	60.32	1.5542195959872

5.1.14 求 2 个变量的反正切

ATAN2 函数（在 MYSQLServer 中这个函数名称为 ATN2）用来计算 2 个变量的反正切，其使用格式为：ATAN2(X,Y)，函数返回 2 个变量 X 和 Y 的反正切。它类似于计算 Y/X 的反正切，除了两个参数的符号被用来决定结果的象限。

执行下面的 SQL 语句：

```
MYSQL,Oracle,DB2:
    SELECT FName,FWeight, ATAN2(FWeight,2)  FROM T_Person
MSSQLServer:
    SELECT FName,FWeight, ATN2(FWeight,2)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	ATAN2(FWeight,2)
Tom	56.67	1.5355189266211
Jim	36.17	1.5155581345314
Lily	40.33	1.5212460438378
Kelly	46.23	1.5275613350507
Sam	48.68	1.5297347852958
Kerry	66.67	1.5408068205144
Smith	51.28	1.5318145240766
BillGates	60.32	1.5376519703496

5.1.15 求余切

用来计算一个数值的反正切值的函数为 **COT**()，它接受一个参数，这个参数为待计算余切值的表达式。在 Oracle 中不支持这个函数，不过根据余切是正切的倒数这一个特性，可以使用 TAN()函数来变通实现。

执行下面的 SQL 语句：

MYSQL,MSSQLServer,DB2:

SELECT FName,FWeight, COT(FWeight) FROM T_Person

Oracle:

SELECT FName,FWeight,1/tan(FWeight) FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	COT(FWeight)
Tom	56.67	8.2013489950015
Jim	36.17	-0.04170864410985
Lily	40.33	-1.7848091276141
Kelly	46.23	-0.8035577215763
Sam	48.68	0.014687186579692
Kerry	66.67	1.1955785941199
Smith	51.28	0.62177767907346
BillGates	60.32	1.3722768180407

5.1.16 求圆周率π值

圆周率π值是一个恒定值，所以在使用它的时候可以使用 3.1415926……来引用它，不过手工输入π值非常容易出错，在 MYSQL 和 MSSQLServer 中提供了 PI()函数用来取得圆周率π值，这个函数不需要使用参数，在 Oracle 和 DB2 中不支持 PI()函数，不过根据-1 的反余弦值等于π值的这一特性，我们可以用 ACOS(-1)来变通实现。

执行下面的 SQL 语句：

MYSQL,MSSQLServer:

SELECT FName,FWeight,FWeight *PI() FROM T_Person

Oracle,DB2:

SELECT FName,FWeight,FWeight * acos(-1) FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	FWeight*PI()
Tom	56.67	178.034056
Jim	36.17	113.631406
Lily	40.33	126.700432
Kelly	46.23	145.235828
Sam	48.68	152.932730
Kerry	66.67	209.449982
Smith	51.28	161.100871
BillGates	60.32	189.500869

5.1.17 弧度制转换为角度制

用来将一个数值从弧度制转换为角度制的函数为 **DEGREES** ()，它接受一个参数，这个参数为待转换的表达式。在 Oracle 和 DB2 中不支持这个函数，不过根据：

角度制=弧度制*180/π

这一个特性，可以用变通方式来实现。

执行下面的 SQL 语句：

MYSQL,MSSQLServer:

SELECT FName,FWeight, DEGREES(FWeight) FROM T_Person

Oracle,DB2:

SELECT FName,FWeight,(FWeight*180)/acos(-1) FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	DEGREES(FWeight)
Tom	56.67	3246.9518250064
Jim	36.17	2072.3883449882
Lily	40.33	2310.7387877626
Kelly	46.23	2648.7838868898
Sam	48.68	2789.1585466968
Kerry	66.67	3819.9096201372
Smith	51.28	2938.1275734309
BillGates	60.32	3456.0814202291

5.1.18 角度制转换为弧度制

用来将一个数值从角度制转换为弧度制的函数为 **RADIANS** ()，它接受一个参数，这个参数为待转换的表达式。在 Oracle 和 DB2 中不支持这个函数，不过根据：

弧度制=角度制*π/180

这一个特性，可以用变通方式来实现。

执行下面的 SQL 语句：

MYSQL,MSSQLServer:

SELECT FName,FWeight, RADIANS(FWeight) FROM T_Person

Oracle,DB2:

SELECT FName,FWeight,(FWeight*acos(-1)/180) FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	RADIANS(FWeight)
Tom	56.67	0.98907808710519
Jim	36.17	0.63128559044635
Lily	40.33	0.70389128732931
Kelly	46.23	0.80686571319698
Sam	48.68	0.84962627987084
Kerry	66.67	1.1636110123046
Smith	51.28	0.89500484042269
BillGates	60.32	1.052782604803

5.1.19 求符号

SIGN()函数用来返回一个数值的符号，如果数值大于 0 则返回 1，如果数值等于 0 则返回 0，如果数值小于 0 则返回-1。该函数接受一个参数，这个参数为待求绝对值的表达式。执行下面的 SQL 语句：

```
SELECT FName,FWeight-48.68,SIGN(FWeight-48.68) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight - 48.68	SIGN(FWeight - 48.68)
Tom	7.99	1
Jim	-12.51	-1
Lily	-8.35	-1
Kelly	-2.45	-1
Sam	0.00	0
Kerry	17.99	1
Smith	2.60	1
BillGates	11.64	1

5.1.20 求符号

SIGN()函数用来返回一个数值的符号，如果数值大于 0 则返回 1，如果数值等于 0 则返回 0，如果数值小于 0 则返回-1。该函数接受一个参数，这个参数为待求绝对值的表达式。执行下面的 SQL 语句：

```
SELECT FName,FWeight-48.68,SIGN(FWeight-48.68) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight - 48.68	SIGN(FWeight - 48.68)
Tom	7.99	1
Jim	-12.51	-1
Lily	-8.35	-1
Kelly	-2.45	-1
Sam	0.00	0
Kerry	17.99	1
Smith	2.60	1
BillGates	11.64	1

5.1.21 求整除余数

MOD()函数用来计算两个数整除后的余数。该函数接受两个参数，第一个参数为除数，而第二个参数则是被除数。在 MYSQL 和 Oracle 中提供了对 MOD()函数的直接支持；在

MSSQLServer 中不支持 MOD()，不过 MSSQLServer 中直接提供了操作符“%”用来计算两个数的整除余数；DB2 中不支持求整除余数操作。

执行下面的 SQL 语句：

```
MYSQL,Oracle:
    SELECT FName,FWeight,MOD(FWeight , 5) FROM T_Person
MSSQLServer:
    SELECT FName,FWeight,FWeight % 5 FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	(FWeight % 5)
Tom	56.67	1.67
Jim	36.17	1.17
Lily	40.33	0.33
Kelly	46.23	1.23
Sam	48.68	3.68
Kerry	66.67	1.67
Smith	51.28	1.28
BillGates	60.32	0.32

5.1.22 求自然对数

LOG ()函数用来计算一个数的自然对数值。该函数接受一个参数，此参数为待计算自然对数的表达式，在 Oracle 中这个函数的名称为 LN()。

执行下面的 SQL 语句：

```
MYSQL,MSSQLServer,DB2:
    SELECT FName,FWeight, LOG(FWeight)  FROM T_Person
Oracle:
    SELECT FName,FWeight, LN(FWeight)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	LOG(FWeight)
Tom	56.67	4.0372449701815
Jim	36.17	3.588230045964
Lily	40.33	3.6970956088853
Kelly	46.23	3.8336289380001
Sam	48.68	3.8852682681193
Kerry	66.67	4.19975507663
Smith	51.28	3.9373008126124
BillGates	60.32	4.0996637236997

5.1.23 求自然对数

LOG ()函数用来计算一个数的自然对数值。该函数接受一个参数，此参数为待计算自然对数的表达式，在 Oracle 中这个函数的名称为 LN()。

执行下面的 SQL 语句：

```
MYSQL,MSSQLServer,DB2:
    SELECT FName,FWeight, LOG(FWeight)  FROM T_Person
Oracle:
    SELECT FName,FWeight, LN(FWeight)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	LOG(FWeight)
Tom	56.67	4.0372449701815
Jim	36.17	3.588230045964
Lily	40.33	3.6970956088853
Kelly	46.23	3.8336289380001
Sam	48.68	3.8852682681193
Kerry	66.67	4.19975507663
Smith	51.28	3.9373008126124
BillGates	60.32	4.0996637236997

5.1.24 求以 10 为底的对数

LOG10()函数用来计算一个数的以 10 为底的对数值。该函数接受一个参数，此参数为待计算对数的表达式，在 Oracle 中不支持这个函数，不过 Oracle 中有一个可以计算任意数为底的对数的函数 LOG(m,n)，它用来计算以 m 为底 n 的对数，我们将 m 设为常量 10 就可以了。

执行下面的 SQL 语句：

```
MYSQL,MSSQLServer,DB2:
    SELECT FName,FWeight, LOG10(FWeight)  FROM T_Person
Oracle:
    SELECT FName,FWeight,LOG(10,FWeight) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	LOG10(FWeight)
Tom	56.67	1.7533532126415
Jim	36.17	1.5583485087616
Lily	40.33	1.6056282220076
Kelly	46.23	1.6649238934381
Sam	48.68	1.687350569558
Kerry	66.67	1.8239304551256
Smith	51.28	1.7099480165108
BillGates	60.32	1.7804613328617

5.1.25 求幂

POWER(X,Y)函数用来计算 X 的 Y 次幂。

执行下面的 SQL 语句：

```
SELECT FName,FWeight, POWER(1.18,FWeight)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	POWER(1.18,FWeight)
Tom	56.67	11845.498826328
Jim	36.17	398.08169169355
Lily	40.33	792.50380246303
Kelly	46.23	2104.2880930071
Sam	48.68	3156.5750223127
Kerry	66.67	61997.392909716
Smith	51.28	4854.1068806426
BillGates	60.32	21673.182135864

5.2 字符串函数

除了数值类型的数据，字符串类型的数据也是数据库系统中经常用到的数据类型，比如用户的密码、电子邮箱地址、证件号码等都是以字符串类型保存在数据库中的。我们经常需要对这些数据进行一些处理，比如截取身份证号码前 5 位、将电子邮箱地址全部改为大写、去掉用户名中的空格，SQL 中提供了丰富的字符串函数用于完成这些功能，本节将对这些字符串函数进行详细讲解。

5.2.1 计算字符串长度

LENGTH()函数用来计算一个字符串的长度。该函数接受一个参数，此参数为待计算的字符串表达式，在 MYSQLServer 中这个函数名称为 LEN()。

执行下面的 SQL 语句：

```
MYSQL,Oracle,DB2:
    SELECT FName, LENGTH(FName)  FROM T_Person
MSSQLServer:
    SELECT FName, LEN(FName)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	LENGTH(FName)
Tom	3
Jim	3
Lily	4
Kelly	5
Sam	3
Kerry	5
Smith	5
BillGates	9

5.2.2 字符串转换为小写

LOWER()函数用来将一个字符串转换为小写。该函数接受一个参数，此参数为待转换的字符串表达式，在 DB2 中这个函数名称为 LCASE()。

执行下面的 SQL 语句：

```
MYSQL,MSSQLServer,Oracle:
    SELECT FName, LOWER(FName)  FROM T_Person
DB2:
    SELECT FName, LCASE(FName)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	LOWER(FName)
Tom	tom
Jim	jim
Lily	lily
Kelly	kelly
Sam	sam
Kerry	kerry
Smith	smith
BillGates	billgates

5.2.3 字符串转换为大写

与 LOWER()函数正好相反，UPPER ()函数用来将一个字符串转换为大写。该函数接受

一个参数，此参数为待转换的字符串表达式，在 DB2 中这个函数名称为 UCASE ()。

执行下面的 SQL 语句：

```
MYSQL,MSSQLServer,Oracle:
    SELECT FName, UPPER(FName)  FROM T_Person
DB2:
    SELECT FName, UCASE(FName)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	UPPER(FName)
Tom	TOM
Jim	JIM
Lily	LILY
Kelly	KELLY
Sam	SAM
Kerry	KERRY
Smith	SMITH
BillGates	BILLGATES

5.2.4 截去字符串左侧空格

LTRIM()函数用来将一个字符串左侧的空格去掉。该函数接受一个参数，此参数为待处理的字符串表达式。

执行下面的 SQL 语句：

```
SELECT FName,LTRIM(FName),LTRIM('  abc    ')  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	LTRIM(FName)	LTRIM(' abc ')
Tom	Tom	abc
Jim	Jim	abc
Lily	Lily	abc
Kelly	Kelly	abc
Sam	Sam	abc
Kerry	Kerry	abc
Smith	Smith	abc
BillGates	BillGates	abc

再执行下面的 SQL 语句：

MYSQL:

SELECT ' 123 abc ', LENGTH(' 123 abc ') ,
LENGTH(LTRIM(' 123 abc '))

MSSQLServer:

SELECT ' 123 abc ', LEN(' 123 abc ') ,
LEN(LTRIM(' 123 abc '))

Oracle:

SELECT ' 123 abc ', LENGTH(' 123 abc ') ,
LENGTH(LTRIM(' 123 abc ')) FROM DUAL

DB2:

SELECT ' 123 abc ', LENGTH(' 123 abc ') ,
LENGTH(LTRIM(' 123 abc ')) FROM SYSIBM.SYSDUMMY1

执行完毕我们就能在输出结果中看到下面的执行结果：

123 abc	LENGTH(' 123 abc ')	LENGTH(LTRIM(' 123 abc '))
123 abc	13	11

5.2.5 截去字符串右侧空格

RTRIM ()函数用来将一个字符串左侧的空格去掉。该函数接受一个参数，此参数为待处理的字符串表达式。

执行下面的 SQL 语句：

```
SELECT FName,RTRIM(FName),RTRIM('   abc   ') FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	RTRIM(FName)	RTRIM(' abc ')
Tom	Tom	abc
Jim	Jim	abc
Lily	Lily	abc
Kelly	Kelly	abc
Sam	Sam	abc
Kerry	Kerry	abc
Smith	Smith	abc
BillGates	BillGates	abc

再执行下面的 SQL 语句：

MYSQL:

SELECT ' 123 abc ', LENGTH(' 123 abc ') ,
LENGTH(RTRIM(' 123 abc '))

MSSQLServer:

SELECT ' 123 abc ', LEN(' 123 abc ') ,
LEN(RTRIM(' 123 abc '))

Oracle:

SELECT ' 123 abc ', LENGTH(' 123 abc ') ,
LENGTH(RTRIM(' 123 abc ')) FROM DUAL

DB2:

SELECT ' 123 abc ', LENGTH(' 123 abc ') ,
LENGTH(RTRIM(' 123 abc ')) FROM SYSIBM.SYSDUMMY1

执行完毕我们就能在输出结果中看到下面的执行结果：

123 abc	LENGTH(' 123 abc ')	LENGTH(RTRIM(' 123 abc '))
123 abc	13	9

5.2.6 截去字符串两侧的空格

TRIM ()函数用来将一个字符串两侧的空格去掉。该函数接受一个参数，此参数为待处理的字符串表达式。此函数只在 MYSQL 和 Oracle 中提供支持，不过在 MSSQLServer 和 DB2 中可以使用 L**TRIM** ()函数和 R**TRIM** ()函数复合来进行变通实现，也就是用 LTRIM(RTRIM(string))来模拟实现 **TRIM** (string)。

执行下面的 SQL 语句：

MYSQL,Oracle:

```
SELECT FName,TRIM(FName),TRIM('   abc   ') FROM T_Person
```

MSSQLServer,DB2:

```
SELECT FName,LTRIM(RTRIM(FName)),LTRIM(RTRIM('    abc    ')) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	TRIM(FName)	TRIM(' abc ')
Tom	Tom	abc
Jim	Jim	abc
Lily	Lily	abc
Kelly	Kelly	abc
Sam	Sam	abc
Kerry	Kerry	abc
Smith	Smith	abc
BillGates	BillGates	abc

再执行下面的 SQL 语句：

MYSQL:

```
SELECT '  123 abc  ', LENGTH('  123 abc  '),
LENGTH(TRIM('  123 abc  '))
```

MSSQLServer:

```
SELECT '  123 abc  ', LEN('  123 abc  '),
LEN(LTRIM(RTRIM('  123 abc  ')))
```

Oracle:

```
SELECT '  123 abc  ', LENGTH('  123 abc  '),
LENGTH(TRIM('  123 abc  ')) FROM DUAL
```

DB2:

```
SELECT '  123 abc  ', LENGTH('  123 abc  '),
LENGTH(LTRIM(RTRIM('  123 abc  '))) FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

123 abc	LENGTH(' 123 abc ')	LENGTH(TRIM(' 123 abc '))
123 abc	13	7

5.2.7 取子字符串

字符串是由多个字符组成的串，比如“HelloWorld”在内存是如下存储的：

1	2	3	4	5	6	7	8	9	10
H	e	l	l	o	W	o	r	l	d

表格第一行的数字表示组成字符串的每个字符的位置，第二行则为各个位置上的字符。由这些字符中连续的多个字符还可以组成新的字符串，新的字符串则被称为“子字符串”。比如从第 3 个字符到第 5 个字符组成的“llo”就是一个子字符串，我们也可以称“llo”为从第 3 个字符开始长度为 3 的子字符串。

SQL 中提供了用来计算子字符串的函数 SUBSTRING()，其参数格式如下：

SUBSTRING(string,start_position,length)

其中参数 string 为主字符串，start_position 为子字符串在主字符串中的起始位置，length 为子字符串的最大长度。在 MYSQL 和 MSSQLServer 中支持这个函数，而在 Oracle 和 DB2 中这个函数的名称则为 SUBSTR()，仅仅是名称不同而已，在用法没有什么不同。

执行下面的 SQL 语句：

MYSQL、MSSQLServer:

SELECT SUBSTRING('abcdef111',2,3)

Oracle:

SELECT SUBSTR('abcdef111',2,3) FROM DUAL

DB2:

SELECT SUBSTR('abcdef111',2,3) FROM SYSIBM.SYSDUMMY1

执行完毕我们就能在输出结果中看到下面的执行结果：

SUBSTRING('abcdef111',2,3)
bcd

再执行下面的 SQL 语句：

MYSQL,MSSQLServer:

SELECT FName, SUBSTRING(FName,2,3) FROM T_Person

Oracle,DB2:

SELECT FName, SUBSTR(FName,2,3) FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	SUBSTRING(FName,2,3)
Tom	om
Jim	im
Lily	ily
Kelly	ell
Sam	am
Kerry	err
Smith	mit
BillGates	ill

5.2.8 计算子字符串的位置

SQL 中提供了计算子字符串在主字符串中位置的函数，这个函数可以检测制定的子字符串是否存在于主字符串中，如果存在则还可以返回所在的位置。这个函数在 MYSQL 和 Oracle 中名称为 INSTR，其参数格式如下：

INSTR(string,substring)

其中参数 string 为主字符串，参数 substring 为待查询的子字符串。如果 string 中存在 substring 子字符串，则返回子字符串第一个字符在主字符串中出现的位置；如果 string 中不存在 substring 子字符串，则返回 0。

在 MSSQLServer 中这个函数名为 CHARINDEX，其参数格式以及返回值规则与 MYSQL 以及 Oracle 一致。

在 DB2 中这个函数名为 LOCATE，其返回值规则与前述几种数据库系统一致，不过参数格式与它们正好相反，其参数格式如下：

LOCATE(substring,string)

其中参数 substring 为待查询的子字符串,参数 string 为主字符串，也就是两个参数的位置是与其它集中数据库系统相反的，这一点在使用的时候需要特别注意的。

执行下面的 SQL 语句：

MYSQL,Oracle:

SELECT FName, INSTR(FName,'m') , INSTR(FName,'ly')

FROM T_Person

MSSQLServer:

SELECT FName,CHARINDEX(FName,'m'), CHARINDEX(FName,'ly')
FROM T_Person

DB2:

SELECT FName, LOCATE('m',FName) , LOCATE('ly',FName)
FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	INSTR(FName,'m')	INSTR(FName,'ly')
Tom	3	0
Jim	3	0
Lily	0	3
Kelly	0	4
Sam	3	0
Kerry	0	0
Smith	2	0
BillGates	0	0

5.2.9 从左侧开始取子字符串

使用 SUBSTRING()函数我们可以从任意位置开始取任意长度的子字符串，不过有的时候我们只需要从左侧开始取子字符串，这样指定主字符串和要取的长度就可以了，不过如果使用 SUBSTRING()函数的话仍然需要指定三个参数，其中第二个参数为常量 1。MYSQL、MSSQLServer、DB2 中提供了 LEFT()函数用于从左侧开始取任意长度的子字符串，其参数格式如下：

LEFT (string,length)

其中参数 string 为主字符串，length 为子字符串的最大长度。

Oracle 中不支持 LEFT()函数，只能使用 SUBSTR()函数进行变通实现，也就是 SUBSTR(string, 1, length)。

执行下面的 SQL 语句：

MYSQL,MSSQLServer,DB2:

SELECT FName, LEFT(FName,3) , LEFT(FName,2)
FROM T_Person

Oracle:

SELECT FName,SUBSTR(FName, 1,3),SUBSTR(FName, 1,2)
FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	LEFT(FName,3)	LEFT(FName,2)
Tom	Tom	To
Jim	Jim	Ji
Lily	Lil	Li
Kelly	Kel	Ke
Sam	Sam	Sa
Kerry	Ker	Ke
Smith	Smi	Sm
BillGates	Bil	Bi

5.2.10 从右侧开始取子字符串

使用 SUBSTRING()函数我们可以从任意位置开始取任意长度的子字符串，不过有的时

候我们只需要从右侧开始取子字符串，这样指定主字符串和要取的长度就可以了，不过如果使用 SUBSTRING()函数的话仍然需要指定三个参数。MYSQL、MSSQLServer、DB2 中提供了 RIGHT ()函数用于从左侧开始取任意长度的子字符串，其参数格式如下：

RIGHT (string,length)

其中参数 string 为主字符串，length 为子字符串的最大长度。

Oracle 中不支持 RIGHT ()函数，只能使用 SUBSTR()函数进行变通实现，其中起始位置用如下表达式计算出来：startposition= LENGTH(string)- length+1

也就是 SUBSTR(string, LENGTH(string)- length+1, length)等价于 RIGHT (string,length)。

执行下面的 SQL 语句：

MYSQL,MSSQLServer,DB2:

```
SELECT FName, RIGHT(FName,3) , RIGHT(FName,2)
FROM T_Person
```

Oracle:

```
SELECT FName,SUBSTR(FName, LENGTH(FName)-3 +1, 3),
SUBSTR(FName, LENGTH(FName)-2 +1, 2) FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	RIGHT(FName,3)	RIGHT(FName,2)
Tom	Tom	om
Jim	Jim	im
Lily	ily	ly
Kelly	lly	ly
Sam	Sam	am
Kerry	rry	ry
Smith	ith	th
BillGates	tes	es

5.2.11 字符串替换

REPLACE()函数可以用来将字符串的指定的子字符串替换为其它的字符串，比如将“Hello World”中的“rl”替换为“ok”后得到“Hello Wookd”，而把“Just so so”中的“s”替换为“z”后得到“Juzt zo zo”。REPLACE()函数的参数格式如下：

REPLACE(string,string_tobe_replace,string_to_replace)

其中参数 string 为要进行替换操作的主字符串，参数 string_tobe_replace 为要被替换的字符串，而 string_to_replace 将替换 string_tobe_replace 中所有出现的地方。

执行下面的 SQL 语句：

```
select FName,REPLACE(FName,'i','e'),FIDNumber,
REPLACE(FIDNumber,'2345','abcd') FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	REPLACE(FName,'i','e')	FIDNumber	REPLACE(FIDNumber,'2345','abcd')
Tom	Tom	123456789120	1abcd6789120
Jim	Jem	123456789121	1abcd6789121
Lily	Lely	123456789122	1abcd6789122
Kelly	Kelly	123456789123	1abcd6789123
Sam	Sam	123456789124	1abcd6789124
Kerry	Kerry	123456789125	1abcd6789125
Smith	Smeth	123456789126	1abcd6789126

BillGates	BellGates	123456789127	1abcd6789127

SQL 中没有提供删除字符串中匹配的子字符串的方法，因为使用 REPLACE()函数就可以达到删除子字符串的方法，那就是将第三个参数设定为空字符串，用空字符串来替换匹配的子字符串也就达到了删除指定子字符串的效果了。比如下面的 SQL 语句用来将 FName 中的 m 以及 FIDNumber 中的 123 删除：

```
SELECT FName, REPLACE(FName,'m','') ,FIDNumber,
REPLACE(FIDNumber,'123','')  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	REPLACE(FName,'m','')	FIDNumber	REPLACE(FIDNumber,'123','')
Tom	To	123456789120	456789120
Jim	Ji	123456789121	456789121
Lily	Lily	123456789122	456789122
Kelly	Kelly	123456789123	456789
Sam	Sa	123456789124	456789124
Kerry	Kerry	123456789125	456789125
Smith	Sith	123456789126	456789126
BillGates	BillGates	123456789127	456789127

LTRIM()、RTRIM()和 TRIM()都只能删除两侧的字符串，无法删除字符串中间的空格，而使用 REPLACE()函数也可以完成这个功能，也就是用空字符串替换中所有的空格。执行下面的 SQL 语句：

MYSQL、MSSQLServer:

```
SELECT  REPLACE(' abc  123  wpf',' ','') , REPLACE(' ccw  enet  wcf f',' ','')
```

Oracle:

```
SELECT  REPLACE(' abc  123  wpf',' ','') ,
REPLACE(' ccw  enet  wcf f',' ','')  FROM DUAL
```

DB2:

```
SELECT  REPLACE(' abc  123  wpf',' ','') ,
REPLACE(' ccw  enet  wcf f',' ','')  FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

REPLACE(' abc 123 wpf',' ','')	REPLACE(' ccw enet wcf f',' ','')
abc123wpf	ccwenetwcff

5.2.12 得到字符的 ASCII 码

ASCII()函数用来得到一个字符的 ASCII 码，它有且只有一个参数，这个参数为待求 ASCII 码的字符，如果参数为一个字符串则函数返回第一个字符的 ASCII 码，比如执行下面的 SQL 语句：

MYSQL,MSSQLServer:

```
SELECT  ASCII('a') , ASCII('abc')
```

Oracle:

```
SELECT  ASCII('a') , ASCII('abc')  FROM DUAL
```

DB2:

```
SELECT  ASCII('a') , ASCII('abc')  FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ASCII('a')	ASCII('abc')
97	97

下面的 SQL 语句用来计算每个员工姓名的第一个字符的 ASCII 码：

MYSQL,MSSQLServer,DB2:

```
SELECT FName, LEFT(FName,1) , ASCII( LEFT(FName,1) ) ,
ASCII(FName)  FROM T_Person
```

Oracle:

```
SELECT FName,SUBSTR(FName, 1,1), ASCII(SUBSTR(FName, 1,1)) ,
ASCII(FName)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	LEFT(FName,1)	ASCII(LEFT(FName,1))	ASCII(FName)
Tom	T	84	84
Jim	J	74	74
Lily	L	76	76
Kelly	K	75	75
Sam	S	83	83
Kerry	K	75	75
Smith	S	83	83
BillGates	B	66	66

5.2.13 得到一个 ASCII 码数字对应的字符

与 ASCII()函数正好相反，SQL 还提供了用来得到一个字符的 ASCII 码的函数。在 MYSQL、MSSQLServer 和 DB2 中，这个函数的名字是 CHAR()，而在 Oracle 中这个函数的名字则为 CHR()。

执行下面的 SQL 语句：

MYSQL,MSSQLServer:

```
SELECT  CHAR(56) , CHAR(90) ,'a', CHAR( ASCII('a') )
```

Oracle:

```
SELECT  CHR(56) , CHR(90) ,'a', CHR( ASCII('a') )
FROM DUAL
```

DB2:

```
SELECT CHR(56),CHR(90),'a',CHR( ASCII('a') )
FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CHAR(56)	CHAR(90)	a	CHAR(ASCII('a'))
8	Z	a	a

下面的 SQL 语句将 FWeight 转换为整数，然后得到它对应的字符：

MYSQL、MSSQLServer:

```
SELECT FWeight, CEILING(FWeight) , CHAR( CEILING(FWeight) )
FROM T_Person
```

Oracle:

```
SELECT FWeight, CEIL(FWeight) , CHR( CEIL(FWeight) )
FROM T_Person
```

DB2:

```
SELECT FWeight, CEILING(FWeight),CHR(int(CEILING(FWeight)))
FROM T_Person
```

由于 DB2 的类型检查机制非常严格，所以在 DB2 中需要用 int()函数将 CEILING()函

数的返回值显示的转换为整数类型。

执行完毕我们就能在输出结果中看到下面的执行结果：

FWeight	CEILING(FWeight)	CHAR(CEILING(FWeight))
56.67	57	9
36.17	37	%
40.33	41	)
46.23	47	/
48.68	49	1
66.67	67	C
51.28	52	4
60.32	61	=

5.2.14 发音匹配度

到目前为止所有关于字符串的匹配都是针对其拼写形式的，比如下面的 SQL 语句用于检索年龄为“jack”的员工：

```
SELECT * from T_Person
WHERE FName='jack'
```

有的时候我们并不知道一个人姓名的准确拼写，只知道它的发音，这是在公安、医疗、教育等系统中是经常需要的功能，比如“检索名字发音为和[jeck]类似的人员”，这时我们就要进行发音的匹配度测试了。

SQL 中提供了 SOUNDEX()函数用于计算一个字符串的发音特征值，这个特征值为一个四个字符的字符串，特征值的第一个字符总是初始字符串中的第一个字符，而其后则是一个三位数字的数值。下面的 SQL 语句用于查询几个名字的发音特征值：

MYSQL,MSSQLServer:

```
SELECT  SOUNDEX('jack') , SOUNDEX('jeck') , SOUNDEX('joke') ,
SOUNDEX('juke') , SOUNDEX('look') , SOUNDEX('jobe')
```

Oracle:

```
SELECT  SOUNDEX('jack') , SOUNDEX('jeck') , SOUNDEX('joke') ,
SOUNDEX('juke') , SOUNDEX('look') , SOUNDEX('jobe')
FROM DUAL
```

DB2:

```
SELECT  SOUNDEX('jack') , SOUNDEX('jeck') , SOUNDEX('joke') ,
SOUNDEX('juke') , SOUNDEX('look') , SOUNDEX('jobe')
FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

SOUNDEX('jack')	SOUNDEX('jeck')	SOUNDEX('joke')	SOUNDEX('juke')	SOUNDEX('look')	SOUNDEX('jobe')
J000	J000	J000	J000	L200	J100

可以看到 jack、jeck、joke、juke 几个字符串的发音非常相似，而 look、jobe 的发音则和它们差距比较大。

下面的 SQL 语句用于查询公司所有员工姓名的发音特征值：

```
SELECT FName, SOUNDEX(FName)  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	SOUNDEX(FName)
Tom	T500

Jim	J500
Lily	L000
Kelly	K400
Sam	S500
Kerry	K600
Smith	S530
BillGates	B4232

发音特征值的含义非常复杂，如果要根据两个发音特征值来分析两个字符串的发音相似度的话非常麻烦。不过在 MSSQLServer 和 DB2 中提供了 DIFFERENCE()用来简化两个字符串的发音相似度比较，它可以计算两个字符串的发音特征值，并且比较它们，然后返回一个 0 至 4 之间的一个值来反映两个字符串的发音相似度，这个值越大则表示两个字符串发音思想度越大。

下面的 SQL 语句用来计算每个人的姓名发音与“Merry”的相似度：

```
SELECT DIFFERENCE(FName,'Merry')  FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName		
Tom	Merry	2
Jim	Merry	1
Lily	Merry	2
Kelly	Merry	3
Sam	Merry	2
Kerry	Merry	3
Smith	Merry	0
BillGates	Merry	1

可以看到 Kerry、Kelly 与 Merry 的发音相似度非常高。

在 WHERE 语句中使用 DIFFERENCE()更有意义，比如下面的 SQL 语句用于查询和“Tim”发音相似度大于 3 的员工：

```
SELECT * FROM T_Person
WHERE DIFFERENCE(FName,'Tim')>=3
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FIdNumber	FName	FBirthDay	FRegDay	FWeight
123456789120	Tom	1981-03-22 00:00:00.0	1998-05-01 00:00:00.0	56.67
123456789121	Jim	1987-01-18 00:00:00.0	1999-08-21 00:00:00.0	36.17

5.3 日期时间函数

日期时间类型的数据也是经常用到的，比如员工出生日期、结账日期、入库日期等等，而且经常需要对这些数据进行处理，比如检索所有超过保质期的商品、将结账日期向后延迟 3 天、检索所有每个月 18 日的入库记录，进行这些处理就需要使用日期时间函数。SQL 中提供了丰富的日期时间函数用于完成这些功能，本节将对这些日期时间函数进行详细讲解。

5.3.1 日期、时间、日期时间与时间戳

根据表示的类型、精度的不同，数据库中的日期时间数据类型分为日期、时间、日期时间以及时间戳四种类型。

日期类型是用来表示“年-月-日”信息的数据类型，其精度精确到“日”，其中包含了

年、月、日三个信息，比如“2008-08-08”。日期类型可以用来表示“北京奥运会开幕式日期”、“王小明的出生年月日”等信息，但是无法表示“最近一次迟到的时间”、“徐总抵京时间”等精确到小时甚至分秒的数据。在数据库中，一般用 Date 来表示日期类型。

时间类型是用来表示“小时:分:秒” 信息的数据类型，其精度精确到“秒”，其中包含了小时、分、秒三个信息，比如“19:00:00”。时间类型可以用来表示“每天《新闻联播》的播出时间”、“每天的下班时间”等信息，但是无法表示“卢沟桥事变爆发日期”、“上次结账时间”等包含“年-月-日”等信息的数据。在数据库中，一般用 Time 来表示时间类型。

日期时间类型是用来表示“年-月-日 小时:分:秒” 信息的数据类型，其精度精确到“秒”，其中包含了年、月、日、小时、分、秒六个信息，比如“2008-08-08 08:00:00”。日期时间类型可以用来表示“北京奥运会开幕式准确时间”、“上次迟到时间”等信息。在数据库中，一般用 DateTime 来表示日期时间类型。

日期时间类型的精度精确到“秒”，这在一些情况下能够满足基本的要求，但是对于精度要求更加高的日期时间信息则无法表示，比如“刘翔跑到终点的时间”、“货物 A 经过射频识别器的时间”等更高精度要求的信息。数据库中提供了时间戳类型用于表示这些对精度要求更加高的场合。时间戳类型还可以用于标记表中数据的版本信息，比如我们想区分表中两条记录插入表中的先后顺序，由于数据库操作速度非常快，如果用 DateTime 类型记录输入插入时间的话，若两条记录插入的时间间隔非常短的话是无法区分它们的，这时就可以使用时间戳类型。在有的数据库系统中，如果对数据表中的记录进行了更新的话，数据库系统会自动更新其中的时间戳字段的值。数据库中，一般用 TimeStamp 来表示日期时间类型。

不同的数据库系统对日期、时间、日期时间与时间戳等数据类型的支持差异性非常大，有的数据类型在有的数据库系统中不被支持，而有的数据类型的表示精度则和其类型名称所暗示的精度不同，比如 MSSQLServer 中不支持 Time 类型、Oracle 中的 Date 类型中包含时间信息。数据库中的日期时间函数对这些类型的支持差别是非常小的，因此在一般情况下我们将这些类型统一称为“日期时间类型”。

5.3.2 主流数据库系统中日期时间类型的表示方式

在 MYSQL、MSSQLServer 和 DB2 中可以用字符串来表示日期时间类型，数据库系统会自动在内部将它们转换为日期时间类型，比如“'2008-08-08'”、“2008-08-08 08:00:00”、“08:00:00” 、“2008-08-08 08:00:00.000000”等。

在 Oracle 中以字符串表示的数据是不能自动转换为日期时间类型的，必须使用 TO_DATE()函数来手动将字符串转换为日期时间类型的，比如 TO_DATE('2008-08-08', 'YYYY-MM-DD HH24:MI:SS') 、 TO_DATE('2008-08-08 08:00:00', 'YYYY-MM-DD HH24:MI:SS')、TO_DATE('08:00:00', 'YYYY-MM-DD HH24:MI:SS')等。

5.3.3 取得当前日期时间

在系统中经常需要使用当前日期时间进行处理，比如将“入库时间”字段设定为当前日期时间，在 SQL 中提供了取得当前日期时间的方式，不过各个数据库中的实现方式各不相同。

5.3.3.1 MYSQL

MYSQL 中提供了 NOW()函数用于取得当前的日期时间，NOW()函数还有 SYSDATE()、CURRENT_TIMESTAMP 等别名。如下：

SELECT NOW(),SYSDATE(),CURRENT_TIMESTAMP

执行完毕我们就能在输出结果中看到下面的执行结果：

NOW()	SYSDATE()	CURRENT_TIMESTAMP
2008-01-12 01:13:19	2008-01-12 01:13:19	2008-01-12 01:13:19

如果想得到不包括时间部分的当前日期，则可以使用 **CURDATE()**函数，**CURDATE()**

函数还有 **CURRENT_DATE** 等别名。如下：

```
SELECT CURDATE(),CURRENT_DATE
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CURDATE()	CURRENT_DATE
2008-01-12	2008-01-12

如果想得到不包括日期部分的当前时间，则可以使用 **CURTIME()**函数，**CURTIME ()** 函数还有 **CURRENT_TIME** 等别名。如下：

```
SELECT CURTIME(),CURRENT_TIME
```

执行完毕我们就能在输出结果中看到下面的执行结果：

CURTIME()	CURRENT_TIME
01:17:09	01:17:09

5.3.3.2 MSQLServer

MSSQLServer 中用于取得当前日期时间的函数为 GETDATE()。如下：

```
SELECT GETDATE() as 当前日期时间
```

执行完毕我们就能在输出结果中看到下面的执行结果：

当前日期时间
2008-01-12 01:02:04.78

可以看到 GETDATE()返回的信息是包括了日期、时间（精确到秒以后部分）的时间戳信息。MSSQLServer 没有专门提供取得当前日期、取得当前时间的函数，不过我们可以将 GETDATE()的返回值进行处理，这里需要借助于 Convert()函数，这个函数的详细介绍后面章节介绍，这里只介绍它在日期处理方面的应用。

使用 **CONVERT(VARCHAR(50) ,**日期时间值**, 101)**可以得到日期时间值的日期部分，因此下面的 SQL 语句可以得到当前的日期值：

```
SELECT CONVERT(VARCHAR(50) ,GETDATE( ), 101) as 当前日期
```

执行完毕我们就能在输出结果中看到下面的执行结果：

当前日期
01/14/2008

使用 **CONVERT(VARCHAR(50) ,**日期时间值**, 108)**可以得到日期时间值的日期部分，因此下面的 SQL 语句可以得到当前的日期值：

```
SELECT CONVERT(VARCHAR(50) ,GETDATE(), 108) as 当前时间
```

执行完毕我们就能在输出结果中看到下面的执行结果：

当前时间
21:37:19

5.3.3.3 Oracle

Oracle 中没有提供取得当前日期时间的函数，不过我们可以到系统表 DUAL 中查询 SYSTIMESTAMP 的值来得到当前的时间戳。如下：

```
SELECT SYSTIMESTAMP
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

SYSTIMESTAMP
2008-1-14 21.46.42.78000000 8:0

同样，我们可以到系统表 DUAL 中查询 SYSDATE 的值来得到当前日期时间。如下：

```
SELECT SYSDATE
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

SYSDATE
2008-01-14 21:47:16.0

同样，Oracle 中也没有专门提供取得当前日期、取得当前时间的函数，不过我们可以将 SYSDATE 的值进行处理，这里需要借助于 TO_CHAR()函数，这个函数的详细介绍后面章节介绍，这里只介绍它在日期处理方面的应用。

使用 TO_CHAR(时间日期值, 'YYYY-MM-DD') 可以得到日期时间值的日期部分，因此下面的 SQL 语句可以得到当前的日期值：

```
SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD')
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

TO_CHAR(SYSDATE,YYYY-MM-DD)
2008-01-14

使用 TO_CHAR(时间日期值, 'HH24:MI:SS') 可以得到日期时间值的时间部分，因此下面的 SQL 语句可以得到当前的日期值：

```
SELECT TO_CHAR(SYSDATE, 'HH24:MI:SS')
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

TO_CHAR(SYSDATE,HH24:MI:SS)
21:56:13

5.3.3.4 DB2

DB2 中同样没有提供取得当前日期时间的函数，不过我们可以到系统表 SYSIBM.SYSDUMMY1 中查询 CURRENT TIMESTAMP 的值来得到当前时间戳。如下：

```
SELECT CURRENT TIMESTAMP
FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
2008-01-14-21.58.20.01515000

从系统表 SYSIBM.SYSDUMMY1 中查询 CURRENT DATE 的值来得到当前日期值。如下：

```
SELECT CURRENT DATE
FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
2008-01-14

从系统表 SYSIBM.SYSDUMMY1 中查询 CURRENT **TIME** 的值来得到当前日期值。如下：

```
SELECT CURRENT TIME
FROM SYSIBM.SYSDUMMY1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1
22:05:48

5.3.4 日期增减

有时我们需要在一个日期的基础上增加某个时间长度或者减去某个时间长度，比如我们

知道每个员工的出生日期，而想计算出他出生后 10000 天的日期，再如我们想计算所有合同的到期日的三月后的日期。由于存在每个月天数不同、闰月等复杂的历法规则，所以不能使用简单的数字加减法进行计算，主流的数据库系统中都提供了对日期增减的计算，下面分别进行介绍。

5.3.4.1 MYSQL

MYSQL中提供了DATE_ADD()函数用于进行日期时间的加法运算，这个函数还有一个别名为ADDDATE()，DATE_ADD()函数的参数格式如下：

DATE_ADD (*date*,INTERVAL *expr type*)

其中参数date为待计算的日期；参数expr为待进行加法运算的增量，它可以是数值类型或者字符串类型，取决于type参数的取值；参数type则为进行加法运算的单位，type参数可选值以及对应的expr参数的格式如下表：

type参数值	expr参数的格式	说明
MICROSECOND	数值类型	以微秒为计算单位
SECOND	数值类型	以秒为计算单位
MINUTE	数值类型	以分钟为计算单位
HOUR	数值类型	以小时为计算单位
DAY	数值类型	以天为计算单位
WEEK	数值类型	以周为计算单位
MONTH	数值类型	以月为计算单位
QUARTER	数值类型	以季度为计算单位
YEAR	数值类型	以年为计算单位
SECOND_MICROSECOND	字符串类型，格式为：**'SECONDS.MICROSECONDS'**	以秒、微秒为计算单位，要求expr参数必须是“秒.微秒”的格式，比如“30.10”表示增加30秒10微秒。
MINUTE_MICROSECOND	字符串类型，格式为：**'MINUTES.MICROSECONDS'**	以分钟、毫秒为计算单位，要求expr参数必须是“分钟.微秒”的格式，比如“30.10”表示增加30分钟10微秒。
MINUTE_SECOND	字符串类型，格式为：**'MINUTES:SECONDS'**	以分钟、秒为计算单位，要求expr参数必须是“分钟:秒”的格式，比如“30:10”表示增加30分钟10秒。
HOUR_MICROSECOND	字符串类型，格式为：**'HOURS.MICROSECONDS'**	以小时、微秒为计算单位，要求expr参数必须是“小时.微秒”的格式，比如“30.10”表示增加30小时10微秒。
HOUR_SECOND	字符串类型，格式为：**'HOURS:MINUTES:SECONDS'**	以小时、分钟、秒为计算单位，要求expr参数必须是“小时:分钟:秒”的格式，比如“1:30:10”表示增加1小时30分钟10秒。
HOUR_MINUTE	字符串类型，格式为：**'HOURS:MINUTES'**	以小时、秒为计算单位，要求expr参数必须是“小时:秒”的格式，比如“30:10”表示增加30小时10秒。
DAY_MICROSECOND	字符串类型，格式为：**'DAYS.MICROSECONDS'**	以天、微秒为计算单位，要求expr参数必须是“天.微秒”的格式，比如

		“30.10”表示增加30天10微秒。
DAY_SECOND	字符串类型，格式为：'DAYS HOURS:MINUTES:SECONDS'	以天、小时、分钟、秒为计算单位，要求expr参数必须是“天 小时:分钟:秒”的格式，比如“1 3:28:36”表示增加1天3小时28分钟36秒。
DAY_MINUTE	字符串类型，格式为：'DAYS HOURS:MINUTES'	以天、小时、分钟为计算单位，要求expr参数必须是“天 小时:分钟”的格式，比如“1 3:15”表示增加1天3小时15分钟。
DAY_HOUR	字符串类型，格式为：'DAYS HOURS'	以天、小时为计算单位，要求expr参数必须是“天 小时”的格式，比如“30 10”表示增加30天10小时。
YEAR_MONTH	字符串类型，格式为：'YEARS-MONTHS'	以年、月为计算单位，要求expr参数必须是“年-月”的格式，比如“2-8”表示增加2年8个月。

表中前九种用法都非常简单，比如DATE_ADD(date,INTERVAL 1 HOUR)就可以得到在日期date基础上增加一小时后的日期时间，而DATE_ADD(date,INTERVAL 1 WEEK)就可以得到在日期date基础上增加一周后的日期时间。下面的SQL语句用来计算每个人出生一周、两个月以及5个季度后的日期：

```
SELECT FBirthDay,
DATE_ADD(FBirthDay,INTERVAL 1 WEEK) as w1,
DATE_ADD(FBirthDay,INTERVAL 2 MONTH) as m2,
DATE_ADD(FBirthDay,INTERVAL 5 QUARTER) as q5
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	w1	m2	q5
1981-03-22 00:00:00	1981-03-29 00:00:00	1981-05-22 00:00:00	1982-06-22 00:00:00
1987-01-18 00:00:00	1987-01-25 00:00:00	1987-03-18 00:00:00	1988-04-18 00:00:00
1987-11-08 00:00:00	1987-11-15 00:00:00	1988-01-08 00:00:00	1989-02-08 00:00:00
1982-07-12 00:00:00	1982-07-19 00:00:00	1982-09-12 00:00:00	1983-10-12 00:00:00
1983-02-16 00:00:00	1983-02-23 00:00:00	1983-04-16 00:00:00	1984-05-16 00:00:00
1984-08-07 00:00:00	1984-08-14 00:00:00	1984-10-07 00:00:00	1985-11-07 00:00:00
1980-01-09 00:00:00	1980-01-16 00:00:00	1980-03-09 00:00:00	1981-04-09 00:00:00
1972-07-18 00:00:00	1972-07-25 00:00:00	1972-09-18 00:00:00	1973-10-18 00:00:00

相对于前九种用法来说，后面几种用法就相对复杂一些，需要根据格式化的类型来决定expr参数的值。比如如果想为日期增加3天2小时10分钟，那么就可以如下使用DATE_ADD()函数：

```
DATE_ADD(date,INTERVAL '3 2:10' DAY_MINUTE)
```

比如下面的SQL语句分别计算出生日期后3天2小时10分钟、1年6个月的日期时间：

```
SELECT FBirthDay,
DATE_ADD(FBirthDay,INTERVAL '3 2:10' DAY_MINUTE) as dm,
DATE_ADD(FBirthDay,INTERVAL '1-6' YEAR_MONTH) as ym
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	dm	ym
1981-03-22 00:00:00	1981-03-25 02:10:00	1982-09-22 00:00:00
1987-01-18 00:00:00	1987-01-21 02:10:00	1988-07-18 00:00:00
1987-11-08 00:00:00	1987-11-11 02:10:00	1989-05-08 00:00:00
1982-07-12 00:00:00	1982-07-15 02:10:00	1984-01-12 00:00:00
1983-02-16 00:00:00	1983-02-19 02:10:00	1984-08-16 00:00:00
1984-08-07 00:00:00	1984-08-10 02:10:00	1986-02-07 00:00:00
1980-01-09 00:00:00	1980-01-12 02:10:00	1981-07-09 00:00:00
1972-07-18 00:00:00	1972-07-21 02:10:00	1974-01-18 00:00:00

几乎所有版本的MYSQL都支持DATE_ADD()函数的前九种用法，但是MYSQL的早期版本不完全支持DATE_ADD()函数的后几种用法，不过在MYSQL的早期版本中可以嵌套调用DATE_ADD()函数来实现后几种用法的效果。下面的SQL语句使用嵌套函数的方式来分别计算出生日期后1年6个月的日期时间：

```
SELECT FBirthDay,
DATE_ADD(DATE_ADD(FBirthDay,INTERVAL 1 YEAR),INTERVAL 6 MONTH) as dm
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	dm
1981-03-22 00:00:00	1982-09-22 00:00:00
1987-01-18 00:00:00	1988-07-18 00:00:00
1987-11-08 00:00:00	1989-05-08 00:00:00
1982-07-12 00:00:00	1984-01-12 00:00:00
1983-02-16 00:00:00	1984-08-16 00:00:00
1984-08-07 00:00:00	1986-02-07 00:00:00
1980-01-09 00:00:00	1981-07-09 00:00:00
1972-07-18 00:00:00	1974-01-18 00:00:00

DATE_ADD()函数不仅可以用来在日期基础上增加指定的时间段，而且还可以在日期基础上减少指定的时间段，只要在expr参数中使用负数就可以，下面的SQL语句用来计算每个人出生一周、两个月以及5个季度前的日期：

```
SELECT FBirthDay,
DATE_ADD(FBirthDay,INTERVAL -1 WEEK) as w1,
DATE_ADD(FBirthDay,INTERVAL -2 MONTH) as m2,
DATE_ADD(FBirthDay,INTERVAL -5 QUARTER) as q5
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	w1	m2	q5
1981-03-22 00:00:00	1981-03-15 00:00:00	1981-01-22 00:00:00	1979-12-22 00:00:00
1987-01-18 00:00:00	1987-01-11 00:00:00	1986-11-18 00:00:00	1985-10-18 00:00:00
1987-11-08 00:00:00	1987-11-01 00:00:00	1987-09-08 00:00:00	1986-08-08 00:00:00
1982-07-12 00:00:00	1982-07-05 00:00:00	1982-05-12 00:00:00	1981-04-12 00:00:00
1983-02-16 00:00:00	1983-02-09 00:00:00	1982-12-16 00:00:00	1981-11-16 00:00:00
1984-08-07 00:00:00	1984-07-31 00:00:00	1984-06-07 00:00:00	1983-05-07 00:00:00
1980-01-09 00:00:00	1980-01-02 00:00:00	1979-11-09 00:00:00	1978-10-09 00:00:00

1972-07-18 00:00:00	1972-07-11 00:00:00	1972-05-18 00:00:00	1971-04-18 00:00:00

在MYSQL中提供了**DATE_SUB()**函数用于计算指定日期前的特定时间段的日期，其效果和在DATE_ADD()函数中使用负数的expr参数值的效果一样，其用法也和DATE_ADD()函数几乎相同。下面的SQL语句用来计算每个人出生一周、两个月以及3天2小时10分钟前的日期：

```
SELECT FBirthDay,
DATE_SUB(FBirthDay,INTERVAL 1 WEEK) as w1,
DATE_SUB(FBirthDay,INTERVAL 2 MONTH) as m2,
DATE_SUB(FBirthDay, INTERVAL '3 2:10' DAY_MINUTE) as dm
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	w1	m2	dm
1981-03-22 00:00:00	1981-03-15 00:00:00	1981-01-22 00:00:00	1981-03-18 21:50:00
1987-01-18 00:00:00	1987-01-11 00:00:00	1986-11-18 00:00:00	1987-01-14 21:50:00
1987-11-08 00:00:00	1987-11-01 00:00:00	1987-09-08 00:00:00	1987-11-04 21:50:00
1982-07-12 00:00:00	1982-07-05 00:00:00	1982-05-12 00:00:00	1982-07-08 21:50:00
1983-02-16 00:00:00	1983-02-09 00:00:00	1982-12-16 00:00:00	1983-02-12 21:50:00
1984-08-07 00:00:00	1984-07-31 00:00:00	1984-06-07 00:00:00	1984-08-03 21:50:00
1980-01-09 00:00:00	1980-01-02 00:00:00	1979-11-09 00:00:00	1980-01-05 21:50:00
1972-07-18 00:00:00	1972-07-11 00:00:00	1972-05-18 00:00:00	1972-07-14 21:50:00

5.3.4.2 MSSQLServer

MSSQLServer中提供了DATEADD()函数用于进行日期时间的加法运算， DATEADD () 函数的参数格式如下：

DATEADD (datepart , number, date)

其中参数date为待计算的日期；参数date制定了用于与 datepart 相加的值，如果指定了非整数值，则将舍弃该值的小数部分；参数datepart指定要返回新值的日期的组成部分，下表列出了 Microsoft SQL Server 2005 可识别的日期部分及其缩写：

取值	别名	说明
year	yy,yyyy	年份
quarter	qq,q	季度
month	mm,m	月份
dayofyear	dy,y	当年度的第几天
day	dd,d	日
week	wk,ww	当年度的第几周
weekday	dw,w	星期几
hour	hh	小时
minute	mi,n	分
second	ss,s	秒
millisecond	ms	毫秒

比如 DATEADD(DAY, 3,date) 为计算日期 date 的 3 天后的日期，而 DATEADD(**MONTH** ,-8,date)为计算日期 date 的 8 个月之前的日期。

下面的 SQL 语句用于计算每个人出生后 3 年、20 个季度、68 个月以及 1000 个周前的日期：

```
SELECT FBirthDay, DATEADD (YEAR ,3,FBirthDay) as threeyrs,
```

```
DATEADD(QUARTER ,20,FBirthDay) as ttqutrs,
DATEADD(MONTH ,68,FBirthDay) as sxtmonths,
DATEADD(WEEK, -1000,FBirthDay) as thweeik
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	threeyrs	ttqutrs	sxtmonths	thweeik
1981-03-22 00:00:00.0	1984-03-22 00:00:00.0	1986-03-22 00:00:00.0	1986-11-22 00:00:00.0	1962-01-21 00:00:00.0
1987-01-18 00:00:00.0	1990-01-18 00:00:00.0	1992-01-18 00:00:00.0	1992-09-18 00:00:00.0	1967-11-19 00:00:00.0
1987-11-08 00:00:00.0	1990-11-08 00:00:00.0	1992-11-08 00:00:00.0	1993-07-08 00:00:00.0	1968-09-08 00:00:00.0
1982-07-12 00:00:00.0	1985-07-12 00:00:00.0	1987-07-12 00:00:00.0	1988-03-12 00:00:00.0	1963-05-13 00:00:00.0
1983-02-16 00:00:00.0	1986-02-16 00:00:00.0	1988-02-16 00:00:00.0	1988-10-16 00:00:00.0	1963-12-18 00:00:00.0
1984-08-07 00:00:00.0	1987-08-07 00:00:00.0	1989-08-07 00:00:00.0	1990-04-07 00:00:00.0	1965-06-08 00:00:00.0
1980-01-09 00:00:00.0	1983-01-09 00:00:00.0	1985-01-09 00:00:00.0	1985-09-09 00:00:00.0	1960-11-09 00:00:00.0
1972-07-18 00:00:00.0	1975-07-18 00:00:00.0	1977-07-18 00:00:00.0	1978-03-18 00:00:00.0	1953-05-19 00:00:00.0

5.3.4.3 Oracle

Oracle中可以直接使用加号“+”来进行日期的加法运算，其计算单位为“天”，比如date+3就表示在日期date的基础上增加三天；同理使用减号“-”则可以用来计算日期前的特定时间段的时间，比如date+3就表示在日期date的三天前的日期。比如下面的SQL语句用于计算每个人出生日期3天后以及10天前的日期：

```
SELECT FBirthDay,
FBirthDay+3,
FBirthDay-10
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FBIRTHDAY+3	FBIRTHDAY-10
1981-03-22 00:00:00.0	1981-03-25 00:00:00.0	1981-03-12 00:00:00.0
1987-01-18 00:00:00.0	1987-01-21 00:00:00.0	1987-01-08 00:00:00.0
1987-11-08 00:00:00.0	1987-11-11 00:00:00.0	1987-10-29 00:00:00.0
1982-07-12 00:00:00.0	1982-07-15 00:00:00.0	1982-07-02 00:00:00.0
1983-02-16 00:00:00.0	1983-02-19 00:00:00.0	1983-02-06 00:00:00.0
1984-08-07 00:00:00.0	1984-08-10 00:00:00.0	1984-07-28 00:00:00.0
1980-01-09 00:00:00.0	1980-01-12 00:00:00.0	1979-12-30 00:00:00.0
1972-07-18 00:00:00.0	1972-07-21 00:00:00.0	1972-07-08 00:00:00.0

可以使用换算的方式来进行以周、小时、分钟等为单位的日期加减运算，比如下面的SQL语句用于计算每个人出生日期2小时10分钟后以及3周后的日期：

```
SELECT FBirthDay,
```

```
FBirthDay+(2/24+10/60/24),
FBirthDay+(3*7)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FBIRTHDAY+(2/24+10/60/24)	FBIRTHDAY+(3*7)
1981-03-22 00:00:00.0	1981-03-22 02:10:00.0	1981-04-12 00:00:00.0
1987-01-18 00:00:00.0	1987-01-18 02:10:00.0	1987-02-08 00:00:00.0
1987-11-08 00:00:00.0	1987-11-08 02:10:00.0	1987-11-29 00:00:00.0
1982-07-12 00:00:00.0	1982-07-12 02:10:00.0	1982-08-02 00:00:00.0
1983-02-16 00:00:00.0	1983-02-16 02:10:00.0	1983-03-09 00:00:00.0
1984-08-07 00:00:00.0	1984-08-07 02:10:00.0	1984-08-28 00:00:00.0
1980-01-09 00:00:00.0	1980-01-09 02:10:00.0	1980-01-30 00:00:00.0
1972-07-18 00:00:00.0	1972-07-18 02:10:00.0	1972-08-08 00:00:00.0

使用加减运算我们可以很容易的实现以周、天、小时、分钟、秒等为单位的日期的增减运算，不过由于每个月的天数是不同的，也就是在天和月之间不存在固定的换算率，所以无法使用加减运算实现以月为单位的计算，为此Oracle中提供了ADD_MONTHS()函数用于以月为单位的日期增减运算，ADD_MONTHS()函数的参数格式如下：

```
ADD_MONTHS(date,number)
```

其中参数date为待计算的日期，参数number为要增加的月份数，如果number为负数则表示进行日期的减运算。下面的SQL语句用于计算每个人的出生日期两个月后以及10个月前的日期：

```
SELECT FBirthDay,
ADD_MONTHS(FBirthDay,2),
ADD_MONTHS(FBirthDay,-10)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	ADD_MONTHS(FBIRTHDAY,2)	ADD_MONTHS(FBIRTHDAY,-10)
1981-03-22 00:00:00.0	1981-05-22 00:00:00.0	1980-05-22 00:00:00.0
1987-01-18 00:00:00.0	1987-03-18 00:00:00.0	1986-03-18 00:00:00.0
1987-11-08 00:00:00.0	1988-01-08 00:00:00.0	1987-01-08 00:00:00.0
1982-07-12 00:00:00.0	1982-09-12 00:00:00.0	1981-09-12 00:00:00.0
1983-02-16 00:00:00.0	1983-04-16 00:00:00.0	1982-04-16 00:00:00.0
1984-08-07 00:00:00.0	1984-10-07 00:00:00.0	1983-10-07 00:00:00.0
1980-01-09 00:00:00.0	1980-03-09 00:00:00.0	1979-03-09 00:00:00.0
1972-07-18 00:00:00.0	1972-09-18 00:00:00.0	1971-09-18 00:00:00.0

综合使用ADD_MONTHS()函数和加、减号运算符则可以实现更加复杂的日期增减运算，比

如下面的SQL语句用于计算每个人的出生日期两个月零10天后以及3个月零10个小时前的日期时间：

```
SELECT FBirthDay,
ADD_MONTHS(FBirthDay,2)+10 as bfd,
ADD_MONTHS(FBirthDay,-3)-(10/24) as afd
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	BFD	AFD
1981-03-22 00:00:00.0	1981-06-01 00:00:00.0	1980-12-21 14:00:00.0
1987-01-18 00:00:00.0	1987-03-28 00:00:00.0	1986-10-17 14:00:00.0
1987-11-08 00:00:00.0	1988-01-18 00:00:00.0	1987-08-07 14:00:00.0
1982-07-12 00:00:00.0	1982-09-22 00:00:00.0	1982-04-11 14:00:00.0
1983-02-16 00:00:00.0	1983-04-26 00:00:00.0	1982-11-15 14:00:00.0
1984-08-07 00:00:00.0	1984-10-17 00:00:00.0	1984-05-06 14:00:00.0
1980-01-09 00:00:00.0	1980-03-19 00:00:00.0	1979-10-08 14:00:00.0
1972-07-18 00:00:00.0	1972-09-28 00:00:00.0	1972-04-17 14:00:00.0

5.3.4.4 DB2

DB2 中可以直接使用加减运算符进行日期的增减运算，只要在要增减的数目后加上单位就可以了。其使用格式如下：

date+length unit

其中 date 参数为待计算的日期；

length 为进行增减运算的日期，当 length 为正值的时候为向时间轴正向计算，而当 length 为负值的时候为向时间轴负向计算；

unit 为进行计算的单位，此参数可取值以及响应函数如下：

计算单位	说明
YEAR	年
MONTH	月
DAY	日
HOUR	小时
MINUTE	分
SECOND	秒

比如 date+3 DAY 为计算日期 date 的 3 天后的日期，而 date-8 MONTH 为计算日期 date 的 8 个月之前的日期，而且我们还可以连续使用加减运算符进行更加复杂的日期运算，比如 date+3 YEAR+10 DAY 用于计算 date 的 3 个月零 10 天后的日期。

下面的 SQL 语句用于计算每个人出生后 3 年、20 个季度、68 个月以及 1000 个周前的日期：

```
SELECT FBirthDay, FBirthDay+3 YEAR + 10 DAY,
FBirthDay-100 MONTH
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	2	3
1981-03-22	1984-04-01	1972-11-22
1987-01-18	1990-01-28	1978-09-18
1987-11-08	1990-11-18	1979-07-08

1982-07-12	1985-07-22	1974-03-12
1983-02-16	1986-02-26	1974-10-16
1984-08-07	1987-08-17	1976-04-07
1980-01-09	1983-01-19	1971-09-09
1972-07-18	1975-07-28	1964-03-18

5.3.5 计算日期差额

有时候我们需要计算两个日期的差额，比如计算“回款日”和“验收日”之间所差的天数或者检索所有“最后一次登录日期”与当前日期的差额大于100天的用户信息。主流的数据库系统中都提供了对计算日期差额的支持，下面分别进行介绍。

5.3.5.1 MYSQL

MYSQL中使用DATEDIFF()函数用于计算两个日期之间的差额，其参数调用格式如下：

DATEDIFF(date1,date2)

函数将返回date1与date2之间的天数差额，如果date2在date1之后返回正值，否则返回负值。

比如下面的SQL语句用于计算注册日期和出生日期之间的天数差额：

```
SELECT FRegDay,FBirthDay, DATEDIFF(FRegDay, FBirthDay) ,
DATEDIFF(FBirthDay ,FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FRegDay	FBirthDay	DATEDIFF(FRegDay, FBirthDay)	DATEDIFF(FBirthDay ,FRegDay)
1998-05-01 00:00:00	1981-03-22 00:00:00	6249	-6249
1999-08-21 00:00:00	1987-01-18 00:00:00	4598	-4598
2001-09-18 00:00:00	1987-11-08 00:00:00	5063	-5063
2000-03-01 00:00:00	1982-07-12 00:00:00	6442	-6442
1998-05-01 00:00:00	1983-02-16 00:00:00	5553	-5553
1999-03-01 00:00:00	1984-08-07 00:00:00	5319	-5319
2002-09-23 00:00:00	1980-01-09 00:00:00	8293	-8293
1995-06-19 00:00:00	1972-07-18 00:00:00	8371	-8371

DATEDIFF()函数只能计算两个日期之间的天数差额，如果要计算两个日期的周差额等就需要进行换算，比如下面的SQL语句用于计算注册日期和出生日期之间的周数差额：

```
SELECT FRegDay,FBirthDay, DATEDIFF(FRegDay, FBirthDay)/7
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FRegDay	FBirthDay	DATEDIFF(FRegDay, FBirthDay)/7

1998-05-01 00:00:00	1981-03-22 00:00:00	892.7143
1999-08-21 00:00:00	1987-01-18 00:00:00	656.8571
2001-09-18 00:00:00	1987-11-08 00:00:00	723.2857
2000-03-01 00:00:00	1982-07-12 00:00:00	920.2857
1998-05-01 00:00:00	1983-02-16 00:00:00	793.2857
1999-03-01 00:00:00	1984-08-07 00:00:00	759.8571
2002-09-23 00:00:00	1980-01-09 00:00:00	1184.7143
1995-06-19 00:00:00	1972-07-18 00:00:00	1195.8571

5.3.5.2 MSSQLServer

MSSQLServer中同样提供了DATEDIFF()函数用于计算两个日期之间的差额，与MYSQL中的DATEDIFF()函数不同，它提供了一个额外的参数用于指定计算差额时使用的单位，其参数调用格式如下：

DATEDIFF (datepart , startdate , enddate)

其中参数datepart为计算差额时使用的单位，可选值如下：

单位	别名	说明
year	yy, yyyy	年
quarter	qq, q	季度
month	mm, m	月
dayofyear	dy, y	工作日
day	dd, d	天数
week	wk, ww	周
Hour	hh	小时
minute	mi, n	分钟
second	ss, s	秒
millisecond	ms	毫秒

参数startdate为起始日期；参数enddate为结束日期。

下面的SQL语句用于计算注册日期和出生日期之间的周数差额：

SELECT FRegDay,FBirthDay,DATEDIFF(WEEK, FBirthDay, FRegDay) FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FRegDay	FBirthDay	
1998-05-01 00:00:00.0	1981-03-22 00:00:00.0	892
1999-08-21 00:00:00.0	1987-01-18 00:00:00.0	656
2001-09-18 00:00:00.0	1987-11-08 00:00:00.0	723
2000-03-01 00:00:00.0	1982-07-12 00:00:00.0	920
1998-05-01 00:00:00.0	1983-02-16 00:00:00.0	793
1999-03-01 00:00:00.0	1984-08-07 00:00:00.0	760
2002-09-23 00:00:00.0	1980-01-09 00:00:00.0	1185
1995-06-19 00:00:00.0	1972-07-18 00:00:00.0	1196

5.3.5.3 Oracle

在Oracle中，可以在两个日期类型的数据之间使用减号运算符“-”，其计算结果为两个日期之间的天数差，比如执行下面的SQL语句用于计算注册日期FRegDay和出生日期FBirthDay之间的时间间隔：

```
SELECT FRegDay,FBirthDay,FRegDay-FBirthDay
```

```
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FREGDAY	FBIRTHDAY	FREGDAY-FBIRTHDAY
1998-05-01 00:00:00.0	1981-03-22 00:00:00.0	6249
1999-08-21 00:00:00.0	1987-01-18 00:00:00.0	4598
2001-09-18 00:00:00.0	1987-11-08 00:00:00.0	5063
2000-03-01 00:00:00.0	1982-07-12 00:00:00.0	6442
1998-05-01 00:00:00.0	1983-02-16 00:00:00.0	5553
1999-03-01 00:00:00.0	1984-08-07 00:00:00.0	5319
2002-09-23 00:00:00.0	1980-01-09 00:00:00.0	8293
1995-06-19 00:00:00.0	1972-07-18 00:00:00.0	8371

注意通过减号运算符“-”计算的两个日期之间的天数差是包含有小数部分的，小数部分表示不足一天的部分，比如执行下面的SQL语句用于计算当前时刻和出生日期FBirthDay之间的时间间隔：

```
SELECT SYSDATE,FBirthDay,SYSDATE-FBirthDay
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

SYSDATE	FBIRTHDAY	SYSDATE-FBIRTHDAY
2008-01-16 23:11:52.0	1981-03-22 00:00:00.0	9796.9665740740740740740740740740740740
2008-01-16 23:11:52.0	1987-01-18 00:00:00.0	7668.9665740740740740740740740740740740
2008-01-16 23:11:52.0	1987-11-08 00:00:00.0	7374.9665740740740740740740740740740740
2008-01-16 23:11:52.0	1982-07-12 00:00:00.0	9319.9665740740740740740740740740740740
2008-01-16 23:11:52.0	1983-02-16 00:00:00.0	9100.9665740740740740740740740740740740
2008-01-16 23:11:52.0	1984-08-07 00:00:00.0	8562.9665740740740740740740740740740740
2008-01-16 23:11:52.0	1980-01-09 00:00:00.0	10234.966574074074074074074074074074
2008-01-16 23:11:52.0	1972-07-18 00:00:00.0	12965.966574074074074074074074074074

可以看到天数差的小数部分是非常精确的，所以完全可以精确的表示两个日期时间值之间差的小时、分、秒甚至毫秒部分。所以如果要计算两个日期时间值之间的小时、分、秒以及毫秒差的话，只要进行相应的换算就可以，比如下面的SQL用来计算当前时刻和出生日期FBirthDay之间的时间间隔（小时、分以及秒）：

```
SELECT (SYSDATE-FBirthDay)*24,(SYSDATE-FBirthDay)*24*60,
(SYSDATE-FBirthDay)*24*60*60
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

(SYSDATE-FBIRTHDAY)*24	(SYSDATE-FBIRTHDAY)*24 *60	(SYSDATE-FBIRTHDAY)*24 *60*60

235127.289166666666666666666666666666667	14107637.35	846458241
184055.289166666666666666666666666666667	11043317.35	662599041
176999.289166666666666666666666666666667	10619957.35	637197441
223679.289166666666666666666666666666667	13420757.35	805245441
218423.289166666666666666666666666666667	13105397.35	786323841
205511.289166666666666666666666666666667	12330677.35	739840641
245639.289166666666666666666666666666656	14738357.34999999999999999999999999999994	884301440.99999999999999999999999999999996
311183.289166666666666666666666666666656	18670997.34999999999999999999999999999994	1120259840.99999999999999999999999999999996

下面的SQL语句用来计算当前时刻和出生日期FBirthDay之间的周间隔：

```
SELECT SYSDATE,FBirthDay,(SYSDATE-FBirthDay)/7
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

SYSDATE	FBIRTHDAY	(SYSDATE-FBIRTHDAY)/7
2008-01-16 23:22:17.0	1981-03-22 00:00:00.0	1399.5676868386243386243386243386243386243386624
2008-01-16 23:22:17.0	1987-01-18 00:00:00.0	1095.5676868386243386243386243386243386243386624
2008-01-16 23:22:17.0	1987-11-08 00:00:00.0	1053.5676868386243386243386243386243386243386624
2008-01-16 23:22:17.0	1982-07-12 00:00:00.0	1331.4248296957671957671957671957671957671957 67
2008-01-16 23:22:17.0	1983-02-16 00:00:00.0	1300.139115410052910052910052910052910053
2008-01-16 23:22:17.0	1984-08-07 00:00:00.0	1223.281972552910052910052910052910052910 05291
2008-01-16 23:22:17.0	1980-01-09 00:00:00.0	1462.139115410052910052910052910052910057
2008-01-16 23:22:17.0	1972-07-18 00:00:00.0	1852.281972552910052910052910052910052914

可以看到计算结果含有非常精确的小数部分，不过如果对这些小数部分没有需求的话则可以使用数值函数进行四舍五入、取最大整数等处理，比如下面的SQL用来计算当前时刻和出生日期FBirthDay之间的时间间隔（小时、分以及秒），并且对于计算结果进行四舍五入运算：

```
SELECT
ROUND((SYSDATE-FBirthDay)*24),ROUND((SYSDATE-FBirthDay)*24*60),
ROUND((SYSDATE-FBirthDay)*24*60*60)
```

```
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ROUND((SYSDATE-FBIRTHDAY)*24)	ROUND((SYSDATE-FBIRTHDAY)*24*60)	ROUND((SYSDATE-FBIRTHDAY)*24*60*60)
235127	14107641	846458455
184055	11043321	662599255
176999	10619961	637197655
223679	13420761	805245655
218423	13105401	786324055
205511	12330681	739840855
245639	14738361	884301655
311183	18671001	1120260055

5.3.5.4 DB2

DB2中提供了DAYS()函数，这个函数接受一个时间日期类型的参数，返回结果为从0001年1月1日到此日期的天数，比如下面的SQL语句用于计算出生日期FBirthDay、注册日期FRegDay以及当前日期距0001年1月1日的天数差：

```
SELECT DAYS(FBirthDay),DAYS(FRegDay),DAYS(CURRENT DATE)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	2	3
723261	729510	733057
725389	729987	733057
725683	730746	733057
723738	730180	733057
723957	729510	733057
724495	729814	733057
722823	731116	733057
720092	728463	733057

借助于DAYS()函数我们可以轻松计算两个日期之间的天数间隔，很显然DAYS(date1)-DAYS(date2)的计算结果就是日期date1和日期date2之间的天数间隔，比如下面的SQL语句用于计算出生日期FBirthDay与注册日期FRegDay之间的天数间隔：

```
SELECT FBirthDay,FRegDay,
DAYS(FRegDay)-DAYS(FBirthDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FREGDAY	3
1981-03-22	1998-05-01	6249
1987-01-18	1999-08-21	4598
1987-11-08	2001-09-18	5063
1982-07-12	2000-03-01	6442
1983-02-16	1998-05-01	5553
1984-08-07	1999-03-01	5319
1980-01-09	2002-09-23	8293

1972-07-18	1995-06-19	8371

如果要计算两个日期时间值之间的周间隔的话，只要进行相应的换算就可以，比如下面的SQL用来计算出生日期FBirthDay和注册日期FRegDay之间的周数间隔：

```
SELECT FBirthDay,FRegDay,
(DAYS(FRegDay)-DAYS(FBirthDay))/7
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FREGDAY	3
1981-03-22	1998-05-01	892
1987-01-18	1999-08-21	656
1987-11-08	2001-09-18	723
1982-07-12	2000-03-01	920
1983-02-16	1998-05-01	793
1984-08-07	1999-03-01	759
1980-01-09	2002-09-23	1184
1972-07-18	1995-06-19	1195

5.3.6 计算一个日期是星期几

计算一个日期是星期几是非常有用的，比如如果安排的报到日期是周末那么就向后拖延报到日期，在主流数据库中对这个功能都提供了很好的支持，下面分别进行介绍。

5.3.6.1 MYSQL

MYSQL中提供了DAYNAME()函数用于计算一个日期是星期几，比如下面的SQL语句用于计算出生日期和注册日期各是星期几：

```
SELECT FBirthDay,DAYNAME(FBirthDay),
FRegDay,DAYNAME(FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	DAYNAME(FBirthDay)	FRegDay	DAYNAME(FRegDay)
1981-03-22 00:00:00	Sunday	1998-05-01 00:00:00	Friday
1987-01-18 00:00:00	Sunday	1999-08-21 00:00:00	Saturday
1987-11-08 00:00:00	Sunday	2001-09-18 00:00:00	Tuesday
1982-07-12 00:00:00	Monday	2000-03-01 00:00:00	Wednesday
1983-02-16 00:00:00	Wednesday	1998-05-01 00:00:00	Friday
1984-08-07 00:00:00	Tuesday	1999-03-01 00:00:00	Monday
1980-01-09 00:00:00	Wednesday	2002-09-23 00:00:00	Monday
1972-07-18 00:00:00	Tuesday	1995-06-19 00:00:00	Monday

注意MYSQL中DAYNAME()函数返回的是英文的日期表示法。

5.3.6.2 MSQLServer

MSQLServer中提供了DATENAME()函数，这个函数可以返回一个日期的特定部分，并且尽量用名称来表述这个特定部分，其参数格式如下：

DATENAME(datepart,date)

其中参数date为待计算日期，date 参数也可以是日期格式的字符串；参数datepart指定要返回的日期部分的参数，其可选值如下：

可选值	别名	说明

Year	yy、yyyy	年份
Quarter	qq, q	季度
Month	mm, m	月份
Dayofyear	dy, y	每年的某一日
Day	dd, d	日期
Week	wk, ww	星期
Weekday	dw	工作日
Hour	hh	小时
Minute	mi, n	分钟
Second	ss, s	秒
Millisecond	ms	毫秒

如果使用Weekday（或者使用别名dw）做为datepart参数调用DATENAME()函数就可以得到一个日期是星期几，比如下面的SQL语句用于计算出生日期和注册日期各是星期几：

```
SELECT FBirthDay,DATENAME(Weekday,FBirthDay),
FRegDay,DATENAME(DW, FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay		FRegDay	
1981-03-22 00:00:00.0	星期日	1998-05-01 00:00:00.0	星期五
1987-01-18 00:00:00.0	星期日	1999-08-21 00:00:00.0	星期六
1987-11-08 00:00:00.0	星期日	2001-09-18 00:00:00.0	星期二
1982-07-12 00:00:00.0	星期一	2000-03-01 00:00:00.0	星期三
1983-02-16 00:00:00.0	星期三	1998-05-01 00:00:00.0	星期五
1984-08-07 00:00:00.0	星期二	1999-03-01 00:00:00.0	星期一
1980-01-09 00:00:00.0	星期三	2002-09-23 00:00:00.0	星期一
1972-07-18 00:00:00.0	星期二	1995-06-19 00:00:00.0	星期一

5.3.6.3 Oracle

Oracle中提供了TO_CHAR()函数用于将数据转换为字符串类型，当针对时间日期类型数据进行转换的时候，它接受两个参数，其参数格式如下：

TO_CHAR(date,format)

其中参数date为待转换的日期，参数format为格式化字符串，数据库系统将按照这个字符串对date进行转换，格式化字符串中可以采用如下的占位符：

占位符	说明
YEAR	年份（英文拼写），比如NINETEEN NINETY-EIGHT
YYYY	4位年份，比如1998

YYY	年份后3位，比如998
YY	年份后2位，比如98
Y	年份后1位，比如8
IYYY	符合ISO标准的4位年份，比如1998
IYY	符合ISO标准的年份后3位，比如998
IY	符合ISO标准的年份后2位，比如98
I	符合ISO标准的年份后1位，比如8
Q	以整数表示的季度，比如1
MM	以整数表示的月份，比如01
MON	月份的名称，比如2月
MONTH	月份的名称，补足9个字符
RM	罗马表示法的月份，比如VIII
WW	日期属于当年的第几周，比如30
W	日期属于当月的第几周，比如2
IW	日期属于当年的第几周（按照ISO标准），比如30
D	日期属于周几，以整数表示，返回值范围为1至7
DAY	日期属于周几，以名字的形式表示，比如星期五
DD	日期属于当月的第几天，比如2
DDD	日期属于当年的第几天，比如168
DY	日期属于周几，以名字的形式表示，比如星期五
HH	小时部分（12小时制）
HH12	小时部分（12小时制）
HH24	小时部分（24小时制）
MI	分钟部分
SS	秒部分
SSSSS	自从午夜开始的秒数

可以简单的将占位符做为参数传递给TO_CHAR()函数，下面的SQL语句用于计算出生日期的年份、月份以及周数：

```
SELECT FBirthDay,
TO_CHAR(FBirthDay, 'YYYY') as yyyy,
TO_CHAR(FBirthDay, 'MM') as mm,
TO_CHAR(FBirthDay, 'MON') as mon,
TO_CHAR(FBirthDay, 'WW') as ww
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	YYYY	MM	MON	WW
1981-03-22 00:00:00.0	1981	03	3 月	12
1987-01-18 00:00:00.0	1987	01	1 月	03
1987-11-08 00:00:00.0	1987	11	11 月	45
1982-07-12	1982	07	7 月	28

00:00:00.0				
1983-02-16 00:00:00.0	1983	02	2 月	07
1984-08-07 00:00:00.0	1984	08	8 月	32
1980-01-09 00:00:00.0	1980	01	1 月	02
1972-07-18 00:00:00.0	1972	07	7 月	29

同样还可以将占位符组合起来实现更加复杂的转换逻辑，比如下面的SQL语句用于以“2008-08-08”这样的形式显示出生日期以及以“31-2007-02”这样的形式显示注册日期：

```
SELECT FBirthDay,
TO_CHAR(FBirthDay, 'YYYY-MM-DD') as yyymmdd,
FRegDay,
TO_CHAR(FRegDay, 'DD-YYYY-MM') as ddyyyymm
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	YYYMMDD	FREGDAY	DDYYYYMM
1981-03-22 00:00:00.0	1981-03-22	1998-05-01 00:00:00.0	01-1998-05
1987-01-18 00:00:00.0	1987-01-18	1999-08-21 00:00:00.0	21-1999-08
1987-11-08 00:00:00.0	1987-11-08	2001-09-18 00:00:00.0	18-2001-09
1982-07-12 00:00:00.0	1982-07-12	2000-03-01 00:00:00.0	01-2000-03
1983-02-16 00:00:00.0	1983-02-16	1998-05-01 00:00:00.0	01-1998-05
1984-08-07 00:00:00.0	1984-08-07	1999-03-01 00:00:00.0	01-1999-03
1980-01-09 00:00:00.0	1980-01-09	2002-09-23 00:00:00.0	23-2002-09
1972-07-18 00:00:00.0	1972-07-18	1995-06-19 00:00:00.0	19-1995-06

我们前面提到了，当用“DAY”做为参数的时候就可以将日期格式化为名字的形式表示的星期几，比如下面的SQL语句用于计算出生日期以及注册日期各属于星期几：

```
SELECT
FBirthDay,TO_CHAR(FBirthDay, 'DAY') as birthwk,
FRegDay,TO_CHAR(FRegDay, 'DAY') as regwk
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	BIRTHWK	FREGDAY	REGWK
1981-03-22 00:00:00.0	星期日	1998-05-01 00:00:00.0	星期五

1987-01-18 00:00:00.0	星期日	1999-08-21 00:00:00.0	星期六
1987-11-08 00:00:00.0	星期日	2001-09-18 00:00:00.0	星期二
1982-07-12 00:00:00.0	星期一	2000-03-01 00:00:00.0	星期三
1983-02-16 00:00:00.0	星期三	1998-05-01 00:00:00.0	星期五
1984-08-07 00:00:00.0	星期二	1999-03-01 00:00:00.0	星期一
1980-01-09 00:00:00.0	星期三	2002-09-23 00:00:00.0	星期一
1972-07-18 00:00:00.0	星期二	1995-06-19 00:00:00.0	星期一

5.3.6.4 DB2

DB2中提供了DAYNAME()函数用于计算一个日期是星期几，执行下面的SQL语句我们可以得到出生日期和注册日期各是星期几：

```
SELECT
FBirthDay,DAYNAME(FBirthDay) as birthwk,
FRegDay,DAYNAME(FRegDay) as regwk
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	BIRTHWK	FREGDAY	REGWK
1981-03-22	星期日	1998-05-01	星期五
1987-01-18	星期日	1999-08-21	星期六
1987-11-08	星期日	2001-09-18	星期二
1982-07-12	星期一	2000-03-01	星期三
1983-02-16	星期三	1998-05-01	星期五
1984-08-07	星期二	1999-03-01	星期一
1980-01-09	星期三	2002-09-23	星期一
1972-07-18	星期二	1995-06-19	星期一

5.3.7 取得日期的指定部分

提取日期的特定部分是非常有必要的，比如检索本年的每个月的16日的销售量、检索访问用户集中的时间段，这些都需要对日期的特定部分进行提取，在主流数据库中对这个功能都提供了很好的支持，下面分别进行介绍。

5.3.7.1 MYSQL

MYSQL中提供了一个DATE_FORMAT()函数用来将日期按照特定各是进行格式化，这个函数的参数格式如下：

DATE_FORMAT(date,format)

这个函数用来按照特定的格式化指定的日期，其中参数date为待计算的日期值，而参数format为格式化字符串，格式化字符串中可以采用如下的占位符：

占位符	说明
%a	缩写的星期几(Sun..Sat)
%b	缩写的月份名(Jan..Dec)

%c	数字形式的月份(0..12)
%D	当月的第几天，带英文后缀(0th, 1st, 2nd, 3rd, ...)
%d	当月的第几天，两位数字形式，不足两位则补零(00..31)
%e	当月的第几天，数字形式(0..31)
%f	毫秒
%H	24小时制的小时 (00..23)
%h	12小时制的小时(01..12)
%I	12小时制的小时(01..12)
%i	数字形式的分钟(00..59)
%j	日期在当年中的天数(001..366)
%k	24进制小时(0..23)
%l	12进制小时(1..12)
%M	月份名(January..December)
%m	两位数字表示的月份(00..12)
%p	上午还是下午（AM.. PM）
%r	12小时制时间，比如08:09:29 AM
%S	秒数(00..59)
%s	秒数(00..59)
%T	时间，24小时制，格式为hh:mm:ss
%U	所属周是当年的第几周，周日当作第一天(00..53)
%u	所属周是当年的第几周，周一当作第一天(00..53)
%V	所属周是当年的第几周，周日当作第一天(01..53)
%v	所属周是当年的第几周，周一当作第一天(01..53)
%W	星期几(Sunday..Saturday)
%w	星期几，数字形式(0=Sunday..6=Saturday)
%X	本周所属年，周日当作第一天
%x	本周所属年，周一当作第一天
%Y	年份数，四位数字
%y	年份数，两位数字

组合使用这些占位符就可以实现非常复杂的字符串格式化逻辑，比如下面的SQL语句实现了将出生日期FBirthDay和注册日期FRegDay分别按照两种格式进行格式化：

```
SELECT
FBirthDay,
DATE_FORMAT(FBirthDay,'%y-%M %D %W') AS bd,
FRegDay,
DATE_FORMAT(FRegDay,'%Y年%m月%e日') AS rd
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	bd	FRegDay	rd
1981-03-22 00:00:00	81-March 22nd Sunday	1998-05-01 00:00:00	1998 年 05 月 1 日
1987-01-18 00:00:00	87-January 18th Sunday	1999-08-21 00:00:00	1999 年 08 月 21 日

1987-11-08 00:00:00	87-November 8th Sunday	2001-09-18 00:00:00	2001 年 09 月 18 日
1982-07-12 00:00:00	82-July 12th Monday	2000-03-01 00:00:00	2000 年 03 月 1 日
1983-02-16 00:00:00	83-February 16th Wednesday	1998-05-01 00:00:00	1998 年 05 月 1 日
1984-08-07 00:00:00	84-August 7th Tuesday	1999-03-01 00:00:00	1999 年 03 月 1 日
1980-01-09 00:00:00	80-January 9th Wednesday	2002-09-23 00:00:00	2002 年 09 月 23 日
1972-07-18 00:00:00	72-July 18th Tuesday	1995-06-19 00:00:00	1995 年 06 月 19 日

很显然，如果只使用单独的占位符那么就可以实现提取日期特定部分的功能了，比如DATE_FORMAT(date,'%Y')可以用来提取日期的年份部分、DATE_FORMAT(date,'%H')可以用来提取日期的小时部分、DATE_FORMAT(date,'%M')可以用来提取日期的月份名称。下面的SQL用于提取每个人员的出生年份、出生时是当年的第几天、出生时是当年的第几周：

```
SELECT
FBirthDay,
DATE_FORMAT(FBirthDay,'%Y') AS y,
DATE_FORMAT(FBirthDay,'%j') AS d,
DATE_FORMAT(FBirthDay,'%U') AS u
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	y	d	u
1981-03-22 00:00:00	1981	081	12
1987-01-18 00:00:00	1987	018	03
1987-11-08 00:00:00	1987	312	45
1982-07-12 00:00:00	1982	193	28
1983-02-16 00:00:00	1983	047	07
1984-08-07 00:00:00	1984	220	32
1980-01-09 00:00:00	1980	009	01
1972-07-18 00:00:00	1972	200	29

5.3.7.2 MSSQLServer

在5.3.6.2一节中我们介绍了DATENAME()函数，使用它就可以提取日期的任意部分，比如下面的SQL用于提取每个人员的出生年份、出生时是当年的第几天、出生时是当年的第几周：

```
SELECT
FBirthDay,
DATENAME(year,FBirthDay) AS y,
DATENAME(dayofyear,FBirthDay) AS d,
DATENAME(week,FBirthDay) AS u
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	y	d	u
1981-03-22 00:00:00.0	1981	81	13
1987-01-18 00:00:00.0	1987	18	4

1987-11-08 00:00:00.0	1987	312	46
1982-07-12 00:00:00.0	1982	193	29
1983-02-16 00:00:00.0	1983	47	8
1984-08-07 00:00:00.0	1984	220	32
1980-01-09 00:00:00.0	1980	9	2
1972-07-18 00:00:00.0	1972	200	30

在MSSQLServer中还提供了一个DATEPART()函数，这个函数也可以用来返回一个日期的特定部分，其参数格式如下：

DATEPART (datepart,date)

其中参数date为待计算日期，date 参数也可以是日期格式的字符串；参数datepart指定要返回的日期部分的参数，其可选值如下：

可选值	别名	说明
Year	yy、yyyy	年份
Quarter	qq, q	季度
Month	mm, m	月份
Dayofyear	dy, y	每年的某一日
Day	dd, d	日期
Week	wk, ww	星期
Weekday	dw	工作日
Hour	hh	小时
Minute	mi, n	分钟
Second	ss, s	秒
Millisecond	ms	毫秒

显然使用Dayofyear做为datepart参数调用DATEPART ()函数就可以得到一个日期是当年的第几天；使用Year做为datepart参数调用DATEPART ()函数就可以得到一个日期的年份数；以此类推……。下面的SQL语句用于计算出生日期是当年第几天以及注册日期中的年份部分：

```
SELECT FBirthDay, DATEPART(Dayofyear,FBirthDay),
FRegDay, DATEPART(Year, FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay		FRegDay	
1981-03-22 00:00:00.0	81	1998-05-01 00:00:00.0	1998
1987-01-18 00:00:00.0	18	1999-08-21 00:00:00.0	1999
1987-11-08 00:00:00.0	312	2001-09-18 00:00:00.0	2001
1982-07-12 00:00:00.0	193	2000-03-01 00:00:00.0	2000
1983-02-16 00:00:00.0	47	1998-05-01 00:00:00.0	1998
1984-08-07 00:00:00.0	220	1999-03-01 00:00:00.0	1999
1980-01-09 00:00:00.0	9	2002-09-23 00:00:00.0	2002
1972-07-18 00:00:00.0	200	1995-06-19 00:00:00.0	1995

粗看起来，DATEPART()函数和DATENAME()函数完全一样，不过其实它们并不是只是名称不同的别名函数，虽然都是用来提取日期的特定部分的，不过DATEPART()函数的返回值是数字而DATENAME()函数则会将尽可能的以名称的方式做为返回值。

5.3.7.3 Oracle

在5.3.6.3一节中我们介绍了Oracle中使用TO_CHAR()函数格式化日期的方法，使用它就可以提取日期的任意部分，比如下面的SQL用于提取每个人员的出生年份、出生时是当年的第几天、出生时是当年的第几周：

```
SELECT
FBirthDay,
TO_CHAR(FBirthDay,'YYYY') AS y,
TO_CHAR(FBirthDay,'DDD') AS d,
TO_CHAR(FBirthDay,'WW') AS u
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	Y	D	U
1981-03-22 00:00:00.0	1981	081	12
1987-01-18 00:00:00.0	1987	018	03
1987-11-08 00:00:00.0	1987	312	45
1982-07-12 00:00:00.0	1982	193	28
1983-02-16 00:00:00.0	1983	047	07
1984-08-07 00:00:00.0	1984	220	32
1980-01-09 00:00:00.0	1980	009	02
1972-07-18 00:00:00.0	1972	200	29

5.3.7.4 DB2

DB2中没有提供像MYSQL、Oracle中那样的日期格式化函数，也没有提供像MSSQLServer中DATENAME()那样通用的取日期的特定部分的函数，DB2中对于提取日期的不同的部分需要使用不同的函数，这些函数的列表如下：

函数名	功能说明
YEAR()	取参数的年份部分
MONTH()	取参数的月份部分，返回值为整数
MONTHNAME()	对于参数的月部分的月份，返回一个大小写混合的字符串（例如，January）。
QUARTER()	取参数的季度数
DAYOFYEAR()	返回参数中一年中的第几天，用范围在 1-366 的整数值表示。
DAY()	取参数的日部分
DAYNAME()	返回一个大小写混合的字符串，对于参数的日部分，用星期表示这一天的名称（例如，Friday）。
WEEK()	返回参数是一年中的第几周
DAYOFWEEK()	返回参数中的星期几，用范围在 1-7 的整数值表示，其中 1 代表星期日。
HOUR()	取参数的小时部分
MINUTE()	取参数的分钟部分
SECOND()	取参数的秒钟部分
MICROSECOND()	取参数的微秒部分

下面的SQL语句用于计算出生日期的年份部分并且计算注册日期的月份名以及是一年中的第几周：

```
SELECT
```

```
FBirthDay,
YEAR(FBirthDay),
FRegDay,
MONTHNAME(FRegDay),
WEEK(FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	2	FREGDAY	4	5
1981-03-22	1981	1998-05-01	五月	18
1987-01-18	1987	1999-08-21	八月	34
1987-11-08	1987	2001-09-18	九月	38
1982-07-12	1982	2000-03-01	三月	10
1983-02-16	1983	1998-05-01	五月	18
1984-08-07	1984	1999-03-01	三月	10
1980-01-09	1980	2002-09-23	九月	39
1972-07-18	1972	1995-06-19	六月	25

5.4 其他函数

除了数学函数、字符串函数、日期函数之外，数据库中还有其他一些函数，比如进行类型转换的函数、进行非空逻辑判断的函数等，这些函数也是非常重要的，因此在本节中我们将对这些函数进行介绍。

5.4.1 类型转换

在使用SQL语句的时候，我们使用的数据的类型不一定符合函数或者运算符的需要，比如函数需要整数类型的数据而我们使用的则是一个字符串，在一些情况下数据库系统会替我们自动将字符串类型转换为整数类型，这种转换称为隐式转换。但是在有的情况下数据库系统不会进行隐式转换，这时就要使用类型转换函数了，这种转换称为显式转换。使用类型转换函数不仅可以保证类型转换的正确性，而且可以提高数据处理的速度，因此应该使用显式转换，尽量避免使用隐式转换。

在主流数据库系统中都提供了类型转换函数，下面分别进行介绍。

5.4.1.1 MYSQL

MYSQL中提供了CAST()函数和CONVERT()函数用于进行类型转换，CAST()是符合ANSI SQL99的函数，CONVERT() 是符合ODBC标准的函数，这两个函数只是参数的调用方式略有差异，其功能几乎相同。这两个函数的参数格式如下：

CAST(expression AS type)

CONVERT(expression,type)

参数expression为待进行类型转换的表达式，而type为转换的目标类型，type可以是下面的任一个：

可选值	缩写	说明
BINARY		BINARY字符串
CHAR		字符串类型
DATE		日期类型
DATETIME		时间日期类型
SIGNED INTEGER	SIGNED	有符号整数
TIME		时间类型
UNSIGNED INTEGER	UNSIGNED	无符号整数

下面的SQL语句分别演示以有符号整形、无符号整形、日期类型、时间类型为目标类型的数据转换：

```
SELECT
CAST('-30' AS SIGNED) as sig,
CONVERT ('36', UNSIGNED INTEGER) as usig,
CAST('2008-08-08' AS DATE) as d,
CONVERT ('08:09:10', TIME) as t
```

执行完毕我们就能在输出结果中看到下面的执行结果：

sig	usig	d	t
-30	36	2008-08-08	08:09:10

5.4.1.2 MSSQLServer

与MYSQL类似，MSSQLServer中同样提供了名称为CAST()和CONVERT()两个函数用于进行类型转换，CAST()是符合ANSI SQL99的函数，CONVERT() 是符合ODBC标准的函数。与MYSQL中的CONVERT()函数不同的是MSSQLServer中的CONVERT()函数参数顺序正好与MYSQL中的CONVERT()函数参数顺序相反。这两个函数的参数格式如下：

CAST (expression AS data_type)

CONVERT (data_type, expression)

参数expression为待进行类型转换的表达式，而type为转换的目标类型，与MYSQL不同，MYSQLServer中的目标类型几乎可以是数据库系统支持的任何类型。

下面的SQL语句分别演示以整形、数值、日期时间类型为目标类型的数据转换：

```
SELECT
CAST('-30' AS INTEGER) as i,
CONVERT(DECIMAL,'3.1415726') as d,
CONVERT(DATETIME,'2008-08-08 08:09:10') as dt
```

执行完毕我们就能在输出结果中看到下面的执行结果：

i	d	dt
-30	3	2008-08-08 08:09:10.0

下面的SQL语句用于将每个人的身份证后三位转换为整数类型并且进行相关的计算：

```
SELECT FIdNumber,
RIGHT(FIdNumber,3) as 后三位,
CAST(RIGHT(FIdNumber,3) AS INTEGER) as 后三位的整数形式,
CAST(RIGHT(FIdNumber,3) AS INTEGER)+1 as 后三位加1,
CONVERT(INTEGER,RIGHT(FIdNumber,3))/2 as 后三位除以2
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FIdNumber	后三位	后三位的整数形式	后三位加 1	后三位除以 2
123456789120	120	120	121	60
123456789121	121	121	122	60
123456789122	122	122	123	61
123456789123	123	123	124	61
123456789124	124	124	125	62
123456789125	125	125	126	62
123456789126	126	126	127	63
123456789127	127	127	128	63

5.4.1.3 Oracle

Oracle中也有一个名称为CONVERT()的函数，不过这个函数是用来进行字符集转换的。Oracle中不支持用做数据类型转换的CAST()和CONVERT()两个函数，它提供了针对性更强的类型TO_CHAR()、TO_DATE()、TO_NUMBER()等函数，这些函数可以将数据显式的转换为字符串类型、日期时间类型或者数值类型。Oracle中还提供了HEXTORAW()、RAWTOHEX()、TO_MULTI_BYTE()、TO_SINGLE_BYTE()等函数用于存储格式的转换。下面我们将对这些函数进行分别介绍。

1) TO_CHAR()

TO_CHAR()函数用来将时间日期类型或者数值类型的数据转换为字符串，其参数格式如下：

TO_CHAR(expression,format)

参数expression为待转换的表达式，参数format为转换后的字符串格式，参数format可以省略，如果省略参数format将会按照数据库系统内置的转换规则进行转换。参数format的可以采用的格式非常丰富，具体可以参考Oracle的联机文档。

下面的SQL语句将出生日期和身高按照不同的格式转换为字符串类型：

```
SELECT FBirthDay,
TO_CHAR(FBirthDay,'YYYY-MM-DD') as c1,
FWeight,
TO_CHAR(FWeight,'L99D99MI') as c2,
TO_CHAR(FWeight) as c3
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	C1	FWEIGHT	C2	C3
1981-03-22 00:00:00.0	1981-03-22	56.67	￥56.67	56.67
1987-01-18 00:00:00.0	1987-01-18	36.17	￥36.17	36.17
1987-11-08 00:00:00.0	1987-11-08	40.33	￥40.33	40.33
1982-07-12 00:00:00.0	1982-07-12	46.23	￥46.23	46.23
1983-02-16 00:00:00.0	1983-02-16	48.68	￥48.68	48.68
1984-08-07 00:00:00.0	1984-08-07	66.67	￥66.67	66.67
1980-01-09 00:00:00.0	1980-01-09	51.28	￥51.28	51.28
1972-07-18 00:00:00.0	1972-07-18	60.32	￥60.32	60.32

2) TO_DATE()

TO_DATE()函数用来将字符串转换为时间类型，其参数格式如下：

TO_DATE (expression,format)

参数expression为待转换的表达式，参数format为转换格式，参数format可以省略，如果省略参数format将会按照数据库系统内置的转换规则进行转换。

下面的SQL语句用于将字符串形式的数据按照特定的格式解析为日期类型：

```
SELECT
TO_DATE('2008-08-08 08:09:10', 'YYYY-MM-DD HH24:MI:SS') as dt1,
TO_DATE('20080808 080910', 'YYYYMMDD HH24MISS') as dt2
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

DT1	DT2
2008-08-08 08:09:10.0	2008-08-08 08:09:10.0

3) TO_NUMBER()

TO_NUMBER()函数用来将字符串转换为数值类型，其参数格式如下：

TO_NUMBER (expression,format)

参数expression为待转换的表达式，参数format为转换格式，参数format可以省略，如果省略参数format将会按照数据库系统内置的转换规则进行转换。参数format的可以采用的格式非常丰富，具体可以参考Oracle的联机文档。

下面的SQL语句用于将字符串形式的数据按照特定的格式解析为数值类型：

```
SELECT
TO_NUMBER('33.33') as n1,
TO_NUMBER('100.00', '9G999D99') as n2
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

N1	N2
33.33	100.55

4) HEXTORAW()、RAWTOHEX()

HEXTORAW()用于将十六进制格式的数据转换为原始值，而RAWTOHEX()函数用来将原始值转换为十六进制格式的数据。例子如下：

```
SELECT HEXTORAW('7D'),
RAWTOHEX ('a'),
HEXTORAW(RAWTOHEX('w'))
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

HEXTORAW(7D)	RAWTOHEX(A)	HEXTORAW(RAWTOHEX(W))
}	61	w

5) TO_MULTI_BYTE()、TO_SINGLE_BYTE()

TO_MULTI_BYTE()函数用于将字符串中的半角字符转换为全角字符，而TO_SINGLE_BYTE()函数则用来将字符串中的全角字符转换为半角字符。例子如下：

```
SELECT
TO_MULTI_BYTE('moring'),
TO_SINGLE_BYTE('ｈｅｌｌｏ')
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

TO_MULTI_BYTE(MORING)	TO_SINGLE_BYTE(ＨＥＬＬＯ)
ｍｏｒｉｎｇ	hello

5.4.1.4 DB2

DB2中没有提供专门进行显式类型转换的函数，取而代之的是借用了很多高级语言中的

强制类型转换的概念，也就是使用目标类型名做为函数名来进行类型转换，比如要将expr转换为日期类型，那么使用DATE(expr)即可。这种实现机制非常方便，降低了学习难度。

下面的SQL语句展示了DB2中类型转换的方式：

```
SELECT CHAR(FRegDay),
INT('33'),
DOUBLE('-3.1415926')
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

1	2	3
1998-05-01	33	-3.1415926
1999-08-21	33	-3.1415926
2001-09-18	33	-3.1415926
2000-03-01	33	-3.1415926
1998-05-01	33	-3.1415926
1999-03-01	33	-3.1415926
2002-09-23	33	-3.1415926
1995-06-19	33	-3.1415926

5.4.2 空值处理

在数据库中经常需要对空值（NULL）做处理，比如“如果名称为空值则返回别名”，甚至还有更复杂的需求，比如“如果名称为空值则返回别名，如果别名也为空则返回‘佚名’两个字”、“如果名称为与别名相等则返回空值，否则返回名称”。这些需求已经带有流程控制的色彩了，一般来说需要在宿主语言中使用流程控制语句来进行处理，可是如果是在报表程序等大数据量的程序中把这些任务交给宿主语言的话会大大降低运行速度，因此我们必须想办法在SQL这一层进行处理。

为了更好的演示本节中的例子，我们需要对T_Person表中的数据进行一下修改，也就是将Kerry的出生日期修改为空值，将Smith的出生日期和注册日期都修改为空值，执行下面的SQL语句：

```
UPDATE T_Person SET FBirthDay=null WHERE FName='Kerry';
UPDATE T_Person SET FBirthDay=null AND FRegDay=null WHERE FName='Smith';
```

执行完毕我们查看T_Person表中中的数据如下：

FIDNUMBER	FNAME	FBIRTHDAY	FREGDAY	FWEIGHT
123456789120	Tom	1981-03-22	1998-05-01	56.67
123456789121	Jim	1987-01-18	1999-08-21	36.17
123456789122	Lily	1987-11-08	2001-09-18	40.33
123456789123	Kelly	1982-07-12	2000-03-01	46.23
123456789124	Sam	1983-02-16	1998-05-01	48.68
123456789125	Kerry	<NULL>	1999-03-01	66.67
123456789126	Smith	<NULL>	<NULL>	51.28
123456789127	BillGates	1972-07-18	1995-06-19	60.32

5.4.2.1 COALESCE()函数

主流数据库系统都支持COALESCE()函数，这个函数主要用来进行空值处理，其参数格式如下：

```
COALESCE ( expression,value1,value2……,valuen)
```

COALESCE()函数的第一个参数expression为待检测的表达式，而其后的参数个数不定。

COALESCE()函数将会返回包括expression在内的所有参数中的第一个非空表达式。如果expression不为空值则返回expression；否则判断value1是否是空值，如果value1不为空值则返回value1；否则判断value2是否是空值，如果value2不为空值则返回value3；……以此类推，如果所有的表达式都为空值，则返回NULL。

我们将使用COALESCE()函数完成下面的功能，返回人员的“重要日期”：如果出生日期不为空则将出生日期做为“重要日期”，如果出生日期为空则判断注册日期是否为空，如果注册日期不为空则将注册日期做为“重要日期”，如果注册日期也为空则将“2008年8月8日”做为“重要日期”。实现此功能的SQL语句如下：

MYSQL、MSSQLServer、DB2:

```
SELECT FName,FBirthDay,FRegDay,
COALESCE(FBirthDay,FRegDay,'2008-08-08')  AS ImportDay
FROM T_Person
```

Oracle:

```
SELECT FBirthDay,FRegDay,
COALESCE(FBirthDay,FRegDay,TO_DATE('2008-08-08', 'YYYY-MM-DD HH24:MI:SS'))
AS ImportDay
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FBirthDay	FRegDay	ImportDay
Tom	1981-03-22 00:00:00	1998-05-01 00:00:00	1981-03-22 00:00:00
Jim	1987-01-18 00:00:00	1999-08-21 00:00:00	1987-01-18 00:00:00
Lily	1987-11-08 00:00:00	2001-09-18 00:00:00	1987-11-08 00:00:00
Kelly	1982-07-12 00:00:00	2000-03-01 00:00:00	1982-07-12 00:00:00
Sam	1983-02-16 00:00:00	1998-05-01 00:00:00	1983-02-16 00:00:00
Kerry	<NULL>	1999-03-01 00:00:00	1999-03-01 00:00:00
Smith	<NULL>	<NULL>	2008-08-08
BillGates	1972-07-18 00:00:00	1995-06-19 00:00:00	1972-07-18 00:00:00

这里边最关键的就是Kerry和Smith这两行，可以看到这里的计算逻辑是完全符合我们的需求的。

5.4.2.2 COALESCE()函数的简化版

COALESCE()函数可以用来完成几乎所有的空值处理，不过在很多数据库系统中都提供了它的简化版，这些简化版中只接受两个变量，其参数格式如下：

MYSQL:

```
IFNULL(expression,value)
```

MSSQLServer:

```
ISNULL(expression,value)
```

Oracle:

```
NVL(expression,value)
```

这几个函数的功能和COALESCE(expression,value)是等价的。比如SQL语句用于返回人员的“重要日期”，如果出生日期不为空则将出生日期做为“重要日期”，如果出生日期为空则返回NULL：

MYSQL:

```
SELECT FBirthDay,FRegDay,
IFNULL(FBirthDay,FRegDay)  AS ImportDay
```

FROM T_Person

MSSQLServer:

SELECT FBirthDay,FRegDay,

ISNULL(FBirthDay,FRegDay) AS ImportDay

FROM T_Person

Oracle:

SELECT FBirthDay,FRegDay,

NVL(FBirthDay,FRegDay) AS ImportDay

FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FBIRTHDAY	FREGDAY	IMPORTDAY
1981-03-22 00:00:00.0	1998-05-01 00:00:00.0	1981-03-22 00:00:00.0
1987-01-18 00:00:00.0	1999-08-21 00:00:00.0	1987-01-18 00:00:00.0
1987-11-08 00:00:00.0	2001-09-18 00:00:00.0	1987-11-08 00:00:00.0
1982-07-12 00:00:00.0	2000-03-01 00:00:00.0	1982-07-12 00:00:00.0
1983-02-16 00:00:00.0	1998-05-01 00:00:00.0	1983-02-16 00:00:00.0
<NULL>	1999-03-01 00:00:00.0	1999-03-01 00:00:00.0
<NULL>	<NULL>	<NULL>
1972-07-18 00:00:00.0	1995-06-19 00:00:00.0	1972-07-18 00:00:00.0

5.4.2.3 NULLIF()函数

主流数据库都支持NULLIF()函数，这个函数的参数格式如下：

NULLIF (expression1 , expression2)

如果两个表达式不等价，则 NULLIF 返回第一个 expression1的值。如果两个表达式等价，则 NULLIF 返回第一个 expression1类型的空值。也就是返回类型与第一个 expression 相同。

下面的SQL演示了NULLIF()函数的用法：

```
SELECT FBirthDay,FRegDay,
NULLIF(FBirthDay,FRegDay)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBirthDay	FRegDay	
1981-03-22 00:00:00.0	1998-05-01 00:00:00.0	1981-03-22 00:00:00.0
1987-01-18 00:00:00.0	1999-08-21 00:00:00.0	1987-01-18 00:00:00.0
1987-11-08 00:00:00.0	2001-09-18 00:00:00.0	1987-11-08 00:00:00.0
1982-07-12 00:00:00.0	2000-03-01 00:00:00.0	1982-07-12 00:00:00.0
1983-02-16 00:00:00.0	1998-05-01 00:00:00.0	1983-02-16 00:00:00.0
<NULL>	1999-03-01 00:00:00.0	<NULL>
<NULL>	<NULL>	<NULL>
1972-07-18 00:00:00.0	1995-06-19 00:00:00.0	1972-07-18 00:00:00.0

5.4.3 CASE函数

COALESCE()函数只能用来进行空值的逻辑判断处理，如果要实现“如果年龄大于25则返回姓名，否则返回别名”这样的逻辑判断就比较麻烦了。在主流数据库系统中提供了CASE函数的支持，严格意义上来讲CASE函数已经是流程控制语句了，不是简单意义上的函数，不过为了方便，很多人都将CASE称作“流程控制函数”。

CASE函数有两种用法，下面分别介绍。

5.4.3.1 用法一

CASE函数的语法如下：

```
CASE expression
WHEN value1 THEN returnvalue1
WHEN value2 THEN returnvalue2
WHEN value3 THEN returnvalue3
……
ELSE defaultreturnvalue
END
```

CASE函数对表达式expression进行测试，如果expression等于value1则返回returnvalue1，如果expression等于value2则返回returnvalue2，expression等于value3则返回returnvalue3，……以此类推，如果不符合所有的WHEN条件，则返回默认值defaultreturnvalue。

可见CASE函数和普通编程语言中的SWITCH……CASE语句非常类似。使用CASE函数我们可以实现非常复杂的业务逻辑。下面的SQL用于判断谁是“好孩子”，我们比较偏爱Tom和Lily，所以我们将他们认为是好孩子，而我们比较不喜欢Sam和Kerry，所以认为他们是坏孩子，其他孩子则为普通孩子：

```
SELECT
    FName,
    (CASE FName
    WHEN 'Tom' THEN 'GoodBoy'
    WHEN 'Lily' THEN 'GoodGirl'
    WHEN 'Sam' THEN 'BadBoy'
    WHEN 'Kerry' THEN 'BadGirl'
    ELSE 'Normal'
    END) as isgood
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	ISGOOD
Tom	GoodBoy
Jim	Normal
Lily	GoodGirl
Kelly	Normal
Sam	BadBoy
Kerry	BadGirl
Smith	Normal
BillGates	Normal

CASE函数在制作报表的时候非常有用。比如表T_Customer中的FLevel字段是整数类型，它记录了客户的级别，如果为1则是VIP客户，如果为2则是高级客户，如果为3则是普通客户，在制作报表的时候显然不应该把1、2、3这样的数字显示到报表中，而应该显示相应的文字，这里就可以使用CASE函数进行处理，SQL语句如下：

```
SELECT
    FName,
    (CASE FLevel
```

```
    WHEN 1 THEN 'VIP客户'
    WHEN 2 THEN '高级客户'
    WHEN 3 THEN '普通客户'
    ELSE '客户类型错误'
    END) as FLevelName
FROM T_Customer
```

5.4.3.2 用法二

上边一节中介绍的CASE语句的用法只能用来实现简单的“等于”逻辑的判断，要实现“如果年龄小于18则返回‘未成年人’，否则返回‘成年人’”是无法完成的。值得庆幸的是，CASE函数还提供了第二种用法，其语法如下：

CASE
WHEN condition1 THEN returnvalue1
WHEN condition 2 THEN returnvalue2
WHEN condition 3 THEN returnvalue3
……
ELSE defaultreturnvalue
END

其中的condition1 、condition 2、condition 3……为条件表达式，CASE函数对各个表达式从前向后进行测试，如果条件condition1为真则返回returnvalue1，否则如果条件condition2为真则返回returnvalue2，否则如果条件condition3为真则返回returnvalue3，……以此类推，如果不符合所有的WHEN条件，则返回默认值defaultreturnvalue。

这种用法中没有限制只能对一个表达式进行判断，因此使用起来更加灵活。比如下面的SQL语句用来判断一个人的体重是否正常，如果体重小于40则认为太瘦，而如果体重大于50则认为太胖，介于40和50之间则认为是正常：

```
SELECT
    FName,
    FWeight,
    (CASE
    WHEN FWeight<40 THEN 'thin'
    WHEN FWeight>50 THEN 'fat'
    ELSE 'ok'
    END) as isnormal
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FWEIGHT	ISNORMAL
Tom	56.67	fat
Jim	36.17	thin
Lily	40.33	ok
Kelly	46.23	ok
Sam	48.68	ok
Kerry	66.67	fat
Smith	51.28	fat
BillGates	60.32	fat

5.5 各数据库系统独有函数

前边几章我们讲解了一些常用的函数，这些函数不是在各个主流数据库系统中有着相同的名称和用法，就是在各个主流数据库系统中有等价的实现，这些函数可以基本满足我们大部分需求。不过在各个主流数据库系统还提供了一些自身独有的函数，这些函数在其他的数据库系统中一般都没有等价的实现，使用这些函数以后会给系统的跨数据库移植带来一定的麻烦，不过如果系统对跨数据库移植没有要求的话，那么使用这些函数不仅能提高开发速度，而且能够更好发挥数据库系统的性能，所以了解它们还是非常有必要的，因此这里我们专门安排一节来介绍这些函数。

5.5.1 MYSQL中的独有函数

5.5.1.1 IF()函数

使用CASE函数可以实现非常复杂的逻辑判断，可是若只是实现“如果符合条件则返回A，否则返回B”这样简单的判断逻辑的话，使用CASE函数就过于繁琐，如下：

```
CASE
WHEN condition THEN A
ELSE B
END
```

MYSQL提供了IF()函数用于简化这种逻辑判断，其语法格式如下：

```
IF(expr1,expr2,expr3)
```

如果 expr1 为真(expr1 <> 0 以及 expr1 <> NULL)，那么 IF() 返回 expr2，否则返回 expr3。IF()返回一个数字或字符串，这取决于它被使用的语境。

这里使用IF()函数来判断一个人的体重是否太胖，而如果体重大于50则认为太胖，否则认为正常：

```
SELECT
    FName,
    FWeight,
    IF(FWeight>50,'太胖','正常') AS ISTooFat
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight	ISTooFat
Tom	56.67	太胖
Jim	36.17	正常
Lily	40.33	正常
Kelly	46.23	正常
Sam	48.68	正常
Kerry	66.67	太胖
Smith	51.28	太胖
BillGates	60.32	太胖

5.5.1.2 CONV()函数

CONV()函数用于对数字进行进制转换，比如将十进制的26转换为2进制显示，其参数格式如下：

```
CONV(N,from_base,to_base)
```

将数字 N 从 from_base进制转换到 to_base进制，并以字符串表示形式返回。from_base和to_base的最小值为 2，最大值为 36。如果 to_base 是一个负值，N 将被看作为是一个有符号数字。否则，N 被视为是无符号的。

下面的SQL语句用于将十进制的26转换为2进制显示、将十六进制的7D转换为八进制显

示：

SELECT CONV('26',10,2), CONV(26,10,2),CONV('7D',16,8)

可以看到数字N既可以为字符串也可以为整数，如果为整数则它被解释为十进制数字。执行完毕我们就能在输出结果中看到下面的执行结果：

CONV('26',10,2)	CONV(26,10,2)	CONV('7D',16,8)
11010	11010	175

下面的SQL语句用来将每个人的体重四舍五入为整数，然后以二进制的形式显示它们：

```
SELECT FWeight,Round(FWeight),
CONV(Round(FWeight),10,2)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FWeight	Round(FWeight)	CONV(Round(FWeight),10,2)
56.67	57	111001
36.17	36	100100
40.33	40	101000
46.23	46	101110
48.68	49	110001
66.67	67	1000011
51.28	51	110011
60.32	60	111100

在日常系统开发过程中，我们最常进行的进制转换是将十进制的整数转换为十六进制显示，如果使用CONV()函数来进行转换的话比较麻烦，为此MYSQL提供了简化调用的函数BIN(N)、OCT(N)、HEX(N)它们分别用于返回 N的字符串表示的二进制、八进制值和十六进制形式。下面的SQL语句将每个人的体重四舍五入为整数，然后以二进制、八进制值和十六进制的形式显示它们：

```
SELECT FWeight,Round(FWeight),
BIN(Round(FWeight)) as b,
OCT(Round(FWeight)) as o,
HEX(Round(FWeight)) as h
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FWeight	Round(FWeight)	b	o	h
56.67	57	111001	71	39
36.17	36	100100	44	24
40.33	40	101000	50	28
46.23	46	101110	56	2E
48.68	49	110001	61	31
66.67	67	1000011	103	43
51.28	51	110011	63	33
60.32	60	111100	74	3C

5.5.1.3 填充函数

在数据显示的时候，出于美化的需要，经常规定数据显示的宽度，比如单元格中填充的文字要正好20个字符，如果超过20个字符则裁剪到20个字符，如果不足20字符，则在右侧用空格填充直至达到20个字符。在MYSQL中提供了LPAD()、RPAD()函数用于对字符串进行左

填充和右填充，其参数格式如下：

LPAD(str,len,padstr)

RPAD(str,len,padstr)

用字符串 padstr 对 str 进行左（右）边填补直至它的长度达到 len 个字符长度，然后返回 str。如果 str 的长度长于 len'，那么它将被截除到 len 个字符。

下面的SQL语句分别将每个人的姓名用星号左填充和右填充到5个字符：

```
SELECT FName,LPAD(FName,5,'*'),RPAD(FName,5,'*')
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	LPAD(FName,5,'*')	RPAD(FName,5,'*')
Tom	**Tom	Tom**
Jim	**Jim	Jim**
Lily	*Lily	Lily*
Kelly	Kelly	Kelly
Sam	**Sam	Sam**
Kerry	Kerry	Kerry
Smith	Smith	Smith
BillGates	BillG	BillG

5.5.1.4 REPEAT()函数

REPEAT()函数用来得到一个子字符串重复了若干次所组成的字符串，其参数格式如下：

REPEAT(str,count)

参数str为子字符串，而count为重复次数。

下面的SQL语句用于得到一个由5个星号以及一个由3个“OK”组成的字符串：

SELECT REPEAT('*',5), REPEAT('OK',3)

执行完毕我们就能在输出结果中看到下面的执行结果：

REPEAT('*',5)	REPEAT('OK',3)
*****	OKOKOK

MYSQL中提供了一个简化REPEAT()的函数SPACE(N)，它用来得到一个有 N 空格字符组成的字符串，可以看做是REPEAT(' ',N)的等价形式。

5.5.1.5 字符串颠倒

REVERSE()函数用来将一个字符串的顺序颠倒，下面的SQL语句将所有人员的姓名进行了颠倒：

SELECT FName, REVERSE(FName)

FROM T_Person

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	REVERSE(FName)
Tom	moT
Jim	miJ
Lily	yliL
Kelly	ylleK
Sam	maS
Kerry	yrreK
Smith	htimS
BillGates	setaGlliB

5.5.1.6 字符串的集合操作

使用CASE函数可以完成“将1翻译成VIP客户，将2翻译成高级客户，将3翻译成普通客户”这样的任务，但是使用起来比较麻烦，MYSQL中提供了几个字符串集合操作函数，分别是ELT()、FIELD()和FIND_IN_SET()，它们将“VIP客户”、“高级客户”、“普通客户”这样的匹配目标字符串当作集合处理，而将“1、2、3”这样的数字当成待匹配项。

首先来看ELT()函数，它的参数格式如下：

ELT(N,str1,str2,str3,...)

如果 N = 1，返回 str1，如果N = 2，返回 str2，等等。如果 N 小于 1 或大于参数的数量，返回 NULL。下面的SQL演示了ELT()函数的使用：

```
SELECT
ELT(2, 'ej', 'Heja', 'hej', 'foo'),
ELT(4, 'ej', 'Heja', 'hej', 'foo')
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ELT(2, 'ej', 'Heja', 'hej', 'foo')	ELT(4, 'ej', 'Heja', 'hej', 'foo')
Heja	foo

ELT()函数在制作报表的时候非常有用。比如表T_Customer中的FLevel字段是整数类型，它记录了客户的级别，如果为1则是VIP客户，如果为2则是高级客户，如果为3则是普通客户，在制作报表的时候显然不应该把1、2、3这样的数字显示到报表中，而应该显示相应的文字，这里就可以使用ELT()函数进行处理，SQL语句如下：

```
SELECT
    FName,
    ELT(FLevel, 'VIP客户', '高级客户', '普通客户')
FROM T_Customer
```

与ELT()函数正好相反，FIELD()函数用于计算字符串在一个字符串集合中的位置，它可以看做是ELT()的反函数。FIELD()函数的参数格式如下：

FIELD(str,str1,str2,str3,...)

返回 str 在列表 str1, str2, str3, ... 中的索引。如果没有发现匹配项，则返回 0。下面的SQL演示了FIELD ()函数的使用：

```
SELECT FIELD('vip','normal','member','vip') as f1,
FIELD('ej', 'Hej', 'ej', 'Heja', 'hej', 'foo') as f2
```

执行完毕我们就能在输出结果中看到下面的执行结果：

f1	f2
3	2

在数据库中有时存储的是字符串，有的情况下需要将字符串转换成整数，方便后续系统的处理，这时就可以使用CASE()函数，但是如果是在MYSQL中，则使用FIELD ()函数更方便。假设客户信息表T_Customer中的FCustomerTypeName保存的是“VIP”、“会员”、“普通客户”这样的文本信息，我们可以使用下面的SQL语句将这些文本信息转换为整数来表示：

```
SELECT
    FName,
    FIELD(FCustomerTypeName, 'VIP', '会员', '普通客户')
FROM T_Customer
```

FIELD()函数将中的参数个数是不确定的，但是在使用的时候参数的个数又是确定，是不能在运行时动态改变的。有时待匹配的字符串集合也是不确定的，这时就无法使用FIELD()函数函数了，MYSQL中提供了FIND_IN_SET()函数，它用一个分隔符分割的字符串做为待匹配字

符串集合，它的参数格式如下：

```
FIND_IN_SET(str,strlist)
```

如果字符串 str 在由 N 个子串组成的列表 strlist 中，返回它在strlist中的索引次序（从1开始计数）。一个字符串列表是由通过字符 “,” 分隔的多个子串组成。如果 str 在不 strlist 中或者如果 strlist 是一个空串，返回值为 0。如果任何一个参数为 NULL，返回值也是 NULL。如果第一个参数包含一个 “,”，这个函数将抛出错误信息。下面的SQL演示了FIELD ()函数的使用：

```
SELECT FIND_IN_SET('b','a,b,c,d') as f1,
FIND_IN_SET('d','a,b,c,d') as f2,
FIND_IN_SET('w','a,b,c,d') as f3
```

执行完毕我们就能在输出结果中看到下面的执行结果：

f1	f2	f3
2	4	0

5.5.1.7 计算集合中的最大最小值

MYSQL中的GREATEST()函数和LEAST()函数用于计算一个集合中的最大和最小值，它们的参数个数都是不定的，也就是它们可以对多个值进行比较。使用演示如下：

```
SELECT GREATEST(2,7,1,8,30,4,3,99,2,222,12),
LEAST(2,7,1,8,30,4,3,99,2,222,12)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

GREATEST(2,7,1,8,30,4,3,99,2,222,12)	LEAST(2,7,1,8,30,4,3,99,2,222,12)
222	1

5.5.1.8 辅助功能函数

DATABASE()函数返回当前数据库名；**VERSION()**函数以一个字符串形式返回 MySQL 服务器的版本；**USER()**函数（这个函数还有**SYSTEM_USER**、**SESSION_USER**两个别名）返回当前 MySQL 用户名。下面的SQL语句演示了这几个函数的使用：

SELECT **DATABASE(),VERSION(),USER()**

执行完毕我们就能在输出结果中看到下面的执行结果：

DATABASE()	VERSION()	USER()
demo	5.0.27-community-nt	yzk@192.168.88.2

ENCODE(str,pass_str)函数使用 **pass_str** 做为密钥加密 **str**，函数的返回结果是一个与 **string** 一样长的二进制字符。如果希望将它保存到一个列中，需要使用 **BLOB** 列类型。

与ENCODE()函数相反，**DECODE()**函数使用 **pass_str** 作为密钥解密经**ENCODE**加密后的字符串 **crypt_str**。

下面的SQL语句演示了这两个函数的使用：

```
SELECT FName,
Length(ENCODE(FName,'aha')),
DECODE(ENCODE(FName,'aha'),'aha')
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	Length(ENCODE(FName,'aha'))	DECODE(ENCODE(FName,'aha'),'aha')
Tom	3	Tom
Jim	3	Jim

Lily	4	Lily
Kelly	5	Kelly
Sam	3	Sam
Kerry	5	Kerry
Smith	5	Smith
BillGates	9	BillGates

除了加解密函数，MYSQL中还提供了对摘要算法的支持，**MD5(string)**、**SHA1(string)**两个函数就是分别用来使用**MD5**算法和**SHA1**算法来进行字符串的摘要计算的函数，下面的**SQL**语句用来计算每个人的姓名的**MD5**摘要和**SHA1**摘要：

```
SELECT FName,
MD5(FName),
SHA1(FName)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	MD5(FName)	SHA1(FName)
Tom	d9ffaca46d5990ec39501bcdf22ee7a1	26d58cf3df0903a2298788b72fced5bca9ea7144
Jim	d54b3c8fcd5ba07e47b400e69a287966	7627f9952b9c378edc494c5751804403e5973074
Lily	951cefd3ca5cb6773251e773379ff26a	89a254a753cf0fe40eadd6b6c9f76a99d41b01c2
Kelly	dce0d99515076c6a24e246952d635fc7	15299a0c1f3b95b73dc012d2185a2aded345edef
Sam	ba0e0cde1bf72c28d435c89a66afc61a	265dc298e8767361f7e407e746c90f399678a5ea
Kerry	a214c0e659e36eb1194137976cab3619	985f2f99dd67185bfa493786e8defbe722ccbeac
Smith	e95f770ac4fb91ac2e4873e4b2dfc0e6	96bcf8c98f94b6ace4a4b716cf0e3b32743a08b1
BillGates	baa81671b06cec362191b2d4fdd78435	ba64a865bdd75c81def59d63b61c24ce3b09bb85

使用UUID算法来生成一个唯一的字符串序列被越来越多的开发者所使用，MYSQL中也提供了对UUID算法的支持，UUID()函数就是用来生成一个UUID字符串的，使用方法如下：

SELECT UUID(),UUID()

执行完毕我们就能在输出结果中看到下面的执行结果（由于UUID算法生成的字符串是全局唯一的，所以你的运行结果会与这里显示的不同）：

UUID()	UUID()
d7495ecd-1863-102b-9b74-218a53021251	d7495ef7-1863-102b-9b74-218a53021251

5.5.2 MSSQLServer中的独有函数

5.5.2.1 PATINDEX()函数

MSSQLServer的CHARINDEX()函数用来计算字符串中指定表达式的开始位置，它是一种确定值的匹配，有时我们需要按照一定模式进行匹配，比如“计算字符串中第一个长度为2并且第二个字符为m的子字符串的位置”，这时使用CHARINDEX()函数就不凑效了。MSSQLServer中PATINDEX()函数就是用来进行这种模式字串匹配的，其参数格式如下：

PATINDEX ('%pattern%' , expression)

它返回指定表达式中模式'%pattern%'第一次出现的起始位置；如果在全部有效的文本和字符数据类型中没有找到该模式，则返回零。在模式中可以使用通配符。

下面的 SQL 语句用来查找每个人的姓名中第一个长度为 2 并且第二个字符为 m 的子字符串的位置：

```
SELECT FName,PATINDEX('%_m%',FName)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	
Tom	2
Jim	2
Lily	0
Kelly	0
Sam	2
Kerry	0
Smith	1
BillGates	0

5.5.2.2 REPLICATE ()函数

REPLICATE ()函数用来得到一个子字符串重复了若干次所组成的字符串，它和MYSQL中的REPEAT()函数是一样的，其参数格式如下：

REPLICATE (str,count)

参数str为子字符串，而count为重复次数。

下面的SQL语句用于将每个人的姓名重复n次，n等于体重与20的整除结果：

```
SELECT FName,FWeight,
CAST(FWeight/20 AS INT),
REPLICATE(FName, CAST(FWeight/20 AS INT))
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FWeight		
Tom	56.67	2	TomTom
Jim	36.17	1	Jim
Lily	40.33	2	LilyLily
Kelly	46.23	2	KellyKelly
Sam	48.68	2	SamSam
Kerry	66.67	3	KerryKerryKerry
Smith	51.28	2	SmithSmith
BillGates	60.32	3	BillGatesBillGatesBillGates

和MYSQL一样，MYSQL中同样提供了一个简化REPLICATE ()调用的函数SPACE(N)，它用来得到一个有 N 空格字符组成的字符串，可以看做是REPLICATE (' ',N)的等价形式。

5.5.2.3 字符串颠倒

REVERSE()函数用来将一个字符串的顺序颠倒，下面的SQL语句将所有人员的姓名进行了颠倒：

```
SELECT FName, REVERSE(FName)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	
Tom	moT
Jim	miJ
Lily	yliL
Kelly	ylleK
Sam	maS
Kerry	yrreK
Smith	htimS
BillGates	setaGlliB

5.5.2.4 ISDATE()函数

ISDATE()函数用来确定输入表达式是否为有效日期。如果输入表达式是有效日期，那么 ISDATE 返回 1；否则，返回 0。其参数格式如下：

ISDATE (expression)

expression 参数为要验证其是否为日期的表达式。expression 可以是 text、ntext 表达式和 image 表达式以外的任意表达式，可以隐式转换为 nvarchar。

下面的 SQL 语句演示了这个函数的使用：

```
SELECT
ISDATE(NULL) as d1,
ISDATE('13/43/3425') as d2,
ISDATE('1995-10-1a') as d3,
ISDATE(19920808) as d4,
ISDATE('1/23/95') as d5,
ISDATE('1995-10-1') as d6,
ISDATE('19920808') as d7,
ISDATE(' Abc') as d8
```

执行完毕我们就能在输出结果中看到下面的执行结果：

d1	d2	d3	d4	d5	d6	d7	d8
0	0	0	1	0	1	1	0

5.5.2.5 ISNUMERIC()函数

ISNUMERIC ()函数用来确定表达式是否为有效的数值类型。如果输入表达式的计算值为有效的整数、浮点数、money 或 decimal 类型时，ISNUMERIC 返回 1；否则返回 0。其参数格式如下：

ISNUMERIC (expression)

expression 参数为要计算的表达式。下面的 SQL 语句演示了这个函数的使用：

```
SELECT
ISNUMERIC(NULL) as d1,
ISNUMERIC('13/43/3425') as d2,
ISNUMERIC('30a.8') as d3,
ISNUMERIC(19920808) as d4,
ISNUMERIC('1/23/95') as d5,
ISNUMERIC('3E-3') as d6,
ISNUMERIC('19920808') as d7,
ISNUMERIC('-30.3') as d8
```

执行完毕我们就能在输出结果中看到下面的执行结果：

d1	d2	d3	d4	d5	d6	d7	d8
0	0	0	1	0	1	1	1

5.5.2.6 辅助功能函数

APP_NAME()函数返回当前会话的应用程序名称；CURRENT_USER函数（注意这个函数不能带括号调用）返回当前登陆用户名；HOST_NAME()函数返回工作站名。下面的SQL语句演示了这几个函数的使用：

```
SELECT APP_NAME() as appname,
CURRENT_USER as cu,
HOST_NAME() as hostname
```

执行完毕我们就能在输出结果中看到下面的执行结果：

appname	cu	hostname
jTDS	dbo	YANGZK

与 MYSQL 类似，MSSQLServer 中同样提供了生成全局唯一字符串的函数 NEWID()，下面生成三个 UUID 字符串：

```
SELECT NEWID() AS id1,
NEWID() AS id2
```

执行完毕我们就能在输出结果中看到下面的执行结果：

id1	id2
705FAA88-12B9-4C52-9B77-589DD20256C3	A110A5E5-92C7-461F-91F8-BF35129FE7B4

5.5.3 Oracle 中的独有函数

5.5.3.1填充函数

与MYSQL类似，Oracle中也提供了用于进行字符串填充的函数LPAD()、RPAD()，其参数格式如下：

LPAD(char1,n [,char2])

RPAD(char1, n [,char2])

与MYSQL中不同的是，Oracle中LPAD()和RPAD()函数的第三个参数是可以省略的，如果省略第三个参数，则使用单个空格进行填充。

下面的SQL语句分别将每个人的姓名用星号左填充和井号右填充到5个字符：

```
SELECT FName,
LPAD(FName,5,'*'),
RPAD(FName,5,'#')
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	LPAD(FNAME,5,*)	RPAD(FNAME,5,#)
Tom	**Tom	Tom##
Jim	**Jim	Jim##
Lily	*Lily	Lily#
Kelly	Kelly	Kelly
Sam	**Sam	Sam##
Kerry	Kerry	Kerry
Smith	Smith	Smith
BillGates	BillG	BillG

5.5.3.2 返回当月最后一天

Oracle 中的 LAST_DAY()函数可以用来计算指定日期所在月份的最后一天的日期。下面的 SQL 语句用于计算每个人出生时当月的最后一天的日期：

```
SELECT FName,FBirthDay,
LAST_DAY(FBirthDay)
FROM T_Person
WHERE FBirthDay IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FBIRTHDAY	LAST_DAY(FBIRTHDAY)
Tom	1981-03-22 00:00:00.0	1981-03-31 00:00:00.0
Jim	1987-01-18 00:00:00.0	1987-01-31 00:00:00.0
Lily	1987-11-08 00:00:00.0	1987-11-30 00:00:00.0
Kelly	1982-07-12 00:00:00.0	1982-07-31 00:00:00.0
Sam	1983-02-16 00:00:00.0	1983-02-28 00:00:00.0
BillGates	1972-07-18 00:00:00.0	1972-07-31 00:00:00.0

5.5.3.3 计算最大最小值

和MYSQL类似，Oracle中提供了用来计算一个集合中的最大和最小值的GREATEST()函数和LEAST()函数。其使用方法和MYSQL一致::

```
SELECT GREATEST(2,7,1,8,30,4,5566,99,2,222,12),
LEAST(2,7,1,8,30,4,3,99,-2,222,12)
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

GREATEST(2,7,1,8,30,4,5566,99,2,222,12)	LEAST(2,7,1,8,30,4,3,99,-2,222,12)
5566	-2

5.5.3.4 辅助功能函数

USER 函数用来取得当前登录用户名，注意使用这个函数的时候不能使用括号形式的空参数列表，也就是 USER()这种使用方式是不对的。正确使用方式如下：

```
SELECT USER
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

USER
SYS

USERENV()函数用来取得当前登录用户相关的环境信息，这个函数的返回值为字符串类型，需要根据情况将返回值转换为合适的类型。它的参数格式如下：

USERENV(option)

option 参数为要取得的环境信息的名称，可取值如下：

可取值	说明
'ISDBA'	如果当前登录用户有 DBA 的角色则返回 TRUE，否则返回 FALSE
'LANGUAGE'	返回当前登录用户使用的语言和字符集，返回格式为“语言.字符集”
'TERMINAL'	返回当前登录用户的操作系统标识
'SESSIONID'	返回当前登录用户的会话标识
'ENTRYID'	返回当前登录用户的认证标识
'LANG'	返回当前用户使用的语言，它比'LANGUAGE'的返回值短
'INSTANCE'	返回当前实例的标识

下面的 SQL 语句用来取得当前登录用户的语言信息和权限信息：

```
SELECT USERENV('ISDBA') AS ISDBA,
USERENV('LANGUAGE') AS LANGUAGE,
USERENV('LANG') AS LANG
FROM DUAL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

ISDBA	LANGUAGE	LANG
TRUE	SIMPLIFIED CHINESE_CHINA.AL32UTF8	ZHS

到这里本章即将结束，请执行下面的 SQL 语句将本章中用到的数据表删除：

DROP TABLE T_Person

第六章 索引与约束

前面的章节我们讲解了数据表的创建以及数据的增删改查，掌握了这些知识我们已经可以对数据库进行基本的操作了，但是在使用一段时间后我们就发现很多问题，比如按照年龄进行数据检索的时候速度非常快但是按照姓名进行数据检索的时候则非常慢、一个人的姓名不应该是未知的但是还是录入了大量的值为 NULL 的姓名到系统中、注册会员的 Email 地址不应该重复的可是在数据表中却能够插入重复的 Email 地址。在数据库系统中解决这些问题的技术就是索引与约束，索引用来提高数据的检索速度，而约束则用来保证数据的完整性。本章将对这两种技术进行详细讲解。

6.1 索引

在第一章中已经讲解了索引的用途并且对它的原理做了介绍，本节将介绍在数据库中创建索引的方法。索引是建立在数据表上的，因此需要首先创建一张数据表，创建 SQL 语句如下：

MYSQL、MSSQLServer、DB2:

```
CREATE TABLE T_Person (FNumber VARCHAR(20),
FName VARCHAR(20),FAge INT)
```

Oracle:

```
CREATE TABLE T_Person (FNumber VARCHAR2(20),
FName VARCHAR2(20),FAge NUMBER (10))
```

请在不同的数据库系统中运行相应的 SQL 语句。T_Person 为记录人员信息的数据表，其中字段 FNumber 为人员的编号，FName 为人员姓名，FAge 为人员年龄。

索引是针对字段的，因此创建索引索引的时候需要指定要在那个字段上创建索引，还可以为多个字段创建一个索引，这样还可以指定索引相关的字段列表。创建索引的 SQL 语句是 CREATE INDEX，其语法如下：

CREATE INDEX 索引名 ON 表名(字段 1, 字段 2,……字段 n)

其中【索引名】为被创建的索引的名称，这个名称必须是唯一的；【表名】为要创建索引的表；【字段 1, 字段 2,……字段 n】为组成这个索引的字段列表，允许一到多个。

下面的 SQL 语句在 T_Person 表的 FName 字段上创建索引，索引名为 idx_person_name：

```
CREATE INDEX  idx_person_name ON T_Person(FName)
```

下面的 SQL 语句在 T_Person 表的 FName 和 FAge 字段上创建索引，索引名为 idx_person_nameage：

```
CREATE INDEX idx_person_nameage ON T_Person(FName,FAge)
```

索引创建后是可以被删除的，删除索引使用的语句为 DROP INDEX。不同的数据库系统的 DROP INDEX 语法是不同的，下面分别介绍：

MYSQL 中的 DROP INDEX 语法如下：

DROP INDEX 索引名 ON 表名

比如下面的 SQL 语句用来删除刚才我们创建了两个索引：

```
DROP INDEX idx_person_name ON T_Person;
DROP INDEX idx_person_nameage ON T_Person;
```

MSSQLServer 中的 DROP INDEX 语法如下：

DROP INDEX 表名.索引名

比如下面的 SQL 语句用来删除刚才我们创建了两个索引：

DROP INDEX T_Person.idx_person_name;

DROP INDEX T_Person.idx_person_nameage;

Oracle 和 DB2 中的 DROP INDEX 语句不要求指定表名，只要指定索引名即可，语法如下：

DROP INDEX 索引名

比如下面的 SQL 语句用来删除刚才我们创建了两个索引：

DROP INDEX idx_person_name;

DROP INDEX idx_person_nameage;

到这里，索引相关的知识点就介绍完毕了，请执行下面的 SQL 语句删除刚才创建的 T_Person 表：

DROP TABLE T_Person;

6.2 约束

使用 CREATE TABLE 语句创建数据表的时候，通过定义一个字段的类型，我们规范了一个字段所能存储的数据类型，但是在有的情况下这种类型的约束是远远不够的。加入一个用户向 T_Person 表中录入数据，由于疏忽它录入的数据中有一条记录中的 FName 字段忘记了填入数据，这就造成了一个没有名字的人员的出现，即数据库中的数据遭到了污染。

可以在宿主程序中通过应用逻辑来保证数据的正确性，比如在用户点击【保存】按钮的时候去校验是否录入了“人员名称”，如果没有录入则提示用户“人员名称不能为空!”。这样可以保证绝大多数情况下的数据的正确性，但是在如下几种情况下仍然无法保证数据的正确性：宿主程序中存在 Bug，导致不正确的数据被保存到了数据表中；有一定技术条件的用户跳过宿主程序，直接修改数据库破坏数据的正确性。因此需要数据库系统提供指定数据表中数据约束条件的机制，这样校验在数据库系统这一最终层面来完成，保证了数据万无一失的正确性，而且在数据库系统中比在宿主程序中的校验更加高效。

数据库系统中主要提供了如下几种约束：非空约束；唯一约束； CHECK 约束；主键约束；外键约束。本节中将会对这些约束做一一介绍。

6.2.1 非空约束

在定义数据表的时候，默认情况下所有字段都是允许为空值的，如果需要禁止字段为空，那么就需要在创建表的时候显示指定。指定一个字段为空的方式就是在字段定义后增加 NOT NULL，比如下面的 SQL 语句创建了表 T_Person，并且设置 FNumber 字段不允许为空：

MYSQL,MSSQLServer,DB2:

CREATE TABLE T_Person (FNumber VARCHAR(20) NOT NULL ,FName VARCHAR(20),FAge INT)

Oracle:

CREATE TABLE T_Person (FNumber VARCHAR2(20) NOT NULL ,FName VARCHAR2(20),FAge NUMBER (10))

创建 T_Person 表后我们执行下面的 SQL 语句进行测试：

```
INSERT INTO T_Person (FNumber, FName, FAge) VALUES ( NULL , 'kingchou', 20)
```

因为在定义 T_Person 表的时候设定字段 FNumber 不能为空，而这个 SQL 语句中将 FNumber 字段设置为 NULL，所以在数据库中执行此 SQL 语句后数据库会报出下面错误信息：

```
不能将值 NULL 插入列 'FNumber'，表 'demo.dbo.T_Person'；列不允许有空值。INSERT 失败。
```

而下面的 SQL 语句则可以正确的执行：

```
INSERT INTO T_Person (FNumber, FName, FAge) VALUES ( '1' , 'kingchou', 20)
```

非空约束不仅对通过 INSERT 语句插入的数据起作用，而且对于使用 UPDATE 语句进行更新时也起作用。执行下面的 SQL 语句尝试将刚才插入的那条数据的 FNumber 字段更新为 NULL：

```
UPDATE T_Person SET FNumber = NULL
```

在数据库中执行此 SQL 语句后数据库会报出下面错误信息：

```
不能将值 NULL 插入列 'FNumber'，表 'demo.dbo.T_Person'；列不允许有空值。UPDATE 失败。
```

非空约束的相关知识就介绍到这里，请执行下面的 SQL 语句将 T_Person 表删除：

```
DROP TABLE T_Person
```

6.2.2 唯一约束

唯一约束又称为 UNIQUE 约束，它用于防止一个特定的列中两个记录具有一致的值，比如在员工信息表中希望防止两个或者多个人具有相同的身份证号码。唯一约束分为单字段唯一约束与复合唯一约束两种类型，下面分别介绍。

如果希望一个字段在表中的值是唯一的，那么就可以将唯一约束设置到这个字段上，设置方式就是在字段定义后增加 UNIQUE，如果是 DB2，那么还要同时将 NOT NULL 约束设置到这个字段上。下面的 SQL 语句创建了表 T_Person，并且将唯一约束设置到 FNumber 字段上：

MYSQL、MSSQLServer：

```
    CREATE TABLE T_Person (FNumber VARCHAR(20) UNIQUE,
    FName VARCHAR(20),FAge INT)
```

Oracle：

```
    CREATE TABLE T_Person (FNumber VARCHAR2(20)    UNIQUE,
    FName VARCHAR2(20),FAge NUMBER (10))
```

DB2：

```
    CREATE TABLE T_Person (FNumber VARCHAR(20) NOT NULL UNIQUE,
    FName VARCHAR(20),FAge INT)
```

创建 T_Person 表后我们执行下面的 SQL 语句向数据库中插入初始的一些测试数据：

```
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '1' , 'kingchou', 20);
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '2' , 'stef', 22);
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '3' , 'long', 26);
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '4' , 'yangzk', 27);
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '5' , 'beansoft', 26);
```

执行完毕后就能在表 T_Person 中的看到下面的数据：

FNUMBER	FNAME	FAGE
1	kingchou	20
2	stef	22
3	long	26
4	yangzk	27
5	beansoft	26

接着执行下面的 SQL 语句进行测试：

```
INSERT INTO T_Person (FNumber, FName, FAge) VALUES ( '2' , 'kitty', 20)
```

在数据库中执行此 SQL 语句后数据库会报出下面错误信息：

违反了 UNIQUE KEY 约束 'UQ__T_Person__1A14E395'。不能在对象 'dbo.T_Person' 中插入重复键。

单字段唯一约束的相关知识就介绍到这里，请执行下面的 SQL 语句将 T_Person 表删除：

```
DROP TABLE T_Person
```

唯一约束可以添加到多个字段中，也就是一张表中的唯一约束可以有不止一个，但是这样的单字段唯一约束只能约束“字段 A 的值在表中不重复”、“字段 B 的值在表中不重复”等，却不能约束“字段 A 的值在表中可以重复，字段 B 的值在表中也可以重复，但是不能存在字段 A 的值和字段 B 的值同时重复的记录”，这种约束也是有应用场景的：公司中每个部门单独进行工号编号，而且每个部门拥有唯一的部门编号，这样每个员工所属的部门编号是可以在表内重复的，而且每个员工的工号也是可以在表内重复的，但是不能存在所属的部门编号和每个员工的工号同时重复的员工。复合唯一约束是建立在多个字段上的约束，被约束的字段在不能同时重复。

定义复合唯一约束需要定义在所有字段列表之后，语法如下：

```
CONSTRAINT 约束名 UNIQUE(字段 1,字段 2……字段 n)
```

这里的“字段 1,字段 2……字段 n”为组成约束的多个字段，如果只有一个字段则可以看做是单字段唯一约束定义的另外一种形式。通过这种形式定义的唯一约束由于有一个确定的名称，所以可以很容易的通过这个名字来删除这个约束。

下面的 SQL 语句创建了表 T_Person，并且将在部门编号字段 FDepartmentNumber 和工号字段 FNumber 上设置复合唯一约束，并且命名为 unic_dep_num：

MYSQL、MSSQLServer：

```
CREATE TABLE T_Person (FNumber VARCHAR(20),
FDepartmentNumber VARCHAR(20),
FName VARCHAR(20),FAge INT,
CONSTRAINT unic_dep_num UNIQUE(FNumber,FDepartmentNumber))
```

Oracle：

```
CREATE TABLE T_Person (FNumber VARCHAR2(20),
FDepartmentNumber VARCHAR(20),
FName VARCHAR2(20),FAge NUMBER (10),
CONSTRAINT unic_dep_num UNIQUE(FNumber,FDepartmentNumber))
```

DB2：

```
CREATE TABLE T_Person (FNumber VARCHAR(20) NOT NULL,
FDepartmentNumber VARCHAR(20) NOT NULL,
FName VARCHAR(20),FAge INT,
CONSTRAINT unic_dep_num UNIQUE(FNumber,FDepartmentNumber))
```

创建 T_Person 表后我们执行下面的 SQL 语句向数据库中插入初始的一些测试数据：

```
INSERT INTO T_Person (FNumber, FDepartmentNumber,FName, FAge)
VALUES ( '1' , 'dev001','kingchou', 20);
INSERT INTO T_Person (FNumber, FDepartmentNumber,FName, FAge)
VALUES ( '2' , 'dev001', 'stef', 22);
INSERT INTO T_Person (FNumber, FDepartmentNumber,FName, FAge)
VALUES ( '1' , 'sales001',  'long', 26);
INSERT INTO T_Person (FNumber, FDepartmentNumber,FName, FAge)
VALUES ( '2' , 'sales001', 'yangzk', 27);
INSERT INTO T_Person (FNumber, FDepartmentNumber,FName, FAge)
VALUES ( '3' , 'sales001', 'beansoft', 26);
```

执行完毕后就能在表 T_Person 中的看到下面的数据：

FNumber	FDepartmentNumber	FName	FAge
1	dev001	kingchou	20
2	dev001	stef	22
1	sales001	long	26
2	sales001	yangzk	27
3	sales001	beansoft	26

可以看到 FNumber 和 FDepartmentNumber 字段的值在表中都有重复的值，但是没有这两个字段同时重复的值，如果这两个字段同时重复的话执行就会失败，执行下面的 SQL 语句来验证一下：

```
INSERT INTO T_Person (FNumber, FDepartmentNumber,FName, FAge)
VALUES ( '2' , 'sales001', 'daxia', 30);
```

因为 FNumber 等于'2'且 FDepartmentNumber 等于'sales001'的记录在表中已经存在了，所以在数据库中执行此 SQL 语句后数据库会报出下面错误信息：

违反了 UNIQUE KEY 约束 'unic_dep_num'。不能在对象 'dbo.T_Person' 中插入重复键。

为了运行后面的例子，请首先将表 T_Person 删除：DROP TABLE T_Person。

可以在一个表中添加多个复合唯一约束，只要为它们指定不同的名称即可。下面的 SQL 语句创建表 T_Person，并且为字段 FNumber 和 FDepartmentNumber 创建一个复合唯一约束以及为 FDepartmentNumber 和 FName 创建一个复合唯一约束：

MYSQL、MSSQLServer：

```
CREATE TABLE T_Person (FNumber VARCHAR(20),
FDepartmentNumber VARCHAR(20),
FName VARCHAR(20),FAge INT,
CONSTRAINT unic_1 UNIQUE(FNumber,FDepartmentNumber) ,
CONSTRAINT unic_2 UNIQUE(FDepartmentNumber, FName))
```

Oracle：

```
CREATE TABLE T_Person (FNumber VARCHAR2(20),
FDepartmentNumber VARCHAR(20),
FName VARCHAR2(20),FAge NUMBER (10) ,
CONSTRAINT unic_1 UNIQUE(FNumber,FDepartmentNumber) ,
CONSTRAINT unic_2 UNIQUE(FDepartmentNumber, FName))
```

DB2：

```
CREATE TABLE T_Person (FNumber VARCHAR(20) NOT NULL,
FDepartmentNumber VARCHAR(20) NOT NULL,
```

FName VARCHAR(20) NOT NULL,FAge INT NOT NULL,
CONSTRAINT unic_1 UNIQUE(FNumber,FDepartmentNumber) ,
CONSTRAINT unic_2 UNIQUE(FDepartmentNumber，FName))

到目前为止，我们已经讲了如何在创建数据表的时候创建唯一约束了，可是有时我们需要在已经创建好的数据表上添加新的唯一约束，这时就需要使用 ALTER TABLE 语句了，使用它我们可以为一张已经存在的数据表添加新的约束，语法如下：

ALTER TABLE 表名 ADD CONSTRAINT 唯一约束名 UNIQUE(字段 1,字段 2……字段 n)

比如下面的 SQL 语句为 T_Person 表添加一个建立在字段 FName 和字段 FAge 上的新的唯一约束：

ALTER TABLE T_Person ADD CONSTRAINT unic_3 UNIQUE(FName, FAge)

同样 ALTER TABLE 语句我们也可以删除已经创建好的复合唯一约束，语法如下：

ALTER TABLE 表名 DROP CONSTRAINT 唯一约束名

不过上边的语法不能在 MYSQL 中执行，MYSQL 中删除约束的语法为：

ALTER TABLE 表名 DROP INDEX 唯一约束名

比如下面的 SQL 语句将刚才创建的三个复合唯一约束删除：

MSQLServer、Oracle、DB2：

ALTER TABLE T_Person DROP CONSTRAINT unic_1;
ALTER TABLE T_Person DROP CONSTRAINT unic_2;
ALTER TABLE T_Person DROP CONSTRAINT unic_3;

MYSQL：

ALTER TABLE T_Person DROP INDEX unic_1;
ALTER TABLE T_Person DROP INDEX unic_2;
ALTER TABLE T_Person DROP INDEX unic_3;

现在可以删除 T_Person 表了：

DROP TABLE T_Person;

6.2.3 CHECK 约束

CHECK 约束会检查输入到记录中的值是否满足一个条件，如果不满足这个条件则对数据库做的修改不会成功。比如，一个人的年龄是不可能为负数的，一个人的入学日期不可能早于出生日期，出厂月份不可能大于 12。可以在 CHECK 条件中使用任意有效的 SQL 表达式，CHECK 约束对于插入、更新等任何对数据进行变化的操作都进行检查。

在字段定义后添加 CHECK 表达式就可以为这个字段添加 CHECK 约束，几乎所有字段中都可以添加 CHECK 约束，也就是一张表中可以存在多个 CHECK 约束。

下面的 SQL 语句创建了一张用于保存人员信息的表 T_Person，其中字段 FNumber 为人员编号，字段 FName 为人员姓名，字段 FAge 为人员年龄，字段 FWorkYear 为人员工龄：

MYSQL,MSSQLServer,DB2:

CREATE TABLE T_Person (
FNumber VARCHAR(20),FName VARCHAR(20),
FAge INT CHECK(FAge >0),
FWorkYear INT CHECK(FWorkYear>0))

Oracle:

CREATE TABLE T_Person (

FNumber VARCHAR2(20),FName VARCHAR2(20),
FAge NUMBER (10) CHECK(FAge >0),
FWorkYear NUMBER (10) CHECK(FWorkYear>0))

一个人的年龄和工龄显然不应该为负值的，所以为 FAge 和 FWorkYear 两个字段增加了 CHECK 约束“FAge >0”和“FWeight>0”。表创建完毕后执行下面的 SQL 语句进行测试：

INSERT INTO T_Person(FNumber, FName, FAge, FWorkYear)
VALUES('001','John',25,-3)

因为这里将 FWorkYear 字段设置成了-3，这是违反“CHECK(FWorkYear>0)”这个 CHECK 约束，所以在数据库中执行此 SQL 语句后数据库会报出下面错误信息：

INSERT 语句与 CHECK 约束"CK__T_Person__FWorkY__24927208"冲突。该冲突发生于数据库"demo"，表"dbo.T_Person", column 'FWorkYear'。

而执行下面的 SQL 语句则可以成功执行：

INSERT INTO T_Person(FNumber, FName, FAge, FWorkYear)
VALUES('001','John',25,3)

现在可以删除 T_Person 表了：

DROP TABLE T_Person;

除了可以在 CHECK 约束中使用常量表达式之外，还可以在 CHECK 约束中使用函数，比如人员编号长度要大于 12，那么就需要如下编写建表语句：

MYSQL,DB2:

```
CREATE TABLE T_Person (
FNumber VARCHAR(20) CHECK (LENGTH(FNumber)>12),
FName VARCHAR(20),
FAge INT CHECK(FAge >0),
FWorkYear INT CHECK(FWorkYear>0))
```

MSSQLServer:

```
CREATE TABLE T_Person (
FNumber VARCHAR(20) CHECK (LEN(FNumber)>12),
FName VARCHAR(20),
FAge INT CHECK(FAge >0),
FWorkYear INT CHECK(FWorkYear>0))
```

Oracle:

```
CREATE TABLE T_Person (
FNumber VARCHAR2(20) CHECK (LENGTH(FNumber)>12),
FName VARCHAR2(20),
FAge NUMBER (10) CHECK(FAge >0),
FWorkYear NUMBER (10) CHECK(FWorkYear>12))
```

表创建完毕后执行下面的 SQL 语句进行测试：

INSERT INTO T_Person(FNumber, FName, FAge, FWorkYear)
VALUES('001','John',25, 3)

因为这里将 FNumber 字段设置成了'001'，这是违反“CHECK (LENGTH(FNumber)>12)”这个 CHECK 约束的，所以在数据库中执行此 SQL 语句后数据库会报出下面错误信息：

INSERT 语句与 CHECK 约束"CK__T_Person__FNumbe__267ABA7A"冲突。该冲突发生于数据库"demo"，表"dbo.T_Person", column 'FNumber'。

而执行下面的 SQL 语句则可以成功执行：

INSERT INTO T_Person(FNumber, FName, FAge, FWorkYear)
VALUES('001001001001001','John',25,3)

现在可以删除 T_Person 表了：

DROP TABLE T_Person;

这种直接在列定义中通过 CHECK 子句添加 CHECK 约束的方式的缺点是约束条件不能引用其他列。比如我们想约束“人员的工龄必须小于他的年龄”，那么我们执行下面的 SQL 语句：

MYSQL,DB2:

```
CREATE TABLE T_Person (
FNumber VARCHAR(20),
FName VARCHAR(20),
FAge INT,
FWorkYear INT CHECK(FWorkYear< FAge))
```

MSSQLServer:

```
CREATE TABLE T_Person (
FNumber VARCHAR(20),
FName VARCHAR(20),
FAge INT,
FWorkYear INT CHECK(FWorkYear< FAge))
```

Oracle:

```
CREATE TABLE T_Person (
FNumber VARCHAR2(20),
FName VARCHAR2(20),
FAge NUMBER (10),
FWorkYear NUMBER (10) CHECK(FWorkYear< FAge))
```

执行这个 SQL 语句以后，数据库会报出如下的错误信息：

表 'T_Person' 的列 'FWorkYear' 的列 CHECK 约束引用了另一列。

出现这个错误的原因是因为在这种方式定义的 **CHECK** 子句中是不能引用其他列的，如果希望 **CHECK** 子句中的条件语句中使用其他列，则必须在 **CREATE TABLe** 语句的末尾使用 **CONSTRAINT** 关键字定义它。语法为：

CONSTRAINT 约束名 **CHECK(**约束条件**)**

重新编写上述的 **SQL** 语句，如下：

MYSQL,DB2:

```
CREATE TABLE T_Person (
FNumber VARCHAR(20),
FName VARCHAR(20),
FAge INT,
FWorkYear INT,
CONSTRAINT ck_1 CHECK(FWorkYear< FAge))
```

MSSQLServer:

```
CREATE TABLE T_Person (
FNumber VARCHAR(20),
FName VARCHAR(20),
```

FAge INT,
FWorkYear INT ,
CONSTRAINT ck_1 CHECK(FWorkYear< FAge))
Oracle:
CREATE TABLE T_Person (
FNumber VARCHAR2(20),
FName VARCHAR2(20),
FAge NUMBER (10),
FWorkYear NUMBER (10) ,
CONSTRAINT ck_1 CHECK(FWorkYear< FAge))
表创建完毕后执行下面的 SQL 语句进行测试：

INSERT INTO T_Person(FNumber, FName, FAge, FWorkYear)
VALUES('001','John',25, 30)

因为这里将 FWorkYear 字段设置成了 30，比如年龄 25 岁还大，这是违反“CHECK(FWorkYear< FAge)”这个 CHECK 约束的，所以在数据库中执行此 SQL 语句后数据库会报出下面错误信息：

INSERT 语句与 CHECK 约束"ck_1"冲突。该冲突发生于数据库"demo"，表"dbo.T_Person"。

而执行下面的 SQL 语句则可以成功执行：

INSERT INTO T_Person(FNumber, FName, FAge, FWorkYear)
VALUES('001001001001001','John',25,3)

可以看到，这种定义 CHECK 约束的方式几乎与定义一个复合唯一约束的方式一致。同样，可以通过 ALTER TABLE 的方式为已经存在的数据表添加 CHECK 约束。下面的 SQL 语句在 T_Person 上添加新的约束：

ALTER TABLE T_Person
ADD CONSTRAINT ck_2 CHECK(FAge>14)

上面的 SQL 语句中为约束指定了显式的名称，所以可以通过下面的 SQL 语句将 CHECK 约束 ck_2 删除（这个语句在 MYSQL 中无效）：

```
ALTER TABLE T_Person
DROP CONSTRAINT ck_2;
```

现在可以删除 T_Person 表了：

DROP TABLE T_Person;

6.2.4 主键约束

第一范式要求每张表都要有主键，因此主键约束是非常重要的，而且主键约束是外键关联的基础条件。主键约束为表之间的关联提供了链接点。

主键必须能够唯一标识一条记录，也就是主键字段中的值必须是唯一的，而且不能包含 NULL 值。从这种意义上来说，主键约束是 UNIQUE 约束和非空约束的组合。虽然一张表中可以有多个 UNIQUE 约束和非空约束，但是每个表中却只能有一个主键约束。

在 CREATE TABLE 语句中定义主键约束非常简单，和 UNIQUE 约束和非空约束非常类似，只要在字段定义后添加 PRIMARY KEY 关键字即可。不过在 DB2 中，主键列也必须显式的定义为 NOT NULL。下面的代码创建了员工信息表，并且将字段 FNumber 设置为主键字段：

MYSQL、MSSQLServer:

CREATE TABLE T_Person (FNumber VARCHAR(20) PRIMARY KEY,
FName VARCHAR(20),FAge INT)

Oracle:

CREATE TABLE T_Person (FNumber VARCHAR2(20) PRIMARY KEY,
FName VARCHAR2(20),FAge NUMBER (10))

DB2：

```
CREATE TABLE T_Person (FNumber VARCHAR(20) NOT NULL PRIMARY KEY,
FName VARCHAR(20),FAge INT)
```

创建完 T_Person 表后，请执行下面的 SQL 语句预置一些初始数据到 T_Person 表中：

```
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '1' , 'kingchou', 20);
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '2' , 'stef', 22);
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '3' , 'long', 26);
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '4' , 'yangzk', 27);
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '5' , 'beansoft', 26);
```

执行完毕后就能在表 T_Person 中的看到下面的数据：

FNUMBER	FNAME	FAGE
1	kingchou	20
2	stef	22
3	long	26
4	yangzk	27
5	beansoft	26

接着执行下面的 SQL 语句进行测试：

```
INSERT INTO T_Person (FNumber, FName, FAge)
VALUES ( '3' , 'sunny', 22);
```

由于表 T_Person 中已经存在 FNumber 等于 3 的值了，所以执行上边的 SQL 语句后数据库系统会报出如下的错误信息：

```
违反了 PRIMARY KEY 约束 'PK__T_Person__2E1BDC42'。不能在对象 'dbo.T_Person' 中插入重复键。
```

现在可以删除 T_Person 表了：

```
DROP TABLE T_Person;
```

除了这种由单一字段组成的主键之外，还可以由多个字段来组成主键，这样的主键被称为复合主键或者联合主键。复合主键的定义和复合唯一约束的定义类似，下面的 SQL 语句用来创建员工信息表，并且将字段 FNumber 和 FName 设置为复合主键：

MYSQL、MSSQLServer:

CREATE TABLE T_Person (FNumber VARCHAR(20),
FName VARCHAR(20),FAge INT,
CONSTRAINT pk_1 PRIMARY KEY(FNumber,FName))

Oracle:

CREATE TABLE T_Person (FNumber VARCHAR2(20)
FName VARCHAR2(20),FAge NUMBER (10) ,
CONSTRAINT pk_1 PRIMARY KEY(FNumber,FName))

DB2：

```
CREATE TABLE T_Person (FNumber VARCHAR(20) NOT NULL,
FName VARCHAR(20) NOT NULL,FAge INT,
CONSTRAINT pk_1 PRIMARY KEY(FNumber,FName))
```

现在可以删除 T_Person 表了：

```
DROP TABLE T_Person;
```

尽管在创建表的时候就定义主键是一个好的习惯，但是如果表创建了时候没有定义主键，那么也可以在以后添加主键，其添加方式与添加 UNIQUE 约束类似，也就是使用 ALTER TABLe 语句。不过通过这种方式添加主键的时候有一个附加条件，那就是组成主键的字段必须包含 NOT NULL 约束。如果在没有添加非空约束的字段上创建主键，系统将会爆出错误信息。

首先创建一个没有主键的 T_Person 表，注意其中的字段 FNumber 和 FName 添加了非空约束：

MYSQL、MSSQLServer:

```
CREATE TABLE T_Person (FNumber VARCHAR(20) NOT NULL,
FName VARCHAR(20) NOT NULL,FAge INT)
```

Oracle:

```
CREATE TABLE T_Person (FNumber VARCHAR2(20) NOT NULL,
FName VARCHAR2(20) NOT NULL,FAge NUMBER (10))
```

DB2：

```
CREATE TABLE T_Person (FNumber VARCHAR(20) NOT NULL,
FName VARCHAR(20) NOT NULL,FAge INT)
```

可以执行下面的 SQL 语句为 T_Person 创建主键约束：

```
ALTER TABLE T_Person
ADD CONSTRAINT pk_1 PRIMARY KEY(FNumber,FName)
```

最后删除主键约束的方式与删除 UNIQUE 约束以及 CHECK 约束的方式相同，只要使用带有 DROP 子句的 ALTER TABLE 语句即可：

```
ALTER TABLE T_Person
DROP CONSTRAINT pk_1;
```

这个语句在 MYSQL 中无效，在 MYSQL 中要执行下面的 SQL 语句才能删除主键：

```
ALTER TABLE T_Person
DROP PRIMARY KEY;
```

现在可以删除 T_Person 表了：

```
DROP TABLE T_Person;
```

6.2.5 外键约束

当一些信息在表中重复出现的时候，我们就要考虑要将它们提取到另外一张表中了，然后在源表中引用新创建的中的数据。比如很多作者都著有不止一本著作，所以在保存书籍信息的时候，应该把作者信息放到单独的表中，创建表的 SQL 语句如下：

MYSQL、MSSQLServer：

```
CREATE TABLE T_AUTHOR
(
    FId VARCHAR(20) PRIMARY KEY,
    FName VARCHAR(100),
```

```
    FAge INT,
    FEmail VARCHAR(20)
);
CREATE TABLE T_Book
(
    FId VARCHAR(20) PRIMARY KEY,
    FName VARCHAR(100),
    FPageCount INT,
    FAuthorId VARCHAR(20)
);
```

Oracle：

```
CREATE TABLE T_AUTHOR
(
    FId VARCHAR2(20) PRIMARY KEY,
    FName VARCHAR2(100),
    FAge NUMBER (10),
    FEmail VARCHAR2(20)
);
CREATE TABLE T_Book
(
    FId VARCHAR2(20) PRIMARY KEY,
    FName VARCHAR2(100),
    FPageCount NUMBER (10),
    FAuthorId VARCHAR2(20)
);
```

DB2：

```
CREATE TABLE T_AUTHOR
(
    FId VARCHAR(20) NOT NULL PRIMARY KEY,
    FName VARCHAR(100),
    FAge INT,
    FEmail VARCHAR(20)
);
CREATE TABLE T_Book
(
    FId VARCHAR(20) NOT NULL PRIMARY KEY,
    FName VARCHAR(100),
    FPageCount INT,
    FAuthorId VARCHAR(20)
);
```

表 T_AUTHOR 是作者信息表，FId 字段为主键，FName 字段为作者姓名，FAge 字段为作者年龄，FEmail 字段为作者 Email 地址；表 T_Book 是书籍信息表，FId 字段为主键，FName 字段为书名，FPageCount 字段为页数，FAuthorId 字段储存的对应的 T_AUTHOR 表中主键字段 FId 的值。

这里最重要的就是 T_Book 表的 FAuthorId 字段，它的值来自于 T_AUTHOR 表的 FId 字段。为了更清楚的表述这两个表的关系，我们插入一些演示数据，执行下面的 SQL 语句：

```
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('1','lily',20,'lily@cownew.com');
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('2','kingchou',23,'kingchou@cownew.com');
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('3','stef',28,'stef@cownew.com');
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('4','long',26,'long@cownew.com');
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('5','badboy',31,'badboy@cownew.com');

INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('1','About Java',300,'1');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('2','Inside Ruby',330,'2');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('3','Inside Curses',200,'5');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('4','Python In Action',450,'4');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('5','WPF Anywhere',250,'1');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('6','C# KickStart',280,'3');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('7','Compling',800,'1');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('8','Faster VB.Net',300,'5');
```

执行完毕查看这两张表中的内容，我们可以看到表 T_AUTHOR 中的内容如下：

FId	FName	FAge	FEmail
1	lily	20	lily@cownew.com
2	kingchou	23	kingchou@cownew.com
3	stef	28	stef@cownew.com
4	long	26	long@cownew.com
5	badboy	31	badboy@cownew.com

表 T_Book 中的内容如下：

FId	FName	FPageCount	FAuthorId
1	About Java	300	1
2	Inside Ruby	330	2
3	Inside Curses	200	5
4	Python In Action	450	4
5	WPF Anywhere	250	1
6	C# KickStart	280	3

7	Compling	800	1
8	Faster VB.Net	300	5

表 T_Book 的 FAuthorId 字段存储的是代表每个作者的主键值，如果要查询一本书的作者信息，那么只要按照 FAuthorId 字段的值到 T_AUTHOR 表中查询就可了。不过这样的表结构仍然存在问题，那就是不能约束表 T_Book 的 FAuthorId 字段中存储在表 T_AUTHOR 中不存在的值，比如执行下面的 SQL 语句向 T_Book 表中插入新数据：

```
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('9','About WinCE',320,'9');
```

执行完毕后就能在表 T_Book 中的看到下面的数据：

FId	FName	FPageCount	FAuthorId
1	About Java	300	1
2	Inside Ruby	330	2
3	Inside Curses	200	5
4	Python In Action	450	4
5	WPF Anywhere	250	1
6	C# KickStart	280	3
7	Compling	800	1
8	Faster VB.Net	300	5
9	About WinCE	320	9

表 T_Book 中最后一条数据的 FAuthorId 字段值为 9，但是在 T_AUTHOR 表中却没有主键值为 9 的记录，也就是这条记录引用了不存在的作者。

同样，这种结果也不能约束删除 T_AUTHOR 中已经被 T_Book 表引用的记录，比如我们执行下面的 SQL 语句删除 T_AUTHOR 表中的部分记录：

```
DELETE FROM T_AUTHOR
WHERE FAge>30
```

执行完毕后就能在表 T_AUTHOR 中的看到下面的数据：

FId	FName	FAge	FEmail
1	lily	20	lily@cownew.com
2	kingchou	23	kingchou@cownew.com
3	stef	28	stef@cownew.com
4	long	26	long@cownew.com

虽然表 T_Book 中《Inside Curses》这本书还引用着 badboy 这个作者，但是这个作者仍然被删除了，这样同样造成了 T_Book 表中引用了 T_AUTHOR 表中不存在的记录。

现在可以删除 T_AUTHOR 表和 T_Book 表了：

```
DROP TABLE T_AUTHOR;
DROP TABLE T_Book;
```

如何防止数据表之间的关系被破坏呢？SQL 提供的外键约束机制可以解决这个问题，它允许指定一个表中的一个列的值是另外一个表的外间，即一个表中的一个列是引用另外一个表中的记录。例如，可以设定 T_Book 表中的 FAuthorId 字段是一个依赖于 T_AUTHOR 表的 FId 列中存在的主键值。

我们可以在创建表的时候就添加外键约束，其定义方式和复合主键类似，语法如下：

```
FOREIGN KEY 外键字段 REFERENCES 外键表名(外键表的主键字段)
```

比如下面的 SQL 语句就是添加了外键约束的 T_AUTHOR 表和 T_Book 表的创建语句：

MYSQL、MSSQLServer：

```
CREATE TABLE T_AUTHOR
(
    FId VARCHAR(20) PRIMARY KEY,
    FName VARCHAR(100),
    FAge INT,
    FEmail VARCHAR(20)
);
CREATE TABLE T_Book
(
    FId VARCHAR(20) PRIMARY KEY,
    FName VARCHAR(100),
    FPageCount INT,
    FAuthorId VARCHAR(20) ,
    FOREIGN KEY (FAuthorId) REFERENCES T_AUTHOR(FId)

);
```

Oracle：

```
CREATE TABLE T_AUTHOR
(
    FId VARCHAR2(20) PRIMARY KEY,
    FName VARCHAR2(100),
    FAge NUMBER (10),
    FEmail VARCHAR2(20)
);
CREATE TABLE T_Book
(
    FId VARCHAR2(20) PRIMARY KEY,
    FName VARCHAR2(100),
    FPageCount NUMBER (10),
    FAuthorId VARCHAR2(20) ,
    FOREIGN KEY (FAuthorId) REFERENCES T_AUTHOR(FId)
);
```

DB2：

```
CREATE TABLE T_AUTHOR
(
    FId VARCHAR(20) NOT NULL PRIMARY KEY,
    FName VARCHAR(100),
    FAge INT,
    FEmail VARCHAR(20)
);
CREATE TABLE T_Book
(
    FId VARCHAR(20) NOT NULL PRIMARY KEY,
    FName VARCHAR(100),
```

```
    FPageCount INT,
    FAuthorId VARCHAR(20) ,
    FOREIGN KEY (FAuthorId) REFERENCES T_AUTHOR(FId)
);
```

我们插入一些演示数据，执行下面的 SQL 语句：

```
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('1','lily',20,'lily@cownew.com');
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('2','kingchou',23,'kingchou@cownew.com');
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('3','stef',28,'stef@cownew.com');
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('4','long',26,'long@cownew.com');
INSERT INTO T_AUTHOR(FId,FName,FAge,FEmail)
VALUES('5','badboy',31,'badboy@cownew.com');

INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('1','About Java',300,'1');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('2','Inside Ruby',330,'2');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('3','Inside Curses',200,'5');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('4','Python In Action',450,'4');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('5','WPF Anywhere',250,'1');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('6','C# KickStart',280,'3');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('7','Compling',800,'1');
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('8','Faster VB.Net',300,'5');
```

执行完毕查看这两张表中的内容，我们可以看到表 T_AUTHOR 中的内容如下：

FId	FName	FAge	FEmail
1	lily	20	lily@cownew.com
2	kingchou	23	kingchou@cownew.com
3	stef	28	stef@cownew.com
4	long	26	long@cownew.com
5	badboy	31	badboy@cownew.com

表 T_Book 中的内容如下：

FId	FName	FPageCount	FAuthorId
1	About Java	300	1
2	Inside Ruby	330	2
3	Inside Curses	200	5

4	Python In Action	450	4
5	WPF Anywhere	250	1
6	C# KickStart	280	3
7	Compling	800	1
8	Faster VB.Net	300	5

尝试向表中插入违反外键约束的数据，不如：

```
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('9','About WinCE',320,'9');
```

因为表 T_AUTHOR 中没有主键值等于 9 的记录，所以上面的 SQL 语句执行后数据库系统会报出如下的错误信息：

```
INSERT 语句与 FOREIGN KEY 约束"FK__T_Book__FAuthorI__38996AB5"冲突。该冲突发生于数据库"demo"，表"dbo.T_AUTHOR", column 'FId'。
```

我们修改上面的 SQL 语句，使其 FAuthorId 字段的值为在表 T_AUTHOR 中存在的主键值：

```
INSERT INTO T_Book(FId,FName,FPageCount,FAuthorId)
VALUES('9','About WinCE',320,'3');
```

执行完毕我们可以看到表 T_Book 中的内容如下：

FId	FName	FPageCount	FAuthorId
1	About Java	300	1
2	Inside Ruby	330	2
3	Inside Curses	200	5
4	Python In Action	450	4
5	WPF Anywhere	250	1
6	C# KickStart	280	3
7	Compling	800	1
8	Faster VB.Net	300	5
9	About WinCE	320	3

同样，我们不能删除被 T_Book 表引用的 T_AUTHOR 表中的数据，比如我们想执行下面的 SQL 语句将作者 long 从 T_AUTHOR 表中删除：

```
DELETE FROM T_AUTHOR
WHERE FName=' badboy'
```

因为《Inside Curses》、《Faster VB.Net》这两本书的作者为主键值等于 5 的作者 badboy，所以上面的 SQL 语句执行后数据库系统会报出如下的错误信息：

```
DELETE 语句与 REFERENCE 约束"FK__T_Book__FAuthorI__38996AB5"冲突。该冲突发生于数据库"demo"，表"dbo.T_Book", column 'FAuthorId'。
```

在删除一个表中的数据的时候，如果有其他的表中存在指向这些数据的外键关系，那么这个删除操作会是失败的，除非将所有的相关的数据。比如我们可以首先执行下面的 SQL 语句将作者 long 的所有著作删除：

```
DELETE FROM T_Book
WHERE FAuthorId =5;
```

然后就可以执行下面的 SQL 语句将作者 long 从 T_AUTHOR 表中删除了：

```
DELETE FROM T_AUTHOR
WHERE FName='badboy'
```

执行完成以后查看 T_AUTHOR 表中的内容，可以看到作者 badboy 已经被删除了：

FId	FName	FAge	FEmail

1	lily	20	lily@cownew.com
2	kingchou	23	kingchou@cownew.com
3	stef	28	stef@cownew.com
4	long	26	long@cownew.com

如果在创建表的时候没有添加外键约束，也可以使用 ALTER TABLE 语句添加外键约束，其语法与添加 UNIQUE 约束类似，比如下面的 SQL 语句就是以 ALTER TABLE 语句的方式添加外键约束：

```
ALTER TABLE T_Book
ADD CONSTRAINT fk_book_author
FOREIGN KEY (FAuthorId) REFERENCES T_AUTHOR(FId)
```

现在可以删除 T_AUTHOR 表和 T_Book 表了：

```
DROP TABLE T_Book;
DROP TABLE T_AUTHOR;
```

注意这里的删除顺序是首先删除 T_Book，再删除 T_AUTHOR，否则会因为违反外键约束而执行失败。

第七章 表连接

7.1 表连接的作用

7.2 表连接实战

7.2.1 INNER JOIN

7.2.2 LEFT JOIN

7.2.3 RIGHT JOIN

7.2.3 CROSS JOIN

第七章 表连接

到目前为止，我们讲解的数据查询都是针对单张数据表的，但是在真实的业务系统中，各个表之间都存在这种联系，很少存在不与其他表存在关联关系的表，而在实现业务功能的时候也经常需要从多个表中进行数据的检索，而进行多表检索最常用的技术就是表连接。因此本章将对表连接技术进行讲解。

为了更容易的运行本章中的例子，必须首先创建所需要的数据表，因此下面列出本章中要用到数据表的创建 SQL 语句：

MYSQL,MSSQLServer,DB2:

```
CREATE TABLE T_Customer (FId INT NOT NULL ,FName VARCHAR(20) NOT NULL ,
FAge INT,PRIMARY KEY (FId))
```

Oracle:

```
CREATE TABLE T_Customer (FId NUMBER (10) NOT NULL ,
FName VARCHAR2(20) NOT NULL ,FAge NUMBER (10),PRIMARY KEY (FId))
```

MYSQL,MSSQLServer,DB2:

```
CREATE TABLE T_OrderType (FId INT NOT NULL ,FName VARCHAR(20) NOT NULL,
PRIMARY KEY (FId))
```

Oracle:

```
CREATE TABLE T_OrderType (FId NUMBER (10) NOT NULL ,
FName VARCHAR2(20) NOT NULL,PRIMARY KEY (FId))
```

MYSQL,DB2:

CREATE TABLE T_Order (FId INT NOT NULL ,FNumber VARCHAR(20) NOT NULL ,
FPrice DECIMAL(10,2),FCustomerId INT,FTypeId INT,PRIMARY KEY (FId))

MSSQLServer:

CREATE TABLE T_Order (FId INT NOT NULL ,FNumber VARCHAR(20) NOT NULL ,
FPrice NUMERIC(10,2),FCustomerId INT, FTypeId INT,PRIMARY KEY (FId))

Oracle:

CREATE TABLE T_Order (FId NUMBER (10) NOT NULL ,
FNumber VARCHAR2(20) NOT NULL ,FPrice NUMERIC(10,2),
FCustomerId NUMBER (10), FTypeId INT,PRIMARY KEY (FId))

请在不同的数据库系统中运行相应的 SQL 语句。其中表 T_Customer 保存的是客户信息，FId 为主键、FName 为客户姓名、FAge 为客户年龄；表 T_OrderType 保存的是订单类型，FId 为主键、FName 为类型名；表 T_Order 为保存的是订单信息，FId 为主键、FNumber 为订单号、FPrice 为价格、FCustomerId 为客户的主键。

为了更加直观的验证本章中函数使用方法的正确性，我们需要在两张表中预置一些初始数据，请在数据库中执行下面的数据插入 SQL 语句：

```
INSERT INTO T_Customer(FId,FName,FAge)
VALUES(1,'TOM',21);
INSERT INTO T_Customer(FId,FName,FAge)
VALUES(2,'MIKE',24);
INSERT INTO T_Customer(FId,FName,FAge)
VALUES(3,'JACK',30);
INSERT INTO T_Customer(FId,FName,FAge)
VALUES(4,'TOM',25);
INSERT INTO T_Customer(FId,FName,FAge)
VALUES(5,'LINDA',NULL);

INSERT INTO T_OrderType(FId,FName)
VALUES(1,'MarketOrder');
INSERT INTO T_OrderType(FId,FName)
VALUES(2,'LimitOrder');
INSERT INTO T_OrderType(FId,FName)
VALUES(3,'Stop Order');
INSERT INTO T_OrderType(FId,FName)
VALUES(4,'StopLimit Order');

INSERT INTO T_Order(FId,FNumber,FPrice,FCustomerId, FTypeId)
VALUES(1,'K001',100,1,1);
INSERT INTO T_Order(FId,FNumber,FPrice,FCustomerId, FTypeId)
VALUES(2,'K002',200,1,1);
INSERT INTO T_Order(FId,FNumber,FPrice,FCustomerId, FTypeId)
VALUES(3,'T003',300,1,2);
INSERT INTO T_Order(FId,FNumber,FPrice,FCustomerId, FTypeId)
VALUES(4,'N002',100,2,2);
INSERT INTO T_Order(FId,FNumber,FPrice,FCustomerId, FTypeId)
```

VALUES(5,'N003',500,3,4);
INSERT INTO T_Order(FId,FNumber,FPrice,FCustomerId, FTypeId)
VALUES(6,'T001',300,4,3);
INSERT INTO T_Order(FId,FNumber,FPrice,FCustomerId, FTypeId)
VALUES(7,'T002',100,NULL,1);

初始数据预置完毕以后执行 SELECT * FROM T_Customer 查看 T_Customer 表中的数据，内容如下：

FId	FName	FAge
1	TOM	21
2	MIKE	24
3	JACK	30
4	TOM	25
5	LINDA	<NULL>

然后执行 `SELECT * FROM T_OrderType` 查看 `T_OrderType` 表中的数据，内容如下：

FId	FName
1	MarketOrder
2	LimitOrder
3	Stop Order
4	StopLimit Order

最后执行 SELECT * FROM T_Order 查看 T_Order 表中的数据，内容如下：

FId	FNumber	FPrice	FCustomerId	FTypeId
1	K001	100.00	1	1
2	K002	200.00	1	1
3	T003	300.00	1	2
4	N002	100.00	2	2
5	N003	500.00	3	4
6	T001	300.00	4	3
7	T002	100.00	<NULL>	1

7.1 表连接简介

使用本书目前所介绍的 SQL 知识，我们仅能够从一张数据表中检索数据，这在很多情况下是不能满足要求的，因为经常需要从多个表中进行检索才能得到想要的数据，SQL 中的“表连接”就是用来解决这个问题的。表连接使用 JOIN 关键字将一个或者多个表按照彼此间的关系连接为一个结果集。

表连接在 SQL 中的地位是非常重要的。假设需要查找姓名为 MIKE 的客户的订单号和票价。如果 SQL 不支持表连接，那么完成这个功能的第一个任务将是在 T_Customer 表中检索姓名为 MIKE 的客户的 FId 值：

```
SELECT FId
FROM T_Customer
WHERE FName='MIKE'
```

这个 SQL 语句返回 2，也就是姓名为 MIKE 的客户的 FId 值为 2，这样就可以到 T_Order 中检索 FCustomerId 等于 2 的记录：

```
SELECT FNumber,FPrice
FROM T_Order
WHERE FCustomerId=2
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FPrice
N002	100.00

由于这个功能比较简单，而且表中的数据也不复杂，所以使用这种分步处理的方式并没有感到有多么慢，但是如果 T_Customer 表有不止一个用户叫 MIKE 甚至有更多表参与检索的话使用这种方式不仅非常烦琐，而且会大大降低检索效率。使用 SQL 中的表连接则可以简化开发，并且由于数据库系统会对表连接进行查询优化，所以使用表连接进行数据的检索也会非常快速。

表连接就像连接两张数据表的连线，线的两端是分别在两张表的特定字段上的。在这里的例子中 T_Customer 表的 FId 字段和 T_Order 表的 FCustomerId 字段就是关联关系的两个端点。在使用表连接的时候必须显式的指定这个关联关系。

SQL 中使用 JOIN 关键字来使用表连接。表连接有多种不同的类型，被主流数据库系统支持的有交叉连接（CROSS JOIN）、内连接（INNER JOIN）、外连接（OUTTER JOIN），另外在有的数据库系统中还支持联合连接（UNION JOIN）。

7.2 内连接（INNER JOIN）

内连接组合两张表，并且基于两张表中的关联关系来连接它们。使用内连接需要指定表中哪些字段组成关联关系，并且需要指定基于什么条件进行连接。内连接的语法如下：

INNER JOIN table_name

ON condition

其中 table_name 为被关联的表名，condition 则为进行连接时的条件。

下面的 SQL 语句检索所有的客户姓名为 MIKE 的客户的订单号以及订单价格：

```
SELECT FNumber,FPrice
FROM T_Order INNER JOIN T_Customer
ON FCustomerId= T_Customer.FId
WHERE T_Customer.FName='TOM'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FPrice
K001	100.00
K002	200.00
T003	300.00
T001	300.00

在这个 SQL 语句中，首先列出了组成结果集所需要的列名，而后则是在 FROM 关键字后指定需要的表，在 INNER JOIN 关键字后指明要被连接的表，而在 ON 关键字后则指定了进行连接时所使用的条件。由于 T_Customer 和 T_Order 表中都有名称为 FId 的列，所以在 ON 关键字后的条件中使用 FId 字段的时候必须显示的指明这里使用 FId 字段属于哪个表。比如下面的 SQL 语句在执行的时候则会报出“列名 FId 不明确”的错误信息：

```
SELECT FNumber,FPrice
FROM T_Order INNER JOIN T_Customer
ON FCustomerId= FId
WHERE T_Customer.FName='TOM'
```

同样如果在 SELECT 语句后的字段列表中也不能存在有歧义的字段，比如下面的 SQL 语句执行会出错：

```
SELECT FId,FNumber,FPrice
FROM T_Order INNER JOIN T_Customer
```

```
ON FCustomerId= T_Customer.FId
WHERE T_Customer.FName='TOM'
```

必须为 FId 字段显式的指定所属的表，修正后的 SQL 语句如下：

```
SELECT T_Order.FId,FNumber,FPrice
FROM T_Order INNER JOIN T_Customer
ON FCustomerId= T_Customer.FId
WHERE T_Customer.FName='TOM'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FNumber	FPrice
1	K001	100.00
2	K002	200.00
3	T003	300.00
6	T001	300.00

为了避免列名歧义并且提高可读性，这里建议使用表连接的时候要显式列所属的表，如下：

```
SELECT T_Order.FId,T_Order.FNumber,T_Order.FPrice
FROM T_Order INNER JOIN T_Customer
ON T_Order.FCustomerId= T_Customer.FId
WHERE T_Customer.FName='TOM'
```

指定列所属的表后，我们就可以很轻松的引用同名的字段了，比如下面的 SQL 语句检索所有的订单以及它们对应的客户的相关信息：

```
SELECT T_Order.FId,T_Order.FNumber,T_Order.FPrice,
T_Customer.FId,T_Customer.FName,T_Customer.FAge
FROM T_Order INNER JOIN T_Customer
ON T_Order.FCustomerId= T_Customer.FId
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FNumber	FPrice	FId	FName	FAge
1	K001	100.00	1	TOM	21
2	K002	200.00	1	TOM	21
3	T003	300.00	1	TOM	21
4	N002	100.00	2	MIKE	24
5	N003	500.00	3	JACK	30
6	T001	300.00	4	TOM	25

在大多数数据库系统中，INNER JOIN 中的 INNER 是可选的，INNER JOIN 是默认的连接方式。也就是下面的 SQL 语句同样可以完成和检索所有的订单以及它们对应的客户的相关信息的功能：

```
SELECT T_Order.FId,T_Order.FNumber,T_Order.FPrice,
T_Customer.FId,T_Customer.FName,T_Customer.FAge
FROM T_Order JOIN T_Customer
ON T_Order.FCustomerId= T_Customer.FId
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FNumber	FPrice	FId	FName	FAge
1	K001	100.00	1	TOM	21
2	K002	200.00	1	TOM	21

3	T003	300.00	1	TOM	21
4	N002	100.00	2	MIKE	24
5	N003	500.00	3	JACK	30
6	T001	300.00	4	TOM	25

为了明确指定字段所属的表，上面的 SQL 语句中多次出现了 T_Order、T_Customer，当字段比较多的时候这样的 SQL 语句看起来非常繁杂，为此可以使用表别名来简化 SQL 语句的编写，比如下面的 SQL 语句就与上面的 SQL 语句是等价的：

```
SELECT o.FId,o.FNumber,o.FPrice,
c.FId,c.FName,c .FAge
FROM T_Order o JOIN T_Customer c
ON o.FCustomerId= c.FId
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FNumber	FPrice	FId	FName	FAge
1	K001	100.00	1	TOM	21
2	K002	200.00	1	TOM	21
3	T003	300.00	1	TOM	21
4	N002	100.00	2	MIKE	24
5	N003	500.00	3	JACK	30
6	T001	300.00	4	TOM	25

在使用表连接的时候可以不局限于只连接两张表，因为有很多情况下需要联系许多表。例如，我们需要检索每张订单的订单号、价格、客户姓名、订单类型等信息，由于客户信息和订单类型信息是保存在另外的表中的，因此需要同时连接 T_Customer 和 T_OrderType 两张表才能检索到所需要的信息，编写如下 SQL 语句即可：

```
INNER JOIN T_OrderType
ON T_Order.FTypeId= T_OrderType.FId
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FPrice	FName	FName
K001	100.00	TOM	MarketOrder
K002	200.00	TOM	MarketOrder
T003	300.00	TOM	LimitOrder
N002	100.00	MIKE	LimitOrder
N003	500.00	JACK	StopLimit Order
T001	300.00	TOM	Stop Order

7.3 不等值连接

到目前为止，本书中所有的连接几乎都是等值连接，也就是在这种连接的ON子句的条件包含一个等号运算。等值连接是最常用的连接，因为它指定的连接条件是一个表中的一个字段必须等于另一个表中的一个字段。

处理等值连接，还存在另外一种不等值连接，也就是在连接的条件中可以使用小于（<）、大于（>）、不等于（<>）等运算符，而且还可以使用LIKE、BETWEEN AND等运算符，甚至还可以使用函数。

例如，如果需要检索价格小于每个客户的年龄的五倍值的订单列表，那么就可以使用不

等值连接，实现的SQL语句如下所示：

```
SELECT T_Order.FNumber,T_Order.FPrice,
T_Customer.FName,T_Customer.FAge
FROM T_Order
INNER JOIN T_Customer
ON T_Order.FPrice< T_Customer.FAge*5
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FPrice	FName	FAge
K001	100.00	TOM	21
K001	100.00	MIKE	24
K001	100.00	JACK	30
K001	100.00	TOM	25
N002	100.00	TOM	21
N002	100.00	MIKE	24
N002	100.00	JACK	30
N002	100.00	TOM	25
T002	100.00	TOM	21
T002	100.00	MIKE	24
T002	100.00	JACK	30
T002	100.00	TOM	25

不等值连接产生了大量的查询结果，因为它是对被连接的两张表做了笛卡尔运算，所以如果只想查看与客户对应的订单，那么就要在不等值连接后添加等值连接匹配条件。实现的SQL语句如下：

```
SELECT T_Order.FNumber,T_Order.FPrice,
T_Customer.FName,T_Customer.FAge
FROM T_Order
INNER JOIN T_Customer
ON T_Order.FPrice< T_Customer.FAge*5
and T_Order.FCustomerId=T_Customer.FId
```

这里添加了“`and T_Order.FCustomerId=T_Customer.FId`”这个条件来限制匹配规则。执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FPrice	FName	FAge
K001	100.00	TOM	21
N002	100.00	MIKE	24

7.4 交叉连接

与内连接比起来，交叉连接非常简单，因为它不存在ON子句。交叉连接会将涉及到的所有表中的所有记录都包含在结果集中。可以采用两种方式来定义交叉连接，分别是隐式的和显式的。

隐式的连接只要在SELECT语句的FROM语句后将要进行交叉连接的表名列出即可，这种方式可以被几乎任意数据库系统支持。比如下面的SQL语句为将`T_Customer`表和`T_Order`做交叉连接：

```
SELECT T_Customer.FId, T_Customer.FName, T_Customer.FAge,
T_Order.FId, T_Order.FNumber, T_Order.FPrice
FROM T_Customer, T_Order
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FAge	FId	FNumber	FPrice
1	TOM	21	1	K001	100.00
1	TOM	21	2	K002	200.00
1	TOM	21	3	T003	300.00
1	TOM	21	4	N002	100.00
1	TOM	21	5	N003	500.00
1	TOM	21	6	T001	300.00
1	TOM	21	7	T002	100.00
2	MIKE	24	1	K001	100.00
2	MIKE	24	2	K002	200.00
2	MIKE	24	3	T003	300.00
2	MIKE	24	4	N002	100.00
2	MIKE	24	5	N003	500.00
2	MIKE	24	6	T001	300.00
2	MIKE	24	7	T002	100.00
3	JACK	30	1	K001	100.00
3	JACK	30	2	K002	200.00
3	JACK	30	3	T003	300.00
3	JACK	30	4	N002	100.00
3	JACK	30	5	N003	500.00
3	JACK	30	6	T001	300.00
3	JACK	30	7	T002	100.00
4	TOM	25	1	K001	100.00
4	TOM	25	2	K002	200.00
4	TOM	25	3	T003	300.00
4	TOM	25	4	N002	100.00
4	TOM	25	5	N003	500.00
4	TOM	25	6	T001	300.00
4	TOM	25	7	T002	100.00
5	LINDA	<NULL>	1	K001	100.00
5	LINDA	<NULL>	2	K002	200.00
5	LINDA	<NULL>	3	T003	300.00
5	LINDA	<NULL>	4	N002	100.00
5	LINDA	<NULL>	5	N003	500.00
5	LINDA	<NULL>	6	T001	300.00
5	LINDA	<NULL>	7	T002	100.00

在交叉连接中同样可以对表使用别名，比如上面的SQL语句来代替：

```
SELECT c.FId, c.FName, c.FAge,
o.FId, o.FNumber, o.FPrice
FROM T_Customer c, T_Order o
```

执行完毕我们就能在输出结果中看到下面的执行结果，可以看到执行结果与上面的一模一样：

FId	FName	FAge	FId	FNumber	FPrice
1	TOM	21	1	K001	100.00
1	TOM	21	2	K002	200.00
1	TOM	21	3	T003	300.00
1	TOM	21	4	N002	100.00
1	TOM	21	5	N003	500.00
1	TOM	21	6	T001	300.00
1	TOM	21	7	T002	100.00
2	MIKE	24	1	K001	100.00
2	MIKE	24	2	K002	200.00
2	MIKE	24	3	T003	300.00
2	MIKE	24	4	N002	100.00
2	MIKE	24	5	N003	500.00
2	MIKE	24	6	T001	300.00
2	MIKE	24	7	T002	100.00
3	JACK	30	1	K001	100.00
3	JACK	30	2	K002	200.00
3	JACK	30	3	T003	300.00
3	JACK	30	4	N002	100.00
3	JACK	30	5	N003	500.00
3	JACK	30	6	T001	300.00
3	JACK	30	7	T002	100.00
4	TOM	25	1	K001	100.00
4	TOM	25	2	K002	200.00
4	TOM	25	3	T003	300.00
4	TOM	25	4	N002	100.00
4	TOM	25	5	N003	500.00
4	TOM	25	6	T001	300.00
4	TOM	25	7	T002	100.00
5	LINDA	<NULL>	1	K001	100.00
5	LINDA	<NULL>	2	K002	200.00
5	LINDA	<NULL>	3	T003	300.00
5	LINDA	<NULL>	4	N002	100.00
5	LINDA	<NULL>	5	N003	500.00
5	LINDA	<NULL>	6	T001	300.00
5	LINDA	<NULL>	7	T002	100.00

交叉连接的显式定义方式为使用CROSS JOIN关键字，其语法与INNER JOIN类似，比如下面的SQL将T_Customer表和T_Order做交叉连接：

```
SELECT T_Customer.FId, T_Customer.FName, T_Customer.FAge,
T_Order.FId, T_Order.FNumber, T_Order.FPrice
FROM T_Customer
CROSS JOIN T_Order
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FAge	FId	FNumber	FPrice
1	TOM	21	1	K001	100.00
2	MIKE	24	1	K001	100.00
3	JACK	30	1	K001	100.00
4	TOM	25	1	K001	100.00
5	LINDA	<NULL>	1	K001	100.00
1	TOM	21	2	K002	200.00
2	MIKE	24	2	K002	200.00
3	JACK	30	2	K002	200.00
4	TOM	25	2	K002	200.00
5	LINDA	<NULL>	2	K002	200.00
1	TOM	21	3	T003	300.00
2	MIKE	24	3	T003	300.00
3	JACK	30	3	T003	300.00
4	TOM	25	3	T003	300.00
5	LINDA	<NULL>	3	T003	300.00
1	TOM	21	4	N002	100.00
2	MIKE	24	4	N002	100.00
3	JACK	30	4	N002	100.00
4	TOM	25	4	N002	100.00
5	LINDA	<NULL>	4	N002	100.00
1	TOM	21	5	N003	500.00
2	MIKE	24	5	N003	500.00
3	JACK	30	5	N003	500.00
4	TOM	25	5	N003	500.00
5	LINDA	<NULL>	5	N003	500.00
1	TOM	21	6	T001	300.00
2	MIKE	24	6	T001	300.00
3	JACK	30	6	T001	300.00
4	TOM	25	6	T001	300.00
5	LINDA	<NULL>	6	T001	300.00
1	TOM	21	7	T002	100.00
2	MIKE	24	7	T002	100.00
3	JACK	30	7	T002	100.00
4	TOM	25	7	T002	100.00
5	LINDA	<NULL>	7	T002	100.00

使用CROSS JOIN的方式声明的交叉连接只能被MYSQL、MSSQLServer和Oracle所支持，在DB2中是不被支持的。因为所有的数据库系统都支持隐式的交叉连接，所以它是执行交叉连接的最好方法。

7.5 自连接

到目前为止，我们讲解的连接都是在不同的数据表之间进行的，其实参与连接的表完全可以是同一样表，也就是表与其自身相连接，这样连接就被称为自连接。自连接并不是独立于交叉连接、内连接、外连接等这些连接方式之外的另外一种连接方式，而只是这些连接方

式的一种特例，也就是交叉连接、内连接、外连接等连接方式中只要参与连接的表是同一张表，那么它们就可以被称为自连接。

虽然大部分时间使用连接都是在连接不同的表，但是有的时候表也需要与自身连接，其主要用途就是检索一张表内部的匹配情况。下面就通过一个实例来演示自连接的使用。

假设需要检索与另外一个订单的订单类型一样的所有订单的列表。有的开发人员可能会写出下面的SQL语句：

```
SELECT FNumber,FPrice,FTypeId
FROM T_Order
WHERE FTypeId= FTypeId
```

执行以后我们在输出结果中看到下面的执行结果：

FNumber	FPrice	FTypeId
K001	100.00	1
K002	200.00	1
T003	300.00	2
N002	100.00	2
N003	500.00	4
T001	300.00	3
T002	100.00	1

这里显示出了T_Order表中的所有数据，而不是想像中的结果。因为这里的WHERE语句条件永远为真，因为向同行的相同列总是等于它自己，因此结果集中包含了表中的所有记录。

如果要实现要求的功能，可以假象存在另外一个与T_Order表完全相同的表，这样我们就可以在这两个表之间进行任意的连接操作了。我们尝试套用INNER JOIN的写法，只是将参与连接的两个表名都设置为T_Order，SQL语句如下：

```
SELECT FNumber,FPrice,FTypeId
FROM T_Order
INNER JOIN T_Order
ON T_Order.FTypeId=T_Order.FTypeId
```

这句SQL语句执行以后数据库系统会报出如下的错误信息：

```
FROM 子句中的对象 "T_Order" 和 "T_Order" 具有相同的表现名称。请使用相关名称来区分它们。
```

很显然，因为这里两次使用了T_Order表，但是数据库系统无法区分这两个T_Order表，因此必须为它们指定不同的别名，修改后的SQL语句如下：

```
SELECT o1.FNumber,o1.FPrice,o1.FTypeId,
o2.FNumber,o2.FPrice,o2.FTypeId
FROM T_Order o1
INNER JOIN T_Order o2
ON o1.FTypeId=o2.FTypeId
```

这里为T_Order表取了两个别名o1和o2，并且在引用表中列的时候也明确的指定了列属于那个表下的，这样数据库系统就能区分这两个别名代表的表了。使用别名以后我们可以将这两个别名看作结构相同、数据相同的两个不同的表，这样就可以避免思维上的障碍。

上边的SQL语句执行以后我们在输出结果中看到下面的执行结果：

FNumber	FPrice	FTypeId	FNumber	FPrice	FTypeId
K001	100.00	1	K001	100.00	1
K002	200.00	1	K001	100.00	1

T002	100.00	1	K001	100.00	1
K001	100.00	1	K002	200.00	1
K002	200.00	1	K002	200.00	1
T002	100.00	1	K002	200.00	1
T003	300.00	2	T003	300.00	2
N002	100.00	2	T003	300.00	2
T003	300.00	2	N002	100.00	2
N002	100.00	2	N002	100.00	2
N003	500.00	4	N003	500.00	4
T001	300.00	3	T001	300.00	3
K001	100.00	1	T002	100.00	1
K002	200.00	1	T002	100.00	1
T002	100.00	1	T002	100.00	1

这个SQL语句执行成功，没有语法错误，它是一个有效的自连接，不过它执行所产生的结果却不是正确的。比如第一行中“订单号为K001的订单与订单号为K001的订单的订单类型相同”，自己的订单类型当然与自己相同，这当然是正确的，可是这样的结果对我们来说是没有意义的。ON子句中指定两个表的FTypeId字段必须相同，当然对于同一个订单来说，它们肯定是相同的，而这里真正要查询的是具有相同的FTypeId字段值的两个不同的订单，因此需要在连接条件中添加一个新的条件，修改后的SQL语句如下：

```
SELECT o1.FNumber,o1.FPrice,o1.FTypeId,
o2.FNumber,o2.FPrice,o2.FTypeId
FROM T_Order o1
INNER JOIN T_Order o2
ON o1.FTypeId=o2.FTypeId and o1.FId<>o2.FId
```

ON子句末端添加的新条件“`and o1.FId<>o2.FId`”检查了别名为o1的表的主键不等于名为o2的表的主键，因为主键是唯一的，所以这样就可以确保得到的是一个不同的订单，从而不包含同一张订单。这个SQL语句执行以后我们在输出结果中看到下面的执行结果：

FNUMBER	FPRICE	FTYPEID	FNUMBER	FPRICE	FTYPEID
T002	100	1	K001	100	1
K002	200	1	K001	100	1
T002	100	1	K002	200	1
K001	100	1	K002	200	1
N002	100	2	T003	300	2
T003	300	2	N002	100	2
K002	200	1	T002	100	1
K001	100	1	T002	100	1

可以看到执行结果中已经去掉了相同订单的匹配，但是仔细观察仍然会发现存在重复的行。比如第一行的最后一行。o1表中的T002订单与o2表中的K001订单匹配，然后o2表中的K001订单与o1表中的T002订单匹配，也就是说数据库系统把“A匹配B”与“B匹配A”看成了两个不同的匹配，而实质上它们只是方向不同的相同的匹配，因此需要防止出现这样相同的匹配结果。因为出现上面这种问题的原因是因为存在“A匹配B”与“B匹配A”这两个方向的匹配，那么我们只要破坏这种双向匹配就可以了，最简单的方式就是要求o1的表的主键值于o2的表的主键值。修改后的SQL语句如下：

```
SELECT o1.FNumber,o1.FPrice,o1.FTypeId,
```

```
o2.FNumber,o2.FPrice,o2.FTypeId
FROM T_Order o1
INNER JOIN T_Order o2
ON o1.FTypeId=o2.FTypeId and o1.FId<o2.FId
```

这里仅有的改变是ON子句中连接条件的最后面部分，其原来的形式是：

```
o1.FId<>o2.FId
```

这个ON子句仅仅应用于两个表中FId字段值不同的记录。只要o1表与o2表中的FId字段值不同，则记录就会被包含字结果集中，因此将导致重复，所以这里将ON子句的这个SQL片段替换为：

```
o1.FId<o2.FId
```

现在o1表的一个记录行仅仅在它的FId字段值小于o2表的一个记录行仅仅在它的FId字段值的时候，才出现在结果集中。这确保了一行数据仅出现在结果集中一次。

这个SQL语句执行以后我们在输出结果中看到下面的执行结果：

FNUMBER	FPRICE	FTYPEID	FNUMBER	FPRICE	FTYPEID
K001	100	1	K002	200	1
T003	300	2	N002	100	2
K002	200	1	T002	100	1
K001	100	1	T002	100	1

7.6 外部连接

内部连接要求组成连接的两个表必须具有匹配的记录，T_Order表中的数据如下：

FId	FNumber	FPrice	FCustomerId	FTypeId
1	K001	100.00	1	1
2	K002	200.00	1	1
3	T003	300.00	1	2
4	N002	100.00	2	2
5	N003	500.00	3	4
6	T001	300.00	4	3
7	T002	100.00	<NULL>	1

使用内部连接可以查询每张订单的订单号、价格、对应的客户姓名以及客户年龄，SQL语句如下：

```
SELECT o.FNumber,o.FPrice,o.FCustomerId,
c.FName,c.FAge
FROM T_Order o
INNER JOIN T_Customer c
ON o.FCustomerId=c.FId
```

执行以后我们在输出结果中看到下面的执行结果：

FNumber	FPrice	FCustomerId	FName	FAge
K001	100.00	1	TOM	21
K002	200.00	1	TOM	21
T003	300.00	1	TOM	21
N002	100.00	2	MIKE	24
N003	500.00	3	JACK	30
T001	300.00	4	TOM	25

仔细观察我们可以看到T_Order表中有7行数据，而通过上面的内部连接查询出来的结果

只有6条，其中订单号为T002的订单没有显示到结果集中。这是因为订单号为T002的订单的FCustomerId字段值为空，显然是无法与T_Customer表中的任何行匹配了，所以它没有显示到结果集中。在一些情况下这种处理方式能够满足要求，但是有时我们要求无法匹配的NULL值也要显示到结果集中，比如“查询每张订单的订单号、价格、对应的客户姓名以及客户年龄，如果没有对应的客户，则在客户信息处显示空格”，希望的查询结果是这样的：

FNumber	FPrice	FCustomerId	FName	FAge
K001	100.00	1	TOM	21
K002	200.00	1	TOM	21
T003	300.00	1	TOM	21
N002	100.00	2	MIKE	24
N003	500.00	3	JACK	30
T001	300.00	4	TOM	25
T002	100.00	<NULL>	<NULL>	<NULL>

使用内部连接是很难达到这种效果的，可以使用外部连接来实现。外部连接主要就是用来解决这种空值匹配问题的。

外部连接的语法与内部连接几乎是一样的，主要区别就是对于空值的处理。外部连接不需要两个表具有匹配记录，这样可以指定某个表中的记录总是放到结果集中。根据哪个表中的记录总是放到结果集中，外部连接分为三种类型：右外部连接（RIGHT OUTER JOIN）、左外部连接（LEFT OUTER JOIN）和全外部连接（FULL OUTER JOIN）。

三者的共同点是都返回符合连接条件的数据，这一点是和内部连接是一样的，不同点在于它们对不符合连接条件的数据处理，三者不同点说明如下：

- 左外部连接还返回左表中不符合连接条件的数据；
- 左外部连接还返回右表中不符合连接条件的数据；
- 全外部连接还返回左表中不符合连接条件的数据以及右表中不符合连接条件的数据，它其实是左外部连接和左外部连接的合集。

这里的左表和右表是相对于JOIN关键字来说的，位于JOIN关键字左侧的表即被称为左表，而位于JOIN关键字右侧的表即被称为右表。比如：

```
SELECT o.FNumber,o.FPrice,o.FCustomerId,
c.FName,c.FAge
FROM T_Order o
INNER JOIN T_Customer c
ON o.FCustomerId=c.FId
```

这里的T_Order就是左表，而T_Customer则是右表。

7.6.1 左外部连接

在左外部连接中，左表中所有的记录都会被放到结果集中，无论是否在右表中存在匹配记录。比如下面的SQL语句用来实现“查询每张订单的订单号、价格、对应的客户姓名以及客户年龄，如果没有对应的客户，则在客户信息处显示空格”：

```
SELECT o.FNumber,o.FPrice,o.FCustomerId,
c.FName,c.FAge
FROM T_Order o
LEFT OUTER JOIN T_Customer c
ON o.FCustomerId=c.FId
```

执行以后我们在输出结果中看到下面的执行结果：

FNumber	FPrice	FCustomerId	FName	FAge

K001	100.00	1	TOM	21
K002	200.00	1	TOM	21
T003	300.00	1	TOM	21
N002	100.00	2	MIKE	24
N003	500.00	3	JACK	30
T001	300.00	4	TOM	25
T002	100.00	<NULL>	<NULL>	<NULL>

在T_Order表中有7条记录，其中最后一条不满足连接条件，但是也放到了结果集中，只是在不存在匹配条件的列中显示为NULL。

虽然左外部连接包含左表中的所有记录，但是它只提供出示的结果集，WHERE语句仍然会改变最终的结果集。比如为上面的SQL语句添加一个WHERE子句，使得结果中不包含价格小于150元的订单：

```
SELECT o.FNumber,o.FPrice,o.FCustomerId,
c.FName,c.FAge
FROM T_Order o
LEFT OUTER JOIN T_Customer c
ON o.FCustomerId=c.FId
WHERE o.FPrice>=150
```

执行以后我们在输出结果中看到下面的执行结果：

FNumber	FPrice	FCustomerId	FName	FAge
K002	200.00	1	TOM	21
T003	300.00	1	TOM	21
N003	500.00	3	JACK	30
T001	300.00	4	TOM	25

尽管左外部连接返回了T_Order表中的所有记录，但是由于WHERE语句的过滤，包括订单号为T002在内的所有价格小于150元的订单全部被排除在了结果集之外。

7.6.2 右外部连接

与左外部连接正好相反，在右外部连接中不管是否成功匹配连接条件都会返回右表中的所有记录。比如下面的SQL语句使用右外部连接查询每张订单的信息以及对应的客户信息：

```
SELECT o.FNumber,o.FPrice,o.FCustomerId,
c.FName,c.FAge
FROM T_Order o
RIGHT OUTER JOIN T_Customer c
ON o.FCustomerId=c.FId
```

执行以后我们在输出结果中看到下面的执行结果：

FNumber	FPrice	FCustomerId	FName	FAge
K001	100.00	1	TOM	21
K002	200.00	1	TOM	21
T003	300.00	1	TOM	21
N002	100.00	2	MIKE	24
N003	500.00	3	JACK	30
T001	300.00	4	TOM	25
<NULL>	<NULL>	<NULL>	LINDA	<NULL>

可以看到前6条记录都是符合连接条件的，而T_Customer表中姓名为LINDA的客户没有

对应的订单，但是仍然被放到了结果集中，其无法匹配的字段填充的都是NULL。

7.6.3 全外部连接

几乎所有的数据库系统都支持左外部连接和右外部连接，但是全外部连接则不是所有数据库系统都支持，特别是最常使用的MYSQL就不支持全外部连接。全外部连接是左外部连接和右外部连接的合集，因为即使在右表中不存在匹配连接条件的数据，左表中的所有记录也将被放到结果集中，同样即使在左表中不存在匹配连接条件的数据，右表中的所有记录也将被放到结果集中。

比如下面的SQL语句使用全外部连接查询每张订单的信息以及对应的客户信息：

```
SELECT o.FNumber,o.FPrice,o.FCustomerId,
c.FName,c.FAge
FROM T_Order o
FULL OUTER JOIN T_Customer c
ON o.FCustomerId=c.FId
```

执行以后我们在输出结果中看到下面的执行结果：

FNUMBER	FPRICE	FCUSTOMERID	FNAME	FAGE
K001	100.00	1	TOM	21
K002	200.00	1	TOM	21
T003	300.00	1	TOM	21
N002	100.00	2	MIKE	24
N003	500.00	3	JACK	30
T001	300.00	4	TOM	25
<NULL>	<NULL>	<NULL>	LINDA	<NULL>
T002	100.00	<NULL>	<NULL>	<NULL>

可以看到前6条记录都是符合连接条件的，而T_Customer表中姓名为LINDA的客户没有对应的订单，但是仍然被放到了结果集中，其无法匹配的字段填充的都是NULL，同样订单号为T002的订单没有匹配任何一个客户，但是仍然被放到了结果集中。

虽然在MYSQL中不支持全外部连接，不过由于全外部连接是左外部连接和右外部连接的合集，所以可以使用左外部连接和右外部连接来模拟实现全外部连接：使用左外部连接和右外部连接分别进行匹配查询，然后使用UNION运算符来取两个查询结果集的合集。比如可以在MYSQL中执行下面的SQL来实现T_Order表和T_Customer表的全外部连接：

```
SELECT o.FNumber,o.FPrice,o.FCustomerId,
c.FName,c.FAge
FROM T_Order o
LEFT OUTER JOIN T_Customer c
ON o.FCustomerId=c.FId
UNION
SELECT o.FNumber,o.FPrice,o.FCustomerId,
c.FName,c.FAge
FROM T_Order o
RIGHT OUTER JOIN T_Customer c
ON o.FCustomerId=c.FId
```

执行以后我们在输出结果中看到下面的执行结果：

FNUMBER	FPRICE	FCUSTOMERID	FNAME	FAGE
K001	100.00	1	TOM	21

K002	200.00	1	TOM	21
T003	300.00	1	TOM	21
N002	100.00	2	MIKE	24
T001	300.00	4	TOM	25
N003	500.00	3	JACK	30
<NULL>	<NULL>	<NULL>	LINDA	<NULL>
T002	100.00	<NULL>	<NULL>	<NULL>

可以看到和全外部连接的执行结果是完全一致的。

到这里本章即将结束，请执行下面的 SQL 语句将本章中用到的数据表删除：

```
DROP TABLE T_Order;
DROP TABLE T_OrderType;
DROP TABLE T_Customer;
```

第六章 子查询

6.1 IN 语句中的子查询

6.2 子查询用作计算列

6.3 ANY 与 SOME、EXISTS

6.4 使用子查询进行数据批量处理（Insert…Select）

6.5 子查询与组合查询

6.6 案例分析

6.6.1 历史表备份的两种迥异的方式

6.6.2 表间数据批处理的两种迥异的方式

6.6.3 查找欠债人的两种迥异的方式

6.6.4 进销存汇总报表

但返回值子查询可以看作是函数，使用=，+等运算符

第八章 子查询

SQL 语句允许将一个查询语句做为一个结果集供其他 SQL 语句使用，就像使用普通的表一样，被当作结果集的查询语句被称为子查询。所有可以使用表的地方几乎都可以使用子查询来代替，比如 SELECT * FROM T 中就可以用子查询来代替表 T，比如 SELECT * FROM(SELECT * FROM T2 where FAge<30)，这里的“SELECT * FROM T2 where FAge<30”就是子查询，可以将子查询看成一张暂态的数据表，这张表在查询开始时被创造，在查询结束时被删除。子查询大大简化了复杂 SQL 语句的编写，不过使用不当也容易造成性能问题。

子查询的语法与普通的 SELECT 语句语法相同，所有可以在普通 SELECT 语句中使用的特性都可以在子查询中使用，比如 WHERE 子句过滤、UNION 运算符、HAVING 子句、GROUPBY 子句、ORDER BY 子句，甚至在子查询中还可以包含子查询。同时，不仅可以在 SELECT 语句中使用子查询，还可以在 UPDATE、DELETE 等语句中使用子查询。

为了更容易的运行本章中的例子，必须首先创建所需要的数据表，因此下面列出本章中要用到数据表的创建 SQL 语句：

MYSQL,MSSQLServer,DB2:

CREATE TABLE T_Reader (FId INT NOT NULL ,FName VARCHAR(50),FYearOfBirth INT,FCity VARCHAR(50),FProvince VARCHAR(50), FYearOfJoin INT)

Oracle:

```
CREATE TABLE T_Reader (FId NUMBER (10) NOT NULL ,FName VARCHAR2(50),FYearOfBirth NUMBER (10),FCity VARCHAR2(50),FProvince VARCHAR2(50), FYearOfJoin NUMBER (10))

MYSQL,MSSQLServer,DB2:
CREATE TABLE T_Book (FId INT NOT NULL ,FName VARCHAR(50),FYearPublished INT,FCategoryId INT)
Oracle:
CREATE TABLE T_Book (FId NUMBER (10) NOT NULL ,FName VARCHAR2(50),FYearPublished NUMBER (10),FCategoryId NUMBER (10))
 MYSQL,MSSQLServer,DB2:
CREATE TABLE T_Category (FId INT NOT NULL ,FName VARCHAR(50))
Oracle:
CREATE TABLE T_Category (FId NUMBER (10) NOT NULL ,FName VARCHAR2(50))

MYSQL,MSSQLServer,DB2:
CREATE TABLE T_ReaderFavorite (FCategoryId INT,FReaderId INT)
Oracle:
CREATE TABLE T_ReaderFavorite (FCategoryId NUMBER (10),FReaderId NUMBER (10))
```

请在不同的数据库系统中运行相应的 SQL 语句。其中表 T_Reader 保存的是读者信息，FId 为主键、FName 为读者姓名、FYearOfBirth 为读者出生年份、FCity 为读者所在城市、FProvince 为读者所在省份、FYearOfJoin 为读者入会年份；表 T_Book 保存的是书籍信息，FId 为主键、FName 为书名、FYearPublished 为出版年份、FCategoryId 为所属分类；表 T_Category 保存的是分类信息，FId 为主键、FName 为分类名；表 T_ReaderFavorite 保存的是读者和读者喜爱的类别之间的对应关系，FCategoryId 为分类主键、FReaderId 为读者主键。

为了更加直观的验证本章中函数使用方法的正确性，我们需要在这几张表中预置一些初始数据，请在数据库中执行下面的数据插入 SQL 语句：

```
INSERT INTO T_Category(FId,FName)
VALUES(1,'Story');
INSERT INTO T_Category(FId,FName)
VALUES(2,'History');
INSERT INTO T_Category(FId,FName)
VALUES(3,'Theory');
INSERT INTO T_Category(FId,FName)
VALUES(4,'Technology');
INSERT INTO T_Category(FId,FName)
VALUES(5,'Art');
INSERT INTO T_Category(FId,FName)
VALUES(6,'Philosophy');

INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(1,'Tom',1979,'TangShan','Hebei',2003);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
```

```
VALUES(2,'Sam',1981,'LangFang','Hebei',2001);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(3,'Jerry',1966,'DongGuan','GuangDong',1995);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(4,'Lily',1972,'JiaXing','ZheJiang',2005);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(5,'Marry',1985,'BeiJing','BeiJing',1999);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(6,'Kelly',1977,'ZhuZhou','HuNan',1995);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(7,'Tim',1982,'YongZhou','HuNan',2001);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(8,'King',1979,'JiNan','ShanDong',1997);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(9,'John',1979,'QingDao','ShanDong',2003);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(10,'Lucy',1978,'LuoYang','HeNan',1996);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(11,'July',1983,'ZhuMaDian','HeNan',1999);
INSERT INTO T_Reader(FId,FName,FYearOfBirth,FCity,FProvince,FYearOfJoin)
VALUES(12,'Fige',1981,'JinCheng','ShanXi',2003);

INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(1,'About J2EE',2005,4);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(2,'Learning Hibernate',2003,4);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(3,'Two Cites',1999,1);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(4,'Jane Eyre',2001,1);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(5,'Oliver Twist',2002,1);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(6,'History of China',1982,2);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(7,'History of England',1860,2);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(8,'History of America',1700,2);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(9,'History of The World',2008,2);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(10,'Atom',1930,3);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(11,'RELATIVITY',1945,3);
```

```
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(12,'Computer',1970,3);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(13,'Astronomy',1971,3);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(14,'How To Singing',1771,5);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(15,'DaoDeJing',2001,6);
INSERT INTO T_Book(FId,FName,FYearPublished,FCategoryId)
VALUES(16,'Obedience to Authority',1995,6);

INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(1,1);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(5,2);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(2,3);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(3,4);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(5,5);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(1,6);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(1,7);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(4,8);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(6,9);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(5,10);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(2,11);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(2,12);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(1,12);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(3,1);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(1,3);
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
VALUES(4,4);
```

初始数据预置完毕以后执行 SELECT * FROM T_Reader 查看 T_Reader 表中的数据，内

容如下：

FId	FName	FYearOfBirth	FCity	FProvince	FYearOfJoin
1	Tom	1979	TangShan	Hebei	2003
2	Sam	1981	LangFang	Hebei	2001
3	Jerry	1966	DongGuan	GuangDong	1995
4	Lily	1972	JiaXing	ZheJiang	2005
5	Marry	1985	BeiJing	BeiJing	1999
6	Kelly	1977	ZhuZhou	HuNan	1995
7	Tim	1982	YongZhou	HuNan	2001
8	King	1979	JiNan	ShanDong	1997
9	John	1979	QingDao	ShanDong	2003
10	Lucy	1978	LuoYang	HeNan	1996
11	July	1983	ZhuMaDian	HeNan	1999
12	Fige	1981	JinCheng	ShanXi	2003

接着执行 SELECT * FROM T_Book 查看 T_Book 表中的数据，内容如下：

FId	FName	FYearPublished	FCategoryId
1	About J2EE	2005	4
2	Learning Hibernate	2003	4
3	Two Cites	1999	1
4	Jane Eyre	2001	1
5	Oliver Twist	2002	1
6	History of China	1982	2
7	History of England	1860	2
8	History of America	1700	2
9	History of The World	2008	2
10	Atom	1930	3
11	RELATIVITY	1945	3
12	Computer	1970	3
13	Astronomy	1971	3
14	How To Singing	1771	5
15	DaoDeJing	2001	6
16	Obedience to Authority	1995	6

接着执行 SELECT * FROM T_Category 查看 T_Category 表中的数据，内容如下：

FId	FName
1	Story
2	History
3	Theory
4	Technology
5	Art
6	Philosophy

最后执行 SELECT * FROM T_ReaderFavorite 查看 T_ReaderFavorite 表中的数据，内容如下：

FCategoryId	FReaderId
1	1
5	2
2	3
3	4
5	5
1	6
1	7
4	8
6	9
5	10
2	11
2	12
1	12
3	1
1	3
4	4

8.1 子查询入门

SELECT 语句可以嵌套在其他语句中,比如 SELECT,INSERT,UPDATE 以及 DELETE 等，这些被嵌套的 SELECT 语句就称为子查询，可以这么说当一个查询依赖于另外一个查询结果时就可以使用子查询。子查询有两种类型，一种是只返回一个单值的子查询，这时它可以用在一个单值可以使用的地方，这时子查询可以看作是一个拥有返回值的函数；另外一种是返回一列值的子查询，这时子查询可以看作是一个在内存中临时存在的数据表。

8.1.1 单值子查询

单值子查询的语法和普通的 SELECT 语句没有什么不同，唯一的限制就是子查询的返回值必须只有一行记录，而且只能有一个列。这样的子查询又被称为标量子查询，标量子查询可以用在 SELECT 语句的列表中、表达式中、WHERE 语句中等很多场合。

首先来看一个在 SELECT 语句列表中使用的最简单的标量子查询。SQL 语句如下：

MYSQL,MSSQLServer:

```
SELECT 1 AS f1,2,(SELECT MIN(FYearPublished) FROM T_Book),(SELECT MAX(FYearPublished) FROM T_Book) AS f4
```

Oracle:

```
SELECT 1 AS f1,2,(SELECT MIN(FYearPublished) FROM T_Book),(SELECT MAX(FYearPublished) FROM T_Book) AS f4 FROM DUAL
```

DB2:

```
SELECT 1 AS f1,2,(SELECT MIN(FYearPublished) FROM T_Book),(SELECT MAX(FYearPublished) FROM T_Book) AS f4 FROM SYSIBM.SYSDUMMY1
```

这个 SQL 语句一共返回四列，第一列是数字 1，第二列是数字 2，第三列则是一个标量子查询，它返回最早出版图书的年份，第四列也是一个标量子查询，它返回最晚出版图书的年份。这里完全可以将标量子查询当成一个普通的列，而且还可以为标量子查询列取一个别名。

执行完毕我们就能在输出结果中看到下面的执行结果：

f1			f4
1	2	1700	2008

如果一个子查询返回值不止一行记录或者有多个列的话都不能当作标量子查询使用，否则会出错。比如下面 SQL 语句是错的：

```
SELECT 1 AS f1,2,(SELECT FYearPublished FROM T_Book)
```

由于这句 SQL 语句中的子查询会返回多行记录，所以在执行的时候数据库会提示如下的错误信息：

子查询返回的值不止一个。当子查询跟随在 =、!=、<、<=、>、>= 之后，或子查询用作表达式时，这种情况是不允许的。

下面的 SQL 语句也是错误的：

```
SELECT 1 AS f1,2,(SELECT MAX(FYearPublished),MIN(FYearPublished) FROM T_Book)
```

由于这句 SQL 语句中的子查询会返回包含两列数据的结果集，所以在执行的时候数据库会提示如下的错误信息：

当没有用 EXISTS 引入子查询时，在选择列表中只能指定一个表达式。

上面举的例子都是在执行前就能确定是否正确的，但是有的时候一个子查询是否使用正确是要到运行时才能确定的。比如在数据库系统中执行下面的 SQL 语句：

```
SELECT 1 AS f1,2,(SELECT FYearPublished FROM T_Book where FYearPublished<2000)
```

在数据库系统中执行后这句 SQL 语句会报错，因为发行日期小于 2000 年的书有不止一本，所以子查询的返回结果集也就有不止一条记录。如果我们调整这句 SQL 语句，转而查询发行日期小于 1750 年的书，SQL 语句如下：

```
SELECT 1 AS f1,2,(SELECT FYearPublished FROM T_Book where FYearPublished<1750)
```

因为数据库中发行日期小于 1750 年的书恰恰只有一本，所以这句 SQL 语句能够执行成功，执行结果如下：

f1			
1	2	1700	

8.1.2 列值子查询

与标量子查询不同，列值子查询可以返回一个多行多列的结果集。这样的子查询又被称为表子查询，表子查询可以看作一个临时的表，表子查询可以用在 SELECT 语句的 FROM 子句中、INSERT 语句、连接、IN 子句等很多场合。

首先来看一个在 FROM 子句中使用的最简单的表子查询。SQL 语句如下：

```
SELECT T_Reader.FName,t2.FYearPublished,t2.FName
FROM T_Reader,
(SELECT * FROM T_Book WHERE FYearPublished < 1800) t2
```

这里将 T_Reader 表和表子查询做交叉连接，并且将“SELECT * FROM T_Book WHERE FYearPublished < 1800”做为表子查询，还可以为表子查询执行表别名，在 SELECT 的列表中也可以使用和表一样的列名饮用方式，这与使用一个普通的数据表没有什么区别。

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FYearPublished	FName
Tom	1700	History of America
Sam	1700	History of America
Jerry	1700	History of America
Lily	1700	History of America
Marry	1700	History of America

Kelly	1700	History of America
Tim	1700	History of America
King	1700	History of America
John	1700	History of America
Lucy	1700	History of America
July	1700	History of America
Fige	1700	History of America
Tom	1771	How To Singing
Sam	1771	How To Singing
Jerry	1771	How To Singing
Lily	1771	How To Singing
Marry	1771	How To Singing
Kelly	1771	How To Singing
Tim	1771	How To Singing
King	1771	How To Singing
John	1771	How To Singing
Lucy	1771	How To Singing
July	1771	How To Singing

表子查询可以看作一张临时的表，所以引用子查询中列的时候必须使用子查询中定义的列名，也就是如果子查询中为列定义了别名，那么在引用的时候也要使用别名。比如下面的 SQL 语句：

```
SELECT T_Reader.FName,t2.FYear,t2.FName ,t2.F3
FROM T_Reader,
(SELECT FYearPublished AS FYear,FName,1+2 as F3 FROM T_Book WHERE FYearPublished < 1800) t2
```

这里的表子查询为 `FYearPublished` 列取了一个别名 `FYear`，这样在引用它的时候就必须使用 `FYear` 而不能继续使用 `FYearPublished` 这个名字，这里子查询中还增加了一个新列 `F3`，同样可以在 `SELECT` 列表中引用它。

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FYear	FName	F3
Tom	1700	History of America	3
Sam	1700	History of America	3
Jerry	1700	History of America	3
Lily	1700	History of America	3
Marry	1700	History of America	3
Kelly	1700	History of America	3
Tim	1700	History of America	3
King	1700	History of America	3
John	1700	History of America	3
Lucy	1700	History of America	3
July	1700	History of America	3
Fige	1700	History of America	3
Tom	1771	How To Singing	3

Sam	1771	How To Singing	3
Jerry	1771	How To Singing	3
Lily	1771	How To Singing	3
Marry	1771	How To Singing	3
Kelly	1771	How To Singing	3
Tim	1771	How To Singing	3
King	1771	How To Singing	3
John	1771	How To Singing	3
Lucy	1771	How To Singing	3
July	1771	How To Singing	3

8.2 SELECT 列表中的标量子查询

在上一节中我们讲到可以将标量子查询当成 SELECT 列表中的一个列，唯一的约束就是子查询的返回值必须只有一行记录，而且只能有一个列。看完上章的例子有的读者可能认为标量子查询只能返回唯一的值，其实标量子查询完全可以返回随当前查询记录而变化的值。比如下面的 SQL 语句可以清楚的说明这一点：

```
SELECT FId,FName,
(
    SELECT MAX(FYearPublished)
    FROM T_Book
    WHERE T_Book. FCategoryId= T_Category.FId
)
FROM T_Category
```

这个 SELECT 语句首先检索 FId、FName 两个字段，而第三个字段不是一个列二是一个子查询。这个子查询位于主查询的内部，它返回一类图书的最新出版年份。因为聚合函数仅返回一行记录，所以这满足标量子查询的条件。通过 WHERE 语句，这个子查询也被连接到外部的 SELECT 查询语句中，因为这个连接，MAX(FYearPublished)将返回每类图书的最新出版年份。

需要注意的是这里的子查询与前边讲的有所不同，前面用到的子查询没有依赖于外部查询中的字段，也就是可以直接单独执行，而这里的子查询是依赖于外部查询中的 T_Category.FId 字段的，这个子查询是无法单独执行的，尝试在数据库系统中执行下面的 SQL 语句：

```
SELECT MAX(FYearPublished)
FROM T_Book
WHERE T_Book. FCategoryId= T_Category.FId
```

执行后数据库系统会报出如下的错误信息：

```
无法绑定由多个部分组成的标识符 "T_Category.FId"。
```

因为这个子查询中引用了外部查询中的字段。这种引用了外部查询中字段的子查询被称为相关子查询。

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	
1	Story	2002
2	History	2008
3	Theory	1971
4	Technology	2005

5	Art	1771
6	Philosophy	2001

下面结合执行结果来仔细分析一下这句 SQL 语句。首先看执行结果中的第一行，它的 FId 是 1.子查询通过 T_Book 表中的 FCategoryId 字段和 T_Category 表中的 FId 连接到外部查询。对于第一行，FId 是 1，因此子查询在 T_Book 表中检索 FCategoryId 字段等于 1 的所有图书的 FYearPublished 字段的最大值；接着查看外部查询的第二行，FId 是 2，这次子查询检索 T_Book 表中 FCategoryId 字段等于 2 的所有图书的 FYearPublished 字段的最大值；然后查看外部查询的第三行……以此类推。

如果没有子查询中的 WHERE 子句将子查询连接到外部查询，则结果将只是子查询返回的所有记录的最大值。比如修改上面的 SQL 语句，将子查询中的 WHERE 子句删除，将得到下面的 SQL 语句：

```
SELECT FId,FName,
(
    SELECT MAX(FYearPublished)
    FROM T_Book
)
FROM T_Category
```

执行这个 SQL 语句则会得到下面的执行结果：

FId	FName	
1	Story	2008
2	History	2008
3	Theory	2008
4	Technology	2008
5	Art	2008
6	Philosophy	2008

MAX(FYearPublished)现在是 T_Book 表中所有记录的最大出版年份，它不与任何书籍分类相关。

8.3 WHERE 子句中的标量子查询

标量子查询不仅可以用在 SELECT 语句的列表中，它还可以用在 WHERE 子句中，而且实际应用中子查询很多的时候都是用在 WHERE 子句中的。

先来看一个简单的例子，我们要检索喜欢“Story”的读者主键列表，那么这可以使用连接来完成，不过这里我们将使用子查询来完成。

使用子查询的实现思路也比使用连接简单。首先肯定要到 T_Category 表中查找 FName 等于“Story”的记录的 FId 字段值：

```
SELECT FId FROM T_Category
WHERE FName=' Story '
```

因为这个查询的返回值是单列且单行的，所以可以当作标量子查询使用。将这个子查询结果来构造外部查询：

```
SELECT FReaderId FROM T_ReaderFavorite
WHERE FCategoryId=
(
    SELECT FId FROM T_Category
    WHERE FName='Story'
)
```

执行这个 SQL 语句则会得到下面的执行结果：

FReaderId
1
6
7
12
3

下面来看一个稍微复杂一点的例子。假设需要检索每一种书籍类别中出版年份最早的书籍的名称，如果有两本或者多本书籍在同一年出版，则均显示它们的名字。要求检索结果中显示出类型的名字、书的名字和它的出版年份。

检索每种类型图书中出版时间最早的图书非常简单，只要使用 GROUP BY 子句以及聚合函数就可以轻松完成这个任务，SQL 语句如下：

```
SELECT T_Category.FId,MIN(T_Book.FYearPublished)
FROM T_Category
INNER JOIN T_Book ON T_Category.FId=T_Book.FCategoryId
GROUP BY T_Category.FId
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	
1	1999
2	1700
3	1930
4	2003
5	1771
6	1995

查询结果是正确的，不过这个查询结果没有提供书名，只提供了类型主键和出版时间最早的图书的出版年份。尝试将图书的名字加入到 SELECT 语句中，如下：

```
SELECT T_Category.FId, T_Book. FName,MIN(T_Book.FYearPublished)
FROM T_Category
INNER JOIN T_Book ON T_Category.FId=T_Book.FCategoryId
GROUP BY T_Category.FId
```

在数据库系统中执行这个 SQL 语句会报出如下的错误信息：

选择列表中的列 'T_Book.FName' 无效，因为该列没有包含在聚合函数或 GROUP BY 子句中。

出现这个错误的原因是所有在 SELECT 列表中的字段如果没有包含在聚合函数中，则必须放到 GROUP BY 子句中，所以将 T_Book. FName 加入到 GROUP BY 子句中，修改后的 SQL 语句如下：

```
SELECT T_Category.FId, T_Book. FName,MIN(T_Book.FYearPublished)
FROM T_Category
INNER JOIN T_Book ON T_Category.FId=T_Book.FCategoryId
GROUP BY T_Category.FId, T_Book. FName
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	
1	Jane Eyre	2001
1	Oliver Twist	2002
1	Two Cites	1999
2	History of America	1700

2	History of China	1982
2	History of England	1860
2	History of The World	2008
3	Astronomy	1971
3	Atom	1930
3	Computer	1970
3	RELATIVITY	1945
4	About J2EE	2005
4	Learning Hibernate	2003
5	How To Singing	1771
6	DaoDeJing	2001
6	Obedience to Authority	1995

这个执行结果显然是错误的，因为它们是根据 T_Category.FId 和 T_Book. FName 这两个字段进行的分组，所以 MIN(T_Book.FYearPublished)返回值不是一个特定书籍类型的最早出版年份，而是每本图书中的最早出版年份。而真正需要的是查询每种书籍类型中的最早出版的书籍，可以使用子查询来轻松完成这个任务。在 SQL 查询中，需要将一本书籍的出版年份与该类型的所有书籍的出版年份进行比较，并且仅仅在它们匹配时，才返回一个记录，实现 SQL 语句如下：

```
SELECT T_Category.FId, T_Book. FName,T_Book.FYearPublished
FROM T_Category
INNER JOIN T_Book ON T_Category.FId=T_Book.FCategoryId
WHERE T_Book.FYearPublished=
    (
        SELECT MIN(T_Book.FYearPublished)
        FROM T_Book
        WHERE T_Book.FCategoryId=T_Category.FId
    )
```

在这个 SQL 语句中，T_Category 表和 T_Book 表首先进行内部连接，然后使用 WHERE 子句中使用子查询来进行数据的过滤。这个子查询是一个相关子查询，它返回外部查询中当前图书类别中的图书的最早出版年份。在外部查询的 WHERE 子句中，T_Book 的 FYearPublished 与子查询的返回值进行比较，这样就可以得到每种书籍类型中的出版最早的书籍了。

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FYearPublished
1	Two Cites	1999
2	History of America	1700
3	Atom	1930
4	Learning Hibernate	2003
5	How To Singing	1771
6	Obedience to Authority	1995

8.4 集合运算符与子查询

标量子查询对子查询的要求非常高，而很多情况下查询结果并不能做为标量子查询。如果子查询是多行多列的表子查询，那么可以将其看成一个临时的数据表使用，而如果子查询是多行单列的表子查询，这样的子查询的结果集其实是一个集合，SQL 提供了对这样的集合进行操作的运算符，包括 IN、ANY、ALL 以及 EXISTS 等。本节将会对这些运算符在处

理子查询中的应用进行介绍。

8.4.1 IN 运算符

在前面章节已经介绍了 IN 运算符的简单使用，使用 IN 运算符可以用来匹配一个固定集合中的某一项。比如下面的 SQL 语句检索在 2001、2003 和 2005 年出版的所有图书：

```
SELECT * FROM T_Book
WHERE FYearPublished IN(2001,2003,2005)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FYearPublished	FCategoryId
1	About J2EE	2005	4
2	Learning Hibernate	2003	4
4	Jane Eyre	2001	1
15	DaoDeJing	2001	6

这里进行匹配的集合是已经确定的集合“2001,2003,2005”，如果要匹配的集合是动态的则无法用这种方式来进行匹配了。比如，需要检索所有图书出版年份内入会的读者信息，可以使用简单的 SQL 语句检索出所有的图书的出版年份：

```
select FYearPublished FROM T_Book
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FYearPublished
2005
2003
1999
2001
2002
1982
1860
1700
2008
1930
1945
1970
1971
1771
2001
1995

这个查询结果是多行单列的，因此可以将其用来与 IN 运算符进行匹配运算，SQL 语句如下：

```
SELECT * FROM T_Reader
WHERE FYearOfJoin IN
(
select FYearPublished FROM T_Book
)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FYearOfBirth	FCity	FProvince	FYearOfJoin

1	Tom	1979	TangShan	Hebei	2003
2	Sam	1981	LangFang	Hebei	2001
3	Jerry	1966	DongGuan	GuangDong	1995
4	Lily	1972	JiaXing	ZheJiang	2005
5	Marry	1985	BeiJing	BeiJing	1999
6	Kelly	1977	ZhuZhou	HuNan	1995
7	Tim	1982	YongZhou	HuNan	2001
9	John	1979	QingDao	ShanDong	2003
11	July	1983	ZhuMaDian	HeNan	1999
12	Fige	1981	JinCheng	ShanXi	2003

8.4.2 ANY 和 SOME 运算符

在 SQL 中 ANY 和 SOME 是同义词，所以下面介绍的时候只使用 ANY，SOME 的用法和功能和 ANY 一模一样。和 IN 运算符不同，ANY 必须和其他的比较运算符共同使用，而且必须将比较运算符放在 ANY 关键字之前，所比较的值需要匹配子查询中的任意一个值，这也就是 ANY 在英文中所表示的意义。

首先看一个 ANY 运算符和等于运算符（=）共同使用的例子，下面的 SQL 语句检索所有图书出版年份内入会的读者信息：

```
SELECT * FROM T_Reader
WHERE FYearOfJoin =ANY
(
select FYearPublished FROM T_Book
)
```

外部查询中的 `WHERE` 子句指定 `FYearOfJoin` 必须等于子查询 `select FYearPublished FROM T_Book` 所返回的集合中的任意一个值。

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNAME	FYEAROFBIRTH	FCITY	FPROVINCE	FYEAROFJOIN
1	Tom	1979	TangShan	Hebei	2003
2	Sam	1981	LangFang	Hebei	2001
3	Jerry	1966	DongGuan	GuangDong	1995
4	Lily	1972	JiaXing	ZheJiang	2005
5	Marry	1985	BeiJing	BeiJing	1999
6	Kelly	1977	ZhuZhou	HuNan	1995
7	Tim	1982	YongZhou	HuNan	2001
9	John	1979	QingDao	ShanDong	2003
11	July	1983	ZhuMaDian	HeNan	1999
12	Fige	1981	JinCheng	ShanXi	2003

这个 SQL 语句的检索结果与上一节介绍的使用 IN 运算符得到的结果是一致的：

```
SELECT * FROM T_Reader
WHERE FYearOfJoin IN
(
select FYearPublished FROM T_Book
)
```

也就是说“=ANY”等价于 IN 运算符，而“<>ANY”则等价于 NOT IN 运算符。

除了等于运算符，ANY 运算符还可以和大于（>）、小于(<)、大于等于（>=）、小于等

于（<=）等比较运算符共同使用。比如下面的 SQL 语句用于检索在任何一个会员出生之前出版的图书：

```
SELECT * FROM T_Book
WHERE FYearPublished<ANY
(
    SELECT FYearOfBirth FROM T_Reader
)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNAME	FYEARPUBLISHED	FCATEGORYID
6	History of China	1982	2
7	History of England	1860	2
8	History of America	1700	2
10	Atom	1930	3
11	RELATIVITY	1945	3
12	Computer	1970	3
13	Astronomy	1971	3
14	How To Singing	1771	5

注意，和 IN 运算符不同，ANY 运算符不能与固定的集合相匹配，比如下面的 SQL 语句是错误的：

```
SELECT * FROM T_Book
WHERE FYearPublished<ANY(2001,2003,2005)
```

不过这个限制并不会妨碍功能的实现，因为没有对固定的集合进行 ANY 匹配的必要，因为待匹配的集合是固定的，所以上面的 SQL 语句完全可以用下面的 SQL 语句来代替：

```
SELECT * FROM T_Book
WHERE FYearPublished<2005
```

8.4.3 ALL 运算符

ALL 在英文中的意思是“所有”，ALL 运算符要求比较的值需要匹配子查询中的所有值。ALL 运算符同样不能单独使用，必须和比较运算符共同使用。

下面的 SQL 语句用来检索在所有会员入会之前出版的图书：

```
SELECT * FROM T_Book
WHERE FYearPublished<ALL
(
    SELECT FYearOfJoin FROM T_Reader
)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FYearPublished	FCategoryId
6	History of China	1982	2
7	History of England	1860	2
8	History of America	1700	2
10	Atom	1930	3
11	RELATIVITY	1945	3
12	Computer	1970	3
13	Astronomy	1971	3
14	How To Singing	1771	5

与 ANY 运算符相同，ALL 运算符同样不能与固定的集合相匹配，比如下面的 SQL 语句是错误的：

```
SELECT * FROM T_Book
WHERE FYearPublished<ALL(2001,2003,2005)
```

不过这个限制并不会妨碍功能的实现，因为没有对固定的集合进行 ALL 匹配的必要，因为待匹配的集合是固定的，所以上面的 SQL 语句完全可以用下面的 SQL 语句来代替：

```
SELECT * FROM T_Book
WHERE FYearPublished<2001
```

另外需要注意的就是，当使用 ALL 运算符的时候，如果带匹配的集合为空，也就是子查询没有返回任何数据的时候，不论与什么比较运算符搭配使用 ALL 的返回值将永远是 true。比如下面的 SQL 语句用于检索在所有江苏省会员入会之前出版的图书：

```
SELECT * FROM T_Book
WHERE FYearPublished<ALL
(
    SELECT FYearOfJoin FROM T_Reader
    WHERE FProvince = 'JiangSu'
)
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FYearPublished	FCategoryId
1	About J2EE	2005	4
2	Learning Hibernate	2003	4
3	Two Cites	1999	1
4	Jane Eyre	2001	1
5	Oliver Twist	2002	1
6	History of China	1982	2
7	History of England	1860	2
8	History of America	1700	2
9	History of The World	2008	2
10	Atom	1930	3
11	RELATIVITY	1945	3
12	Computer	1970	3
13	Astronomy	1971	3
14	How To Singing	1771	5
15	DaoDeJing	2001	6
16	Obedience to Authority	1995	6

这个查询结果将所有的会员都检索出来了，可是根本没有江苏省的会员，应该是返回空结果才对的。看起来这是错误的，其实这完全符合 ALL 运算符的语义，因为没有江苏省的会员，所以每本书的出版年份就在所有的江苏省的会员之前，所以每一本书都符合匹配条件。在使用 ALL 运算符的时候，这一个问题很容易在系统中造成 BUG，因此使用时必须注意。

8.4.4 EXISTS 运算符

和 IN、ANY、ALL 等运算符不同，EXISTS 运算符是单目运算符，它不与列匹配，因此它也不要求待匹配的集合是单列的。EXISTS 运算符用来检查每一行是否匹配子查询，可以认为 EXISTS 就是用来测试子查询的结果是否为空，如果结果集为空则匹配结果为 false，否则匹配结果为 true。

先来看一个简单的 SQL 语句：

```
SELECT * FROM T_Book
WHERE EXISTS
(
    SELECT * FROM T_Reader
    WHERE FProvince='ShanDong'
)
```

这句 SQL 语句对 T_Book 表中的每行数据进行匹配，测试是否存在山东省的读者，因为系统中存在山东省的读者，所以这个 SQL 语句将检索出所有的图书。执行结果如下：

FId	FName	FYearPublished	FCategoryId
1	About J2EE	2005	4
2	Learning Hibernate	2003	4
3	Two Cites	1999	1
4	Jane Eyre	2001	1
5	Oliver Twist	2002	1
6	History of China	1982	2
7	History of England	1860	2
8	History of America	1700	2
9	History of The World	2008	2
10	Atom	1930	3
11	RELATIVITY	1945	3
12	Computer	1970	3
13	Astronomy	1971	3
14	How To Singing	1771	5
15	DaoDeJing	2001	6
16	Obedience to Authority	1995	6

再来看另外一个 SQL 语句：

```
SELECT * FROM T_Book
WHERE EXISTS
(
    SELECT * FROM T_Reader
    WHERE FProvince='YunNan'
)
```

这句 SQL 语句对 T_Book 表中的每行数据进行匹配，测试是否存在云南省的读者，因为系统中不存在云南省的读者，所以这个 SQL 语句的执行结果为空。

从前面几个例子来看，使用 EXISTS 运算符要么就是匹配返回表中所有数据，要么就是不匹配不返回任何数据，好像 EXISTS 运算符并没有太大意义，其实上面这两个例子在实际中并不常用，EXISTS 运算符的真正意义只有和相关子查询一起使用才更有意义。相关子查询中引用外部查询中的这个字段，这样在匹配外部子查询中的每行数据的时候相关子查询就会根据当前行的信息来进行匹配判断了，这样就可以完成非常丰富的功能。

下面的 SQL 语句用来检索存在 1950 年以前出版的图书的图书类别：

```
SELECT * FROM T_Category
WHERE EXISTS
(
```

```
    SELECT * FROM T_Book
    WHERE T_Book. FCategoryId = T_Category.FId
        AND T_Book. FYearPublished<1950
)
```

在 **EXISTS 后的子查询中，SQL 对** T_Category 表中的每一行数据到子查询中进行匹配，测试 T_Book 表中是否存在 FCategoryId 字段值等于当前类别主键值且出版年份在 1950 年之前的书籍。执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName
2	History
3	Theory
5	Art

8.5 在其他类型 SQL 语句中的子查询应用

到目前为止，本书讲解的子查询都是应用在使用 SELECT 进行数据检索，其实子查询完全可以应用在 INSERT、UPDATE 以及 DELETE 等语句中。合理的使用子查询将能够简化功能的实现并且极大的提高 SQL 语句执行的效率。本节将介绍子查询在 INSERT、UPDATE 以及 DELETE 等语句中的应用。

8.5.1 子查询在 INSERT 语句中的应用

在使用 INSERT 语句的时候，一般都是使用它向数据库中一条条的插入数据，比如：

```
INSERT INTO MyTable(FId,FName,FAge)
VALUES(1,'John',20)
```

但是有时我们可能需要将数据批量插入表中，比如创建一个和 T_ReaderFavorite 表结构完全相同的表 T_ReaderFavorite2，然后将 T_ReaderFavorite 中的输入复制插入到 T_ReaderFavorite2 表。

首先我们创建 T_ReaderFavorite2 表：

```
MYSQL,MSSQLServer,DB2:
CREATE TABLE T_ReaderFavorite2 (FCategoryId INT,FReaderId INT)
Oracle:
CREATE TABLE T_ReaderFavorite2 (FCategoryId NUMBER (10),FReaderId NUMBER (10))
```

按照普通的实现思路，我们需要编写如下的宿主语言代码：

```
rs = ExecuteQuery("SELECT * from T_ReaderFavorite");
while(rs.next())
{
    int categoryId = rs.get("FCategoryId");
    int readerId = rs.get("FReaderId");
    Execute("INSERT INTo T_ReaderFavorite2(FCategoryId,FReaderId) VALUES(?,?)",
            categoryId, readerId);
}
```

在宿主语言中逐条读取 T_ReaderFavorite 表中的记录，然后将它们逐条插入 T_ReaderFavorite2 表中。这样的处理方式能够正确的完成要求的功能，而且由于目前 T_ReaderFavorite 表中的数据量非常少，所以这样处理速度上也没有什么影响。但是如果 T_ReaderFavorite 表中有大量的数据的话，由于每插入一条数据都要执行两次数据库操作，所以这种处理方式效率非常低，因此必须采用其他的处理方式。

除了 INSERT……VALUES……这种用法外，INSERT 语句还支持另外一种语法，那就是 INSERT……SELECT……，采用这种使用方式可以将 SELECT 语句返回的结果集直接插

入到目标表中，因为这一切都是都数据库内部完成的，所以效率非常高。下面看一下使用 INSERT……SELECT……来实现将 T_ReaderFavorite 中的输入复制插入到 T_ReaderFavorite2 表，SQL 语句如下：

```
INSERT INTO T_ReaderFavorite2(FCategoryId,FReaderId)
SELECT FCategoryId,FReaderId FROM T_ReaderFavorite
```

这里使用 SELECT FCategoryId,FReaderId FROM T_ReaderFavorite 将 T_ReaderFavorite 表中的数据读出，然后使用 INSERT INTO T_ReaderFavorite2(FCategoryId,FReaderId)将检索结果插入到 T_ReaderFavorite2 表中，注意上下的列顺序必须是一一对应的。

执行完毕我们后我们查看 T_ReaderFavorite2 表中的内容：

FCategoryId	FReaderId
1	1
5	2
2	3
3	4
5	5
1	6
1	7
4	8
6	9
5	10
2	11
2	12
1	12
3	1
1	3
4	4

可以看到 T_ReaderFavorite 表中的数据已经被正确的复制到 T_ReaderFavorite2 表中了。

继续下面内容之前请执行：DELETE FROM T_ReaderFavorite2 来清空 T_ReaderFavorite2 表。

使用 INSERT……SELECT……不仅能够实现简单的将一个表中的数据导出到另外一个表中的功能，还能在将输入插入目标表之前对数据进行处理，比如下面的 SQL 语句用于将 T_ReaderFavorite 表中的数据复制到 T_ReaderFavorite2 表中，但是如果 T_ReaderFavorite 表中的 FReaderId 列的值大于 10，则将 FReaderId 的值减去 FCategoryId 的值后再复制到 T_ReaderFavorite2 表中。编写如下的 SQL 语句：

```
INSERT INTO T_ReaderFavorite2(FCategoryId,FReaderId)
SELECT FCategoryId,
(CASE
    WHEN FReaderId<=10 THEN FReaderId
    ELSE FReaderId- FCategoryId
END
)
FROM T_ReaderFavorite
```

这里在 SELECT 语句中使用 CASE 函数来实现对数据插入前的处理。执行完毕我们后我们查看 T_ReaderFavorite2 表中的内容：

FCategoryId	FReaderId
1	1
5	2
2	3
3	4
5	5
1	6
1	7
4	8
6	9
5	10
2	9
2	10
1	11
3	1
1	3
4	4

使用这种插入前的数据处理可以完成诸如“将数据从 A 表导出到 B 表，并且将 B 表的主键全部加上 bak 前缀”、“将 A 公司的所有员工插入到我们的会员表，自动导入所有的客户信息，并且为其自动生成会员编号”等复杂的任务。

现在可以删除 T_ReaderFavorite2 表了：

```
DROP TABLE T_ReaderFavorite2;
```

因为可以在插入目标表前可以对数据进行处理，所以 INSERT……SELECT……语句不局限于同结构表间的数据插入，也可以实现异构表见输入的插入。假设要将所有会员爱好的图书统一增加“小说”，也就是为 T_Reader 表中的每个读者都在 T_ReaderFavorite 表中创建一条 FCategoryId 等于 1 的记录，实现 SQL 语句如下：

```
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
SELECT 1,FId FROM T_Reader
```

SELECT 语句从 T_Reader 表中检索所有的读者信息，并且将第一列设定为固定值 1，而将第二列设定为读者的主键，执行完毕查看 `T_ReaderFavorite` 表中的内容：

FCategoryId	FReaderId
1	1
5	2
2	3
3	4
5	5
1	6
1	7
4	8
6	9
5	10
2	11
2	12

1	12
3	1
1	3
4	4
1	1
1	2
1	3
1	4
1	5
1	6
1	7
1	8
1	9
1	10
1	11
1	12

数据被正确的插入了，但是仔细查看 T_ReaderFavorite 表中的内容，不难发现其中有重复的数据，比如 FCategoryId 等于 1、FReaderId 等于 1 的记录出现了两次，这就出现了数据冗余，必须修改 SQL 语句防止这种情况的发生。也就是在将输入插入 T_ReaderFavorite 表之前要检查 T_ReaderFavorite 表中是否已经存在了同样的数据。

在继续进行之前请首先清空 T_ReaderFavorite 表中的数据，然后执行本章开头的 T_ReaderFavorite 表数据的初始化代码。

首先需要在 SELECT 语句中添加 WHERE 子句，这个 WHERE 子句检查每个读者是否有 Story 的爱好，即是否存在图书分类主键为 1 的的爱好，只有没有的时候才返回记录：

```
SELECT 1,FId FROM T_Reader
WHERE NOT EXISTS
(
    SELECT * FROM T_ReaderFavorite
    WHERE T_ReaderFavorite. FCategoryId=1
    AND T_ReaderFavorite. FReaderId= T_Reader.FId
)
```

执行这个子查询片段，执行完毕我们就能在输出结果中看到下面的执行结果：

	FId
1	2
1	4
1	5
1	8
1	9
1	10
1	11

可以看到检索出的确实是还没有以 Story 做为爱好的读者，所以编写下面的 SQL 语句来执行数据的插入：

```
INSERT INTO T_ReaderFavorite(FCategoryId,FReaderId)
SELECT 1,FId FROM T_Reader
```

```
WHERE NOT EXISTS
(
    SELECT * FROM T_ReaderFavorite
    WHERE T_ReaderFavorite. FCategoryId=1
AND T_ReaderFavorite. FReaderId= T_Reader.FId
)
```

执行完毕查看 T_ReaderFavorite 表中的内容：

FCategoryId	FReaderId
1	1
5	2
2	3
3	4
5	5
1	6
1	7
4	8
6	9
5	10
2	11
2	12
1	12
3	1
1	3
4	4
1	2
1	4
1	5
1	8
1	9
1	10
1	11

可以看到表中的数据没有重复的了，即使重复执行这个 SQL 语句也都不会添加新的记录，因为这个 SQL 语句已经对数据的重复做了检查。

8.5.2 子查询在 UPDATE 语句中的应用

在 UPDATE 语句中可以在更新列表中以及 WHERE 语句使用子查询。下面演示一个将图书的出版日期全部更新为所有图书中的最新出版日期，SQL 语句如下：

```
UPDATE T_Book
SET FYearPublished=
        (SELECT MAX(FYearPublished) FROM T_Book)
```

注意，在 MYSQL 中是不支持使用子查询来更新一个列的，所以这个 UPDATE 语句无法在 MYSQL 中执行。

执行完毕查看 T_Book 表中的内容：

FID	FNAME	FYEARPUBLISHED	FCATEGORYID

1	About J2EE	2008	4
2	Learning Hibernate	2008	4
3	Two Cites	2008	1
4	Jane Eyre	2008	1
5	Oliver Twist	2008	1
6	History of China	2008	2
7	History of England	2008	2
8	History of America	2008	2
9	History of The World	2008	2
10	Atom	2008	3
11	RELATIVITY	2008	3
12	Computer	2008	3
13	Astronomy	2008	3
14	How To Singing	2008	5
15	DaoDeJing	2008	6
16	Obedience to Authority	2008	6

如果 UPDATE 语句拥有 WHERE 子句，那么还可以在 WHERE 子句中使用子查询，其使用方式与 SELECT 语句中的子查询基本相同，而且也可以使用相关子查询等高级的特性。下面的 SQL 语句用来将所有同类书本书超过 3 本的图书的出版日期更新为 2005：

```
UPDATE T_Book b1
SET b1.FYearPublished=2005
WHERE
(
    SELECT COUNT(*) FROM T_Book b2
    WHERE b1. FCategoryId=b2. FCategoryId
)>3
```

上面的 SQL 语句使用相关子查询来查询所有与待更新的书籍属于同类别的书籍的总数，如果总数大于 3 则将当前书籍的出版日期更新为 2005。

执行完毕查看 T_Book 表中的内容：

FID	FNAME	FYEARPUBLISHED	FCATEGORYID
1	About J2EE	2008	4
2	Learning Hibernate	2008	4
3	Two Cites	2008	1
4	Jane Eyre	2008	1
5	Oliver Twist	2008	1
6	History of China	2005	2
7	History of England	2005	2
8	History of America	2005	2
9	History of The World	2005	2
10	Atom	2005	3
11	RELATIVITY	2005	3
12	Computer	2005	3
13	Astronomy	2005	3

14	How To Singing	2008	5
15	DaoDeJing	2008	6
16	Obedience to Authority	2008	6

8.5.3 子查询在 DELETE 语句中的应用

子查询在 DELETE 中唯一可以应用的位置就是 WHERE 子句，使用子查询可以完成复杂的数据删除控制。其使用方式与 SELECT 语句中的子查询基本相同，而且也可以使用相关子查询等高级的特性。下面的 SQL 语句用来将所有同类书本书超过 3 本的图书删除：

```
DELETE FROM T_Book b1
WHERE
(
    SELECT COUNT(*) FROM T_Book b2
    WHERE b1.FCategoryId=b2.FCategoryId
)>3
```

上面的 SQL 语句使用相关子查询来查询所有与待更新的书籍属于同类别的书籍的总数，如果总数大于 3 则将当前书籍删除。

执行完毕查看 T_Book 表中的内容：

FID	FNAME	FYEARPUBLISHED	FCATEGORYID
1	About J2EE	2008	4
2	Learning Hibernate	2008	4
3	Two Cites	2008	1
4	Jane Eyre	2008	1
5	Oliver Twist	2008	1
14	How To Singing	2008	5
15	DaoDeJing	2008	6
16	Obedience to Authority	2008	6

现在可以删除本章用到的表了：

```
DROP TABLE T_Reader;
DROP TABLE T_Book;
DROP TABLE T_Category;
DROP TABLE T_ReaderFavorite;
```

第八章 主流数据库的 SQL 语法差异性以及解决方案

8.1 SQL 语法差异性简介

8.1.1 SQL 语法差异的由来

8.1.2 SQL 语法差异的分类

8.2 数据类型差异

8.3 基本元素差异

8.3.1 日期数据表示

8.3.2 字符串相加

8.4 基本语句差异

8.4.1 删除索引

8.4.2 子查询

8.4.3 表联接

取元数据的语法不同。比如取所有表名、取表中的字段（参考 CNSQL 和 CookBook）

8.5 消除差异性的方案

8.5.1 不同的数据库不同的 DAO

8.5.2 使用语法交集

8.5.3 使用语法适配器

8.5.4 使用 ORM 或者 RAD

8.5.5 使用 SQL 翻译器

8.5.6 几种方案的优劣比较

8.6 CowNewSQL 翻译器

8.6.1 CowNewSQL 简介

8.6.2 CowNewSQL 的使用

8.6.3 CowNewSQL 的语法

8.6.4 CowNewSQL 的扩展

独特的数据类型

Sequence

高级语法的差异：Connect by

第九章　主流数据库的 SQL 语法差异以解决方案

现在市场上存在很多厂商推出的数据库管理系统，商业化的有 Oracle、MSSQLServer、DB2、SybaseSQLServer、Informix，开源的有 MYSQL、SQLite、SimpleSQL、Berkely DB、Minosse、Firebird、HSQLDB 等，这些数据库产品的出现给了开发者更多的选择余地，比如：系统对安全性、稳定性以及售后技术支持要求非常高，那么就可以选择 Oracle 或者 DB2；如果系统只运行在 Windows 操作系统下那么可以选择 MSSQLServer；如果不想为数据库管理系统承担费用，那么可以选择免费版的 MYSQL；如果对数据库的要求不高只是提供一个方便数据存取机制，那么可以使用 HSQLDB 等嵌入式数据库系统。不过，这在给我们带来选择便利的同时也给开发者带来了麻烦，比如：开发的系统是以产品的形式发售的，要求能够运行在所有主流的数据库管理系统下；为了降低成本，在开发阶段使用免费版的 MYSQL 进行开发，然后在部署阶段要运行在 Oracle 下；旧有系统运行在 Informix 下，现在要求平滑迁移到 DB2 下。众所周知，各个数据库管理系统支持的 SQL 语法是存在一定差异的，在 Oracle 下能够成功运行的 SQL 迁移到 DB2 下可能就无法运行，因此如何使得系统能够在多种数据库管理系统下运行就成了一个非常棘手的问题，本章将首先对主流数据库管理系统的 SQL 语法差异进行介绍，然后提出相应的解决方案。

9.1 SQL 语法差异分析

主流数据库系统支持的 SQL 语句的差异主要有以下几点：数据类型的差异；运算符的差异；函数的差异；常用 SQL 的差异；取元数据信息的差异。

9.1.1 数据类型的差异

整数类型：在 MYSQL 中整数相关的类型有 tinyint、smallint、mediumint、int、integer 和 bigint；在 MSSQLServer 中整数相关的类型有 bit、int、smallint、tinyint 和 bigint；在 Oracle 中整数相关的类型有 number；在 DB2 中整数相关的类型有 smallint、integer 和 bigint。

数值类型：在 MYSQL 中数值相关的类型有 **float**、**double**、**real**、**decimal** 和 **numeric**；在 MSSQLServer 中数值相关的类型有 decimal、numeric、money、smallmoney 、float 和 real；在 Oracle 中数值相关的类型有 number；在 DB2 中数值相关的类型有 decimal、numeric、real 和 double。

字符类型：在 MYSQL 中字符相关的类型有 char、varchar、tinytext、text、mediumtext、longtext、enum 和 set；在 MSSQLServer 中字符相关的类型有 char、varchar、text、**nchar**、**nvarchar** 和 **ntext**；在 Oracle 中字符相关的类型有 char、varchar2、nvarchar2、clob 和 nclob；在 DB2 中字符相关的类型有 CHARACTER、VARCHAR、LONG VARCHAR、CLOB、GRAPHIC、VARGRAPHIC 和 LONG VARGRAPHIC。

日期时间类型：在 MYSQL 中日期时间相关的类型有 date、time、datetime、timestamp 和 year；在 MSSQLServer 中日期时间相关的类型有 **datetime**、**smalldatetime** 和 **timestamp**；在 Oracle 中日期时间相关的类型有 date 和 timestamp；在 DB2 中日期时间相关的类型有 DATE、TIME 和 TIMESTAMP。

二进制类型：MYSQL、Oracle 和 DB2 都支持 Blob 类型，而在 MSSQLServer 中支持 **image** 类型。

关于这些数据类型的具体介绍请参照第二章相关内容。

9.1.2 运算符的差异

在不同的数据库系统中字符串拼接的方式是不同的，下面的主流数据库系统对字符串拼接的支持：

MYSQL：在 MYSQL 中进行字符串的拼接要使用 CONCAT 函数，CONCAT 函数支持一个或者多个参数，比如 `CONCAT('Hello',1,'World')`；MYSQL 中还提供了另外一个进行字符串拼接的函数 CONCAT_WS，CONCAT_WS 可以在待拼接的字符串之间加入指定的分隔符，比如 CONCAT_WS `('Hello',1,'World')`。

MSSQLServer：MSSQLServer 中可以直接使用加号“+”来拼接字符串，比如'Hello'+'World'。

Oracle：Oracle 中使用“||”进行字符串拼接，比如'Hello'||'World'；除了“||”，Oracle 还支持使用 CONCAT()函数进行字符串拼接，不过与 MYSQL 的 CONCAT()函数不同，Oracle 的 CONCAT()函数只支持两个参数，不支持两个以上字符串的拼接。

DB2：DB2 中使用“||”进行字符串拼接，比如'Hello'||'World'。

9.1.3 函数的差异

不同数据库系统对函数的差异是非常大的，不仅同样功能的函数在不同数据库系统中的名称不同，而且一些高级的函数也并不是在所有数据库系统中都有提供支持。比如将一个字符串转换为小写的函数在 **MYSQL，MSSQLServer 和 Oracle 中为 LOWER，而在 DB2 中则为 LCASE；MYSQL 中支持** IF 函数，而在其他数据库系统中则只有通过变通方式才能实现。主流数据库系统对函数支持的差异性在第五章有详细介绍，这里不再赘述。

9.1.4 常用 SQL 的差异

主流数据库系统对 SELECT、UPDATE、DELETE、CREATE、DROP 等基本语法的支持是相同，不过在一些高级特性支持方面仍然有差异。

9.1.4.1 限制结果集行数

在实现分页检索、排行榜等功能的时候，需要限制检索的结果集行数，不同的数据库系统对此的支持是不同的。

MYSQL 中提供了 LIMIT 关键字用来限制返回的结果集，比如：

```
SELECT * FROM T_Employee
ORDER BY FSalary DESC LIMIT 2,5
```

MSSQLServer：MSSQLServer中提供了TOP关键字用来返回结果集中的前N条记录，比如：

```
select top 5 * from T_Employee
order by FSalary Desc;
```

在MSSQLServer2005中还可以使用窗口函数`ROW_NUMBER()`实现限制结果集行数，比

如：

```
SELECT ROW_NUMBER() OVER(ORDER BY FSalary),
FNumber,FName,FSalary,FAge
FROM T_Employee。
```

Oracle：Oracle中支持窗口函数ROW_NUMBER()，其用法和MSSQLServer2005中相同；除了窗口函数ROW_NUMBER()，Oracle中还提供了更方便的rownum机制，Oracle为每个结果集都增加了一个默认的表示行号的列，这个列的名称为rownum。使用rownum可以很轻松的取得结果集中前N条的数据行，比如：

```
SELECT * FROM T_Employee
WHERE rownum<=6
ORDER BY FSalary Desc
```

DB2：DB2中支持窗口函数ROW_NUMBER()，其用法和MSSQLServer2005以及Oracle中相同。除此之外，DB2还提供了FETCH关键字用来提取结果集的前N行，比如：

```
SELECT * FROM T_Employee
ORDER BY FSalary Desc
FETCH FIRST 6 ROWS ONLY
```

9.1.4.2 删除索引

索引的定义在各个数据库系统中基本相同，但是删除索引的语法则各有不同，比如删除T_Person表中定义的名称为idx1的索引在不同数据库系统下的SQL语句如下：

MYSQL:

DROP INDEX idx1 ON T_Person

MSSQLServer:

DROP INDEX T_Person.idx1

Oracle,DB2:

DROP INDEX idx1

9.1.5取元数据信息的差异

在开发一些功能的时候有时需要查询数据的一些信息，比如数据库的名称、当前用户名、数据库中有哪些表、指定表的字段定义等，这些信息被称为元数据。对元数据的支持在不同的数据库系统下的差异性是非常大的。

9.1.5.1 取数据库信息

MYSQL中可以通过函数来取得数据库的信息，包括当前数据库名、版本、当前登录用户等信息：DATABASE()函数返回当前数据库名；VERSION()函数以一个字符串形式返回 MySQL 服务器的版本；USER()函数（这个函数还有SYSTEM_USER、SESSION_USER两个别名）返回当前 MySQL 用户名。

MSSQLServer中也可以通过函数来取得数据库的信息：APP_NAME()函数返回当前会话的应用程序名称；CURRENT_USER函数（注意这个函数不能带括号调用）返回当前登陆用户名；HOST_NAME()函数返回工作站名。

不过，在MSSQLServer中如果要查询当前数据库名，则必须到系统表sysprocesses中查询，SQL语句如下：

```
select
dbname =
    case when dbid = 0 then null
    when dbid <> 0 then db_name(dbid)
    end
```

```
from master..sysprocesses
where spid=@@SPID
```

系统表“master..sysprocesses”中存储了当前数据库系统中的进程信息，而“@@SPID”则表示当前进程号。

Oracle 中使用 USER 函数用来取得当前登录用户名，注意使用这个函数的时候不能使用括号形式的空参数列表，也就是 USER()这种使用方式是不对的。正确使用方式如下：

```
SELECT USER FROM DUAL
```

Oracle中使用USERENV()函数用来取得当前登录用户相关的环境信息，USERENV()函数有一个参数，参数的可选值如下：ISDBA、LANGUAGE、TERMINAL、SESSIONID、ENTRYID、LANG和INSTANCE。

DB2中可以通过**CURRENT_USER来取得当前登陆用户名，而CURRENT_SERVER用来取得当前服务名，比如：**

```
SELECT  CURRENT_USER,CURRENT_SERVER
FROM sysibm.sysdummy1
```

DB2中取得当前数据库的版本的SQL语句如下：

SELECT * FROM sysibm.sysversions

9.1.5.2 取得所有表

MYSQL中取得当前数据库中所有表定义的SQL语句如下：

SHOW TABLES

MSSQLServer中的系统表sysobjects中记录了当前系统中定义的对象，其中xtype字段等于U的记录为表定义，因此取得当前数据库中所有表定义的SQL语句如下：

SELECT name FROM sysobjects where xtype='U'

Oracle中的系统表all_objects中记录了当前系统中定义的对象，其中Object_Type字段等于TABLE的记录为表定义，因此取得当前数据库中所有表定义的SQL语句如下：

select Object_Name from all_objects where Object_Type='TABLE'

DB2中的系统表all_ syscat.tables中记录了当前系统中定义的表和视图，其中TYPE字段等于T的记录为表定义，因此取得当前数据库中所有表定义的SQL语句如下：

SELECT TABNAME FROM syscat.tables where TYPE='T'

9.1.5.3 取得指定Schema下的表

MYSQL中取得指定Schema下所有表定义的SQL语句如下（假设Schema名为demoschema）：

SHOW TABLES FROM demoschema

MSSQLServer中的系统表sysobjects中记录了当前系统中定义的对象，其中xtype字段等于U的记录为表定义，因此取得当前数据库中所有表定义的SQL语句如下（假设Schema名为demoschema）：

SELECT name FROM demoschema.sysobjects where xtype='U'

Oracle中的系统表all_objects中记录了当前系统中定义的对象，其中Object_Type字段等于TABLE的记录为表定义，OWNER字段为Schema，因此取得当前数据库中所有表定义的SQL语句如下（假设Schema名为demoschema）：

select Object_Name from all_objects

where Object_Type='TABLE' and OWNER='demoschema'

DB2中的系统表all_ syscat.tables中记录了当前系统中定义的表和视图，其中TYPE字段等于T的记录为表定义，TABSCHEMA字段为Schema，因此取得当前数据库中所有表定义的SQL语句如下（假设Schema名为demoschema）：

```
SELECT TABNAME FROM syscat.tables
where    TYPE='T' and TABSCHEMA='demoschema'
```

9.1.5.4 取得指定表的字段定义

MYSQL中取得指定表的字段定义（假设表名为mytable）：

```
DESCRIBE mytable
```

MYSQLServer中取得指定表的字段定义（假设表名为mytable）：

```
SELECT syscols.name as COLUMN_NAME,
st.name as DATA_TYPE,
syscomm.text as DATA_DEFAULT,
syscols.isnullable as NULLABLE
FROM syscolumns syscols
left join systypes st on syscols.xusertype=st.xusertype
left join syscomments syscomm on syscols.cdefault=syscomm.id
where syscols.id=OBJECT_ID(N'mytable')
order by syscols.id,syscols.colorder
```

Oracle中的all_tab_columns表是系统中所有表的字段定义，其中TABLE_NAME字段为表名，因此取得指定表的字段定义（假设表名为mytable）：

```
select COLUMN_NAME,DATA_TYPE,DATA_DEFAULT,NULLABLE
from all_tab_columns where TABLE_NAME ='MYTABLE'
```

DB2中的syscat.columns表是系统中所有表的字段定义，其中TABNAME字段为表名，因此取得指定表的字段定义（假设表名为mytable）：

```
select COLNAME as COLUMN_NAME, TYPENAME as DATA_TYPE,DEFAULT as
DATA_DEFAULT,NULLS as NULLABLE
from syscat.columns where TABNAME='MYTABLE'
```

9.2消除差异性的方案

由于不同数据库系统的语法有差异，所以如果想要开发的系统能够运行于多数据库系统下就必须通过一定的方法来消除这些差异性。消除差异性的主要方法有：为每种数据库编写不同的SQL语句；使用语法交集；使用抽象SQL；使用ORM工具；使用SQL翻译器。下面对这几种方案进行分析。

9.2.1为每种数据库编写不同的SQL语句

采用这种方案时，对于有语法差异的SQL语句则为为每种数据库编写不同的SQL，然后在运行时根据当前数据库类型来执行不同的SQL语句，比如：

```
if(currentdatabase='MYSQL')
{
   executeQuery(' SELECT * FROM T_Person LIMIT 0, 10');
}
else if(currentdatabase='MSSQLServer')
{
   executeQuery(' SELECT    TOP 10 * FROM T_Person');
} else if(currentdatabase='Oracle')
{
   executeQuery(' SELECT * FROM T_Person WHERE ROWNUM <= 10');
}
else if(currentdatabase='DB2')
```

```
{
   executeQuery(' SELECT * FROM T_Person FETCH FIRST 10 ROWS ONLY');
}
```

采用这种方案的时候能够比较好的解决多数据库的问题，但是要求开发人员对每种数据库的语法差异性非常精通，而且这增加了开发的工作量。

9.2.2使用语法交集

为了避免多数据库的问题，在开发的时候避免使用各个数据库系统语法的差异部分，只使用所有数据库系统都支持的SQL语句。采用这种方案的时候能够比较好的解决多数据库的问题，但是由于不能使用一些高级的语法，因此有的功能无法实现或者必须在宿主语言中通过代码来实现，这不仅限制了系统功能的实现而且降低了运行效率，最重要的问题是：我既然花大价钱买了Oracle数据，为什么不用Oracle提供的一些耗用的特性呢？

9.2.3使用SQL实体对象

该方案下，开发人员访问数据库方法并不是直接执行SQL语句，而是以SQL实体对象的方式描述出对应SQL语句的语意；然后调用SQL实体执行器对SQL实体描述对象处理，生成对应的数据类型的SQL语句，并执行。在SQL执行器中，为每种数据库实现一个适配翻译器，该适配翻译器接收传入的SQL实体对象，并能根据SQL实体对象描述的语意，生成符合对应数据库语法的SQL语句。SQL实体执行器在运行期间才与具体适配翻译器关联，不必事先知道由何种适配翻译器处理SQL实体对象；当需要增加对新数据库的支持时，不必修改任何原有软件，只需要实现一个新的适配翻译器即可。

采用这种方案，开发人员不能直接编写SQL语句，只能编写抽象的语法结构，比如下面的代码来实现取得表T_Person中前10行数据的功能：

```
Query query = new Query();
query.SetColumn("*");
query.SetTableName("T_Person");
query.SetLimit(0,10);
ExecuteQuery(query);
```

系统框架会将Query翻译成对应数据库系统支持的SQL语句，比如：

```
MYSQL:
SELECT * FROM T_Person LIMIT 0, 10
MSSQLServer:
SELECT    TOP 10 * FROM T_Person
Oracle:
SELECT * FROM T_Person WHERE ROWNUM <= 10
DB2:
SELECT * FROM T_Person FETCH FIRST 10 ROWS ONLY
```

采用这种方式能最大程度的利用目标数据库的高级特性，而且开发人员甚至不需要对SQL语法有任何了解，其缺点是编写的代码量增加了，同时如果要实现子查询、连接等复杂功能就编写非常冗长且难懂的代码，使用普通SQL语句三五行就能完成的功能如果采用这种方式可能就需要几十行代码。

9.2.4使用ORM工具

Java中的Hibernate、EJB以及.Net中的Linq、NHibernate等都是非常优秀的ORM工具，这些ORM工具提供了以面向对象的方式使用数据库，开发人员只要操作实体对象就可以，从而避免了直接编写SQL语句，比如下面的代码用来向人员表中加入一条记录：

```
Person person = new Person();
```

```
person.Name="Tom";
person.Age=22;
ormTool.Save(person);
```

ORM工具会将其翻译成如下的SQL语句：

```
INSERT INTO T_Person(FName,FAge)
VALUES('Tom',22);
```

下面的代码用来取得人员表中排名前十的人员：

```
Query query = new Query();
query.SetLimit(0,10);
query.SetEntityName("Person");
ormTool.ExecuteQuery(query);
```

系统框架会将Query翻译成对应数据库系统支持的SQL语句，比如：

```
MYSQL:
SELECT * FROM T_Person LIMIT 0, 10
MSSQLServer:
SELECT    TOP 10 * FROM T_Person
Oracle:
SELECT * FROM T_Person WHERE ROWNUM <= 10
DB2:
SELECT * FROM T_Person FETCH FIRST 10 ROWS ONLY
```

ORM工具将对实体对象的操作翻译成SQL语句，这本质上也是一种“使用SQL实体对象”的解决方案。

除了支持以操作实体对象的方式使用ORM工具，很多ORM工具都提供了以一种类似于SQL语句的语法工具，比如EJB中的EJB-SQL以及Hibernate中的HSQL，我们可以统称为ORMSQL，在实现复杂功能的时候使用ORMSQL可以避免编写过长的对象操作代码，ORM工具会将ORMSQL语句翻译成目标数据库平台支持的语法。ORMSQL简化了开发，但是目前的ORMSQL支持的语法主要集中在数据查询上，对DELETE、INSERT、UPDATE以及DDL语句的支持非常有限，而且对常用函数的支持也明显不足。

9.2.5使用SQL翻译器

SQL翻译器是这样一种翻译器，它接受开发人员编写的SQL，然后会将SQL翻译成目标数据库系统支持的SQL语句。比如开发人员编写下面的SQL语句来取得人员表中排名前十的人员：

```
SELECT TOP 10 * FROM T_Person
```

SQL翻译器会将其翻译成目标数据库系统支持的SQL语句：

```
MYSQL:
SELECT * FROM T_Person LIMIT 0, 10
MSSQLServer:
SELECT    TOP 10 * FROM T_Person
Oracle:
SELECT * FROM T_Person WHERE ROWNUM <= 10
DB2:
SELECT * FROM T_Person FETCH FIRST 10 ROWS ONLY
```

SQL翻译器支持完整的SELECT、INSERT、UPDATE、DELETE以及DDL语句语法，而且支持任意复杂度的SQL语句，而且开发人员只要熟悉一种SQL语法就可以了，无需对SQL

语句在不同数据库系统中的实现差异性有了解。

目前SQL翻译器产品有三个，分别是SwisSQL、LDBC和CowNewSQL，SwisSQL是一个非开源的商业公司的公开产品，LDBC和CowNewSQL是开源项目。

9.2.5.1 SwisSQL

SwisSQL是商业公司的一个产品，产品网站为：http://www.swissql.com/。该公司有很多数据库相关产品，SQL翻译器在AdventNetSwisSQLSQLOneAPI.zip中，lib目录下，有一个SwisSQLAPI.jar，这个jar文件中的类就是SQL翻译的入口。使用方式如下：

```
SwisSQLAPI api = new SwisSQLAPI("select top 10 * from T1 order by FId ");
System.out.println("DB2:"+api.convert(SwisSQLAPI.DB2));
```

SwisSQL 支持的数据库非常多，包括 DB2、ORACLE、MYSQL、INFORMIX、MSSQLSERVER、SYBASE、POSTGRESQL 等，但是有如下一些重大的缺陷。

- 部分 SQL 语句解析速度太慢，一些简单 SQL 的解析竟然用了将近一秒钟，这在大并发的系统中显然是无法忍受的；
- 一些重要的 SQL 语句有翻译错误；
- 代码可扩展性非常差，对各个数据库的翻译支持是定义在 SwisSQLStatement 接口的 toOracleString、toMSSQLServerString、toDB2String 等方法中的要扩展的话必须到 SwisSQLStatement 中添加一个新的 toxxxString 方法，并在所有的实现类（共 15 个）中添加相应的实现代码；

9.2.5.2 LDBC

LDBC 是位于 SourceForge 上的开源项目，地址是 http://sourceforge.net/projects/ldbc。要使用 LDBC 需要下载 ldbc.jar，如果需要分析源代码则需要下载 ldbc-src.zip 或者连接其 CVS 服务器。

LDBC 是以一个 JDBC 驱动的形式运行的，它会接管系统的 SQL 请求，将 SQL 翻译成目标平台 SQL，然后再转发给底层数据库的 JDBC 驱动运行。由于 LDBC 是以 JDBC 驱动的形式运行的，所以只要使用标准的 JDBC 方式使用即可。

使用 org.ldbc.jdbcDriver 代替换来的数据库 JDBC 驱动，使用 jdbc:ldbc:<原数据库 URL> 代替原 JDBCURL。无需注册原来的数据库 JDBC 驱动，因为 LDBC 会完成此项工作，当然注册了也不会出现错误。下面是使用 LDBC 连接 PointBase 数据库的方法：

```
Class.forName("org.ldbc.jdbc.jdbcDriver");
Connection conn=DriverManager
    .getConnection("jdbc:ldbc:pointbase:sample",user,password);
```

在通过这样的方式得到的 Connection 中执行 SQL 语句，LDBC 就能自动翻译了。

LDBC 的缺点如下：

（1）对复杂的 SQL 语句不支持，比如报表开发中常用的子查询、union 等不支持；

（2）支持的函数数量非常少，对 DateDiff、DateAdd、Trim、ABS 等函数不支持；

（3）可扩展性差，特别是在函数这方面更差，因为在解析SQL语句的时候把函数名当成和Select、Insert一样的关键字，而不是一个标识名，如果要增加新函数的支持必须在语法文件中增加新的关键字；

9.2.5.3 CowNewSQL

CowNewSQL 也是一个开源产品，开源项目地址：http://www.cownew.com。CowNewSQL 有如下优点：

- 支持较复杂的语法。对于子查询、union 等都有良好的支持，支持查询数据库元数据、数据表管理、索引管理、外键管理等高级特性。
- 支持 MYSQL、MSSQLServer、Oracle、DB2 等主流数据库系统。
- 支持的函数比较丰富，包括 ABS、ACOS、ATAN、CEILING、COS、EXP、LOG、RAND、SQRT 等数学函数，CURDATE、DATEADD、DATEDIFF、DAYOFMONTH、HOUR、MONTHS_BETWEEN、DAYS_BETWEEN 等日期函数，CHARINDEX、LEFT、LENGTH、LCASE、REPLACE、RIGHT、SUBSTRING 等字符串函数，ISNULL、NULLIF 等逻辑函数，TO_INT、TO_NUMBER 等转型函数，SUM、MAX、MIN、AVG 等聚合函数。
- 支持函数的各种常见别名，提高了容错性。比如 NOW 与 GETDATE、CHAR 与 CHR、LENGTH 与 LEN、LCASE 与 LOWER、SUBSTRING 与 SUBSTR、TOCHAR 与 TO_CHAR。
- 解析过程分层明确。首先将源 SQL 语句翻译为一棵抽象语法树（AST），然后将此 AST 交给各个数据库平台的翻译器去翻译成平台相关 SQL。
- 易于增加新的数据库支持。根据 SQL 语句生成的 AST 是一棵普通的树，因此任何熟悉树操作的开发人员都可以开发对应数据库平台的翻译器，并不需要开发人员有编译原理方面的基础；
- 易于增加新的函数支持。翻译器把函数名当成普通的标识符看待，这样增加新的函数支持无需修改语法文件，只要在方法翻译器（MethodTranslator）中添加数行代码即可，这也无需开发人员有编译原理的基础；

其缺点如下：对 Informix、Sybase、Firbird、Posgtreql 等数据库系统还没有提供支持；目前仅能在 Java 平台下使用，不过在其发布的开发计划列表中，已经将对.Net、Python 等语言的支持加入了开发计划中。

在这三种 SQL 翻译器产品中，CowNewSQL 拥有非常大的优势，因此下面我们将会对 CowNewSQL 进行详细的介绍。

9.3 CowNewSQL翻译器

CowNewSQL支持如下几种类型的SQL语句：CreateTable/AlterTable/DropTable/CreateIndex/DropIndex/Select/Insert/Delete/Update/Show；支持子查询、Join、Union等高级的SQL特性；支持日期（包括取当前日期、从日期中提取任意部分、计算日期差异、日期前后推算等）、数学（包括取绝对值、取PI值、四舍五入、对数计算、随机数等）、字符串（包括取子字符串、取字符串长度、字符串截断、大小写转换等）、基本数据处理（包括数字字符串互转、日期转字符串、非空判断等）等函数。

9.3.1 CowNewSQL 支持的数据类型

整数类型：int、integer、tinyint、smallint。
布尔类型：bit、 boolean。
字符类型：varchar、tinytext、longtext、text、longvarchar、char、nchar、nvarchar、clob、nclob。
数值类型：decimal、numeric、real、float、double。
时间日期类型：datetime、date、timestamp、time。
二进制类型：blob、tinyblob、longblob、binary、varbinary、longvarbinary、image。

9.3.1.1 CowNewSQL 中日期常量的表示

各个不同数据库中对日期常量的表示各不相同，如果混用的话很容易造成数据错误，因此 CowNewSQL 采用了特殊的标识来表示日期常量，格式如下：
{‘日期值’}。举例：{‘2008-08-08’}、{‘2007-07-06’}。

9.3.1.2 字符串的连接

各个不同数据库中字符串的连接也是不同的，如果混用的话很容易造成数据错误，因此 CowNewSQL 采用了特殊的方式来表示字符串的连接，格式如下：
字符串 1||字符串 2。

举例：’abce’||’1235’、f1||’.exe’。

9.3.2 CowNewSQL 支持的 SQL 语法

9.3.2.1 Create Table

```
CREATE TABLE   tableName (
  columnDefinition [,...]
  [,PRIMARY KEY(column [,...])]
  [,FOREIGN KEY(column [,...]) REFERENCES tableName ( column [,...])]
  [UNIQUE (column [,...])]
)
```

例句：

（1）标准 SQL：

```
create table T_Person(FId VARCHAR(20),FName varchar(20),FAge int,primary key FId)
```

MSSQLServer、MYSQL、DB2 下的翻译结果：

```
CREATE TABLE T_Person (FId VARCHAR(20),FName VARCHAR(20),
FAge INT,PRIMARY KEY (FId))
```

Oracle 的翻译结果：

```
CREATE TABLE T_Person (FId VARCHAR2(20),FName VARCHAR2(20),
FAge NUMBER (10),PRIMARY KEY (FId))
```

（2）标准 SQL：

```
create table T_SaleInvoiceDetails(FId VARCHAR(20),FParentId varchar(20),
FMaterialId varchar(20),FCount int default 0,FPrice decimal(10,2),
primary key FId,
FOREIGN KEY FParentId REFERENCES T_SaleInvoice(FId),
FOREIGN KEY FMaterialId REFERENCES T_Material(FId))
```

MYSQL,DB2 的翻译结果:

```
CREATE TABLE T_SaleInvoiceDetails (FId VARCHAR(20),FParentId
VARCHAR(20),FMaterialId VARCHAR(20),FCount INT DEFAULT 0,FPrice
DECIMAL(10,2),PRIMARY KEY (FId),FOREIGN KEY (FParentId) REFERENCES
T_SaleInvoice(FId),FOREIGN KEY (FMaterialId) REFERENCES T_Material(FId))
```

MSSQLServer 的翻译结果:

```
CREATE TABLE T_SaleInvoiceDetails (FId VARCHAR(20),FParentId
VARCHAR(20),FMaterialId VARCHAR(20),FCount INT DEFAULT 0,FPrice
NUMERIC(10,2),PRIMARY KEY (FId),FOREIGN KEY (FParentId) REFERENCES
T_SaleInvoice(FId),FOREIGN KEY (FMaterialId) REFERENCES T_Material(FId))
```

Oracle 的翻译结果:

```
CREATE TABLE T_SaleInvoiceDetails (FId VARCHAR2(20),FParentId
VARCHAR2(20),FMaterialId VARCHAR2(20),FCount NUMBER (10) DEFAULT 0,FPrice
```

NUMERIC(10,2),PRIMARY KEY (FId),FOREIGN KEY (FParentId) REFERENCES T_SaleInvoice(FId),FOREIGN KEY (FMaterialId) REFERENCES T_Material(FId))

9.3.2.2 DropTable

DROP TABLE tableName

例句：drop table T_SaleInvoice

9.3.2.3 Create Index

CREATE INDEX indexName ON tableName (columnName [,...])

例句：

create index idx1 on T_Person(FName)

create index idx2 on T_Person(FName,FAge)

9.3.2.4 Drop Index

DROP INDEX tableName.indexName

例句：标准 SQL： DROP INDEX T_Person.idx1

MSSQLServer 的翻译结果：

DROP INDEX T_Person.idx1

MYSQL 的翻译结果：

DROP INDEX idx1 ON T_Person

Oracle、DB2 的翻译结果：

DROP INDEX idx1

9.3.2.5 SELECT

SELECT [DISTINCT] TOP N { * | selectList }

FROM tableList

[WHERE condition]

[GROUP BY columnName [,...]]

[HAVING condition]

[ORDER BY columnName [{ASC|DESC}] [,...]]

[UNION [ALL]] SELECT

tableList:

tableName [alias] [, | LEFT OUTER JOIN | INNER JOIN tableList]

selectList:

{ tableName.* | expression [AS alias]} [,...]

例句：

（1）标准 SQL：

select trim(person.FName),bill.FNumber from T_SaleInvoice as bill left join T_Person person on bill.FSalesPersonId=person.FId

MSSQLServer 的翻译结果：

SELECT LTRIM(RTRIM(person.FName)),bill.FNumber FROM T_SaleInvoice bill LEFT JOIN T_Person person ON bill.FSalesPersonId = person.FId

MYSQL、Oracle 的翻译结果：

SELECT LTRIM(RTRIM(person.FName)),bill.FNumber FROM T_SaleInvoice bill LEFT JOIN T_Person person ON bill.FSalesPersonId = person.FId

DB2 的翻译结果：

SELECT LTRIM(RTRIM(person.FName)),bill.FNumber FROM T_SaleInvoice bill LEFT JOIN

T_Person person ON (bill.FSalesPersonId = person.FId)
（2）标准 SQL：
select * from T_StockFlow where FBillId in('a','b') or FBillId in(select FBillId from T_StockFlow)
（3）标准 SQL：
select * from t1 union all select * from t2
（4）标准 SQL：
select top 10 * from t where fid<any(select fid from t2)
MSSQLServer 的翻译结果：
SELECT TOP 10 * FROM t WHERE fid < ANY(SELECT fid FROM t2)
MYSQL 的翻译结果：
SELECT * FROM t WHERE fid < ANY(SELECT fid FROM t2) LIMIT 0, 10
Oracle 的翻译结果：
SELECT * FROM t WHERE fid < ANY(SELECT fid FROM t2) AND ROWNUM <= 10
DB2 的翻译结果:
SELECT * FROM t WHERE fid < ANY(SELECT fid FROM t2) FETCH FIRST 10 ROWS ONLY
（5）标准 SQL：
select
to_char(ceil(3333/10)),FLOOR(3.4),MOD(2,3),LOG(1),RAND(),RAND(2.0),ROUND(3.1415926,3),SIGN(3),now()
MSSQLServer 的翻译结果：
SELECT CONVERT(VARCHAR, CEILING(3333 / 10)), FLOOR(3.4) ,2 % 3, LOG(1) ,RAND(),RAND(2.0), ROUND(3.1415926,3) , SIGN(3) ,GETDATE()
MYSQL 的翻译结果：
SELECT CONCAT(", ceil(3333 / 10)), FLOOR(3.4) ,MOD(2 , 3), LOG(1) ,RAND(),RAND(2.0), ROUND(3.1415926,3) , SIGN(3) ,NOW()
Oracle 的翻译结果：
SELECT TO_CHAR(CEIL(3333 / 10)) , FLOOR(3.4) ,MOD(2 , 3), LN(1) ,(select dbms_random.value from dual),(select dbms_random.value*2.0 from dual),ROUND(3.1415926, 3), SIGN(3) ,SYSDATE FROM DUAL
DB2 的翻译结果:
SELECT LTRIM(RTRIM(CHAR(CEILING(3333 / 10)))), FLOOR(3.4) , MOD(2,3) , LOG(1) ,(select SYSFUN.rand() from SYSIBM.SYSDUMMY1),(select SYSFUN.rand()*2.0 from SYSIBM.SYSDUMMY1), ROUND(3.1415926,3) , SIGN(3) ,CURRENT TIMESTAMP FROM SYSIBM.SYSDUMMY1
9.3.2.6 Insert
INSERT INTO tableName [(columnName [,...])] VALUES (value [,...])
INSERT INTO tableName [(columnName [,...])] SELECT ...
例句：（1）标准 SQL：
insert into T_SaleInvoice values('sv001','sv001','p001',{'2007-08-8'})
MSSQLServer 的翻译结果：
INSERT INTO T_SaleInvoice VALUES ('sv001', 'sv001', 'p001', '2007-08-8')
MYSQL 的翻译结果：
INSERT INTO T_SaleInvoice VALUES ('sv001', 'sv001', 'p001', '2007-08-8')

Oracle 的翻译结果：
INSERT INTO T_SaleInvoice VALUES ('sv001', 'sv001', 'p001', TO_DATE('2007-08-8', 'YYYY-MM-DD HH24:MI:SS'))
（2）标准 SQL：
insert into T_Person(FId,FName,FAge) values('p002','小红',22)
（3）标准 SQL：
insert into T_StockFlow(FBillId,FDetailId,FDate,FBillTypeName,FAmount)
select detail.FParentId,detail.FId,parent.FDate,' 销 售 发 票 ',detail.FCount*detail.FPrice from T_SaleInvoiceDetails detail left join T_SaleInvoice parent on parent.FId=detail.FParentId
9.3.2.7 Delete
DELETE [*] FROM tableName
[WHERE condition]
（1）标准 SQL：
delete from T_SaleInvoiceDetails where FCount=?
（2）标准 SQL：
delete * from T_SaleInvoiceDetails where FCount<>0 or FPrice!=0
（3）标准 SQL：
delete from T_SaleInvoice where FId in(select FParentId from T_SaleInvoiceDetails)
9.3.2.8 Update
UPDATE tableName
SET columnName=expression [,...]
[WHERE condition]
例句：
（1）标准 SQL：
update T_SaleInvoice set FDate=DateAdd(dd,-1,Now())
MSSQLServer 的翻译结果：
UPDATE T_SaleInvoice SET FDate = DATEADD(dd, -1, GETDATE())
MYSQL 的翻译结果：
UPDATE T_SaleInvoice SET FDate = DATE_ADD(NOW(), INTERVAL -1 DAY)
Oracle 的翻译结果：
UPDATE T_SaleInvoice SET FDate = (SYSDATE + TRUNC(-1))
DB2 的翻译结果:
UPDATE T_SaleInvoice SET FDate = CURRENT TIMESTAMP+(-1) DAY
9.3.2.9 Show
Show 语句主要用来查看系统中的表定义、字段定义、支持的函数等。由于各个数据库系统中取得这些元信息的方式各有不同，经常需要关联查询很多系统表才能得到，为而 CowNewSQL 创新性的设计了 Show 系列语句，这样使用非常短的语句就可以实现以前需要编写很复杂的语句才能实现的功能。
Show 语句语法列表：
（1）SHOW TABLES：显示系统中默认 Schema 下的所有表的表名。
（2）SHOW TABLES SchemaName 定 Schema 下的所有表的表名。比如：SHOW TABLES DEMO。
（3）SHOW FUNCTIONS：显示系统支持的函数的列表。
（4）SHOW FUNCTIONCOLUMNS：显示系统支持的函数参数以及返回值的详细说明。

（5）SHOW TABLECOLUMNS tablename 列信息。比如：SHOW TABLECOLUMNS table_1。

9.3.3 CowNewSQL 支持的函数

CowNewSQL 支持的函数是非常多的，而且为了方便使用还为支持一些函数的别名。下面是 CowNewSQL 支持的函数列表：

函数名(别名)	说明	例子
ABS	绝对值	ABS(-1)
ACOS	反余弦	ACOS(1)
ADD_DAYS ADDDAYS	将日期增加指定个天数	ADDDAYS (now(),3)
ADD_HOURS ADDHOURS	将日期增加指定个小时数	ADDHOURS (now(),3)
ADD_MINUTES ADDMINUTES	将日期增加指定个分钟数	ADDMINUTES (now(),3)
ADD_MONTHS	将日期增加指定个月数	ADD_MONTHS(now(),3)
ADD_SECONDS ADDSECONDS	将日期增加指定个秒数	ADDSECONDS (now(),3)
ADD_YEARS ADDYEARS	将日期增加指定个年数	ADDYEARS(now(),3)
ASCII	取得字符的 ASCII 值。	ASCII(‘a’)
ASIN	反正弦	ASIN(1)
ATAN	反正切	ATAN(1)
ATN2/ATAN2	反正切	ATAN2(1,2)
AVG	聚合函数，返回表达式的平均值。	AVG(FCount)
CEILING/CEIL	返回大于或等于所给数字表达式的最小整数。	CEIL(2.3)
CHARINDEX	返回字符串中指定表达式的起始位置。CHARINDEX(expression1 , expression2)。expression1 一个表达式，其中包含要寻找的字符的次序。expression2 一个表达式，通常是一个用于搜索指定序列的列。	CHARINDEX(‘very cd’,’w’)
CHR	取得数字对应的字符（ASCII 的反函数）	CHR(80)
CONCAT	连接两个字符串。	CONCAT(‘hi’,’world’)
CONVERT	进行数值类型转换。格式 CONVERT(类型,值)，“类型”可取值为：CHAR、VARCHAR、NCHAR、NVARCHAR、DATETIME、DATE、INT、DECIMAL。	CONVERT（‘INT’,’3’）、CONVERT（‘DATE’,’2008-08-08’）、CONVERT（‘VARCHAR’,3）
COS	余弦	COS(1)
COT	余切	COT(1)
COUNT	聚合函数，返回组中项目的数量。	COUNT(*)
CURDATE	取当前日期	CURDATE()

CURTIME	去当前时间	CURTIME()
DATEADD 、DATE_ADD	在向指定日期加上一段时间的基础上，返回新的 datetime 值。DATEADD (datepart , number, date)。datepart 是规定应向日期的哪一部分返回新值的参数，有如下取值“Year（缩写 yy, yyyy）”、“quarter（缩写 qq, q）”、“Month（缩写 mm, m）”、“dayofyear（缩写 dy, y）”、“Day（缩写 dd, d）”、“Week（缩写 wk, ww）”、“Hour（缩写 hh）”、“minute（缩写 mi, n）”、“second（缩写 ss, s）”、“millisecond（缩写 ms）”。number 是用来增加 datepart 的值。date 是返回 datetime 或 smalldatetime 值或日期格式字符串的表达式。	DateADD(month,3, Now())
DATEDIFF 、DATE_DIFF	返回跨两个指定日期的日期和时间边界数。DATEADD 。datepart 是规定了应在日期的哪一部分计算差额的参数，取值同“DATEADD”。startdate 是计算的开始日期。enddat 是计算的终止日期。	DATEDIFF(day, pubdate, getdate())
DATENAME	返回代表指定日期的指定日期部分的字符串。DATENAME (datepart , date)。datepart 是指定应返回的日期部分的参数，取值同“DATEADD”。	DATENAME（q,now()）
DAYNAME	返回指定日期是星期几。	DAYNAME(bdate)
DAYOFMONTH	返回代表指定日期的在一个月中的天数。	DAYOFMONTH(now())
DAYOFWEEK	返回代表指定日期的在一周中的天数。	DAYOFWEEK (now())
DAYOFYEAR	返回代表指定日期的在一年中的天数。	DAYOFYEAR (now())
DAYS_BETWEEN DAYSBETWEEN	返回两个日期之间的天数间隔	DAYSBETWEEN(d2,d2)
DEGREE DEGREES	弧度转角度	DEGREE(2)
EXP	自然指数	EXP(2)
FLOOR	返回小于或等于所给数字表达式的最大整数。	FLOOR(-2.4)
HOUR	返回指定日期的小时部分	HOUR(now())
ISNULL NVL	使用指定的替换值替换 NULL。ISNULL (check_expression , replacement_value) ：check_expression 将被检查是否为 NULL 的表达式；replacement_value 在 check_expression 为 NULL 时将返回的表达式。如果 check_expression 不为 NULL，那么返回该表达式的值；否则返回 replacement_value。	ISNULL(f1,f2) 、NVL (f1,f2)
LCASE LOWER	返回指定字符串的小写形式。	LCASE(‘aBcD’) 、LOWER(‘1AAcB’)
LEFTSTR	返回从字符串左边开始指定个数的字符。LEFT (character_expression , integer_expression) 。	LEFTSTR(‘abcd’,2)
LENGTH LEN	返回给定字符串表达式的字符（而不是字节）个数。	LENGTH(‘123’) 、LEN(‘012’)

LOG	求自然对数	LOG(3)
LOG10	求以 10 为低的对数	LOG10(4)
LTRIM	删除起始空格后返回字符表达式。	LTRIM(‘ aa f ’)
MAX	聚合函数，返回表达式的最大值。	MAX(price)
MIN	聚合函数，返回表达式的最小值。	MIN (price)
MINUTE	返回指定日期的分钟部分	MINUTE (now())
MOD	整除	3 MOD 4
MONTH	返回指定日期的月份部分	MONTH (now())
MONTHNAME	返回指定日期的月份的名称	MONTHNAME (now())
MONTHS_BETWEEN MONTHSBETWEEN	返回两个日期之间的月份间隔	MONTHS_BETWEEN(d1,d2)
NOW GETDATE TODAY	取得当前时间	NOW()
NULLIF	如果两个指定的表达式相等，则返回空值。NULLIF (expression , expression)：如果两个表达式不相等，NULLIF 返回第一个 expression 的值。如果相等，NULLIF 返回第一个 expression 类型的空值。	NULLIF(f1、f2)
PI	取圆周率	PI()
POWER POW	求 a 的 b 次方	POWER(3,5)
QUARTER	返回指定日期的季度部分	QUARTER(now())
RADIANS	角度转弧度	RADIANS(3)
RAND	取得随机数。如果没有参数，则返回 0..1 之间的一个随机数，如果有参数则返回大于 0、小于指定参数的一个随机数。	RAND() 、RAND(108)
REPLACE	用第三个表达式替换第一个字符串表达式中出现的所有第二个给定字符串表达式。REPLACE ('string_expression1' , 'string_expression2' , 'string_expression3')	REPLACE(‘abc%123%www’,’%’,’@’)
RIGHTSTR	返回字符串中从右边开始指定个数的 integer_expression 字符。RIGHT (character_expression , integer_expression)。	RIGHTSTR(‘aaaa’, 2)
ROUND	四舍五入,第一个参数表示要进行四舍五入的值，第二个表示要取的精度	ROUND(3.38332,3)
RTRIM	截断所有尾随空格后返回一个字符串。	RTRIM (‘ aa f ’)
SECOND	返回指定日期的秒部分	SECOND (now())
SIGN	取一个数的符号，如果此数为正数则返回 1，为负数则返回 -1，为 0 则返回 0.	SIGN(-8)
SIN	正弦	SIN(3)
SOUNDEX	返回由四个字符组成的代码 (SOUNDEX) 以评估两个字符串的相似性。SOUNDEX (character_expression)	SOUNDEX(‘cownew’)
SQRT	开方	SQRT(3)

SUBSTRING SUBSTR	返回字符串表达式的一部分。SUBSTRING (expression , start , length)：expression 是字符串表达式；start 是一个整数，指定子串的开始位置；length 是一个整数，指定子串的长度（要返回的字符数或字节数）。	SUBSTRING(‘ok aha’,1,3)
SUM	聚合函数，返回表达式中所有值的和。	SUM(amount)
TAN	正切	TAN(1)
TO_DATE TODATE	将指定的数值转换为日期类型	TODATE(‘2008-08-08’)
TO_INT TOINT	将指定的值转换为整数类型。	TOINT(3.14) 、TO_INT(‘3.65’)
TO_NUMBER TONUMBER	将指定的值转换为数值类型。	TO_NUMBER(‘3.14’)
TOCHAR TO_CHAR	将指定的值转换为字符串类型。	TOCHAR(33) 、TO_CHAR (33)
TRIM	截断开头和结尾的空格后返回一个字符串。	TRIM (‘ aa f ’)
UCASE UPPER	返回指定字符串的大写形式。	UCASE (‘aBcD’)、UPPER (‘1AAcB’)
WEEK	返回指定日期的在一年中的第几周	WEEK (now())
YEAR	返回指定日期的年部分，4 位数	YEAR (now())

9.3.4 CowNewSQL 的使用方法

CowNewSQL 目前只支持在 Java 语言中调用。CowNewSQL 支持直接调用翻译器和 JDBC 驱动两种使用方式。下面分别进行介绍。

9.3.4.1 直接调用翻译器

首先到 CowNewSQL 的网站上下载最新的 CowNewSQL，下载完成后解压安装包，将 retrotranslator-runtime-1.0.7.jar、antlr.jar、commons-lang-2.3.jar 以及 cownewsql.jar 加入 CLASSPATH，调用类 com.cownew.cownewsql.imsql.common.DialectManager 的 createTranslator 方法创建一个翻译器，然后调用翻译器的 translateSQL 方法来将标准 SQL 语句翻译成特定数据库平台的 SQL 语句。

DialectManager 的 createTranslator 方法接受一个字符串类型的参数来指定目标数据库的类型，目前接受四个值”MSSQLServer”、”MYSQL”、“DB2”和”Oracle”。

由于被翻译的 SQL 语句有可能是多句，所以翻译器的 translateSQL 方法返回值为字符串数组，每个数据元素表示一个翻译后的语句。

举例：

```
import com.cownew.cownewsql.imsql.ISQLTranslator;
import com.cownew.cownewsql.imsql.common.DataBaseTypeEnum;
import com.cownew.cownewsql.imsql.common.DialectManager;
import com.cownew.cownewsql.imsql.common.TranslateException;

public class Main
{
    public static void main(String[] args) throws TranslateException
    {
        ISQLTranslator tx = DialectManager
                .createTranslator(“MYSQL”);
```

```
        String[] venderSQLs;
        venderSQLs = tx.translateSQL("select top 10 * from t");
        System.out.println("翻译后的SQL:" + venderSQLs[0]);
    }
}
```

运行结果：翻译后的SQL:SELECT * FROM t LIMIT 0, 10

9.3.4.2 JDBC 驱动方式使用

CowNewSQL 提供了以 JDBC 驱动方式使用的支持（支持最新的 JDBC4.0 标准）。通过这种方式用户无需修改系统的任何代码，只要修改原有的 JDBC 连接字符串就可以轻松的将 CowNewSQL 融入系统，使用 CowNewSQL 的 JDBC 驱动后系统中所有的 SQL 语句在送到数据库系统中执行前都将会自动进行翻译。

CowNewSQL 的 JDBC 驱动类为：com.cownew.cownewsql.imsql.jdbc.DBDriver；连接字符串格式为：jdbc:cownewsql:目标数据库类型:目标数据库 JDBC 驱动类:原 JDBC 连接字符串。

使用方式举例：

原有程序连接到 Oracle 数据库，使用的 Oracle 驱动类为 oracle.jdbc.driver.OracleDriver，JDBC 连接字符串为：jdbc:oracle:thin:@192.168.88.128:1521:XE。

我们只要将 CowNewSQL 的 Jar 包（包括 cownewsql***.jar、antlr.jar、commons-lang**.jar、retrotranslator-runtime**.jar 等）加入程序的 ClassPath，然后修改使用的数据库驱动为：com.cownew.cownewsql.imsql.jdbc.DBDriver，然后修改 JDBC 连接字符串为：jdbc:cownewsql:oracle:oracle.jdbc.driver.OracleDriver:jdbc:oracle:thin:@192.168.88.128:1521:XE。

使用 JDBC 驱动方式不仅可以支持直接调用 JDBC 来进行数据库操作的系统，而且对于通过数据库连接池或者其他 JDBC 驱动包装方式访问数据库的方式都适合，这样我们无需修改任何代码就将 CowNewSQL 翻译器轻松的植入系统。

第九章 高级话题

9.1 SQL 注入漏洞攻防

//9.2 SQL、ORM 与 OODBMS

9.3 SQL 调优

合理规划索引、合理管理索引（更新索引还是删除后重建索引）

9.4 事务

9.5 自动编号的产生

//9.7 范式不是凡是

12 主键的生成策略、主键与业务键

13 NULL 的学问与注意事项

//9.11 数据库管理工具在位编辑更新的秘密

第十章 高级话题

本章将讨论一些数据库开发中的高级话题，包括 SQL 注入漏洞攻防、数据库调优、范式等。

10.1 SQL 注入漏洞攻防

系统中安全性是非常重要的，为了保证安全性很多解决方案被应用到系统中，比如架设防火墙防止数据库服务器直接暴露给外部访问者、使用数据库的授权机制防止未授权的用户访问数据库，这些解决方案可以很大程度上避免了系统受攻击，但是如果开发人员不注意 SQL 的安全性造成了 SQL 注入漏洞，那么所有的这些解决方案都形同虚设了，因为通过 SQL 注入漏洞，恶意访问者可以堂而皇之的对数据库进行任意操作，因为恶意访问者破坏数据库时所使用的一切操作都看起来是合法。我们来看一下什么是 SQL 注入漏洞。

10.1.1 SQL 注入漏洞原理

到目前为止，本书演示的 SQL 语句都是静态的 SQL 语句，比如下面的 SQL 用于校验用户名为“admin”的用户的密码是否是“123456”，如果密码正确则 PwdCorrect 的值为 true，否则为 false：

```
SELECT (FPassword='123456') AS PwdCorrect
FROM T_User
WHERE FUser='admin'
```

在实际开发过程中一般是开发人员提供一个界面，允许用户输入用户名和密码，然后程序读取用户输入用户名和密码来构造 SQL 语句来校验用户名和密码是否正确。实现的代码如下：

```
string user=txtUser.getText();
string password = txtPassword.getText();
rs = ExuecuteQuery("SELECT (FPassword='"+password+"') AS PwdCorrect  FROM T_User WHERE FUser='"+password+"'");
if(rs.getBool("PwdCorrect ")==true)
{
    ShowMessage("密码正确");
}
else
{
    ShowMessage("密码错误");
}
```

这里采用拼接字符串的方式根据用户录入的用户名和密码来构建 SQL 语句，如果用户名为“guest”，密码为“123456”，那么拼接出来的 SQL 语句如下：

```
SELECT (FPassword='123456') AS PwdCorrect
FROM T_User
WHERE FUser='guest'
```

这看起来是没有任何问题的，但是试想如果恶意攻击者在用户名输入框中随意输入一个“abc”，而在密码数据框中输入“1' or '1'='1”，那么拼接出来的 SQL 语句如下：

```
SELECT (FPassword='1' or '1'='1') AS PwdCorrect
FROM T_User
WHERE FUser='abc'
```

由于“'1'='1'”这个表达式永远返回 true，而 true 与任何布尔值的 or 运算的结果都是 true，那么无论正确密码是什么“FPassword='1' or '1'='1'”的计算值永远是 true，这样恶意攻击者就可以使用任何帐户登录系统了。十分惊讶吧！这样的漏洞就被称作“SQL 注入漏洞（SQL Injection）”。

对付 SQL 注入漏洞有两种方式：过滤敏感字符和使用参数化 SQL。

10.1.2 过滤敏感字符

过滤敏感字符的思路非常简单，由于恶意攻击者一般需要在输入框中输入的文本一般含有 or、and、select、delete 之类的字符串片段，所以在拼接 SQL 之前检查用户提交的文本中是否含有这些敏感字符串，如果含有则终止操作。示例代码如下：

```
string user=txtUser.getText();
string password = txtPassword.getText();
//校验是否含有敏感字符
if(user.contains("or","and","select","delete"))
{
    ShowMessage("可能存在注入漏洞攻击！");
    return;
}
if(password.contains("or","and","select","delete"))
{
    ShowMessage("可能存在注入漏洞攻击！");
    return;
}
rs = ExuecuteQuery("SELECT (FPassword='"+password+"') AS PwdCorrect  FROM T_User WHERE FUser='"+password+"'");
if(rs.getBool("PwdCorrect ")==true)
{
    ShowMessage("密码正确");
}
else
{
    ShowMessage("密码错误");
}
```

这种方式能够过滤大部分注入漏洞攻击，但是有如下两个缺陷：

- 给正常用户的正常操作造成了麻烦。比如一个正常的用户的密码是“more”、“select”甚至就是“1' or '1'='1”，它们是没有恶意的，但是在点击【提交】按钮后，系统却弹出了一个报错信息，用户必须将密码修改为一个不包含这些敏感字符串的密码，无疑这造成系统给用户的友好性非常差。国内著名的 CMS 产品“动易 CMS”采用的就是这种方式来防止注入漏洞攻击的，这带来的麻烦就是如果用户要发表一个 SQL 语句相关的文章，因为文章中有大量敏感字符，这造成这篇文章几乎无法发表。
- 逻辑难以严谨。尽管过滤了大部分的敏感字符串，但是攻击者是非常聪明的，他们也许能构造一个能够骗过敏感字符串过滤的字符串从而绕过这道“防火墙”。谈到安全性的时候人们都会说：所有用户输入都是不可信的！

10.1.3 使用参数化 SQL

Java、C#等语言提供了参数化 SQL 机制，使用参数化 SQL 开发人员为在运行时才能确定的参数值设置占位符，在执行的时候再指定这些占位符所代表的值。示例代码如下：

```
string user=txtUser.getText();
string password = txtPassword.getText();
query = CreateQuery("SELECT (FPassword=:password) AS PwdCorrect  FROM T_User WHERE FUser=:user");
```

```
query.SetParameter(":password ",password);
query.SetParameter(":user", user);
if(rs.getBool("PwdCorrect ")==true)
{
    ShowMessage("密码正确");
}
else
{
    ShowMessage("密码错误");
}
```

在上面的例子中，为运行时才能确定的用户名和密码设置了占位符，然后在运行时再设定占位符的值，在执行时 Java、C#会直接将参数化 SQL 以及对应的参数值传递给 DBMS，在 DBMS 中会将参数值当成一个普通的值来处理而不是将它们拼接到参数化 SQL 中，因此从根本上避免了 SQL 注入漏洞攻击。建议开发人员使用参数化 SQL 来代替字符串拼接，不过如果开发的时候采用的 ASP、PHP 等语言，那么由于这些语言没有提供参数化 SQL 机制，因此只能采用其它方式来避免了 SQL 注入漏洞攻击。

10.2 SQL 调优

在使用 DBMS 时经常对系统的性能有非常高的要求：不能占用过多的系统内存和 CPU 资源、要尽可能快的完成的数据库操作、要有尽可能高的系统吞吐量。如果系统开发出来不能满足要求的所有性能指标，则必须对系统进行调整，这个工作被称为调优。绝对 DBMS 的性能的因素有两个因素：硬件和软件。使用频率的的 CPU、使用多处理器、加大内存容量、增加 Cache、提高网络速度等这些都是非常有效的硬件调优方式，不过对硬件进行调优对系统性能的提高是有限的，如果有非常好的硬件条件但是如果编写的 SQL 质量非常差的话系统的性能并不会有明显的改善，而如果能对 SQL 语句进行充分的优化的话即使硬件条件稍差的话，系统性能的变化也是非常惊人的。硬件的调优涉及到非常多的内容，不是本书所能覆盖的，因此本节将主要讲解 SQL 的调优。

10.2.1 SQL 调优的基本原则

“二八原理”是一个普遍的真理，特别是在计算机的世界中表现的更加明显，那就是 20%的代码的资源消耗占用了 80%的总资源消耗。SQL 语句也是一种代码，因此它也符合这个原理。在进行 SQL 调优的时候应该把主要精力放到这 20%的最消耗系统资源的 SQL 语句中，不要想把所有的 SQL 语句都调整到最优状态。

很多 DBMS 都提供了非常好的工具用来分析系统中所有 SQL 语句资源消耗的工具，借助于这些工具发现占用系统资源排在前面的 SQL 语句，然后尝试对它们进行优化，优化后再次执行分析，迭代这一过程，直到系统中没有明显的系统资源消耗异常的 SQL 语句为止。

10.2.2 索引

索引是数据库调优的最根本的优化方法，很多优化手法都是围绕索引展开的，可以说索引是一切优化手法的“内功”，而所有的优化手法都是由索引衍化出来的招式而已。

根据索引的顺序与数据表的物理顺序是否相同，可以把索引分成两种类型：聚簇索引，数据表的物理顺序与索引顺序相同；非聚簇索引，数据表的物理顺序与索引顺序不相同。下面，我们举例来说明一下聚集索引和非聚集索引的区别：

字典的目录就是一种索引，因为通过目录我们可以很快的定位到要检索的内容，而不用从头到尾把字典翻一遍。汉语字典一般都至少提供两种目录，一种是拼音目录，一种是偏旁部首目录。汉语字典是按照拼音的顺序排列的，因此拼音目录就是聚集索引，

而偏旁部首目录则是非聚集索引。应该在表中经常搜索的列或者按照顺序访问的列上创建聚簇索引。当创建聚簇索引时要需要每一个表只能有一个聚簇索引，因为表中数据的物理顺序只能有一个，而非聚集索引则可以创建多个。

由于索引需要占据一定的存储空间，而且索引也会降低数据插入、更新和删除的速度，所以应该只创建必要的索引，一般是在检索的时候用的字段中创建索引。

索引还会造成存储碎片的问题。当删除一条记录时将会导致对应的索引中的该记录的对应项为空，由于索引是采用 B 树结构存储的，所以对应的索引项并不会被删除，经过一段时间的增删改操作后，数据库中就会出现大量的存储碎片，这和磁盘碎片、内存碎片产生原理是类似的，这些存储碎片不仅占用了存储空间，而且降低了数据库运行的速度。如果发现索引中存在过多的存储碎片的话就要进行“碎片整理”了，最方便的“碎片整理”手段就是重建索引，重建索引会将先前创建的索引删除然后重新创建索引，主流数据库管理系统都提供了重建索引的功能，比如 REINDEX、REBUILD 等，如果使用的数据库管理系统没有提供重建索引的功能，可以首先用 DROP INDEX 语句删除索引，然后用 ALTER TABLE 语句重新创建索引。

10.2.3 表扫描和索引查找

一般地，系统访问数据库中的数据，可以使用两种方法：全表扫描和索引查找。

全表扫描，就是指系统必须在数据表中逐条检索表中的每条记录，以检查该记录是否匹配检索条件。全表扫描有可能会造成巨大的性能损失，当然也有可能不会影响性能，这取决于表中的数据量，如果表中有上千万条甚至上亿条记录的话，全表扫描的速度会非常慢，而如果表中只有几条、几十条记录的话表扫描的性能消耗就可以忽略不计了。当表中数据量比较小的时候，使用全表扫描非常有用。但是随着表中数据量的增加，全表扫描会导致系统性能严重下降。

如果表中有索引并且待匹配条件符合索引的要求的话，DBMS 就不会执行全表扫描，而是直接到索引中查找，这将大大加快检索的速度。

DBMS 中都有查询优化器，它会根据分布的统计信息生成该查询语句的优化执行规划，以提高访问数据的效率为目标，确定是使用全表扫描还是使用索引查找。注意并不是表中存在索引在进行检索的时候就会使用索引查找，如果使用不当检索的过程仍然会是采用全表扫描，这样索引就起不到效果了。关于如何才能避免全表扫描的发生我们会在下一节中介绍。

10.2.4 优化手法

下面将会列出了一些常用的优化手法，注意这些优化手法只是一些常规条件下的优化手法，具体的优化效果是与使用的 DBMS 以及数据的特点密切相关的，需要根据具体情况来使用不同的优化手法，如果使用不当的话有可能会适得其反。

10.2.4.1 创建必要的索引

在经常需要进行检索的字段上创建索引，比如经常要按照图书名称进行检索，那么就应该在图书名称字段上创建索引，如果经常要按照员工部门和员工岗位级别进行检索，那么就应该在员工部门和员工岗位级别这两个字段上创建索引。创建索引给检索带来的性能提升往往是巨大的，因此在发现检索速度过慢的时候应该首先想到的就是创建索引。

10.2.4.2 使用预编译查询

程序中通常是根据用户的输入来动态执行 SQL 语句，这时应该尽量使用参数化 SQL，这样不仅可以避免 SQL 注入漏洞攻击，最重要数据库会对这些参数化 SQL 执行预编译，这样第一次执行的时候 DBMS 会为这个 SQL 语句进行查询优化并且执行预编译，这样以后再执行这个 SQL 的时候就直接使用预编译的结果，这样可以大大提高执

行的速度。

10.2.4.3 调整 WHERE 子句中的连接顺序

DBMS 一般采用自下而上的顺序解析 WHERE 子句，根据这个原理,表连接最好写在其他 WHERE 条件之前，那些可以过滤掉最大数量记录。

比如下面的 SQL 语句性能较差：

```
SELECT *
FROM T_Person
WHERE    FSalary > 50000
AND      FPosition= ‘MANAGER’
AND      25 < (SELECT COUNT(*) FROM T_Manager
WHERE FManagerId=2);
```

我们将子查询的条件放到最前面，下面的 SQL 语句性能比较好：

```
SELECT *
FROM T_Person
WHERE
25 < (SELECT COUNT(*) FROM T_Manager
WHERE FManagerId=2)
AND FSalary > 50000
AND      FPosition= ‘MANAGER’ ;
```

10.2.4.4 SELECT 语句中避免使用'*'

SELECT *比较简单，但是除非确实需要检索所有的列，否则将会检索出不需要的列，这回增加网络的负载和服务器的资源消耗；即使确实需要检索所有列，也不要使用 SELECT *，因为这是一个非常低效的方法，DBMS 在解析的过程中，会将*依次转换成所有的列名，这意味着将耗费更多的时间。

10.2.4.5 尽量将多条 SQL 语句压缩到一句 SQL 中

每次执行 SQL 的时候都要建立网络连接、进行权限校验、进行 SQL 语句的查询优化、发送执行结果，这个过程是非常耗时的，因此应该尽量避免过多的执行 SQL 语句，能够压缩到一句 SQL 执行的语句就不要用多条来执行。

比如 T_Person 表中有如下的初始数据：

FId	FName	FGroupId
0	ABC	
1	CBA	
2	NBA	
3	WTO	
4	WHO	
5	Tom	
6	Jim	
7	Ham	
8	Mam	
9	Lily	
10	Lucy	
11	Linda	
12	Robert	
13	John	

14	Wiki	
15	SVN	

T_Person 表中保存是人员信息，FId 字段为主键，FName 字段为姓名，FGroupId 为分组号。我们需要将这些人员进行分组，每三人一组，将组号更新到 FGroupId 字段中。

可以在宿主语言中通过代码来实现这个功能：

```
//当前组号
int groupid=0;
//计数器
int counter=0;
rs = ExecuteQuery("SELECT FId FROM T_Person");
while(rs.next())
{
    //计数器增加 1
    counter = counter+1;
    if(counter==3)
    {
        //计数器清零
        counter=0;
        //组号加一
        groupid = groupid+1;
    }
    id = rs.Get("FId");
    //将主键为 id 的记录的 FGroupId 字段更新为组号 groupid
    update = CreateUpdate(
        "UPDATE T_Person  SET  FGroupId =:groupId WHERE  FId =:id");
    update.SetParameter(":groupId ", groupid);
    update.SetParameter(":id", id);
    ExecuteUpdate(update);
}
```

这个算法非常简单，逐条处理 T_Person 表中数据，这在数据量小的时候并没有什么问题，但是当表中的数据有大量的数据时性能就非常差了。假设表中 1 万条数据，那么一共就需要执行 10001 次数据库操作，速度将会让人无法忍受。

我们分析 T_Person 表中数据特点，发现 FId 字段是主键，因此每条记录中 FId 字段都是唯一的，同时 FId 字段的值是连续递增的，因此可以更新 FGroupId 的值为 FId 与 3 整除后的值。根据整除运算的特性，FId 等于 0 的记录的 FGroupId 字段被更新为 0，FId 等于 1 的记录的 FGroupId 字段被更新为 0，FId 等于 2 的记录的 FGroupId 字段被更新为 0；FId 等于 3 的记录的 FGroupId 字段被更新为 1，FId 等于 4 的记录的 FGroupId 字段被更新为 1，FId 等于 5 的记录的 FGroupId 字段被更新为 1；FId 等于 6 的记录的 FGroupId 字段被更新为 2……，以此类推，这样所有的记录就被很快的分组了。实现这样的分组操作只要一个 UPDATE 语句即可完成：

```
UPDATE T_Person  SET  FGroupId =FId/3;
```

10.2.4.6 用 Where 子句替换 HAVING 子句

避免使用 HAVING 子句，因为 HAVING 只会在检索出所有记录之后才对结果集

进行过滤。如果能通过 WHERE 子句限制记录的数目，那就能减少这方面的开销。HAVING 中的条件一般用于聚合函数的过滤，除此而外，应该将条件写在 WHERE 子句中。

10.2.4.7 使用表的别名

当在 SQL 语句中连接多个表时，请使用表的别名并把别名前缀于每个列名上。这样就可以减少解析的时间并减少那些由列名歧义引起的语法错误。

10.2.4.8 用 EXISTS 替代 IN

在查询中，为了满足一个条件，往往需要对另一个表进行联接，在这种情况下，使用 EXISTS 而不是 IN 通常将提高查询的效率，因为 IN 子句将执行一个子查询内部的排序和合并。下面的语句 2 就比语句 1 效率更加高。

语句 1：

```
SELECT * FROM T_Employee
WHERE FNumber> 0
AND FDEPTNO IN (SELECT FNumber
FROM T_Department
WHERE FMangerName = 'Tome')
```

语句 2：

```
SELECT * FROM T_Employee
WHERE FNumber > 0
AND EXISTS (SELECT 1
FROM T_Department
WHERE T_Department. FDEPTNO = EMP.FNumber
AND FMangerName = ‘MELB’)
```

10.2.4.9 用表连接替换 EXISTS

通常来说，表连接的方式比 EXISTS 更有效率，因此如果可能的话尽量使用表连接替换 EXISTS。下面的语句 2 就比语句 1 效率更加高。

语句 1：

```
SELECT FName FROM T_Employee
WHERE EXISTS
(
    SELECT 1 FROM T_Department
    WHERE T_Employee.FDepartNo= FNumber
    AND FKind='A'
);
```

语句 2：

```
SELECT FName FROM T_Department, T_Employee
WHERE T_Employee. FDepartNo = T_Departmen. FNumber
AND FKind = ‘A’ ;
```

10.2.4.10 避免在索引列上使用计算

在 WHERE 子句中，如果索引列是计算或者函数的一部分，DBMS 的优化器将不会使用索引而使用全表扫描。

例如下面的 SQL 语句用于检索月薪的 12 倍大于两万五千元的员工：

```
SELECT *FROM T_Employee
WHERE FSalary * 12 >25000;
```

由于在大于号左边的是 FSalary 与 12 的成绩表达式，这样 DBMS 的优化器将不会使用字段 FSalary 的索引，因为 DBMS 必须对 T_Employee 表进行全表扫描，从而计算 FSalary * 12 的值，然后与 25000 进行比较。将上面的 SQL 语句修改为下面的等价写法后 DBMS 将会使用索引查找，从而大大提高了效率：

```
SELECT *FROM T_Employee
WHERE FSalary >25000/12;
```

同样的，不能在索引列上使用函数，因为函数也是一种计算，会造成全表扫描。下面的语句 2 就比语句 1 效率更加高。

语句 1：

```
SELECT * FROM T_Example
WHERE ABS(FAmount)=300
```

语句 2：

```
SELECT * FROM T_Example
WHERE FAmount=300 OR FAmount=-300
```

10.2.4.11 用 UNION ALL 替换 UNION

当 SQL 语句需要 UNION 两个查询结果集合时，即使检索结果中不会有重复的记录，如果使用 UNION 这两个结果集同样会尝试进行合并，然后在输出最终结果前进行排序。

因此，如果检索结果中不会有重复的记录的话，应该用 UNION ALL 替代 UNION，这样效率就会因此得到提高。下面的语句 2 就比语句 1 效率更加高。

语句 1：

```
SELECT ACCT_NUM, BALANCE_AMT
FROM DEBIT_TRANSACTIONS1
WHERE TRAN_DATE = '20010101'
UNION
SELECT ACCT_NUM, BALANCE_AMT
FROM DEBIT_TRANSACTIONS2
WHERE TRAN_DATE ='20010102'
```

语句 2：

```
SELECT ACCT_NUM, BALANCE_AMT
FROM DEBIT_TRANSACTIONS1
WHERE TRAN_DATE ='20010101'
UNION ALL
SELECT ACCT_NUM, BALANCE_AMT
FROM DEBIT_TRANSACTIONS2
WHERE TRAN_DATE = '20010102'
```

10.2.4.12 避免隐式类型转换造成的全表扫描

T_Person 表的字符串类型字段 FLevel 为人员的级别，在 FAge 字段上建有索引。我们执行下面的 SQL 语句用于检索所有级别等于 10 的员工：

```
SELECT FId,FAge,FName
FROM T_Person
WHERE FAge=10
```

在这个 SQL 语句中，将字符串类型字段 FLevel 与数值 10 进行比较，由于在大部分数据库中隐式转换类型中数值类型的优先级高于字符串类型，因此 DBMS 会对 FAge

字段进行隐式类型转换，相当于执行了下面的 SQL 语句：

```
SELECT FId,FAge,FName
FROM T_Person
WHERE TO_INT(FAge)=10
```

由于在索引字段上进行了计算，所以造成了索引失效而使用全表扫描。因此应将 SQL 语句做如下修改：

```
SELECT FId,FAge,FName
FROM T_Person
WHERE FAge='10'
```

10.2.4.13 防止检索范围过宽

如果 DBMS 优化器认为检索范围过宽，那么它将放弃索引查找而使用全表扫描。下面是几种可能造成检索范围过宽的情况：

使用 IS NOT NULL 或者不等于判断，可能造成优化器假设匹配的记录数太多。

使用 LIKE 运算符的时候，"a%"将会使用索引，而"a%c"和"%c"则会使用全表扫描，因此"a%c"和"%c"不能被有效的评估匹配的数量。

10.3 事务

如果要执行一系列的操作，而这些操作最终是以整体的原子操作的形式完成的话，事务就是必须的。关于事务的理论中，银行转账问题是最经典的例子：当把钱从一个银行帐号转移至另外一个银行帐号的时候，这个操作要由两个步骤来完成，首先要将资金从一个银行帐号取出，然后再将其存入另一个银行帐号。如果资金已经从一个银行帐号取出了，在将资金存入另一个银行帐号之前或者进行当中发生异常情况(包括程序内部异常、服务器当机、目标帐号被冻结)，如果没有事务保护就会出现源帐号中的资金已经减少了，但是目标帐号中的资金并没有增加的状况。

事务是关键业务系统开发中非常关键性的服务，对于关键性业务系统如果没有采用事务，那么这个系统可以说是不可用的。

10.3.1 事务简介

从严格的定义来讲“事务是形成一个逻辑工作单位的数据库操作的汇集”。通俗的说，事务(Transaction)是能以整体的原子操作形式完成的一系列操作，事务能保证一个“全有或者全无”的命题的成立，即操作或者全部成功或者全部失败，不会出现部分成功、部分失败的情况。事务以一种可靠的、简洁的方式来解决系统运行中的各种异常问题。

事务具有 4 个基本特性，即原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)和持久性(Durability)。简称 ACID 特性。

- 原子性：一个事务中所有的数据库操作，是一个不可分割的整体，这些操作要么全部执行，要么全部无效果。
- 一致性：一个事务独立执行的结果，将保持数据库的一致性，即数据不会因事务的执行而被破坏。在事务执行过程中，可以违反一致性原则，并产生一个临时的不一致状态。比如在转账过程中，会出现暂时的数据不一致的情况。当事务结束后，系统又回到一致的状态。不过临时的一致性不会导致问题，因为原子性会使得系统始终保持一致性。
- 隔离性：在多个事务并发执行的时候，系统应该保证与这些事务先后单独执行时的结果一样，即并发执行的任务不必关心其他事务。对于每一个事务来讲，那一刻看起来好像只有它在修改数据库一样。事务系统是通过对后台数据库数据使用同步协议来实现隔离性的。同步协议使一个事务与另外一个事务相分离。如果事务对数据进行了锁定，可以使并发的事务无法影响该数据，直到锁定解除为止。

- 持久性：一个事务一旦完成全部操作以后，它对数据库的所有操作将永久地反映在数据库中。持久性保证了系统在操作的时候免遭破坏。持久性主要是为了解决机器故障、突然断电、硬盘损坏等问题而出现的。为了达到持久性，系统一般都保留了一份日志。一旦出现故障，就可以通过日志将数据重建。

10.3.2 事务的隔离

假设同一个 A 和 B 两个同时并发操作数据库，A 和 B 执行的任务如下：从数据库中读取整数 N，将 N 随机加上 10 或者 20，将新的 N 更新回数据库。这两个并发执行的实例可能发生下面的执行顺序。

(1)A 从数据库中读取 N，当前数据库中 N=0；

(2)B 从数据库中读取 N，当前数据库中 N=0；

(3)A 将 N 加 10，并更新入数据库，当前数据库中 N=10

(4)B 将 N 加 20，并更新入数据库，当前数据库中 N=20；

可以看到由于数据库出现了交叉存取的操作，B 所读取的 N 是过期的版本，即 A 在写回数据之前的版本。这样当 B 更新的时候，将会覆盖 A 的操作，这就是著名的“更新丢失”问题。那么应该如何避免这种情况的发生呢？

解决此类问题的方法就是为数据库加锁，以防止多个组件读取数据，通过锁住事务所用的数据，能保证在打开锁之前，只有本事务才能访问数据。这样就避免了交叉存取的问题。这和操作系统中的 PV 操作原理类似。

由于锁将其他并发的事务排除在数据库更新之外，所以这会导致性能的严重下降。为了提高性能，事务将锁分为两种类型：只读锁和写入锁。只读锁是非独占的，多个并发的事务都能获得只读锁；写入锁是独占的，任意时间只能有一个事务可以获得写入锁。

10.3.3 事务的隔离级别

由于隔离性是通过加锁的方式获得的，而锁会降低系统的性能，所以事务提供了控制隔离程度的机制。如果使用较高的隔离级别，则事务会比较好的与其他事务相隔离，当然也会带来大量的系统开销；如果使用较低的隔离级别，则事务的隔离性会比较差，但是能获得更好的性能。

事务的隔离级别分为四种：READ_UNCOMMITED、READ_COMMITED、REPEATABLE_READ、SERIALIZABLE。要理解这些隔离级别的差异必须首先弄清如下几个概念：脏读、不可重复读、幻影读取。

假设同一个 A 和 B 两个同时并发操作数据库，A 和 B 执行的任务如下：从数据库中读取整数 N，将 N 加上 10，将新的 N 更新回数据库。这两个并发执行的实例可能发生下面的执行顺序。

(1)A 从数据库中读取整数 N，当前数据库中 N=0；

(2)N 加上 10，并将其更新到数据库中，当前数据库中 N=10。然而由于 A 的事务还没有提交，所以数据库更新还没有称为持久性的；

(3)B 从数据库中读取整数 N，当前数据库中 N=10；

(4)A 回滚了事务，所以 N 恢复到了 N=0；

(5)B 将 N 加上 10，并将其更新到数据库中，当前数据库中 N=20；

这里出现了 B 在 A 提交之前读取了 A 所更新的数据，由于 A 回滚了事务，所以数据库中出现了错误的数据 20。尽管 A 回滚了事务，但是 A 更新的数据还是间接的通过 B 被更新到了数据库中。这种读取了未提交的数据的方法就叫脏(dirty)读问题。

当一个用户从数据库中读取数据的时候，另外一个用户修改了这条数据，所以数据发生了改变，当再次读取的时候就出现了不可重复读取问题。比如：

(1)A 从数据库中读取整数 N；

(2)B 以一个新的值更新 N；
(3)当 A 再次从数据库中读取 N 的时候，会发现 N 的值变了；

幻影读取指的是在两次数据库操作读取操作之间，一组新的数据会出现在数据库中。比如：
(1)A 从数据库检索到了一些数据；
(2)B 通过 Insert 语句插入了一些新数据；
(3)A 再次查询的时候，新的数据就会出现；

了解了这几个概念，下面来看一下四种事务的隔离级别的区别：

- 使用 READ_UNCOMMITED 级别，会导致脏读问题、幻影读取问题和不可重复读取问题。在需要敏感计算任务的事务中，这样的模式是不太适合的；
- 使用 READ_COMMITED 级别，可以解决脏读问题，但是还会有幻影读取问题和不可重复读取问题。这种级别一般用于制作报表。这种模式是大部分系统的默认级别；
- 使用 REPEATABLE_READ 级别，可以解决脏读问题和不可重复读取问题，但是会有幻影读取问题；
- 使用 SERIALIZABLE 级别可以解决脏读问题、幻影读取问题和不可重复读取问题。这是最严格级别的隔离级别；

10.3.4 事务的使用

主流的 DBMS 都提供了启动、提交以及回滚事务的机制，也提供了指定锁粒度、隔离级别的机制，不过这些机制一般是谁 DBMS 的不同而不同的，请参考具体 DBMS 的说明文档。比如在 MSSQLServer 中执行一个 READ_UNCOMMITED 级别事务的 SQL 语句如下：

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITED
BEGIN TRANSACTION
--具体的操作代码
COMMIT
```

10.4 自动增长字段

在设计数据库的时候，有时需要表的某个字段是自动增长的，最常使用自动增长字段的就是表的主键，使用自动增长字段可以简化主键的生成。不同的 DBMS 中自动增长字段的实现机制也有不同，下面分别介绍。

10.4.1 MYSQL 中的自动增长字段

MYSQL 中设定一个字段为自动增长字段非常简单，只要在表定义中指定字段为 AUTO_INCREMENT 即可。比如下面的 SQL 语句创建 T_Person 表，其中主键 FId 为自动增长字段：

```
CREATE TABLE T_Person
(
FId INT PRIMARY KEY AUTO_INCREMENT,
FName VARCHAR(20),
FAge INT
);
```

执行上面的 SQL 语句后就创建成功了 T_Person 表，然后执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FName,FAge)
VALUES('Tom',18);
```

```
INSERT INTO T_Person(FName,FAge)
VALUES('Jim',81);

INSERT INTO T_Person(FName,FAge)
VALUES('Kerry',33);
```

注意这里的 INSERT 语句没有为 FId 字段设定任何值，因为 DBMS 会自动为 FId 字段设定值。执行完毕后查看 T_Person 表中的内容：

FId	FName	FAge
1	Tom	18
2	Jim	81
3	Kerry	33

可以看到 FId 中确实是自动增长的。

这个例子讲解完了，请删除 T_Person 表：

```
DROP TABLE T_Person;
```

10.4.2 MSSQLServer 中的自动增长字段

MSSQLServer 中设定一个字段为自动增长字段非只要在表定义中指定字段为 IDENTITY 即可，格式为 IDENTITY(startvalue,step)，其中的 startvalue 参数值为起始数字，step 参数值为步长，即每次自动增长时增加的值。

比如下面的 SQL 语句创建 T_Person 表，其中主键 FId 为自动增长字段，并且设定 100 为起始数字，步长为 3：

```
CREATE TABLE T_Person
(
FId INT PRIMARY KEY IDENTITY(100,3),
FName VARCHAR(20),
FAge INT
);
```

执行上面的 SQL 语句后就创建成功了 T_Person 表，然后执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FName,FAge)
VALUES('Tom',18);

INSERT INTO T_Person(FName,FAge)
VALUES('Jim',81);

INSERT INTO T_Person(FName,FAge)
VALUES('Kerry',33);
```

注意这里的 INSERT 语句没有为 FId 字段设定任何值，因为 DBMS 会自动为 FId 字段设定值。执行完毕后查看 T_Person 表中的内容：

FId	FName	FAge
100	Tom	18
103	Jim	81
106	Kerry	33

可以看到 FId 中确实是 100 为起始数字、步长为 3 自动增长的。

这个例子讲解完了，请删除 T_Person 表：

```
DROP TABLE T_Person;
```

10.4.3 Oracle 中的自动增长字段

Oracle 中不像 MYSQL 和 MSSQLServer 中那样指定一个列为自动增长列的方式，不过在 Oracle 中可以通过 SEQUENCE 序列来实现自动增长字段。

在 Oracle 中 SEQUENCE 被称为序列，每次取的时候它会自动增加，一般用在需要按序列号排序的地方。

在使用 SEQUENCE 前需要首先定义一个 SEQUENCE，定义 SEQUENCE 的语法如下：

```
CREATE SEQUENCE sequence_name
INCREMENT BY step
START WITH startvalue;
```

其中 sequence_name 为序列的名字，每个序列都必须有唯一的名字；startvalue 参数值为起始数字，step 参数值为步长，即每次自动增长时增加的值。

一旦定义了 SEQUENCE，你就可以用 CURRVAL 来取得 SEQUENCE 的当前值，也可以通过 NEXTVAL 来增加 SEQUENCE，然后返回 新的 SEQUENCE 值。比如：

```
sequence_name.CURRVAL
sequence_name.NEXTVAL
```

如果 SEQUENCE 不需要的话就可以将其删除：

```
DROP SEQUENCE sequence_name;
```

下面举一个使用 SEQUENCE 序列实现自动增长的例子。

首先创建一个名称为 seq_PersonId 的 SEQUENCE：

```
CREATE SEQUENCE seq_PersonId
INCREMENT BY 1
START WITH 1;
```

然后创建 T_Person 表：

```
CREATE TABLE T_Person
(
FId NUMBER (10) PRIMARY KEY,
FName VARCHAR2(20),
FAge NUMBER (10)
);
```

执行上面的 SQL 语句后就创建成功了 T_Person 表，然后执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FId,FName,FAge)
VALUES(seq_PersonId.NEXTVAL,'Tom',18);

INSERT INTO T_Person(FId,FName,FAge)
VALUES(seq_PersonId.NEXTVAL,'Jim',81);

INSERT INTO T_Person(FId,FName,FAge)
VALUES(seq_PersonId.NEXTVAL,'Kerry',33);
```

注意这里的 INSERT 语句没有为 FId 字段设定任何值，因为 DBMS 会自动为 FId 字段设定值。执行完毕后查看 T_Person 表中的内容：

FID	FNAME	FAGE
1	Tom	18
2	Jim	81
3	Kerry	33

使用 SEQUENCE 实现自动增长字段的缺点是每次向表中插入记录的时候都要显式的到 SEQUENCE 中取得新的字段值，如果忘记了就会造成错误。为了解决这个问题，我们可以使用触发器来解决，创建一个 T_Person 表上的触发器：

```
CREATE OR REPLACE TRIGGER trigger_personIdAutoInc
   BEFORE INSERT ON T_Person
   FOR EACH ROW
DECLARE
BEGIN
   SELECT seq_PersonId.NEXTVAL INTO:NEW.FID FROM DUAL;
END trigger_personIdAutoInc;
```

这个触发器在 T_Person 中插入新记录之前触发，当触发器被触发后则从 seq_PersonId 中取道新的序列号然后设置给 FID 字段。

执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FAge)
VALUES('Wow',22);

INSERT INTO T_Person(FName,FAge)
VALUES('Herry',28);

INSERT INTO T_Person(FName,FAge)
VALUES('Gavin',36);
```

注意在这个 SQL 语句中无需再为 FId 字段赋值。执行完毕后查看 T_Person 表中的内容：

FID	FNAME	FAGE
1	Tom	18
2	Jim	81
3	Kerry	33
4	Wow	22
5	Herry	28
7	Gavin	36

这个例子讲解完了，请删除 T_Person 表以及 SEQUENCE：

```
DROP TABLE T_Person;
DROP SEQUENCE seq_PersonId;
```

10.4.4 DB2 中的自动增长字段

DB2 中实现自动增长字段有两种方式：定义带有 IDENTITY 属性的列；使用 SEQUENCE 对象。

10.4.4.1 定义带有 IDENTITY 属性的列

首先创建 T_Person 表，SQL 语句如下：

```
CREATE TABLE T_Person
(
```

```
FId INT PRIMARY KEY NOT NULL
GENERATED ALWAYS
     AS IDENTITY
     (START WITH 1
      INCREMENT BY 1
     ),
FName VARCHAR(20),
FAge INT
);
```

执行上面的 SQL 语句后就创建成功了 T_Person 表，然后执行下面的 SQL 语句向 T_Person 表中插入一些数据：

```
INSERT INTO T_Person(FName,FAge)
VALUES('Tom',18);

INSERT INTO T_Person(FName,FAge)
VALUES('Jim',81);

INSERT INTO T_Person(FName,FAge)
VALUES('Kerry',33);
```

注意这里的 INSERT 语句没有为 FId 字段设定任何值，因为 DBMS 会自动为 FId 字段设定值。执行完毕后查看 T_Person 表中的内容：

FId	FName	FAge
100	Tom	18
103	Jim	81
106	Kerry	33

这个例子讲解完了，请删除 T_Person 表：

```
DROP TABLE T_Person;
```

10.4.4.2 使用 SEQUENCE 对象

DB2 中的 SEQUENCE 和 Oracle 中的 SEQUENCE 相同，只是定义方式和使用方式略有不同。

下面创建了一个 SEQUENCE：

```
CREATE SEQUENCE seq_PersonId AS INT
INCREMENT BY 1
START WITH 1;
```

使用SEQUENCE的方式如下：

NEXT VALUE FOR sequence_name

这样就可以通过下面的SQL语句来使用SEQUENCE：

```
INSERT INTO T_Person(FId,FName,FAge)
VALUES(NEXT VALUE FOR seq_PersonId,'Kerry',33);
```

如果想在向表中插入记录的时候自动设定 FId 字段的值则同样要使用触发器，具体请参考相关资料，这里不再赘述。

这个例子讲解完了，请删除 seq_PersonId 序列：

```
DROP SEQUENCE seq_PersonId;
```

10.5 业务主键与逻辑主键

一般情况下，一张数据表必须要有一个主键字段，这样这个主键字段就可以唯一标识这条记录了。不过采用什么样的字段来做为主键字段还是一个必须解决的问题，目前有两种常用的主键策略：业务主键与逻辑主键。

业务主键是指采用业务数据中的某个字段做为主键，比如在员工档案表中可以用工号来做为主键、在车辆管理系统中可以用车牌号做为主键字段。逻辑主键，也称代理逐渐，是指不采用任何业务数据做为主键，而是采用一个没有业务意义的不重复值做主键，比如在员工档案表中用一个自动增长的字段来做为主键，这个字段没有任何的业务意义。

使用业务主键是比较简单的，但是会存在潜在的问题，一个是业务主键并不能真正的保证唯一性，第二个是做为主键的数据一旦发生变化就会带来维护的问题。假设在社区人员信息表中使用身份证号码做为主键，由于我国身份证编号制度还存在一定缺陷，所以存在不少的身份证号码重复现象，这样一旦社区中有两个同样身份证号的人员出现，系统将会出现问题；即使能够杜绝身份证号码重复现象，也会存在升级的问题，比如今后出现身份证号码位数升级的问题，那么由于很多表都是通过身份证号码这个主键字段来关联社区人员信息表的，那么不仅要升级社区人员信息表，还要将这些关联表进行升级。因此建议尽量不要用业务字段做主键，而是使用没有业务意义主键。

使用业务主键可以保证主键值的唯一性，并且在业务发生变化时，适应性更强一些。不过使用代理主键也有劣势，那就是主键字段由于没有任何业务意义，所以在使用的时候比较麻烦。不过总的来与逻辑主键比起来，业务主键更有优势，因此除非有特别的里有，否则使用逻辑主键是一个好的习惯。

如果决定采用逻辑主键的话，使用什么样的主键生成策略则是必须考虑的。常用的主键生成策略有：自动增长字段和 UUID。使用自动增长字段就是每次向表中插入记录的时候 DBMS 自动为主键设定一个自动增长的值；使用 UUID 则是为主键字段设置一个 UUID 类型的值，这个 UUID 值采用 UUID 算法来生成，这样可以保证生成的值是绝对唯一的。

使用自动增长字段的优势在于速度比较快，根据统计 UUID 算法要比自动增长字段的生成速度慢约 30 倍；使用自动增长字段的劣势在于进行表合并的时候会存在冲突的问题，比如 A 表和 B 表的结构完全相同，而且它们都采用自动增长字段来生成主键，如果想将 A 表和 B 表合并为一张表的话那么就有可能由于 A 表中的记录的主键值和 B 表中的记录的主键值冲突而造成合并失败，而使用 UUID 算法则不会有这个问题，因为 UUID 算法能够保证两个 UUID 值是唯一的。

10.6 NULL 的学问

在数据库中存在一种特殊的值：NULL（空值）。一个字段如果没有被赋值，那么它的值就是 NULL，NULL 并不代表没有值而是表示值未知。员工信息表中存储着身份证号、姓名、年龄等信息，其中某条记录中年龄字段的值为 NULL，并不表示这个员工没有年龄，而只是他的年龄暂时不知道。因此，在数据库中 NULL 主要用于标识一个字段的值为“未知”。

由于 NULL 在数据库中是比较特殊的，所以在涉及到 NULL 的一些处理中也会存在一些需要特别注意的地方。为了更加清晰的讲解我们将创建一张表，执行下面的 SQL 语句：

MYSQL,MSSQLServer,DB2:

```
CREATE TABLE T_Employee
(
    FId VARCHAR(20),
```

```
    FName VARCHAR(20),
    FSalary INT
)
```

Oracle:

```
CREATE TABLE T_Employee
(
    FId VARCHAR2(20),
    FName VARCHAR2(20),
    FSalary NUMBER (10)
)
```

T_Employee 表保存了员工信息，FId 字段为主键，FName 字段为员工姓名，FSalary 字段为员工工资。请在相应的 DBMS 中执行相应的 SQL 语句，然后执行下面的 SQL 语句向 T_Employee 表中插入一些演示数据：

```
INSERT INTO T_Employee(FId,FName,FSalary)
VALUES('1','Tom',3000);
INSERT INTO T_Employee(FId,FName,FSalary)
VALUES('2','Jim',NULL);
INSERT INTO T_Employee(FId,FName,FSalary)
VALUES('3',NULL,8000);
INSERT INTO T_Employee(FId,FName,FSalary)
VALUES('4','Lily',9000);
INSERT INTO T_Employee(FId,FName,FSalary)
VALUES('5','Robert',2000);
```

执行完毕查看 T_Employee 表中的内容：

FId	FName	FSalary
1	Tom	3000
2	Jim	<NULL>
3	<NULL>	8000
4	Lily	9000
5	Robert	2000

10.6.1 NULL 与比较运算符

NULL 表示未知的值，因此在使用比较运算符的时候就需要注意 NULL 值可能造成的 BUG。比如有的开发人员认为下面的 SQL 语句将返回 Jim、Robert、Tom 三个人的工资，因为他认为 NULL 等于 0：

```
SELECT * FROM T_Employee
WHERE FSalary<5000
```

可是执行上面的查询语句后却得到了下面的结果：

FId	FName	FSalary
1	Tom	3000
5	Robert	2000

Jim 并没有像预想的那样被检索出来。这是因为 NULL 不等于 0，它代表“未知”，Jim 的工资未知，所以 DBMS 不会认为它的工资小于 5000，所以它并不会被检索出来。

有的开发人员认为下面的 SQL 语句将返回所有员工的工资，因为所有员工的工资肯定不是大于 5000 就是小于等于 5000：

SELECT * FROM T_Employee
WHERE FSalary<5000 OR FSalary>=5000

可是执行上面的查询语句后却得到了下面的结果：

FId	FName	FSalary
1	Tom	3000
3	<NULL>	8000
4	Lily	9000
5	Robert	2000

同样，Jim 并没有像预想的那样被检索出来。因为貌似这个查询条件包含了所有的工资金额，可以 DBMS 是无法确认 NULL 值是不是在这个范围之内的，因此 Jim 并不会被检索出来。

因此为了检索所有工资小于 5000 元的员工，包括工资额未知的员工，必须使用 IS NULL 运算符，SQL 语句如下：

SELECT * FROM T_Employee
WHERE FSalary<5000 OR FSalary IS NULL

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FSalary
1	Tom	3000
2	Jim	<NULL>
5	Robert	2000

10.6.2 NULL 和计算字段

如果 NULL 值出现在任何计算字段中，那么计算结果永远是 NULL。为了验证这一点请执行下面的 SQL 语句：

SELECT FId,FName, FSalary ,FSalary+2000
FROM T_Employee

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FSalary	FSalary+2000
1	Tom	3000	5000
2	Jim	<NULL>	<NULL>
3	<NULL>	8000	10000
4	Lily	9000	11000
5	Robert	2000	4000

第二行记录的 FSALARY 字段为 NULL，为一个未知的工资增加 2000 元得到的仍然是未知工资 NULL，这是完全符合逻辑的。

如果这个结果不符合业务系统的要求可以通过两种方式来解决这个问题，一个是过滤掉 NULL 值，一个是将 NULL 值转换为业务系统认为的值。

第一种解决方式例子如下，这里用 IS NOT NULL 运算符将 NULL 值过滤掉：

SELECT FId,FName, FSalary ,FSalary+2000
FROM T_Employee
WHERE FSalary IS NOT NULL

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FSalary	FSalary+2000
1	Tom	3000	5000
3	<NULL>	8000	10000

4	Lily	9000	11000
5	Robert	2000	4000

第二种解决方式例子如下，这里使用 CASE 函数将 NULL 值转换为 0，也就是认为工资未知的工资为 0：

```
SELECT FId,FName, FSalary ,
(CASE
    WHEN FSalary IS NULL THEN 0
    ELSE FSalary
END
)+2000
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FSalary	
1	Tom	3000	5000
2	Jim	<NULL>	2000
3	<NULL>	8000	10000
4	Lily	9000	11000
5	Robert	2000	4000

10.6.3 NULL 和字符串

如果 NULL 值出现在任何和字符串相关计算字段中，那么计算结果永远是 NULL。为了验证这一点请执行下面的 SQL 语句：

MYSQL,Oracle:

```
SELECT FId,FName,FName||'LOL',FSalary
FROM T_Employee
```

MSSQLServer:

```
SELECT FId,FName,FName+'LOL',FSalary
FROM T_Employee
```

DB2:

```
SELECT FId,FName,CONCAT(FName,'LOL'),FSalary
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName		FSalary
1	Tom	TomLOL	3000
2	Jim	JimLOL	<NULL>
3	<NULL>	<NULL>	8000
4	Lily	LilyLOL	9000
5	Robert	RobertLOL	2000

第三行记录的 FName 字段为 NULL，为一个未知姓名的员工的名字后增加“LOL”得到的仍然是未知姓名 NULL，这是完全符合逻辑的。

如果这个结果不符合业务系统的要求，同样可以采用 10.6.2 的解决方案，这里不再赘述。

10.6.4 NULL 和函数

如果 NULL 值出现在普通函数中，那么计算结果永远是 NULL。为了验证这一点请执行下面的 SQL 语句：

SELECT FId,FName, FSalary ,ABS(FSalary-5000)
FROM T_Employee

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FName	FSalary	
1	Tom	3000	2000
2	Jim	<NULL>	<NULL>
3	<NULL>	8000	3000
4	Lily	9000	4000
5	Robert	2000	3000

第二行记录的 FSalary 字段为 NULL，对一个未知值进行函数计算得到的仍然是未知 NULL，这是完全符合逻辑的。

如果这个结果不符合业务系统的要求，同样可以采用 10.6.2 的解决方案，这里不再赘述。

10.6.5 NULL 和聚合函数

和普通的函数不同，如果 NULL 值出现在聚合函数中，那么 NULL 值将会被忽略。为了验证这一点请执行下面的 SQL 语句：

```
SELECT MAX(FSalary) AS MAXSALARY,
MIN(FSalary) AS MINSALARY,
COUNT(FSalary)
FROM T_Employee
```

执行完毕我们就能在输出结果中看到下面的执行结果：

MAXSALARY	MINSALARY	
9000	2000	4

按照前面的分析，一个包含 NULL 值在内的所有员工工资的的最大值和最小值应该是未知 NULL，不过聚合函数是一个例外，NULL 值将会被忽略。这是需要特别注意的。

10.6.6 诀窍

处理含有 NULL 值的运算是非常麻烦的，不过只要记住“NULL 代表未知”这一原则就可以灵活应对很多问题。下面举几个例子：

条件表达式“NULL=3”的返回值为 NULL，因为无法确认 3 是否与一个未知值相等；“NULL=NULL”的返回值也为 NULL，因为无法确认两个未知值是否相等；“NULL<>NULL”的返回值也为 NULL，因为同样无法确认两个未知值是否不相等。

表达式“NULL AND TRUE”的返回值为 NULL，因为无法确认一个未知值与 TRUE 进行 AND 运算的结果；表达式“NULL AND FALSE”的返回值为 TRUE，因为任何一个布尔值与 FALSE 进行 AND 运算的结果都为 TRUE，虽然 NULL 表示未知值，但是 NULL 同样不是 TRUE 就是 FALSE；表达式“NULL OR TRUE”的返回值为 TRUE，因为任何一个布尔值与 TRUE 进行 OR 运算的结果都为 TRUE，虽然 NULL 表示未知值，但是 NULL 同样不是 TRUE 就是 FALSE；表达式“NULL OR FALSE”的返回值为 TRUE，因为无法确认一个未知值与 FALSE 进行 OR 运算的结果。

这个例子讲解完了，请删除 T_Employee 表：

```
DROP TABLE T_Employee;
```

10.7 开窗函数

在开窗函数出现之前存在着很多用 SQL 语句很难解决的问题，很多都要通过复杂的相关子查询或者存储过程来完成。为了解决这些问题，在 2003 年 ISO SQL 标准加入了开窗

函数，开窗函数的使用使得这些经典的难题可以被轻松的解决。目前在 MSSQLServer、Oracle、DB2 等主流数据库中都提供了对开窗函数的支持，不过非常遗憾的是 MYSQL 暂时还未对开窗函数给予支持，因此本节中的例子将无法在 MYSQL 中运行通过。

为了更加清晰的讲解开窗函数我们将创建一张表，执行下面的 SQL 语句：

MYSQL,MSSQLServer,DB2:

```
CREATE TABLE T_Person (FName VARCHAR(20),FCity VARCHAR(20),
FAge INT,FSalary INT)
```

Oracle:

```
CREATE TABLE T_Person (FName VARCHAR2(20),FCity VARCHAR2(20),
FAge INT,FSalary INT)
```

T_Person 表保存了人员信息，FName 字段为人员姓名，FCity 字段为人员所在的城市名，FAge 字段为人员年龄，FSalary 字段为人员工资。请在相应的 DBMS 中执行相应的 SQL 语句，然后执行下面的 SQL 语句向 T_Person 表中插入一些演示数据：

```
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Tom','BeiJing',20,3000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Tim','ChengDu',21,4000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Jim','BeiJing',22,3500);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Lily','London',21,2000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('John','NewYork',22,1000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('YaoMing','BeiJing',20,3000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Swing','London',22,2000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Guo','NewYork',20,2800);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('YuQian','BeiJing',24,8000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Ketty','London',25,8500);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Kitty','ChengDu',25,3000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Merry','BeiJing',23,3500);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Smith','ChengDu',30,3000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Bill','BeiJing',25,2000);
INSERT INTO T_Person(FName,FCity,FAge,FSalary)
VALUES('Jerry','NewYork',24,3300);
```

执行完毕查看 T_Person 表中的内容：

FNAME	FCITY	FAGE	FSALARY
Tom	BeiJing	20	3000
Tim	ChengDu	21	4000
Jim	BeiJing	22	3500
Lily	London	21	2000
John	NewYork	22	1000
YaoMing	BeiJing	20	3000
Swing	London	22	2000
Guo	NewYork	20	2800
YuQian	BeiJing	24	8000
Ketty	London	25	8500
Kitty	ChengDu	25	3000
Merry	BeiJing	23	3500
Smith	ChengDu	30	3000
Bill	BeiJing	25	2000
Jerry	NewYork	24	3300

10.7.1 开窗函数简介

与聚合函数一样，开窗函数也是对行集组进行聚合计算，但是它不像普通聚合函数那样每组只返回一个值，开窗函数可以为每组返回多个值，因为开窗函数所执行聚合计算的行集组是窗口。在 ISO SQL 规定了这样的函数为开窗函数，在 Oracle 中则被称为分析函数，而在 DB2 中则被称为 OLAP 函数。

要计算所有人员的总数，我们可以执行下面的 SQL 语句：

```
SELECT COUNT(*) FROM T_Person
```

除了这种较简单的使用方式，有时需要从不在聚合函数中的行中访问这些聚合计算的值。比如我们想查询每个工资小于 5000 元的员工信息（城市以及年龄），并且在每行中都显示所有工资小于 5000 元的员工个数，尝试编写下面的 SQL 语句：

```
SELECT FCITY , FAGE , COUNT(*)
FROM T_Person
WHERE FSALARY<5000
```

执行上面的 SQL 以后我们会得到下面的错误信息：

```
选择列表中的列 'T_Person.FCity' 无效，因为该列没有包含在聚合函数或 GROUP BY 子句中。
```

这是因为所有不包含在聚合函数中的列必须声明在 GROUP BY 子句中，可以进行如下修改：

```
SELECT FCITY, FAGE, COUNT(*)
FROM T_Person
WHERE FSALARY<5000
GROUP BY FCITY , FAGE
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FCITY	FAGE	
BeiJing	20	2
NewYork	20	1
ChengDu	21	1
London	21	1
BeiJing	22	1

London	22	1
NewYork	22	1
BeiJing	23	1
NewYork	24	1
BeiJing	25	1
ChengDu	25	1
ChengDu	30	1

这个执行结果与我们想像的是完全不同的，这是因为 **GROUP BY 子句对结果集进行了分组，所以聚合函数进行计算的对象不再是所有的结果集，而是每一个分组。**

可以通过子查询来解决这个问题，SQL 如下：

```
SELECT FCITY , FAGE ,
(
    SELECT COUNT(* ) FROM T_Person
    WHERE FSALARY<5000
)
FROM T_Person
WHERE FSALARY<5000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FCITY	FAGE	
BeiJing	20	13
ChengDu	21	13
BeiJing	22	13
London	21	13
NewYork	22	13
BeiJing	20	13
London	22	13
NewYork	20	13
ChengDu	25	13
BeiJing	23	13
ChengDu	30	13
BeiJing	25	13
NewYork	24	13

虽然使用子查询能够解决这个问题，但是子查询的使用非常麻烦，使用开窗函数则可以大大简化实现，下面的 SQL 语句展示了如果使用开窗函数来实现同样的效果：

```
SELECT FCITY , FAGE , COUNT(*) OVER()
FROM T_Person
WHERE FSALARY<5000
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FCITY	FAGE	
BeiJing	20	13
ChengDu	21	13
BeiJing	22	13
London	21	13

NewYork	22	13
BeiJing	20	13
London	22	13
NewYork	20	13
ChengDu	25	13
BeiJing	23	13
ChengDu	30	13
BeiJing	25	13

可以看到与聚合函数不同的是，开窗函数在聚合函数后增加了一个 OVER 关键字。开窗函数的调用格式为：

函数名(列) OVER(选项)

OVER 关键字表示把函数当成开窗函数而不是聚合函数。SQL 标准允许将所有聚合函数用做开窗函数，使用 OVER 关键字来区分这两种用法。

在上边的例子中，开窗函数 `COUNT(*) OVER()`对于查询结果的每一行都返回所有符合条件的行的条数。OVER 关键字后的括号中还经常添加选项用以改变进行聚合运算的窗口范围。如果 OVER 关键字后的括号中的选项为空，则开窗函数会对结果集中的所有行进行聚合运算。

10.7.2 PARTITION BY 子句

开窗函数的 OVER 关键字后括号中的可以使用 PARTITION BY 子句来定义行的分区来供进行聚合计算。与 GROUP BY 子句不同，PARTITION BY 子句创建的分区是独立于结果集的，创建的分区只是供进行聚合计算的，而且不同的开窗函数所创建的分区也不互相影响。下面的 SQL 语句用于显示每一个人员的信息以及所属城市的人员数：

```
SELECT FName,FCITY , FAGE , FSalary,
COUNT(*) OVER(PARTITION BY FCITY)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FCITY	FAGE	FSalary	
Tom	BeiJing	20	3000	6
Jim	BeiJing	22	3500	6
YaoMing	BeiJing	20	3000	6
YuQian	BeiJing	24	8000	6
Merry	BeiJing	23	3500	6
Bill	BeiJing	25	2000	6
Smith	ChengDu	30	3000	3
Kitty	ChengDu	25	3000	3
Tim	ChengDu	21	4000	3
Lily	London	21	2000	3
Ketty	London	25	8500	3
Swing	London	22	2000	3
Guo	NewYork	20	2800	3
John	NewYork	22	1000	3
Jerry	NewYork	24	3300	3

`COUNT(*) OVER(PARTITION BY FCITY)`表示对结果集按照FCITY进行分区，并且计算

当前行所属的组的聚合计算结果。比如对于FName等于Tom的行，它所属的城市是BeiJing，同属于BeiJing的人员一共有6个，所以对于这一列的显示结果为6。

在同一个SELECT语句中可以同时使用多个开窗函数，而且这些开窗函数并不会相互干扰。比如下面的SQL语句用于显示每一个人员的信息、所属城市的人员数以及同龄人的人数：

```
SELECT FName,FCITY, FAGE, FSalary,
COUNT(*) OVER(PARTITION BY FCITY),
COUNT(*) OVER(PARTITION BY FAGE)
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FCITY	FAGE	FSalary		
Tom	BeiJing	20	3000	6	3
YaoMing	BeiJing	20	3000	6	3
Guo	NewYork	20	2800	3	3
Tim	ChengDu	21	4000	3	2
Lily	London	21	2000	3	2
Jim	BeiJing	22	3500	6	3
John	NewYork	22	1000	3	3
Swing	London	22	2000	3	3
Merry	BeiJing	23	3500	6	1
YuQian	BeiJing	24	8000	6	2
Jerry	NewYork	24	3300	3	2
Kitty	ChengDu	25	3000	3	3
Bill	BeiJing	25	2000	6	3
Ketty	London	25	8500	3	3
Smith	ChengDu	30	3000	3	1

在这个查询结果中，可以看到同一城市中的`COUNT(*) OVER(PARTITION BY FCITY)`计算结果相同，而且同龄人中的`COUNT(*) OVER(PARTITION BY FAGE)`计算结果也相同。

10.7.2 ORDER BY子句

MSSQLServer中是不支持开窗函数中的ORDER BY子句的，因此本节演示的内容只适用于Oracle和DB2。开窗函数中可以在OVER关键字后的选项中使用ORDER BY子句来指定排序规则，而且有的开窗函数还要求必须指定排序规则。使用ORDER BY子句可以对结果集按照指定的排序规则进行排序，并且在一个指定的范围内进行聚合运算。ORDER BY子句的语法为：

ORDER BY 字段名 RANGE|ROWS BETWEEN 边界规则1 AND 边界规则2

RANGE表示按照值的范围进行范围的定义，而ROWS表示按照行的范围进行范围的定义；边界规则的可取值见下表：

可取值	说明	示例
CURRENT ROW	当前行	
N PRECEDING	前N行	2 PRECEDING
UNBOUNDED PRECEDING	一直到第一条记录	
N FOLLOWING	后N行	2 FOLLOWING
UNBOUNDED FOLLOWING	一直到最后一条记录	

“RANGE|ROWS BETWEEN 边界规则1 AND 边界规则2”部分用来定位聚合计算范

围，这个子句又被称为定位框架。下面通过例子来展示ORDER BY子句的用法。

例1

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN UNBOUNDED PRECEDING
    AND CURRENT ROW)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	1000
Lily	2000	3000
Swing	2000	5000
Bill	2000	7000
Guo	2800	9800
Tom	3000	12800
YaoMing	3000	15800
Kitty	3000	18800
Smith	3000	21800
Jerry	3300	25100
Jim	3500	28600
Merry	3500	32100
Tim	4000	36100
YuQian	8000	44100
Ketty	8500	52600

这里的开窗函数“`SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)`”表示按照FSalary进行排序，然后计算从第一行（`UNBOUNDED PRECEDING`）到当前行（`CURRENT ROW`）的和，这样的计算结果就是按照工资进行排序的工资值的累积和。

例2

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FSalary RANGE BETWEEN UNBOUNDED PRECEDING
    AND CURRENT ROW)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	1000
Lily	2000	7000
Swing	2000	7000
Bill	2000	7000
Guo	2800	9800
Tom	3000	21800
YaoMing	3000	21800
Kitty	3000	21800
Smith	3000	21800

Jerry	3300	25100
Jim	3500	32100
Merry	3500	32100
Tim	4000	36100
YuQian	8000	44100
Ketty	8500	52600

这个SQL语句与例1中的SQL语句唯一不同的就是“**ROWS**”被替换成了“RANGE”。“**ROWS**”是按照行数进行范围定位的，而“RANGE”则是按照值范围进行定位的，这两个不同的定位方式主要用来处理并列排序的情况。比如Lily、Swing、Bill这三个人的工资都是2000元，如果按照“**ROWS**”进行范围定位，则计算从第一条到当前行的累积和，而如果如果按照“RANGE”进行范围定位，则仍然计算从第一条到当前行的累积和，不过由于等于2000元的工资有三个人，所以计算的累积和为从第一条到2000元工资的人员结，所以对Lily、Swing、Bill这三个人进行开窗函数聚合计算的时候得到的都是7000（“1000+2000+2000+2000”）。

例3

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN 2 PRECEDING AND 2
FOLLOWING)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	5000
Lily	2000	7000
Swing	2000	9800
Bill	2000	11800
Guo	2800	12800
Tom	3000	13800
YaoMing	3000	14800
Kitty	3000	15300
Smith	3000	15800
Jerry	3300	16300
Jim	3500	17300
Merry	3500	22300
Tim	4000	27500
YuQian	8000	24000
Ketty	8500	20500

这里的开窗函数“SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)”表示按照FSalary进行排序，然后计算从当前行前两行（2 PRECEDING）到当前行后两行（2 FOLLOWING）的工资和，注意对于第一条和第二条而言它们的“前两行”是不存在或者不完整的，因此计算的时候也是要按照前两行是不存在或者不完整进行计算，同样对于最后两行数据而言它们的“后两行”也不存在或者不完整的，同样要进行类似的处理。

例4

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN 1 FOLLOWING AND 3
```

```
FOLLOWING)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	6000
Lily	2000	6800
Swing	2000	7800
Bill	2000	8800
Guo	2800	9000
Tom	3000	9000
YaoMing	3000	9300
Kitty	3000	9800
Smith	3000	10300
Jerry	3300	11000
Jim	3500	15500
Merry	3500	20500
Tim	4000	16500
YuQian	8000	8500
Ketty	8500	<NULL>

这里的开窗函数“`SUM(FSalary) OVER(ORDER BY FSalary ROWS BETWEEN 1 FOLLOWING AND 3 FOLLOWING)`”表示按照`FSalary`进行排序，然后计算从当前行后一行（`1 FOLLOWING`）到后三行（`3 FOLLOWING`）的工资和。注意最后一行没有后续行，其计算结果为空值`NULL`而非`0`。

例5

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FName RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
Bill	2000	2000
Guo	2800	4800
Jerry	3300	8100
Jim	3500	11600
John	1000	12600
Ketty	8500	21100
Kitty	3000	24100
Lily	2000	26100
Merry	3500	29600
Smith	3000	32600
Swing	2000	34600
Tim	4000	38600
Tom	3000	41600

YaoMing	3000	44600
YuQian	8000	52600

这里的开窗函数“SUM(FSalary) OVER(ORDER BY FName RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)”表示按照FName进行排序，然后计算从第一行（UNBOUNDED PRECEDING）到当前行（CURRENT ROW）的工资和。

“RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW”是开窗函数中最常使用的定位框架，为了简化使用，如果使用的是这种定位框架，则可以省略定位框架声明部分，也就是说“SUM(FSalary) OVER(ORDER BY FName RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)”等价于“SUM(FSalary) OVER(ORDER BY FName)”，所以这个SQL语句可以简写为：

```
SELECT FName, FSalary,
SUM(FSalary) OVER(ORDER BY FName)
FROM T_Person;
```

例6

```
SELECT FName, FSalary,
COUNT(*) OVER(ORDER BY FSalary ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	3
John	1000	1
Lily	2000	2
Swing	2000	3
Bill	2000	4
Guo	2800	5
Tom	3000	6
YaoMing	3000	7
Kitty	3000	8
Smith	3000	9
Jerry	3300	10
Jim	3500	11
Merry	3500	12
Tim	4000	13
YuQian	8000	14
Ketty	8500	15

这里的开窗函数“COUNT(*) OVER(ORDER BY FSalary RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)”表示按照FSalary进行排序，然后计算从第一行（UNBOUNDED PRECEDING）到当前行（CURRENT ROW）的人员的个数，这个可以看作是计算人员的工资水平排名。

例7

```
SELECT FName, FSalary,FAge,
MAX(FSalary) OVER(ORDER BY FAge)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE	4
Tom	3000	20	3000
YaoMing	3000	20	3000
Guo	2800	20	3000
Tim	4000	21	4000
Lily	2000	21	4000
Jim	3500	22	4000
John	1000	22	4000
Swing	2000	22	4000
Merry	3500	23	4000
YuQian	8000	24	8000
Jerry	3300	24	8000
Ketty	8500	25	8500
Kitty	3000	25	8500
Bill	2000	25	8500
Smith	3000	30	8500

这里的开窗函数“MAX(FSalary) OVER(ORDER BY FAge)”是“MAX(FSalary) OVER(ORDER BY FAge RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)”的简化写法，它表示按照FSalary进行排序，然后计算从第一行（UNBOUNDED PRECEDING）到当前行（CURRENT ROW）的人员的最大工资值。

例8

```
SELECT FName, FSalary,FAge,
MAX(FSalary) OVER(PARTITION BY FAge ORDER BY FSalary)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE	4
Guo	2800	20	2800
Tom	3000	20	3000
YaoMing	3000	20	3000
Lily	2000	21	2000
Tim	4000	21	4000
John	1000	22	1000
Swing	2000	22	2000
Jim	3500	22	3500
Merry	3500	23	3500
Jerry	3300	24	3300
YuQian	8000	24	8000
Bill	2000	25	2000
Kitty	3000	25	3000
Ketty	8500	25	8500
Smith	3000	30	3000

从这个例子可以看出PARTITION BY子句和**ORDER BY可以共同使用，从而可以实现更加复杂的功能。**

10.7.3 高级开窗函数

除了可以在开窗函数中使用COUNT()、SUM()、MIN()、MAX()、AVG()等这些聚合函数，还可以在开窗函数中使用一些高级的函数，有些函数同时被DB2和Oracle同时支持，比如RANK()、DENSE_RANK()、ROW_NUMBER()，而有些函数只被Oracle支持，比如RATIO_TO_REPORT()、NTILE()、LEAD()、LAG()、FIRST_VALUE()、LAST_VALUE()。下面对这几个函数进行详细介绍。

RANK()和DENSE_RANK()函数都可以用于计算一行的排名，不过对于并列排名的处理方式不同；ROW_NUMBER()函数计算一行在结果集中的行号，同样可以将其当成排名函数。这三个函数的功能存在一定的差异，举例如下：

```
SELECT FName, FSalary,FAge,
RANK() OVER(ORDER BY FAGE),
DENSE_RANK() OVER(ORDER BY FAGE),
ROW_NUMBER() OVER(ORDER BY FAGE)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE			
Tom	3000	20	1	1	1
YaoMing	3000	20	1	1	2
Guo	2800	20	1	1	3
Tim	4000	21	4	2	4
Lily	2000	21	4	2	5
Jim	3500	22	6	3	6
John	1000	22	6	3	7
Swing	2000	22	6	3	8
Merry	3500	23	9	4	9
YuQian	8000	24	10	5	10
Jerry	3300	24	10	5	11
Ketty	8500	25	12	6	12
Kitty	3000	25	12	6	13
Bill	2000	25	12	6	14
Smith	3000	30	15	7	15

从上面的执行结果可以看出，使用DENSE_RANK()的时候如果发生并列排名的情况，名次的位置会被占用，而使用RANK()的时候则名次会顺延，而ROW_NUMBER()则会返回一个唯一的排名。

RATIO_TO_REPORT()函数用于计算当前行的某个列的值在当前窗口中所占的百分比，它等价于colname/SUM(colname)。下面的SQL语句演示了RATIO_TO_REPORT()函数的使用：

```
SELECT FName, FSalary,FAge,
RATIO_TO_REPORT(FSalary) OVER(),
FSalary/SUM(FSalary) OVER(),
RATIO_TO_REPORT(FSalary) OVER(PARTITION BY FAge)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE			

YaoMing	3000	20	0.05703422053231939163498098859315589353	0.05703422053231939163498098859315589353	0.340909090909090909090909090909090909
Tom	3000	20	0.05703422053231939163498098859315589353	0.05703422053231939163498098859315589353	0.340909090909090909090909090909090909
Guo	2800	20	0.05323193916349809885931558935361673004	0.05323193916349809885931558935361673004	0.318181818181818181818181818181818182
Tim	4000	21	0.07604562737642585551330798479087452478	0.07604562737642585551330798479087452478	0.666666666666666666666666666666666667
Lily	2000	21	0.03802281368821292775665399239543726235	0.03802281368821292775665399239543726235	0.333333333333333333333333333333333333
John	1000	22	0.01901140684410646387832699619771863117	0.01901140684410646387832699619771863117	0.153846153846153846153846153846153846
Jim	3500	22	0.06653992395437262357414448669201520912	0.06653992395437262357414448669201520912	0.538461538461538461538461538461538461
Swing	2000	22	0.03802281368821292775665399239543726235	0.03802281368821292775665399239543726235	0.307692307692307692307692307692307692
Merry	3500	23	0.06653992395437262357414448669201520912	0.06653992395437262357414448669201520912	1
Jerry	3300	24	0.06273764258555133079847908745247148288	0.06273764258555133079847908745247148288	0.292035398230088495575221238938053097345
YuQian	8000	24	0.15209125475285171102661596958174904942	0.15209125475285171102661596958174904942	0.707964601769911504424778761061946902654
Ketty	8500	25	0.16159695817490494296577946768060836501	0.16159695817490494296577946768060836501	0.629629629629629629629629629629629629
Bill	2000	25	0.03802281368821292775665399239543726235	0.03802281368821292775665399239543726235	0.148148148148148148148148148148148148
Kitty	3000	25	0.05703422053231939163498098859315589353	0.05703422053231939163498098859315589353	0.222222222222222222222222222222222222
Smith	3000	30	0.05703422053231939163498098859315589353	0.05703422053231939163498098859315589353	1

			1558935361	1558935361	

第4列计算当前员工的工资占所有员工工资额的百分比，第5列使用**SUM()函数来模拟实现**RATIO_TO_REPORT()函数，而第6列则计算当前员工的工资占本年龄段内员工工资额的百分比。

NTILE()函数用来将窗口中的行按照某个列的值进行区域的平均分割，然后返回当前行所在的区域编号。NTILE()函数接受一个整数类型的值，这个值表示把结果集分割成的份数，注意必须在NTILE()函数后的OVER()子句中显式的使用ORDER BY关键字来指定排序规则。

下面的SQL语句演示了NTILE()函数的使用：

```
SELECT FName, FSalary,FAge,
NTILE(3) OVER(ORDER BY FSalary)
FROM T_Person;
```

这个SQL语句将所有的人员按照工资的顺序将人员平均分割成3个区域，并且计算每一行所在的区域编号。执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE	
John	1000	22	1
Lily	2000	21	1
Bill	2000	25	1
Swing	2000	22	1
Guo	2800	20	1
YaoMing	3000	20	2
Tom	3000	20	2
Smith	3000	30	2
Kitty	3000	25	2
Jerry	3300	24	2
Merry	3500	23	3
Jim	3500	22	3
Tim	4000	21	3
YuQian	8000	24	3
Ketty	8500	25	3

在窗口函数出现之前，如果要涉及到“取前一行的值”、“取后三行的值”这样的需求的话，必须要借助于存储过程，为了简化开发，Oracle中提供了LEAD()函数用于访问结果集中的其它行，它的行为类似于一个游标，不过使用起来比游标更加简单，因为使用它无需定义游标和关闭游标，而且无需编写存储过程，一个函数调用就能得到像使用游标一样强大的功能。使用LEAD()函数，只要给定一个查询结果集和游标的位置，就可以访问在当前行之前某个游标位置的指定列的值。LEAD()函数的参数格式如下：

```
LEAD(value_expr[,offset,default])
```

其中参数value_expr为指定行的待计算表达式，参数offset表示与当前行的偏移量，而参数default则为游标位置不存在行时的默认值。其中offset和default两个参数可选，如果不指定这两个参数，则表示offset取1、default取NULL。

```
SELECT FName, FSalary,FAge,
LEAD(FName) OVER(ORDER BY FSalary),
LEAD(FName,1,'无后继者') OVER(ORDER BY FSalary)
FROM T_Person;
```

这个SQL语句中第一个窗口函数用于计算按照工资排序后每一个人员的后继人员的姓名；第

二个窗口函数同样用于计算按照工资排序后每一个人员的后继人员的姓名，不过如果没有后继人员则显示为“无后继者”。执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE		
John	1000	22	Lily	Lily
Lily	2000	21	Bill	Bill
Bill	2000	25	Swing	Swing
Swing	2000	22	Guo	Guo
Guo	2800	20	YaoMing	YaoMing
YaoMing	3000	20	Tom	Tom
Tom	3000	20	Smith	Smith
Smith	3000	30	Kitty	Kitty
Kitty	3000	25	Jerry	Jerry
Jerry	3300	24	Merry	Merry
Merry	3500	23	Jim	Jim
Jim	3500	22	Tim	Tim
Tim	4000	21	YuQian	YuQian
YuQian	8000	24	Ketty	Ketty
Ketty	8500	25	<NULL>	无后继者

与LEAD()函数的方向正好相反，LAG()函数用于访问在当前行之后某个游标位置的指定列的值，其参数调用和LEAD()函数一样。下面的SQL语句用于计算按照工资进行排序后每个人员的前N（N等于年龄与5的整除）行的人员姓名，如果不存在这样的人员，则显示其姓名为“查无此人”：

```
SELECT FName, FSalary,FAge,
LEAD(FName,FAge/5,'查无此人') OVER(ORDER BY FSalary)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE	
John	1000	22	Guo
Lily	2000	21	YaoMing
Bill	2000	25	Smith
Swing	2000	22	Smith
Guo	2800	20	Kitty
YaoMing	3000	20	Jerry
Tom	3000	20	Merry
Smith	3000	30	YuQian
Kitty	3000	25	YuQian
Jerry	3300	24	YuQian
Merry	3500	23	Ketty
Jim	3500	22	查无此人
Tim	4000	21	查无此人
YuQian	8000	24	查无此人
Ketty	8500	25	查无此人

如果要计算窗口中第一行或者最后一行的值可以使用FIRST_VALUE()函数和

LAST_VALUE()函数，可以将它们看做LEAD()函数和LAG()函数的特例。这两个函数的参数格式为：

```
FIRST_VALUE(value_expr[IGNORE NULLS])
LAST_VALUE(value_expr[IGNORE NULLS])
```

其中参数value_expr为待计算列表达式，而选项IGNORE NULLS则是可选的，它表示如果第一行（对于FIRST_VALUE）或者最后一行（对于LAST_VALUE）的value_expr是NULL，那么就忽略。

FIRST_VALUE()函数和LAST_VALUE()函数使用的例子如下：

```
SELECT FName, FSalary,FAge,
FIRST_VALUE(FName IGNORE NULLS) OVER(PARTITION BY FAge ORDER BY FSalary),
LAST_VALUE(FName) OVER(ORDER BY FSalary),
LAST_VALUE(FName) OVER(ORDER BY FSalary RANGE BETWEEN UNBOUNDED PRECEDING
AND UNBOUNDED FOLLOWING)
FROM T_Person;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	FSALARY	FAGE			
John	1000	22	John	John	Ketty
Bill	2000	25	Bill	Swing	Ketty
Lily	2000	21	Lily	Swing	Ketty
Swing	2000	22	John	Swing	Ketty
Guo	2800	20	Guo	Guo	Ketty
Kitty	3000	25	Bill	YaoMing	Ketty
Smith	3000	30	Smith	YaoMing	Ketty
Tom	3000	20	Guo	YaoMing	Ketty
YaoMing	3000	20	Guo	YaoMing	Ketty
Jerry	3300	24	Jerry	Jerry	Ketty
Jim	3500	22	John	Merry	Ketty
Merry	3500	23	Merry	Merry	Ketty
Tim	4000	21	Lily	Tim	Ketty
YuQian	8000	24	Jerry	YuQian	Ketty
Ketty	8500	25	Bill	Ketty	Ketty

需要注意的是，LAST_VALUE()是一个窗口函数，当在OVER子句中没有指定范围的时候默认的范围是由第一行到当前行（RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW），所以“LAST_VALUE(FName) OVER(ORDER BY FSalary)”得到的是第一行到当前行这个范围内的最后一行，也就是当前行，而非整个表中最后一行的。如果要得到整个表中最后一行必须明确指定窗口范围“LAST_VALUE(FName) OVER(ORDER BY FSalary RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)”。

到这里，开窗函数的知识就介绍完毕了，请删除T_Person表：

```
DROP TABLE T_Person;
```

10.8 WITH子句与子查询

子查询可以简化SQL语句的编写，不过如果使用不当的话子查询会降低系统性能，为了避免子查询带来的性能问题，除了需要优化SQL语句之外还需要尽量降低使用子查询的次数。比如下面的子查询用来取得系统中所有年龄或者工资与Tom相同的人员：

```
SELECT * FROM T_Person
```

```
WHERE FAge=(SELECT FAge FROM T_Person WHERE FName='TOM')
OR FSalary=(SELECT FSalary FROM T_Person WHERE FName='TOM')
```

这个SQL语句可以完成要求的功能，不过可以看到类似的子查询被用到了两次，这会带来下面的问题：

- 同一个子查询被使用多次会造成这个子查询被执行多次，由于子查询是比较消耗系统资源的操作，所以这会降低系统的性能。
- 同一个子查询在多处被使用，这违反了编程中的DRY（Don't Repeat Yourself）原则，如果要修改子查询就必须对这些子查询同时修改，很容易造成修改不同步。

造成这种问题的原因就是子查询只能在定义的时候使用，这样如果多次使用就必须多次定义，为了解决这种问题，SQL提供了WITH子句用于为子查询定义一个别名，这样就可以通过这个别名来引用这个子查询了，也就是实现“一次定义多次使用”。

使用WITH子句来改造上面的SQL语句：

```
WITH person_tom AS
(
    SELECT * FROM T_Person
    WHERE FName='TOM'
)
SELECT * FROM T_Person
WHERE FAge=person_tom.FAge
OR FSalary=person_tom.FSalary
```

可以看到WITH子句的格式为：

```
WITH 别名 AS
(子查询)
```

定义好别名以后就可以在SQL语句中通过这个别名来引用子查询了，注意这个语句是一个SQL语句，而非存储过程，所以可以远程调用。

还可以在WITH语句中为子查询中的列定义别名，定义的方式就是在子查询别名后列出参数名列表。如下：

```
WITH person_tom(F1,F2,F3) AS
(
    SELECT FAge,FName,FSalary FROM T_Person
    WHERE FName='TOM'
)
SELECT * FROM T_Person
WHERE FAge=person_tom.F1
OR FSalary=person_tom.F3
```

第十一章 案例讲解

前面章节我们对 SQL 相关的知识做了介绍，为了帮助读者更好的理解这些知识点，并且更加清晰的看到这些知识点是如何在实践中解决问题的，我们安排了这章讲解一些实际开发中经常碰到的重点、难点问题，并且对这些问题进行了详细的分析和解答。

在学习本章之前请在相应的数据库中运行下面的 SQL 语句以创建演示用的数据表。这里创建一个简化的进销系统，系统中只有销售单和采购单，不存在红冲单据以及库存、退货等单据。由于销售单和采购单存在主从结构，所以将这两张表中的主从数据分别保存在不同

的表中。下面是这个系统中表之间的关系图：

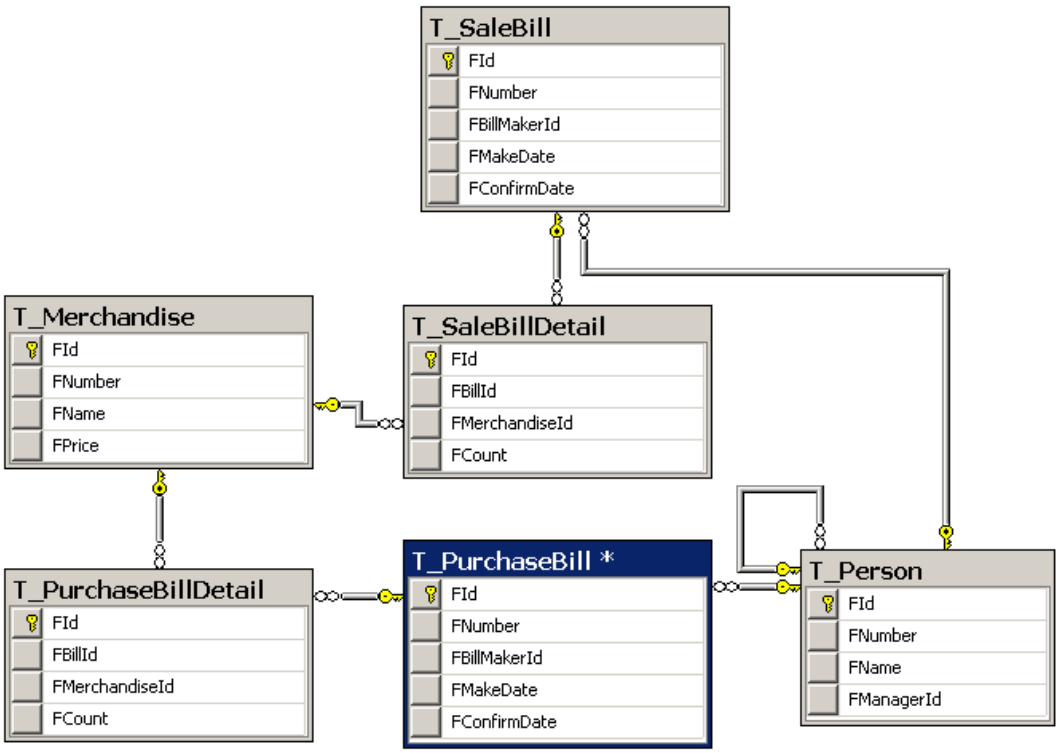


表 T_Person 为人员表，FId 字段为主键，FNumber 字段为人员工号，FName 字段为人员姓名，FManagerId 字段为上级主管主键（指向 T_Person 表的 FId 字段的外键）。

表 T_Merchandise 为商品表，FId 字段为主键，FNumber 字段为产品编号，FName 字段为商品名，FPrice 为商品价格；表 T_SaleBill 为销售单主表，FNumber 字段为销售单编号，FBillMakerId 字段为开单人主键（指向 T_Person 表的 FId 字段的外键），FMakeDate 字段为制单日期，FConfirmDate 字段为确认日期；表 T_SaleBillDetail 为销售单明细记录，FId 字段为主键，FBillId 字段为主表主键（指向 T_SaleBill 表的 FId 字段的外键），FMerchandiseId 字段为商品主键（指向 T_Merchandise 表的 FId 字段的外键），FCount 字段为销售数量。

表 T_PurchaseBill 为采购单主表，FNumber 字段为采购单编号，FBillMakerId 字段为开单人主键（指向 T_Person 表的 FId 字段的外键），FMakeDate 字段为制单日期，FConfirmDate 字段为确认日期；表 T_PurchaseBillDetail 为采购单明细记录，FId 字段为主键，FBillId 字段为主表主键（指向 T_PurchaseBill 表的 FId 字段的外键），FMerchandiseId 字段为商品主键（指向 T_Merchandise 表的 FId 字段的外键），FCount 字段为采购数量。

根据表的设计，我们编写了下面的建表 SQL 语句，请在相应的 DBMS 中执行对应的 SQL 语句：

MYSQL,MSSQLServer:

CREATE TABLE T_Person(FId VARCHAR(20) NOT NULL,FNumber VARCHAR(20),FName VARCHAR(20),FManagerId VARCHAR(20),PRIMARY KEY (FId),FOREIGN KEY (FManagerId) REFERENCES T_Person(FId));

CREATE TABLE T_Merchandise(FId VARCHAR(20) NOT NULL ,FNumber VARCHAR(20),FName VARCHAR(20),FPrice INT,PRIMARY KEY (FId));

CREATE TABLE T_SaleBill(FId VARCHAR(20) NOT NULL ,FNumber VARCHAR(20),FBillMakerId VARCHAR(20),FMakeDate DATETIME,FConfirmDate DATETIME,PRIMARY KEY (FId),FOREIGN KEY (FBillMakerId) REFERENCES

T_Person(FId));
CREATE TABLE T_SaleBillDetail(FId VARCHAR(20) NOT NULL ,FBillId VARCHAR(20),FMerchandiseId VARCHAR(20),FCount INT,PRIMARY KEY (FId),FOREIGN KEY (FBillId) REFERENCES T_SaleBill(FId),FOREIGN KEY (FMerchandiseId) REFERENCES T_Merchandise(FId));
CREATE TABLE T_PurchaseBill(FId VARCHAR(20) NOT NULL ,FNumber VARCHAR(20),FBillMakerId VARCHAR(20),FMakeDate DATETIME,FConfirmDate DATETIME,PRIMARY KEY (FId),FOREIGN KEY (FBillMakerId) REFERENCES T_Person(FId));
CREATE TABLE T_PurchaseBillDetail(FId VARCHAR(20) NOT NULL ,FBillId VARCHAR(20),FMerchandiseId VARCHAR(20),FCount INT,PRIMARY KEY (FId),FOREIGN KEY (FBillId) REFERENCES T_PurchaseBill(FId),FOREIGN KEY (FMerchandiseId) REFERENCES T_Merchandise(FId));
Oracle:
CREATE TABLE T_Person(FId VARCHAR2(20) NOT NULL ,FNumber VARCHAR2(20),FName VARCHAR2(20),FManagerId VARCHAR2(20),PRIMARY KEY (FId),FOREIGN KEY (FManagerId) REFERENCES T_Person(FId));
CREATE TABLE T_Merchandise (FId VARCHAR2(20) NOT NULL ,FNumber VARCHAR2(20),FName VARCHAR2(20),FPrice INT,PRIMARY KEY (FId));
CREATE TABLE T_SaleBill (FId VARCHAR2(20) NOT NULL ,FNumber VARCHAR2(20),FBillMakerId VARCHAR2(20),FMakeDate DATE,FConfirmDate DATE,PRIMARY KEY (FId),FOREIGN KEY (FBillMakerId) REFERENCES T_Person(FId));
CREATE TABLE T_SaleBillDetail(FId VARCHAR2(20) NOT NULL ,FBillId VARCHAR2(20),FMerchandiseId VARCHAR2(20),FCount INT,PRIMARY KEY (FId),FOREIGN KEY (FBillId) REFERENCES T_SaleBill(FId),FOREIGN KEY (FMerchandiseId) REFERENCES T_Merchandise(FId));
CREATE TABLE T_PurchaseBill (FId VARCHAR2(20) NOT NULL ,FNumber VARCHAR2(20),FBillMakerId VARCHAR2(20),FMakeDate DATE,FConfirmDate DATE,PRIMARY KEY (FId),FOREIGN KEY (FBillMakerId) REFERENCES T_Person(FId));
CREATE TABLE T_PurchaseBillDetail(FId VARCHAR2(20) NOT NULL ,FBillId VARCHAR2(20),FMerchandiseId VARCHAR2(20),FCount INT,PRIMARY KEY (FId),FOREIGN KEY (FBillId) REFERENCES T_PurchaseBill(FId),FOREIGN KEY (FMerchandiseId) REFERENCES T_Merchandise(FId));
DB2:
CREATE TABLE T_Person (FId VARCHAR(20) NOT NULL ,FNumber VARCHAR(20),FName VARCHAR(20),FManagerId VARCHAR(20),PRIMARY KEY (FId),FOREIGN KEY (FManagerId) REFERENCES T_Person(FId));
CREATE TABLE T_Merchandise (FId VARCHAR(20) NOT NULL ,FNumber VARCHAR(20),FName VARCHAR(20),FPrice INT,PRIMARY KEY (FId));
CREATE TABLE T_SaleBill (FId VARCHAR(20) NOT NULL ,FNumber VARCHAR(20),FBillMakerId VARCHAR(20),FMakeDate DATE,FConfirmDate DATE,PRIMARY KEY (FId),FOREIGN KEY (FBillMakerId) REFERENCES T_Person(FId));
CREATE TABLE T_SaleBillDetail (FId VARCHAR(20) NOT NULL ,FBillId VARCHAR(20),FMerchandiseId VARCHAR(20),FCount INT,PRIMARY KEY (FId),FOREIGN

KEY (FBillId) REFERENCES T_SaleBill(FId),FOREIGN KEY (FMerchandiseId) REFERENCES T_Merchandise(FId));

CREATE TABLE T_PurchaseBill (FId VARCHAR(20) NOT NULL ,FNumber VARCHAR(20),FBillMakerId VARCHAR(20),FMakeDate DATE,FConfirmDate DATE,PRIMARY KEY (FId),FOREIGN KEY (FBillMakerId) REFERENCES T_Person(FId));

CREATE TABLE T_PurchaseBillDetail (FId VARCHAR(20) NOT NULL ,FBillId VARCHAR(20),FMerchandiseId VARCHAR(20),FCount INT,PRIMARY KEY (FId),FOREIGN KEY (FBillId) REFERENCES T_PurchaseBill(FId),FOREIGN KEY (FMerchandiseId) REFERENCES T_Merchandise(FId));

除了创建数据表，还需要预置一些演示数据。首先向 T_Person、T_Merchandise 两张表中插入演示数据：

```
insert into T_Person(FId,FNumber,FName,FManagerId)
values('00001','1','Robert',NULL);
insert into T_Person(FId,FNumber,FName,FManagerId)
values('00002','2','John','00001');
insert into T_Person(FId,FNumber,FName,FManagerId)
values('00003','3','Tom','00001');
insert into T_Person(FId,FNumber,FName,FManagerId)
values('00004','4','Jim','00003');
insert into T_Person(FId,FNumber,FName,FManagerId)
values('00005','5','Lily','00002');
insert into T_Person(FId,FNumber,FName,FManagerId)
values('00006','6','Merry','00003');

insert into T_Merchandise(FId,FNumber,FName,FPrice)
values('00001','1','Bacon',30);
insert into T_Merchandise(FId,FNumber,FName,FPrice)
values('00002','2','Cake',2);
insert into T_Merchandise(FId,FNumber,FName,FPrice)
values('00003','3','Apple',6);
```

还要向 T_SaleBill 和 T_PurchaseBill 表中插入演示数据：

```
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00001','1','00006','2007-03-15','2007-05-15');
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00002','2',null,'2006-01-25','2006-02-03');
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00003','3','00001','2006-02-12','2007-01-11');
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00004','4','00003','2008-05-25','2008-06-15');
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00005','5','00005','2008-03-17','2007-04-15');
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00006','6','00002','2002-02-03','2007-11-11');
```

```
insert into T_PurchaseBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00001','1','00006','2007-02-15','2007-02-15');
insert into T_PurchaseBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00002','2','00004','2003-02-25','2006-03-03');
insert into T_PurchaseBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00003','3','00001','2007-02-12','2007-07-12');
insert into T_PurchaseBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00004','4','00002','2007-05-25','2007-06-15');
insert into T_PurchaseBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00005','5','00002','2007-03-17','2007-04-15');
insert into T_PurchaseBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00006','6',null,'2006-02-03','2006-11-20');
```

由于 Oracle 中日期常量的表示方法与其他 DBMS 不同，所以在 Oracle 总必须执行下面的 SQL 语句才能向 T_SaleBill 中插入演示数据：

```
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00001','1','00006',TO_DATE('2007-03-15', 'YYYY-MM-DD
HH24:MI:SS'),TO_DATE('2007-05-15', 'YYYY-MM-DD HH24:MI:SS'));
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00002','2',null,TO_DATE('2006-01-25', 'YYYY-MM-DD
HH24:MI:SS'),TO_DATE('2006-02-03', 'YYYY-MM-DD HH24:MI:SS'));
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00003','3','00001',TO_DATE('2006-02-12', 'YYYY-MM-DD
HH24:MI:SS'),TO_DATE('2007-01-11', 'YYYY-MM-DD HH24:MI:SS'));
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00004','4','00003',TO_DATE('2008-05-25', 'YYYY-MM-DD
HH24:MI:SS'),TO_DATE('2008-06-15', 'YYYY-MM-DD HH24:MI:SS'));
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00005','5','00005',TO_DATE('2008-03-17', 'YYYY-MM-DD
HH24:MI:SS'),TO_DATE('2007-04-15', 'YYYY-MM-DD HH24:MI:SS'));
insert into T_SaleBill(FId,FNumber,FBillMakerId,FMakeDate,FConfirmDate)
values('00006','6','00002',TO_DATE('2002-02-03', 'YYYY-MM-DD
HH24:MI:SS'),TO_DATE('2007-11-11', 'YYYY-MM-DD HH24:MI:SS'));

INSERT INTO T_PurchaseBill (FId, FNumber, FBillMakerId, FMakeDate, FConfirmDate)
VALUES ('00001', '1', '00006', TO_DATE('2007-02-15', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2007-02-15', 'YYYY-MM-DD HH24:MI:SS'));
INSERT INTO T_PurchaseBill (FId, FNumber, FBillMakerId, FMakeDate, FConfirmDate)
VALUES ('00002', '2', '00004', TO_DATE('2003-02-25', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2006-03-03', 'YYYY-MM-DD HH24:MI:SS')) ;
INSERT INTO T_PurchaseBill (FId, FNumber, FBillMakerId, FMakeDate, FConfirmDate)
VALUES ('00003', '3', '00001', TO_DATE('2007-02-12', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2007-07-12', 'YYYY-MM-DD HH24:MI:SS')) ;
INSERT INTO T_PurchaseBill (FId, FNumber, FBillMakerId, FMakeDate, FConfirmDate)
VALUES ('00004', '4', '00002', TO_DATE('2007-05-25', 'YYYY-MM-DD HH24:MI:SS'),
```

```
TO_DATE('2007-06-15', 'YYYY-MM-DD HH24:MI:SS')) ;
INSERT INTO T_PurchaseBill (FId, FNumber, FBillMakerId, FMakeDate, FConfirmDate)
VALUES ('00005', '5', '00002', TO_DATE('2007-03-17', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2007-04-15', 'YYYY-MM-DD HH24:MI:SS')) ;
INSERT INTO T_PurchaseBill (FId, FNumber, FBillMakerId, FMakeDate, FConfirmDate)
VALUES ('00006', '6', null, TO_DATE('2006-02-03', 'YYYY-MM-DD HH24:MI:SS'),
TO_DATE('2006-11-20', 'YYYY-MM-DD HH24:MI:SS')) ;
```

向 T_SaleBillDetail 表和 T_PurchaseBillDetail 表中插入演示数据：

```
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00001','00001','00003',20);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00002','00001','00001',30);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00003','00001','00002',22);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00004','00002','00003',12);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00005','00002','00002',11);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00006','00003','00001',60);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00007','00003','00002',2);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00008','00003','00003',5);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00009','00004','00001',16);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00010','00004','00002',8);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00011','00004','00003',9);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00012','00005','00001',6);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00013','00005','00003',26);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00014','00006','00001',66);
insert into T_SaleBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00015','00006','00002',518);

insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00001','00001','00002',12);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00002','00001','00001',20);
```

```
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00003','00002','00001',32);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00004','00002','00003',18);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00005','00002','00002',88);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00006','00003','00003',19);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00007','00003','00002',6);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00008','00003','00001',2);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00009','00004','00001',20);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00010','00004','00003',18);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00011','00005','00002',19);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00012','00005','00001',26);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00013','00006','00003',3);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00014','00006','00001',22);
insert into T_PurchaseBillDetail(FId,FBillId,FMerchandiseId,FCount)
values('00015','00006','00002',168);
```

11.1 报表制作

数据库中的数据是以关系表的形式保存的，非技术人员很难看懂这些表中数据的意思，必须将其转换为业务人员可以看得懂的形式，这就是报表。复杂的分析报表可能需要数据挖掘等技术才能实现，不过普通的报表一般使用 SELECT 语句就可以完成，本节将讲解几个典型的报表制作的案例。

11.1.1 显示制单人详细信息

要求显示每张销售单的编号、制单人、制单日期等信息，可以使用简单的 SELECT 语句来完成这个任务：

```
SELECT FNumber, FBillMakerId, FMakeDate
FROM T_SaleBill
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FBillMakerId	FMakeDate
1	00006	2007-03-15 00:00:00.0
2	<NULL>	2006-01-25 00:00:00.0
3	00001	2006-02-12 00:00:00.0
4	00003	2008-05-25 00:00:00.0
5	00005	2008-03-17 00:00:00.0
6	00002	2002-02-03 00:00:00.0

这里的 FBillMakerId 显示的是制单人在 T_Person 表中的主键，业务人员很难将这个编号与人名对应起来，因此必须将其转换为制单人的姓名。FBillMakerId 字段保存的是 T_Person 表的主键，而 T_Person 表的 FName 字段则为人员的名称，因此将这两个表做连接查询即可，SQL 语句如下：

```
SELECT salebill.FNumber,person.FName,salebill.FMakeDate
FROM T_SaleBill salebill
INNER JOIN T_Person person
ON salebill.FBillMakerId=person.FId;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FMakeDate
1	Merry	2007-03-15 00:00:00.0
3	Robert	2006-02-12 00:00:00.0
4	Tom	2008-05-25 00:00:00.0
5	Lily	2008-03-17 00:00:00.0
6	John	2002-02-03 00:00:00.0

这个查询结果已经能够显示开票人的姓名，不过仔细观察会发现编号为 2 的记录并没有显示在执行结果中，这是因为这条记录的 FBillMakerId 字段为空值，所以不能与 T_Person 表中的任何记录进行匹配，而内连接不会显示没有匹配的行。一般情况下即使没有开单人也要将这张单据显示出来，这时就要使用外部连接了，如下：

```
SELECT salebill.FNumber,person.FName,salebill.FMakeDate
FROM T_SaleBill salebill
LEFT OUTER JOIN T_Person person
ON salebill.FBillMakerId=person.FId;
```

因为 T_SaleBill 表中的记录必须全部显示到结果集中，而 T_SaleBill 为左表，所以使用左外部连接。执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FMakeDate
1	Merry	2007-03-15 00:00:00.0
2	<NULL>	2006-01-25 00:00:00.0
3	Robert	2006-02-12 00:00:00.0
4	Tom	2008-05-25 00:00:00.0
5	Lily	2008-03-17 00:00:00.0
6	John	2002-02-03 00:00:00.0

这样没有开票人的单据也显示出来了，不过这里其对应的开票人处显示的是 NULL，这让业务人员感到难以理解，我们使用 **COALESCE()函数来解决这个问题。前面章节讲到 COALESCE()函数支持多个参数，该函数返回参数中的第一个非空值，这样 COALESCE(f1,f2) 就可以实现“如果 f1 为空则将 f2 做为返回值”这样的空值处理逻辑了。将 SQL 语句做如下改造：**

```
SELECT salebill.FNumber,
COALESCE(person.FName,'没有开单人'),
salebill.FMakeDate
FROM T_SaleBill salebill
LEFT OUTER JOIN T_Person person
ON salebill.FBillMakerId=person.FId;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber		FMakeDate
1	Merry	2007-03-15 00:00:00.0
2	没有开单人	2006-01-25 00:00:00.0
3	Robert	2006-02-12 00:00:00.0
4	Tom	2008-05-25 00:00:00.0
5	Lily	2008-03-17 00:00:00.0
6	John	2002-02-03 00:00:00.0

11.1.2 显示销售单的详细信息

要求列出所有销售单的详细信息，每行显示销售单的每一条销售记录，同时每行头部要显示此行所属的销售单的信息，比如单号、开单人、开单日期等。T_SaleBillDetail 表保存的是销售单的每一条销售记录，T_SaleBill 表保存的是销售单的头信息，T_SaleBillDetail 表的 FMerchandiseId 字段保存的是销售的商品主键，而 T_SaleBill 表的 FBillMakerId 字段保存的是开单人的主键，只要对这四张表做连接查询即可。由于 T_SaleBill 表的 FBillMakerId 字段有可能为空，所以在 T_SaleBill 表和 T_Person 表进行连接的时候要使用左外连接，而为了提高查询效率其他连接都使用内连接。SQL 语句如下：

```
SELECT
salebill.FNumber,person.FName,salebill.FMakeDate,merchandise.FName,salebilldetail.FCount
FROM T_SaleBill saleBill
INNER JOIN T_SaleBillDetail salebilldetail
ON salebilldetail.FBillId=saleBill.FId
INNER JOIN T_Merchandise merchandise
ON salebilldetail.FMerchandiseId=merchandise.FId
LEFT OUTER JOIN T_Person person
ON saleBill.FBillMakerId=person.FId
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	FName	FMakeDate	FName	FCount
1	Merry	2007-03-15 00:00:00.0	Apple	20
1	Merry	2007-03-15 00:00:00.0	Bacon	30
1	Merry	2007-03-15 00:00:00.0	Cake	22
2	<NULL>	2006-01-25 00:00:00.0	Apple	12
2	<NULL>	2006-01-25 00:00:00.0	Cake	11
3	Robert	2006-02-12 00:00:00.0	Bacon	60
3	Robert	2006-02-12 00:00:00.0	Cake	2
3	Robert	2006-02-12 00:00:00.0	Apple	5
4	Tom	2008-05-25 00:00:00.0	Bacon	16
4	Tom	2008-05-25 00:00:00.0	Cake	8
4	Tom	2008-05-25 00:00:00.0	Apple	9
5	Lily	2008-03-17 00:00:00.0	Bacon	6
5	Lily	2008-03-17 00:00:00.0	Apple	26
6	John	2002-02-03 00:00:00.0	Bacon	66
6	John	2002-02-03 00:00:00.0	Cake	518

11.1.3 计算收益

要求计算每种商品的总收益，受收益的定义为所有的销售单中该商品的销售总额减去所

有的采购单中该商品的购买总额。

T_SaleBillDetail 表中保存的所有的销售单详细记录，因此下面的 SQL 语句可以检索所有产品的销售记录，包括产品名和销售额：

```
SELECT merchandise.FName AS MerchandiseName,
merchandise.FPrice*salebilldetail.FCount AS Amount
FROM T_SaleBillDetail salebilldetail
INNER JOIN T_Merchandise merchandise
ON salebilldetail.FMerchandiseId=merchandise.FId
```

这里将 T_SaleBillDetail 表和 T_Merchandise 进行连接查询就可以取得每种商品的名称和单价，而单价与销售量的乘积即为销售额。执行完毕我们就能在输出结果中看到下面的执行结果：

MerchandiseName	Amount
Apple	120
Bacon	900
Cake	44
Apple	72
Cake	22
Bacon	1800
Cake	4
Apple	30
Bacon	480
Cake	16
Apple	54
Bacon	180
Apple	156
Bacon	1980
Cake	1036

同理下面的 SQL 语句则可以检索所有产品的购买记录，包括产品名和采购额：

```
SELECT merchandise.FName AS MerchandiseName,
merchandise.FPrice*purchasebilldetail.FCount AS Amount
FROM T_PurchaseBillDetail purchasebilldetail
INNER JOIN T_Merchandise merchandise
ON purchasebilldetail.FMerchandiseId=merchandise.FId
```

执行完毕我们就能在输出结果中看到下面的执行结果：

MerchandiseName	Amount
Cake	24
Bacon	600
Bacon	960
Apple	108
Cake	176
Apple	114
Cake	12
Bacon	60

Bacon	600
Apple	108
Cake	38
Bacon	780
Apple	18
Bacon	660
Cake	336

将上述两个 SQL 语句进行 UNION 运算就可以将两个检索的结果集合并了，不过这样就无法区分销售单和采购单了。为了区分销售单和采购单，同时方便后续运算，我们将检索产品的购买记录的金额全部取负值，这样就可以表示采购行为的金额为负值：

```
SELECT merchandise.FName AS MerchandiseName,
merchandise.FPrice*purchasebilldetail.FCount*(-1) AS Amount
FROM T_PurchaseBillDetail purchasebilldetail
INNER JOIN T_Merchandise merchandise
ON purchasebilldetail.FMerchandiseId=merchandise.FId
```

这样就可以将销售记录和采购记录同时显示了：

```
SELECT merchandise.FName AS MerchandiseName,
merchandise.FPrice*salebilldetail.FCount AS Amount
FROM T_SaleBillDetail salebilldetail
INNER JOIN T_Merchandise merchandise
ON salebilldetail.FMerchandiseId=merchandise.FId
UNION ALL
SELECT merchandise.FName AS MerchandiseName,
merchandise.FPrice*purchasebilldetail.FCount*(-1) AS Amount
FROM T_PurchaseBillDetail purchasebilldetail
INNER JOIN T_Merchandise merchandise
ON purchasebilldetail.FMerchandiseId=merchandise.FId
```

因为有可能存在重复的记录（即同一种商品有同样的交易额），所以要使用 **UNION ALL，以防止重复的记录被合并。**执行完毕我们就能在输出结果中看到下面的执行结果：

MerchandiseName	Amount
Apple	120
Bacon	900
Cake	44
Apple	72
Cake	22
Bacon	1800
Cake	4
Apple	30
Bacon	480
Cake	16
Apple	54
Bacon	180
Apple	156

Bacon	1980
Cake	1036
Cake	-24
Bacon	-600
Bacon	-960
Apple	-108
Cake	-176
Apple	-114
Cake	-12
Bacon	-60
Bacon	-600
Apple	-108
Cake	-38
Bacon	-780
Apple	-18
Bacon	-660
Cake	-336

这个结果集中列出了每一条详细交易记录，包括商品名和交易金额，销售行为的交易额为正值，而购买行为的交易额为负值。有个这个执行结果，只要将这个 SQL 语句做为子查询，然后按照商品名进行分组，然后计算交易金额的总和。SQL 语句如下：

```
SELECT details.MerchandiseName,SUM(details.Amount)
FROM
(
    SELECT merchandise.FName AS MerchandiseName,
    merchandise.FPrice*salebilldetail.FCount AS Amount
    FROM T_SaleBillDetail salebilldetail
    INNER JOIN T_Merchandise merchandise
    ON salebilldetail.FMerchandiseId=merchandise.FId
    UNION ALL
    SELECT merchandise.FName AS MerchandiseName,
    merchandise.FPrice*purchasebilldetail.FCount*(-1) AS Amount
    FROM T_PurchaseBillDetail purchasebilldetail
    INNER JOIN T_Merchandise merchandise
    ON purchasebilldetail.FMerchandiseId=merchandise.FId
) details
GROUP BY details.MerchandiseName
```

执行完毕我们就能在输出结果中看到下面的执行结果：

MerchandiseName	
Apple	84
Bacon	1680
Cake	536

11.1.4 产品销售额统计

要求统计每种产品的销售额并且在报表最后列出销售额总计。

检索每种产品的销售额可以到 T_SaleBillDetail 中按照产品进行分组，然后使用聚合函

数 SUM()来计算每种产品的销售额。因为 T_SaleBillDetail 只保存了销售量，价格和产品名称保存在 T_Merchandise 表中，因此还需要与 T_Merchandise 表进行连接运算。SQL 语句如下：

```
SELECT merchandise.FName AS MerchandiseName,
SUM(merchandise.FPrice*salebilldetail.FCount) AS Amount
FROM T_SaleBillDetail salebilldetail
INNER JOIN T_Merchandise merchandise
ON salebilldetail.FMerchandiseId=merchandise.FId
GROUP BY merchandise.FName
```

统计销售额总计同样需要与 T_Merchandise 表进行连接运算以取得价格，然后使用聚合函数 SUM()来计算总销售额。SQL 语句如下：

```
SELECT SUM(merchandise.FPrice*salebilldetail.FCount) AS Amount
FROM T_SaleBillDetail salebilldetail
INNER JOIN T_Merchandise merchandise
ON salebilldetail.FMerchandiseId=merchandise.FId
```

由于这两个查询结构是异构的，如果要将第二个查询的结果集添加到第一个查询的结果集后，那么就需要首先为第二个查询增加一列以保证和第一个查询的列数相同，然后将这两个查询语句使用 UNION ALL 联合起来。SQL 语句如下：

```
SELECT merchandise.FName AS MerchandiseName,
SUM(merchandise.FPrice*salebilldetail.FCount) AS Amount
FROM T_SaleBillDetail salebilldetail
INNER JOIN T_Merchandise merchandise
ON salebilldetail.FMerchandiseId=merchandise.FId
GROUP BY merchandise.FName
UNION ALL
SELECT '总计',
SUM(merchandise.FPrice*salebilldetail.FCount) AS Amount
FROM T_SaleBillDetail salebilldetail
INNER JOIN T_Merchandise merchandise
ON salebilldetail.FMerchandiseId=merchandise.FId
```

执行完毕我们就能在输出结果中看到下面的执行结果：

MerchandiseName	Amount
Apple	432
Bacon	5340
Cake	1122
总计	6894

11.1.5 统计销售记录的份额

要求统计所有的销售明细，并且计算每一笔销售记录中销售量占同产品总销售量的百分比。

这里要求列出每一笔销售记录中销售量，而且还要计算销售量占同产品总销售量的百分比，很显然这要使用窗口函数来完成。SQL 语句如下：

```
SELECT merchandise.FName,FCount,
FCount*1.0/SUM(salebilldetail.FCount) OVER(PARTITION BY salebilldetail.FMerchandiseId)
FROM T_SaleBillDetail salebilldetail
```

```
INNER JOIN T_Merchandise merchandise
ON salebilldetail.FMerchandiseId=merchandise.FId
```

为了取得每一笔销售记录中的商品名称，需要连接 T_Merchandise 表；为了统计同产品的总销量，需要使用按 FMerchandiseId 进行分区的 SUM()窗口函数来统计同产品总销售量；由于 FCount 是整数类型，在进行除法运算的时候为了避免精度问题，需要将 FCount 乘以 1.0 这样就可以进行高精度运算了。由于这里使用来的窗口函数，所以这个 SQL 语句不能在 MYSQL 中运行。

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	FCount	
Bacon	30	0.168539325842
Bacon	60	0.337078651685
Bacon	16	0.089887640449
Bacon	6	0.033707865168
Bacon	66	0.370786516853
Cake	518	0.923351158645
Cake	8	0.014260249554
Cake	2	0.003565062388
Cake	22	0.039215686274
Cake	11	0.019607843137
Apple	20	0.277777777777
Apple	12	0.166666666666
Apple	5	0.069444444444
Apple	9	0.125000000000
Apple	26	0.361111111111

11.1.6 为采购单分级

统计每张采购单的单号、总采购额，并且对于总采购额小于等于 500 元的显示为“小额”、总采购额大于等于 1000 元的显示为“大额”、介于 500 元与 1000 元之间的显示为“普通”。

对于这种复杂的问题可以分解成几个小步骤进行处理。首先统计所有采购单的单号以及它们的每条交易明细的交易额，这只要将 T_PurchaseBill、T_PurchaseBillDetail 和 T_Merchandise 这三张表进行连接查询就可以得到。SQL 语句如下：

```
SELECT purchasebill.FNumber,purchasebilldetail.FCount*merchandise.FPrice
FROM T_PurchaseBillDetail purchasebilldetail
INNER JOIN T_PurchaseBill purchasebill
ON purchasebilldetail.FBillId = purchasebill.FId
INNER JOIN T_Merchandise merchandise
ON purchasebilldetail.FMerchandiseId = merchandise.FId;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	
1	24
1	600
2	960
2	108
2	176

3	114
3	12
3	60
4	600
4	108
5	38
5	780
6	18
6	660
6	336

接着需要统计每张采购单的总交易额，因为我们不关系每张采购单内部的交易明细。将上面的结果集按照采购单号进行分组，然后使用聚合函数 SUM()就可以轻松的统计每张采购单的总交易额了。SQL 语句如下：

```
SELECT purchasebill.FNumber,
SUM(purchasebilldetail.FCount*merchandise.FPrice)
FROM T_PurchaseBillDetail purchasebilldetail
INNER JOIN T_PurchaseBill purchasebill
ON purchasebilldetail.FBillId = purchasebill.FId
INNER JOIN T_Merchandise merchandise
ON purchasebilldetail.FMerchandiseId = merchandise.FId
GROUP BY purchasebill.FNumber;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber	
1	624
2	1244
3	186
4	708
5	818
6	1014

这样便统计出了采购单的总交易额，接着需要按照每张采购单的总交易额对采购单进行分级，显然使用 CASE……WHEN 语句很容易的实现这个功能：

```
SELECT purchasebill.FNumber,
SUM(purchasebilldetail.FCount*merchandise.FPrice),
CASE
    WHEN SUM(purchasebilldetail.FCount*merchandise.FPrice)<=500 THEN '小额'
    WHEN SUM(purchasebilldetail.FCount*merchandise.FPrice)>=1000 THEN '大额'
    ELSE '普通'
END
FROM T_PurchaseBillDetail purchasebilldetail
INNER JOIN T_PurchaseBill purchasebill
ON purchasebilldetail.FBillId = purchasebill.FId
INNER JOIN T_Merchandise merchandise
ON purchasebilldetail.FMerchandiseId = merchandise.FId
GROUP BY purchasebill.FNumber;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber		
1	624	普通
2	1244	大额
3	186	小额
4	708	普通
5	818	普通
6	1014	大额

这样就完成了要求的功能。不过上面的 SQL 有一个问题，那就是计算每张采购单总交易额的聚合函数“SUM(purchasebilldetail.FCount*merchandise.FPrice)”出现了很多次，这违反了 DRY 原则造成后期维护的困难，而且有可能造成性能问题，因为乘法运算是非常低效的。可以使用子查询来解决这个问题，在子查询中计算每张采购单总交易额，这样在外部只要引用计算结果就可以了。SQL 语句如下：

```
SELECT t.BillNumber,t.TotalAmount,
CASE
    WHEN t.TotalAmount <=500 THEN '小额'
    WHEN t.TotalAmount >=1000 THEN '大额'
    ELSE '普通'
END
FROM
(
    SELECT purchasebill.FNumber AS BillNumber,
    SUM(purchasebilldetail.FCount*merchandise.FPrice) AS TotalAmount
    FROM T_PurchaseBillDetail purchasebilldetail
    INNER JOIN T_PurchaseBill purchasebill
    ON purchasebilldetail.FBillId = purchasebill.FId
    INNER JOIN T_Merchandise merchandise
    ON purchasebilldetail.FMerchandiseId = merchandise.FId
    GROUP BY purchasebill.FNumber
) t
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNumber		
1	624	普通
2	1244	大额
3	186	小额
4	708	普通
5	818	普通
6	1014	大额

11.1.7 检索所有重叠日期销售单

要求索所有重叠日期销售单。重叠日期销售单的定义如下：如果销售单 t1 的开单日期介于销售单 t2 的开单日期和确认日期之间的话，我们就说 t1 与 t2 日期重叠。

类似的需求在生产排程、工作计划安排等场景也可以看到。这种自身与自身进行比较的需求一般都可以使用自连接来解决。SQL 语句如下：

```
SELECT t1.FNumber,t1.FMakeDate,t1.FConfirmDate,
```

```
t2.FNumber,t2.FMakeDate,t2.FConfirmDate
FROM T_SaleBill t1,T_SaleBill t2
WHERE t2.FMakeDate>=t1.FMakeDate
AND t2.FMakeDate<=t1.FConfirmDate
AND t1.FId<>t2.FId
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FMAKEDATE	FCONFIRMDATE	FNUMBER	FMAKEDATE	FCONFIRMDATE
6	2002-02-03	2007-11-11	1	2007-03-15	2007-05-15
6	2002-02-03	2007-11-11	2	2006-01-25	2006-02-03
6	2002-02-03	2007-11-11	3	2006-02-12	2007-01-11

为了更加清晰的理解上面的 SQL 语句，我们首先查看没有 WHERE 子句的自连接部分，观察一下它是如何能在一行中看到其他所有销售单的：

```
SELECT t1.FNumber,t1.FMakeDate,t1.FConfirmDate,
t2.FNumber,t2.FMakeDate,t2.FConfirmDate
FROM T_SaleBill t1,T_SaleBill t2
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FMAKEDATE	FCONFIRMDATE	FNUMBER	FMAKEDATE	FCONFIRMDATE
1	2007-03-15	2007-05-15	1	2007-03-15	2007-05-15
2	2006-01-25	2006-02-03	1	2007-03-15	2007-05-15
3	2006-02-12	2007-01-11	1	2007-03-15	2007-05-15
4	2008-05-25	2008-06-15	1	2007-03-15	2007-05-15
5	2008-03-17	2007-04-15	1	2007-03-15	2007-05-15
6	2002-02-03	2007-11-11	1	2007-03-15	2007-05-15
1	2007-03-15	2007-05-15	2	2006-01-25	2006-02-03
2	2006-01-25	2006-02-03	2	2006-01-25	2006-02-03
3	2006-02-12	2007-01-11	2	2006-01-25	2006-02-03
4	2008-05-25	2008-06-15	2	2006-01-25	2006-02-03
5	2008-03-17	2007-04-15	2	2006-01-25	2006-02-03
6	2002-02-03	2007-11-11	2	2006-01-25	2006-02-03
1	2007-03-15	2007-05-15	3	2006-02-12	2007-01-11
2	2006-01-25	2006-02-03	3	2006-02-12	2007-01-11
3	2006-02-12	2007-01-11	3	2006-02-12	2007-01-11
4	2008-05-25	2008-06-15	3	2006-02-12	2007-01-11
5	2008-03-17	2007-04-15	3	2006-02-12	2007-01-11
6	2002-02-03	2007-11-11	3	2006-02-12	2007-01-11
1	2007-03-15	2007-05-15	4	2008-05-25	2008-06-15
2	2006-01-25	2006-02-03	4	2008-05-25	2008-06-15
3	2006-02-12	2007-01-11	4	2008-05-25	2008-06-15
4	2008-05-25	2008-06-15	4	2008-05-25	2008-06-15
5	2008-03-17	2007-04-15	4	2008-05-25	2008-06-15
6	2002-02-03	2007-11-11	4	2008-05-25	2008-06-15

1	2007-03-15	2007-05-15	5	2008-03-17	2007-04-15
2	2006-01-25	2006-02-03	5	2008-03-17	2007-04-15
3	2006-02-12	2007-01-11	5	2008-03-17	2007-04-15
4	2008-05-25	2008-06-15	5	2008-03-17	2007-04-15
5	2008-03-17	2007-04-15	5	2008-03-17	2007-04-15
6	2002-02-03	2007-11-11	5	2008-03-17	2007-04-15
1	2007-03-15	2007-05-15	6	2002-02-03	2007-11-11
2	2006-01-25	2006-02-03	6	2002-02-03	2007-11-11
3	2006-02-12	2007-01-11	6	2002-02-03	2007-11-11
4	2008-05-25	2008-06-15	6	2002-02-03	2007-11-11
5	2008-03-17	2007-04-15	6	2002-02-03	2007-11-11
6	2002-02-03	2007-11-11	6	2002-02-03	2007-11-11

结果集数据量是非常大的。这个结果集是 T_SaleBill 表自连接的结果，一共是 36 条记录（6*6）。从上面的结果集中很容易的找到重叠日期：只要返回 t2.FMakeDate 介于 t1.FMakeDate 和 t1.FConfirmDate 之间的记录即可。这样只要再增加下面的 WHERE 子句即可：

```
WHERE t2.FMakeDate>=t1.FMakeDate
AND t2.FMakeDate<=t1.FConfirmDate
AND t1.FId<>t2.FId
```

11.1.8为查询编号

要求按照主键排序，检索所有制单人不为空的销售单，并且为每行显示一个行号。

在MSSQLServer、Oracle、DB2等支持窗口函数的DBMS中，使用窗口函数ROW_NUMBER()可以完成这个功能：

```
SELECT ROW_NUMBER() OVER(ORDER BY FId) AS rn,
FNumber,FMakeDate
FROM T_SaleBill
WHERE FBillMakerId IS NOT NULL
```

执行完毕我们就能在输出结果中看到下面的执行结果：

RN	FNUMBER	FMAKEDATE
1	1	2007-03-15
2	3	2006-02-12
3	4	2008-05-25
4	5	2008-03-17
5	6	2002-02-03

对于MYSQL以及MSSQLServer2000等不支持窗口函数的DBMS函数可以使用子查询来完成这个功能：

```
SELECT
(
    SELECT COUNT(*) FROM T_SaleBill t1
    WHERE t1.FId<=t2.FId
        AND t1.FBillMakerId IS NOT NULL
) AS rn,
t2.FNumber,t2.FMakeDate
FROM T_SaleBill t2
```

```
WHERE t2.FBillMakerId IS NOT NULL
ORDER BY t2.FId
```

由于是按照FId排序，而且FId的值是唯一的，所以使用相关子查询计算小于等于当前FId值的行的个数就可以得到当前行的行号。执行完毕我们就能在输出结果中看到下面的执行结果：

RN	FNUMBER	FMAKEDATE
1	1	2007-03-15
2	3	2006-02-12
3	4	2008-05-25
4	5	2008-03-17
5	6	2002-02-03

11.1.9 标记所有单内最大销售量

要求将每张销售单中销售量最大的明细记录标记出来。

尝试使用下面的 SQL 语句来来完成要求的功能：

```
SELECT FId,FBillId,FMerchandiseId,FCount,
CASE
    WHEN FCount=MAX(FCount)
    THEN '单内最大值'
    ELSE ''
END
FROM T_SaleBillDetail
GROUP BY FBillId
```

在这个 SQL 语句中，首先按照 FBillId 进行分组，然后使用聚合函数 **MAX()来计算组内** FCount **的最大值，最后使用 CASE 函数判断每一行的** FCount 是否等于这个最大值。

执行这个 SQL 语句后 DBMS 会报出如下的错误信息：

选择列表中的列 'T_SaleBillDetail.FId' 无效，因为该列没有包含在聚合函数或 GROUP BY 子句中。

出现这个错误的原因是因为出现在 SELECT 列表中的所有列如果不是在聚合函数中使用则必须加入 GROUP BY 子句中。为了保证这个 SQL 语句能够正确运行，需要将用到的所有列放到 GROUP BY 子句中，SQL 语句如下：

```
SELECT FId,FBillId,FMerchandiseId,FCount,
CASE
    WHEN FCount=MAX(FCount)
    THEN '单内最大值'
    ELSE ''
END
FROM T_SaleBillDetail
GROUP BY FBillId,FId,FBillId,FMerchandiseId,FCount;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FBillId	FMerchandiseId	FCount	
00001	00001	00003	20	单内最大值
00002	00001	00001	30	单内最大值
00003	00001	00002	22	单内最大值
00004	00002	00003	12	单内最大值
00005	00002	00002	11	单内最大值
00006	00003	00001	60	单内最大值

00007	00003	00002	2	单内最大值
00008	00003	00003	5	单内最大值
00009	00004	00001	16	单内最大值
00010	00004	00002	8	单内最大值
00011	00004	00003	9	单内最大值
00012	00005	00001	6	单内最大值
00013	00005	00003	26	单内最大值
00014	00006	00001	66	单内最大值
00015	00006	00002	518	单内最大值

虽然 SQL 语句能够执行通过了，不过非常遗憾的是，这个执行结果是错误的，因为将 SELECT 列表中的所有列都放到 GROUP BY 子句中会破坏原有的分组。这里将讲解使用聚合函数而又不必将 SELECT 列表中的所有列都放到 GROUP BY 子句中的技巧。

在介绍窗口函数的时候曾经提到，使用窗口函数将无需使用 GROUP BY 子句，而且窗口函数中的聚合计算不会影响其他的列，因此对于支持窗口函数的 DBMS 可以使用如下的 SQL 语句：

```
SELECT FId,FBillId,FMerchandiseId,FCount,
CASE
    WHEN FCount=MAX(FCount) OVER(PARTITION BY FBillId)
    THEN '单内最大值'
    ELSE ''
END
FROM T_SaleBillDetail;
```

这里使用窗口函数“`MAX(FCount) OVER(PARTITION BY FBillId)`”计算每一行所属的销售单中的销售量的最大值，然后将其与 `FCount` 进行比较，如果等于 `FCount` 则表示当前行是销售量的最大值所在的行。

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FBillId	FMerchandiseId	FCount	
00001	00001	00003	20	
00002	00001	00001	30	单内最大值
00003	00001	00002	22	
00004	00002	00003	12	单内最大值
00005	00002	00002	11	
00006	00003	00001	60	单内最大值
00007	00003	00002	2	
00008	00003	00003	5	
00009	00004	00001	16	单内最大值
00010	00004	00002	8	
00011	00004	00003	9	
00012	00005	00001	6	
00013	00005	00003	26	单内最大值
00014	00006	00001	66	
00015	00006	00002	518	单内最大值

对于 MYSQL、MSSQLServer2000 等不支持窗口函数的 DBMS 来说，可以使用相关子

查询来达到相同的效果。SQL 语句如下：

```
SELECT FId,FBillId,FMerchandiseId,FCount,
CASE
    WHEN FCount=
        (
            SELECT MAX(t1.FCount) FROM T_SaleBillDetail t1
            WHERE t1.FBillId=t2.FBillId
        )
    THEN '单内最大值'
    ELSE ''
END
FROM T_SaleBillDetail t2;
```

这里使用相关子查询来计算每一个销售单中的销售量的最大值，其余部分与使用窗口函数是一样的。需要注意的是相关子查询中的 WHERE 子句中将 t1.FBillId 和 t2.FBillId 进行了相等性过滤，这样就达到了窗口函数中“PARTITION BY FBillId”一样的分区计算最大值的效果，因此这个 WHERE 语句是不能遗漏的。

这个案例是非常典型的，当需要使用聚合计算，但是又不希望由于引入聚合函数而需要添加额外的 GROUP BY 子句的话可以使用这里介绍的方案，那就是：支持窗口函数的 DBMS 使用窗口函数，不支持窗口函数的 DBMS 使用子查询。

11.2 排序

很多业务数据都是前后顺序敏感的，因此对数据排序也是一个值得讨论的话题，本节将讨论非字段排序规则、随机排序等问题。

11.2.1 非字段排序规则

到目前为止讲解的结果集排序的例子中都是针对字段的排序，比如下面的 SQL 语句是根据姓名字段进行排序：

```
SELECT * FROM T_Person
ORDER BY FName
```

有的时候排序规则并不是按照一个字段进行排序那么简单，比如需要按照姓名的第二个字母进行排序。ORDER BY 子句中不仅可以使用简单的字段，还可以指定计算字段，这样排序时就会根据计算字段的值来进行排序，比如下面的 SQL 语句就是按照人员姓名的第二个字母进行排序：

MYSQL,MSSQLServer:

```
SELECT * FROM T_Person
ORDER BY    SUBSTRING(FName,2,1)
```

Oracle,DB2:

```
SELECT * FROM T_Person
ORDER BY    SUBSTR(FName,2,1)
```

这里使用 SUBSTRING 函数来计算 FName 字段的第二个字母，然后按照函数的返回值进行排序。执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FNumber	FName	FManagerId
00006	6	Merry	00003
00004	4	Jim	00003
00005	5	Lily	00002
00001	1	Robert	<NULL>

00002	2	John	00001
00003	3	Tom	00001

11.2.2 随机排序

在开发一些新闻、娱乐类应用系统的时候，有时候需要随机的显示数据库中的若干条数据，比如随机显示 3 个人员。要实现这个功能首先需要随机对人员表中的数据进行排序，然后再取排序后的结果集的前三条即可。

要进行随机排序，那么使用随机函数做为排序规则就可以，各个数据库的实现方式如下：

MYSQL：

```
SELECT * FROM T_Person
ORDER BY RAND();
```

MSSQLServer：

```
SELECT * FROM T_Person
ORDER BY NEWID();
```

ORACLE：

```
SELECT * FROM T_Person
ORDER BY dbms_random.value();
```

DB2：

```
SELECT * FROM T_Person
ORDER BY RAND();
```

注意 MSSQLServer 中的实现并没有使用随机函数 RAND()来做为排序，这是因为 MSSQLServer 中的 RAND()函数与其他 DBMS 中的随机函数不同，它在运算的期间在各个行中的值是不变的，因此不能使用 RAND()函数做为排序规则，而用来产生 GUID 值的函数 NEWID()产生的字符串在这种场景下可以看作一个可以利用的“伪随机函数”使用用来替代 RAND()函数。

实现了对 T_Person 表中的数据进行随机排序，要取得结果集的前 3 条记录就非常简单了，各个数据库的实现方式如下：

MYSQL：

```
SELECT * FROM T_Person
ORDER BY RAND()
LIMIT 0,3;
```

MSSQLServer：

```
SELECT TOP 3 * FROM T_Person
ORDER BY NEWID();
```

ORACLE：

```
SELECT * FROM
(
SELECT * FROM T_Person
ORDER BY dbms_random.value()
)
WHERE rownum<=3;
```

DB2：

```
SELECT * FROM T_Person
ORDER BY RAND()
FETCH FIRST 3 ROWS ONLY;
```

由于随机排序需要对每一行都执行随机函数运算，所以会造成全表扫描，因此在数据量比较大的表中，需要慎用这种处理方式。

11.3 表间比较

在一些业务报表中经常需要对多组数据进行比较，在技术实现上就是表间的数据比较。使用连接、IN、EXISTS等都可以解决表间比较的问题，不过这些方式中存在着一些差异，本节将通过例子对这些差异进行比较。

11.3.1 检索制作过采购单的人制作的销售单

要求检索出制作过采购单的人制作的销售单，也就是销售单的制单人必须在采购单中有其制作的单据。

使用子查询可以解决这个问题，对于T_SaleBill 中的每一条记录的制单人都到T_PurchaseBill 表中查看是否存在于T_PurchaseBill 表中。SQL语句如下：

```
SELECT * FROM T_SaleBill
WHERE FBillMakerId IN
(
    SELECT FBillMakerId FROM T_PurchaseBill
);
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNUMBER	FBILLMAKERID	FMAKEDATE	FCONFIRMDATE
00001	1	00006	2007-03-15	2007-05-15
00003	3	00001	2006-02-12	2007-01-11
00006	6	00002	2002-02-03	2007-11-11

11.3.2 检索没有制作过采购单的人制作的销售单

要求检索出没有制作过采购单的人制作的销售单，也就是销售单的制单人必须不在采购单中有其制作的单据。

参照上的例子我们也尝试使用子查询解决这个问题，对于T_SaleBill 中的每一条记录的制单人都到T_PurchaseBill 表中查看是否存在于T_PurchaseBill 表中。只需要将IN运算符修改为NOT IN即可，SQL语句如下：

```
SELECT * FROM T_SaleBill
WHERE FBillMakerId NOT IN
(
    SELECT FBillMakerId FROM T_PurchaseBill
);
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNUMBER	FBILLMAKERID	FMAKEDATE	FCONFIRMDATE

这个执行结果出乎我们的意料，因为返回的查询结果是空的，主键为00003和00005的人员只做了销售单、没做过采购单，但是它们并没有被输出的执行结果中。这是因为SELECT FBillMakerId FROM T_PurchaseBill返回的结果中有一个空值，如下：

FBILLMAKERID
00006

00004
00001
00002
00002
<NULL>

我们知道，IN和NOT IN运算符本质上是OR运算，因而必须从OR运算符对于NULL的处理来考虑IN和NOT IN运算符的运算结果。比如下面的两句SQL语句是等价的：

```
SELECT * FROM T_SaleBill
WHERE FBillMakerId IN
('00006','00004','00001','00002', NULL);

SELECT * FROM T_SaleBill
WHERE FBillMakerId ='00006' OR FBillMakerId ='00004' OR FBillMakerId
='00001' OR FBillMakerId ='00002' OR FBillMakerId =NULL;
```

而对于NOT IN运算符来说，下面的两句SQL语句也是等价的：

```
SELECT * FROM T_SaleBill
WHERE FBillMakerId NOT IN('00006','00004','00001','00002', NULL);

SELECT * FROM T_SaleBill
WHERE NOT(FBillMakerId ='00006' OR FBillMakerId ='00004' OR FBillMakerId
='00001' OR FBillMakerId ='00002' OR FBillMakerId =NULL);
```

可以看到，条件FBillMakerId NOT IN('00006','00004','00001','00002', NULL)等价于NOT(FBillMakerId ='00006' OR FBillMakerId ='00004' OR FBillMakerId ='00001' OR FBillMakerId ='00002' OR FBillMakerId =NULL)

在这种情况下，假设当FBillMakerId等于'00005'时，表达式的输出为：

```
NOT('00005'='00006' OR '00005'='00004' OR '00005'='00001' OR
'00005'='00002' OR '00005'=NULL)
```

我们知道，在SQL中NULL代表“值未知”，“'00005'=NULL”表示判断'00005'是否等于NULL，一个已知的值'00005'是无法确定是否等于一个未知的值的，所以“'00005'=NULL”的返回值也是位置的，所以“'00005'=NULL”的计算结果为NULL。因此上边的条件可以简化为：

```
NOT(FALSE OR FALSE OR FALSE OR NULL)
```

布尔值只有TRUE和FALSE两种取值，对于布尔类型而言NULL表示不知道是TRUE还是FALSE。TRUE和任何布尔值的OR运算结果都是TRUE，因此TRUE和未知值NULL的OR运算结果都是TRUE，因此“TURE OR NULL”的计算结果是TRUE；FALSE和TRUE的OR运算结果是TRUE，而FALSE和FALSE的OR运算结果是FALSE，因此FALSE和未知值NULL的OR运算结果是未知的，因此“FALSE OR NULL”的结果结果是NULL。

根据布尔值与NULL的这个运算特性我们得知“FALSE OR FALSE OR FALSE OR NULL”的运算结果为NULL，而一个未知的值做NOT运算，其结果同样是未知的，因此“NOT NULL”的运算结果为NULL，这样“NOT(FALSE OR FALSE OR FALSE OR NULL)”的运算结果为NULL。既然WHERE条件是未知的，那么结果集中的结果是否符合WHERE条件也就是未知的了，因此检索不出任何数据是非常合理的。

既然问题出在空值上面，只要将空值从子查询中过滤掉即可。为子查询中的查询语句增加一个WHERE条件，将所有空值过滤掉。SQL语句如下：

```
SELECT * FROM T_SaleBill
WHERE FBillMakerId NOT IN
(
SELECT FBillMakerId FROM T_PurchaseBill
WHERE FBillMakerId IS NOT NULL
);
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNUMBER	FBILLMAKERID	FMAKEDATE	FCONFIRMDATE
00004	4	00003	2008-05-25	2008-06-15
00005	5	00005	2008-03-17	2007-04-15

11.4 表复制

在进行数据的备份等场景下，需要创建和指定表格式完全相同的表，甚至要将源表中的数据也原封不动的复制到目标表中，而且完成这一任务的时候无需关心源表的具体结构。

11.4.1 复制源表的结构并复制表中的数据

下面演示以T_Person表为源表创建目标表T_Person2，并且将T_Person中的表复制到T_Person2表。

在MYSQL和ORACLE中支持从结果集来创建一张表，并且将结果集中的数据填充到新创建的表中，使用方法也非常简单，只要在CREATE TABLE语句后使用“AS 查询语句”就可以。比如：

```
CREATE TABLE T_Person2
AS
SELECT * FROM T_Person
```

在MSSQLServer中可以使用SELECT……INTO语句来将检索的结果插入到一张新表中，如下：

```
SELECT * INTO T_Person2
FROM T_Person
```

在DB2中可以使用带LIKE子句的CREATE TABLE语句来创建目标表，比如下面的SQL语句表示创建和T_Person表格式一样的表T_Person2：

```
CREATE TABLE T_Person2 LIKE T_Person;
```

不过这个语句只能复制表结构，不能同时将T_Person表中的数据复制到T_Person2表中的。前面介绍过使用INSERT……SELECT语句可以快速将结果集插入到目标表中，因此在执行完上面的**CREATE TABLE语句后还需再执行一个INSERT语句来将**T_Person表中的输入插入到T_Person2表中：

```
INSERT INTO T_Person2
SELECT * FROM T_Person;
```

本案例讲解完毕，请删除T_Person2表：

```
DROP TABLE T_Person2;
```

11.4.2 只复制源表的结构

下面演示以T_Person表为源表创建目标表T_Person2，但是并不将T_Person中的表复制到T_Person2表。

MYSQL、MSSQLServer和Oracle中都是从结果集来创建表的，而且会将结果集中的数据插入到目标表中，因此如果不需要将T_Person中的表复制到T_Person2表只需要让结果集为空就可以了。注意结果集为空并不表示结果集没有意义，因为空的结果集中仍然包含了结果集的列信息。创建一个空结果集的最简单方式就是使用永远为FALSE的WHERE条件“WHERE 1<>1”。

MYSQL和ORACLE：

```
CREATE TABLE T_Person2
AS
SELECT * FROM T_Person
WHERE 1<>1
```

MSSQLServer：

```
SELECT * INTO T_Person2
FROM T_Person
WHERE 1<>1
```

“WHERE 1<>1”这样的查询条件有可能造成全表扫描从而带来性能问题，因此如果数据量比较大的话可以用取结果集前0条这样的方式来得到一个空结果集，具体实现方式请参考前面章节。

而DB2中的带LIKE子句的CREATE TABLE语句本来就不复制数据，因此只要执行下面的SQL语句就可以了：

```
CREATE TABLE T_Person2 LIKE T_Person;
```

本案例讲解完毕，请删除T_Person2表：

```
DROP TABLE T_Person2;
```

11.5 计算字符在字符串中出现的次数

要求计算字符“r”在每个员工名字中出现的次数。

SQL中并没有提供计算字符在字符串中出现的次数的函数，不过可以通过下面的思路来等价实现：首先计算FName的长度，然后将FName中的所有“r”删掉，计算删掉“r”以后的FName的长度，这两个长度之差就是字符“r”在FName中出现的次数。

MYSQL,Oracle,DB2:

```
SELECT FName, LENGTH(FName)  -  LENGTH( REPLACE(FName,'r','') )
FROM T_Person
```

MSSQLServer:

```
SELECT FName, LEN(FName)  -  LEN( REPLACE(FName,'r','') )
FROM T_Person
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FName	
Robert	2
John	0
Tom	0
Jim	0
Lily	0
Merry	2

11.6 去除最高分、去除最低分

“去掉最高分和最低分，最后平均得分……”，这是在很多比赛中经常听到的。最高和最低分对平均分的影响较大，为了避免个别评委的不公平评分而造成得分的不公正，去掉最高和最低分的做法是有效方法。去除最高值、最低值有两种策略：去除所有最低最高值和只去除一个最低最高值。本节将以计算销售单中的销售量的平均值为例进行介绍。

11.6.1 去除所有最低最高值

使用子查询可以排除所有的最高值和最低值，子查询返回表中的最大值和最小值，针对返回的值使用NOT IN就可以从所有值中排除所有的最大值和最小值，然后使用AVG()函数计算平均值。SQL语句如下：

```
SELECT AVG(FCount) FROM T_SaleBillDetail
WHERE FCount NOT IN
(
    (SELECT MIN(FCount) FROM T_SaleBillDetail),
    (SELECT MAX(FCount) FROM T_SaleBillDetail)
)
```

在MSSQLServer、Oracle、DB2等支持窗口函数的DBMS中还可以使用窗口函数来完成这个功能。SQL语句如下：

```
SELECT AVG(t.FCount)
FROM
(
    SELECT FCount,MIN(FCount) OVER() min_count,
    MAX(FCount) OVER() max_count
    FROM T_SaleBillDetail
)t
WHERE t.FCount NOT IN(min_count, max_count)
```

子查询t返回所有的值（包括最大值和最小值）、最大值和最小值，这样从每一行中都可以访问最大值和最小值，因此找出哪些值是最大值和最小值非常简单。外层的查询语句对子查询t中的行做筛选，这样所有与min_count和max_count相等的行都会被排除。

11.6.2 只去除一个最低最高值

上面的实现方式如果存在多个最高值或者最低值，那么它们都会被排除。如果只想排除一个最高值和一个最低值，那么只需要从总和SUM()中减去最高值MAX()和最低值MIN()，然后除以COUNT()-2就可以得到平均值了：

```
SELECT (SUM(FCount)-MIN(FCount)-MAX(FCount))/(COUNT(FCount)-2)
FROM T_SaleBillDetail;
```

注意这个SQL语句是存在BUG的，也就是如果表T_SaleBillDetail中恰好只有两条记录，那么COUNT(FCount)-2就会返回0，这就会造成以0为除数的计算错误。可以使用CASE……WHEN语句来修复这个BUG，SQL语句如下：

```
SELECT
    CASE COUNT(FCount)
    WHEN 2 THEN NULL
    ELSE (SUM(FCount)-MIN(FCount)-MAX(FCount))/(COUNT(FCount)-2)
    END
FROM T_SaleBillDetail;
```

在这个SQL语句中，如果恰好只有两条记录那么就将NULL做为平均值返回。

11.7 日期相关的应用

在实际应用中，很多数据是时间相关的，因此灵活的使用SQL进行日期处理将会大大提高开发速度，本节将会介绍SQL中的常见的日期相关的应用。

11.7.1 计算销售确认日和制单日之间相差的天数

要求计算销售单的确认日和制单日之间相差的天数。

不同的DBMS中计算两个日期之间相差的天数的方式是不同的，具体可以参考第五章。

```
MYSQL:
SELECT FNumber,FMakeDate,FConfirmDate,
DATEDIFF(FConfirmDate,FMakeDate) AS DaysBetween
FROM T_SaleBill

MSSQLServer:
SELECT FNumber,FMakeDate,FConfirmDate,
DATEDIFF(day,FMakeDate,FConfirmDate) AS DaysBetween
FROM T_SaleBill

ORACLE:
SELECT FNumber,FMakeDate,FConfirmDate,
FConfirmDate-FMakeDate AS DaysBetween
FROM T_SaleBill

DB2:
SELECT FNumber,FMakeDate,FConfirmDate,
DAYS(FConfirmDate)-DAYS(FMakeDate) AS DaysBetween
FROM T_SaleBill
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FMAKEDATE	FCONFIRMDATE	DAYSBETWEEN
1	2007-03-15	2007-05-15	61
2	2006-01-25	2006-02-03	9
3	2006-02-12	2007-01-11	333
4	2008-05-25	2008-06-15	21
5	2008-03-17	2007-04-15	-337
6	2002-02-03	2007-11-11	2107

11.7.2 计算两张销售单之间的时间间隔

要求计算以制单时间排序的当前销售单和下一条销售单的制单时间之间相差的天数。

计算两个日期的天数间隔的方式在上一节中已经介绍了，因此只要解决取得以制单时间排序一条销售单的制单时间的方式即可。最通用的解决方案就是使用子查询：

```
MYSQL:
SELECT t.FNumber AS FNumber,t.date1 AS FDate,
DATEDIFF(t.date2,t.date1) AS FDateDiff
FROM
(
    SELECT t2.FNumber,t2.FMakeDate date1,
        (
            SELECT MIN(t1.FMakeDate) FROM T_SaleBill t1
            WHERE t1.FMakeDate>t2.FMakeDate
        ) date2
    FROM T_SaleBill t2
) t
ORDER BY t.date1
```

```
MSSQLServer:
SELECT t.FNumber AS FNumber,t.date1 AS FDate,
DATEDIFF(DAY,t.date1,t.date2) AS FDateDiff
FROM
(
    SELECT t2.FNumber,t2.FMakeDate date1,
    (
        SELECT MIN(t1.FMakeDate) FROM T_SaleBill t1
        WHERE t1.FMakeDate>t2.FMakeDate
    ) date2
    FROM T_SaleBill t2
) t
ORDER BY t.date1

ORACLE:
SELECT t.FNumber AS FNumber,t.date1 AS FDate,
t.date2-t.date1 AS FDateDiff
FROM
(
    SELECT t2.FNumber,t2.FMakeDate date1,
    (
        SELECT MIN(t1.FMakeDate) FROM T_SaleBill t1
        WHERE t1.FMakeDate>t2.FMakeDate
    ) date2
    FROM T_SaleBill t2
) t
ORDER BY t.date1

DB2:
SELECT t.FNumber AS FNumber,t.date1 AS FDate,
DAYS(t.date2)-DAYS(t.date1) AS FDateDiff
FROM
(
    SELECT t2.FNumber,t2.FMakeDate date1,
    (
        SELECT MIN(t1.FMakeDate) FROM T_SaleBill t1
        WHERE t1.FMakeDate>t2.FMakeDate
    ) date2
    FROM T_SaleBill t2
) t
ORDER BY t.date1
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNUMBER	FDATE	FDATEDIFF

6	2002-02-03	1452
2	2006-01-25	18
3	2006-02-12	396
1	2007-03-15	368
5	2008-03-17	69
4	2008-05-25	<NULL>

虽然在不同的DBMS中的SQL语句略有差异，不过它们的差别在于计算两个日期之间相差天数的方式，它们取得下一个日期的方式是一致的，核心实现就在于下面的子查询片段：

```
SELECT t2.FNumber,t2.FMakeDate date1,
(
    SELECT MIN(t1.FMakeDate) FROM T_SaleBill t1
    WHERE t1.FMakeDate>t2.FMakeDate
) date2
FROM T_SaleBill t2
```

这个子查询表示“取得比当前制单日期大的最小日期”，因为结果集是按照制单日期排序的，所以“比当前制单日期大的最小日期”所在的行既是下一条行。

除了这种实现方案，在Oracle中还可以通过窗口函数LEAD()来取得下一个日期，进而进行天数间隔运算：

```
SELECT FNumber,FMakeDate,
LEAD(FMakeDate) OVER(ORDER BY FMakeDate)-FMakeDate
FROM T_SaleBill
ORDER BY FMakeDate
```

这种实现方式更加清晰易读。这里“LEAD(FMakeDate) OVER(ORDER BY FMakeDate)”表示取得以FMakeDate排序的下一个日期，对此不熟悉的读者可以参考前面章节。

11.7.3 计算销售单制单日期所在年份的天数

要求计算计算销售单制单日期所在年份的天数。

解决这个问题的思路为：首先取得当年的第一天，然后在当年的第一天的基础上增加一年以得到第二年的第一天，最后计算这两个日期之间相差的天数就是当前时间的总天数。

MYSQL：

```
SELECT DATEDIFF(FCurrentYear +interval 1 year, FCurrentYear)
FROM
(
    SELECT ADDDATE(FMakeDate,-DAYOFYEAR(FMakeDate)+1) FCurrentYear
    FROM T_SaleBill
) t
```

MSSQLServer：

```
SELECT DATEDIFF(DAY,FCurrentYear,DATEADD(YEAR,1, FCurrentYear))
FROM
(
    SELECT DATEADD(DAY,-DATEPART(DY,FMakeDate)+1,
           FMakeDate) FCurrentYear
    FROM T_SaleBill
) t
```

```
ORACLE:
SELECT ADD_MONTHS(TRUNC(FMakeDate,'y'),12)-TRUNC(FMakeDate,'y')
FROM T_SaleBill

DB2:
SELECT DAYS(FCurrentYear+1 year)-DAYS(FCurrentYear)
FROM
(
    SELECT (FMakeDate-DAYOFYEAR(FMakeDate) DAY + 1 DAY) FCurrentYear
    FROM T_SaleBill
)t
```

11.7.4计算销售单制单日期所在月份的第一天和最后一天

要求计算销售单制单日期所在月份的第一天和最后一天。

MYSQL中使用DAY()函数可以得到制单日期的月份日期，然后用DATE_ADD()函数从制单日期中减去这个值并且加1，这样就可以得到制单日期所在月份的第一天；MYSQL中的LAST_DAY()函数则可以用来得到制单日期所在月份的最后一天。SQL语句如下：

```
SELECT
DATE_ADD(FMakeDate,interval (-DAY(FMakeDate)+1) DAY) AS firstday,
LAST_DAY(FMakeDate) AS lastday
FROM T_SaleBill
```

MSSQLServer中的取得月份的第一天的实现思路和MYSQL中一样，只是实现细节略有不同而已；要得到制单日期所在月份的最后一天，可以给制单日期加一个月，然后从这个值中减去由DAY()函数返回的值。SQL语句如下：

```
SELECT DATEADD(DAY,-DAY(FMakeDate)+1,FMakeDate) AS firstday,
DATEADD(DAY,-DAY(FMakeDate),DATEADD(MONTH,1,FMakeDate)) AS lastday
FROM T_SaleBill
```

ORACLE中函数TRUNC()可以用来将日期截取到任意精度，比如年、月、日等，因此使用TRUNC()截取精度到月就可以得到制单日期所在月份的第一天，而使用函数LAST_DAY()得到制单日期所在月份的最后一天。SQL语句如下：

```
SELECT TRUNC(FMakeDate,'mm') AS firstday,
LAST_DAY(FMakeDate) AS lastday
FROM T_SaleBill
```

DB2中DAY()函数可以返回制单日期所在月份日期，从制单日期中减去这个值，然后加1，就能得到制单日期所在月份日期的第一天；要得到制单日期所在月份日期的最后一天，可以先给制单日期加一个月，然后从这个值中减去由DAY()函数返回的值。SQL语句如下：

```
SELECT (FMakeDate-DAY(FMakeDate) DAY +1 DAY) AS firstday,
(FMakeDate+1 MONTH-DAY(FMakeDate) DAY) AS lastday
FROM T_SaleBill
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FIRSTDAY	LASTDAY
2007-03-01	2007-03-31
2006-01-01	2006-01-31
2006-02-01	2006-02-28

2008-05-01	2008-05-31
2008-03-01	2008-03-31
2002-02-01	2002-02-28

11.8 结果集转置

使用SQL语句可以创建非常复杂的报表，不过出于美观性和兼容使用者喜欢的考虑经常需要调整报表的输出格式，通过结果集转置可以快速调整输出格式，本节将介绍几种常见的结果集转置方式。

11.8.1 将结果集转置为一行

假设只有Bacon、Cake和Apple三种产品，要求统计每种产品的销售量。要求以下面的格式显示：

Apple	Bacon	Cake
20	33	88

使用带GROUP BY子句的SELECT语句可以很容易的统计每种产品的销售量，SQL语句如下：

```
SELECT merchandise.FName,SUM(salebilldetail.FCount)
FROM T_SaleBillDetail salebilldetail
INNER JOIN T_Merchandise merchandise
ON salebilldetail.FMerchandiseId = merchandise.FId
GROUP BY merchandise.FName
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FNAME	
Apple	72
Bacon	178
Cake	561

这样虽然可以统计每种产品的总销量，不过显示格式与要求的不同，必须要将其向顺时针方向转90度成下面的格式：

APPLE	BACON	CAKE
72	178	561

转换方式非常简单，只要使用CASE语句将各行数据费城列，然后统计各列值的合计即可。SQL语句如下：

```
SELECT
SUM(CASE WHEN FMerchandiseId='00003' THEN FCount ELSE 0 END) AS Apple,
SUM(CASE WHEN FMerchandiseId='00001' THEN FCount ELSE 0 END) AS Bacon,
SUM(CASE WHEN FMerchandiseId='00002' THEN FCount ELSE 0 END) AS Cake
FROM T_SaleBillDetail
```

执行完毕我们就能在输出结果中看到下面的执行结果：

APPLE	BACON	CAKE
72	178	561

这与要求的格式完全一致。如果不能理解这种用法，可以首先去掉聚合函数SUM()，执行下面的SQL语句：

```
SELECT
CASE WHEN FMerchandiseId='00003' THEN FCount ELSE 0 END AS Apple,
CASE WHEN FMerchandiseId='00001' THEN FCount ELSE 0 END AS Bacon,
CASE WHEN FMerchandiseId='00002' THEN FCount ELSE 0 END AS Cake
FROM T_SaleBillDetail
```

以Apple列为例，通过CASE函数判断FMerchandiseId是否等于'00003'，如果等于'00003'则表示它是Apple，因此列出FCount，如果不等于'00003'则表示它不是Apple，因此列出0值。执行完毕我们就能在输出结果中看到下面的执行结果：

APPLE	BACON	CAKE
20	0	0
0	30	0
0	0	22
12	0	0
0	0	11
0	60	0
0	0	2
5	0	0
0	16	0
0	0	8
9	0	0
0	6	0
26	0	0
0	66	0
0	0	518

结果集中每行对应一个销售单明细，每一列对应一种商品，如果此销售单明细中的商品是对应的产品则列的值为此产品的销量，否则列的值为0。接下来只要统计每一列的值的总和就可以得到每种商品的总销量了：

```
SELECT
SUM(CASE WHEN FMerchandiseId='00003' THEN FCount ELSE 0 END) AS Apple,
SUM(CASE WHEN FMerchandiseId='00001' THEN FCount ELSE 0 END) AS Bacon,
SUM(CASE WHEN FMerchandiseId='00002' THEN FCount ELSE 0 END) AS Cake
FROM T_SaleBillDetail
```

因为列数是固定的，无法实现动态列，所以实现这种转置要求产品的种类是确定的。

11.8.2 把结果集转置为多行

要求显示每张销售单中明细记录的产品编号。如果使用简单的SQL语句可以显示如下的格式：

FBillId	FMerchandiseId
00001	00003
00001	00001
00001	00002
00002	00003
00002	00002
00003	00001
00003	00002
00003	00003
00004	00001
00004	00002
00004	00003

00005	00001
00005	00003
00006	00001
00006	00002

希望重新调整结果集的格式，使每张销售单使用一列（假设只有这六张销售单）：

Bill_00001	Bill_00002	Bill_00003	Bill_00004	Bill_00005	Bill_00006
00003	00003	00003	00003	00003	00002
00002	00002	00002	00002	00001	00001
00001		00001	00001		

首先，使用窗口函数ROW_NUMBER()使为每个FBillId、FMerchandiseId组合是设定一个编号：

```
SELECT FBillId,FMerchandiseId,
ROW_NUMBER() OVER(PARTITION BY FBillId ORDER BY FMerchandiseId) rn
FROM T_SaleBillDetail
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FBillId	FMerchandiseId	rn
00001	00001	1
00001	00002	2
00001	00003	3
00002	00002	1
00002	00003	2
00003	00001	1
00003	00002	2
00003	00003	3
00004	00001	1
00004	00002	2
00004	00003	3
00005	00001	1
00005	00003	2
00006	00001	1
00006	00002	2

rn为每个FBillId、FMerchandiseId组合指定了一个编号，由于窗口函数是按照FBillId字段进行分区的，所以这个编号是FBillId相同的组内唯一的。

接着，使用CASE表达式，把FMerchandiseId组合对应的FBillId：

```
SELECT
(CASE WHEN FBillId='00001' THEN FMerchandiseId ELSE '' END) AS Bill_00001,
(CASE WHEN FBillId='00002' THEN FMerchandiseId ELSE '' END) AS Bill_00002,
(CASE WHEN FBillId='00003' THEN FMerchandiseId ELSE '' END) AS Bill_00003,
(CASE WHEN FBillId='00004' THEN FMerchandiseId ELSE '' END) AS Bill_00004,
(CASE WHEN FBillId='00005' THEN FMerchandiseId ELSE '' END) AS Bill_00005,
(CASE WHEN FBillId='00006' THEN FMerchandiseId ELSE '' END) AS Bill_00006
FROM
(
    SELECT FBillId,FMerchandiseId,
```

```
    ROW_NUMBER() OVER(PARTITION BY FBillId ORDER BY FMerchandiseId) rn
    FROM T_SaleBillDetail
) t
```

执行完毕我们就能在输出结果中看到下面的执行结果：

Bill_00001	Bill_00002	Bill_00003	Bill_00004	Bill_00005	Bill_00006
00003					
00001					
00002					
	00003				
	00002				
		00001			
		00002			
		00003			
			00001		
			00002		
			00003		
				00001	
				00003	
					00001
					00002

以Bill_00001列为例，CASE函数将不是00001销售单的值全部置为空格，这样保证只有00001销售单明细中所具有的商品列出来。这一步是转置的关键，至此所有的行都变换为列。最后需要剔除空格，使得结果集能够以需求中要求的格式显示出来。接着需要按照rn进行分组，由于空格没有意义，而空格又比所有的销售单主键小，所以使用聚合函数MAX()就可以将所有的空格过滤掉从而得到每一个rn、FBillId、FMerchandiseId组合：

```
SELECT
MAX(CASE WHEN FBillId='00001' THEN FMerchandiseId ELSE '' END) AS
Bill_00001,
MAX(CASE WHEN FBillId='00002' THEN FMerchandiseId ELSE '' END) AS
Bill_00002,
MAX(CASE WHEN FBillId='00003' THEN FMerchandiseId ELSE '' END) AS
Bill_00003,
MAX(CASE WHEN FBillId='00004' THEN FMerchandiseId ELSE '' END) AS
Bill_00004,
MAX(CASE WHEN FBillId='00005' THEN FMerchandiseId ELSE '' END) AS
Bill_00005,
MAX(CASE WHEN FBillId='00006' THEN FMerchandiseId ELSE '' END) AS
Bill_00006
FROM
(
    SELECT FBillId,FMerchandiseId,
    ROW_NUMBER() OVER(PARTITION BY FBillId ORDER BY FMerchandiseId) rn
    FROM T_SaleBillDetail
) t
```

```
GROUP BY rn
```

执行完毕我们就能在输出结果中看到下面的执行结果：

Bill_00001	Bill_00002	Bill_00003	Bill_00004	Bill_00005	Bill_00006
00003	00003	00003	00003	00003	00002
00002	00002	00002	00002	00001	00001
00001		00001	00001		

在MYSQL、MSSQLServer2000等DBMS中并不支持窗口函数，所以上面的SQL语句是无法正确执行的。上面的SQL语句中是使用窗口函数ROW_NUMBER()来为行设定一个编号的，在本书的前面章节已经介绍了在不支持窗口函数的DBMS中使用子查询来变通实现窗口函数ROW_NUMBER()，在此不做赘述。在不支持窗口函数的DBMS中可以使用下面的SQL语句来完成同样的功能：

```
SELECT
MAX(CASE WHEN FBillId='00001' THEN FMerchandiseId ELSE '' END) AS
Bill_00001,
MAX(CASE WHEN FBillId='00002' THEN FMerchandiseId ELSE '' END) AS
Bill_00002,
MAX(CASE WHEN FBillId='00003' THEN FMerchandiseId ELSE '' END) AS
Bill_00003,
MAX(CASE WHEN FBillId='00004' THEN FMerchandiseId ELSE '' END) AS
Bill_00004,
MAX(CASE WHEN FBillId='00005' THEN FMerchandiseId ELSE '' END) AS
Bill_00005,
MAX(CASE WHEN FBillId='00006' THEN FMerchandiseId ELSE '' END) AS
Bill_00006
FROM
(
    SELECT t2.FBillId AS FBillId,t2.FMerchandiseId AS FMerchandiseId,
    (
        SELECT COUNT(*) FROM T_SaleBillDetail t1
        WHERE t2.FBillId=t1.FBillId AND
t2.FMerchandiseId<t1.FMerchandiseId
    ) AS rn
    FROM T_SaleBillDetail t2
) t
GROUP BY rn;
```

11.9 递归查询

在编程语言中，递归是一个非常重要的特性，不过由于关系数据库是以集合的观点来处理数据的，所以在实现递归查询的时候需要使用一些高级的语言特性。在Oracle中可以使用CONNECT BY子句可以轻松的实现递归查询，在MSSQLServer和DB2中则可以使用WITH子句来实现递归查询， MYSQL中即不支持CONNECT BY子句也不支持WITH子句，所以要实现递归查询就必须使用其他的方式来变通实现，而且实现方案也随需求的不同而有差异。本节将介绍递归查询相关的基础知识，并且讲解常见的递归查询的应用。

11.9.1 Oracle中的CONNECT BY子句

T_Person表含有人员主键（FId）和主管领导主键（FManagerId）两列，通过这两列反

映出来的就是人员之间领导和被领导的关系。有些人员领导另一些人员，有些人员被领导，还有些人员领导一些人又被别人领导，他们之间的这种关系就是一种树结构。

树结构的数据可以存放在数据表中，数据之间的层次关系即父子关系通过表中的列与列间的关系来描述，如T_Person表中的FId和FManagerId。FId表示该人员的主键，FManagerId表示领导该人员的人的编号，即子节点的FId值等于父节点的FManagerId值。在表的每一行中都有一个表示父节点的FManagerId（除根节点外），通过每个节点的父节点，就可以确定整个树结构。

在Oracle中，可以在SELECT命令中使用CONNECT BY子句查询表中的树型结构关系。其命令格式如下：

```
CONNECT BY {PRIOR 列名1=列名2|列名1=PRIOR 裂名2}
[START WITH];
```

CONNECT BY子句说明每行数据将是按层次顺序检索，并规定将表中的数据连入树型结构的关系中；PRIORY运算符必须放置在连接关系的两列中某一个的前面，对于节点间的父子关系，PRIOR运算符在的一侧表示父节点，在另一侧表示子节点，从而确定查找树结构是的顺序是自顶向下还是自底向上；START WITH 子句用来标识哪个节点作为查找树型结构的根节点。

11.9.1.1 简单的应用

下面的SQL语句以树结构方式显示以'00001'为根的T_Person表中的数据：

```
SELECT FId,FNumber,FName,FManagerId
FROM T_Person
CONNECT BY PRIOR FId=FManagerId
START WITH FId='00001';
```

START WITH即可以放在CONNECT BY子句的前面，也可以放在CONNECT BY子句的后面，所以下面的SQL语句也是正确的：

```
SELECT FId,FNumber,FName,FManagerId
FROM T_Person
START WITH FId='00001'
CONNECT BY PRIOR FId=FManagerId;
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNUMBER	FNAME	FMANAGERID
00001	1	Robert	<NULL>
00002	2	John	00001
00005	5	Lily	00002
00003	3	Tom	00001
00004	4	Jim	00003
00006	6	Merry	00003

11.9.1.2 PRIOR运算符

运算符PRIOR被放置于等号前后的位置，决定着查询时的顺序。

如果PRIOR被置于CONNECT BY子句中等号的前面时，则从根节点到叶节点的检索，即由父节点向子节点方向通过树结构，我们称之为自顶向下的查询方式。如：

```
CONNECT BY PRIOR FId=FManagerId
```

如果PIROR运算符被置于CONNECT BY 子句中等号的后面时，则从叶节点到根节点的顺序检索，即由子节点向父节点方向检索树结构，我们称之为自底向上的方式。例如：

```
CONNECT BY FId= PRIOR FManagerId
```

下面的SQL语句从00003节点开始自底向上查找T_Person的树结构：

```
SELECT FId,FNumber,FName,FManagerId
FROM T_Person
CONNECT BY FId= PRIOR FManagerId
START WITH FId='00003';
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNUMBER	FNAME	FMANAGERID
00006	6	Merry	00003
00003	3	Tom	00001
00001	1	Robert	<NULL>

可以看到，在这种自底向上的检索过程中，只有树中的一个结点被显示，这是因为在树结构中每一个节点只允许有一个父节点，其查找过程是从开始节点起，找到其父节点，再由其父节点向上，找父节点的父节点。这样一直找到根节点为止，结果就是树中一个节点的数据。

11.9.1.3 计算节点层次

在具有树结构的表中，每一行数据都是树结构中的一个节点，由于节点所处的层次位置不同，所以每行记录都可以有一个层号。不论从哪个节点开始，该起始根节点的层号始终为 1，根节点的子节点为 2……，依此类推。如果使用了 CONNECT BY 子句，那么可以使用名称为 LEVEL 的列来取得每行数据的层次，LEVEL 将返回树型结构中当前节点的层次，可以使用 LEVEL 来控制对树型结构进行遍历的深度。

下面的SQL语句以树结构方式显示以'00001'为根的T_Person表中的数据，并且显示每行数据的层次信息：

```
SELECT FId,FNumber,FName,FManagerId,LEVEL
FROM T_Person
CONNECT BY PRIOR FId=FManagerId
START WITH FId='00001';
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNUMBER	FNAME	FMANAGERID	LEVEL
00001	1	Robert	<NULL>	1
00002	2	John	00001	2
00005	5	Lily	00002	3
00003	3	Tom	00001	2
00004	4	Jim	00003	3
00006	6	Merry	00003	3

使用LEVEL不仅可以得到层次信息，而且还可以使用LEVEL来以更加清晰易懂的方式来显示查询结果。下面的SQL语句以更加清晰的形式展示了树状的结构：

```
SELECT LPAD('-',LEVEL*3,'-')||FId,FNumber,FName,FManagerId,LEVEL
FROM T_Person
CONNECT BY PRIOR FId=FManagerId
START WITH FId='00001';
```

执行完毕我们就能在输出结果中看到下面的执行结果：

LPAD(-,LEVEL*3,-)‖FID	FNUMBER	FNAME	FMANAGERID	LEVEL
---00001	1	Robert	<NULL>	1
------00002	2	John	00001	2
---------00005	5	Lily	00002	3
------00003	3	Tom	00001	2

---------00004	4	Jim	00003	3
---------00006	6	Merry	00003	3

使用了函数LPAD()来得到包含(LEVEL*3)个'-'第字符串，然后将此字符串与人员姓名拼接，这样就可以以这种易于观察的形式来显示层次结构了。

下面的SQL语句使用CASE函数将根节点和最底层节点突出显示，最底层节点的节点也就是Level值最大的节点：

```
SELECT FId,FNumber,FName,FManagerId,LEVEL,
CASE
    WHEN LEVEL=1 THEN '根节点'
    WHEN LEVEL=MAX(LEVEL) OVER() THEN '最底层节点'
    ELSE ' '
END AS FLevelName
FROM T_Person
CONNECT BY PRIOR FId=FManagerId
START WITH FId='00001'
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNUMBER	FNAME	FMANAGERID	LEVEL	FLEVELNAME
00001	1	Robert	<NULL>	1	根节点
00002	2	John	00001	2	
00005	5	Lily	00002	3	最底层节点
00003	3	Tom	00001	2	
00004	4	Jim	00003	3	最底层节点
00006	6	Merry	00003	3	最底层节点

11.9.2 Oracle中的SYS_CONNECT_BY_PATH()函数

使用**CONNECT BY可以显示记录之间的层次结构，而且使用**LPAD()等技巧还可以用更加清晰的方式显示父子结构，不过如果想清晰的查看从根节点开始一直到当前节点的信息就比较麻烦了，必须顺着FManagerId逐级向上看，如果级次比较多的话很容易造成混乱。为了解决这个问题，Oracle中提供了SYS_CONNECT_BY_PATH()函数用来以清晰的形式显示节点的路径。

SYS_CONNECT_BY_PATH()函数必须与**CONNECT BY子句一起使用，**其参数格式如下：

```
SYS_CONNECT_BY_PATH(columnName,seperator)
```

columnName为要连接显示的字段，而seperator则为连接各个层次的分隔符。SYS_CONNECT_BY_PATH()函数的返回值为将层次路径中各个节点的columnName字段值用seperator为分隔符连接起来的字符串。

下面的SQL语句显示每个人员的直接和间接领导者信息：

```
SELECT FId,FNumber,FName,FManagerId,
sys_connect_by_path(FName,'>') AS Name_Path
FROM T_Person
CONNECT BY PRIOR FId=FManagerId
START WITH FId='00001';
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNUMBER	FNAME	FMANAGERID	NAME_PATH
00001	1	Robert	<NULL>	>Robert
00002	2	John	00001	>Robert>John
00005	5	Lily	00002	>Robert>John>Lily

00003	3	Tom	00001	>Robert>Tom
00004	4	Jim	00003	>Robert>Tom>Jim
00006	6	Merry	00003	>Robert>Tom>Merry

SYS_CONNECT_BY_PATH()函数返回的是路径上的节点的连接字符串，所以如果修改遍历顺序的话，SYS_CONNECT_BY_PATH()函数的返回值也会发生变化。下面的SQL语句将遍历顺序修改为自底向上的方式：

```
SELECT FId,FNumber,FName,FManagerId,
sys_connect_by_path(FName,'>') AS Name_Path
FROM T_Person
CONNECT BY FId= PRIOR FManagerId
START WITH FId='00006';
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FID	FNUMBER	FNAME	FMANAGERID	NAME_PATH
00006	6	Merry	00003	>Merry
00003	3	Tom	00001	>Merry>Tom
00001	1	Robert	<NULL>	>Merry>Tom>Robert

可以看到NAME_PATH的值为父节点在子节点的后面。

11.9.3 MYSQLServer和DB2中递归查询

MYSQLServer和DB2不支持CONNECT BY子句，不过可以使用WITH子句来变通实现递归查询，其核心思想就是子查询的递归调用。

在前面章节介绍了可以使用WITH子句来为子查询定义一个别名，这样就可以通过这个别名来引用这个子查询了。如下：

```
WITH RPL AS
(
    SELECT FId, FNumber,FName,FManagerId
    FROM T_Person
    WHERE FManagerId IS NOT NULL
)
SELECT FId, FNumber,FName,FManagerId
FROM RPL
```

除了在外部查询中使用定义的子查询RPL，也可以在RPL内部使用RPL，这种“定义中使用定义”的方式就构成了递归查询。

下面的SQL语句使用嵌套定义的子查询RPL来实现递归查询：

```
WITH RPL (FId, FNumber,FName,FManagerId) AS
(
    SELECT ROOT.FId, ROOT.FNumber,ROOT.FName,ROOT.FManagerId
    FROM T_Person ROOT
    WHERE ROOT.FId='00001'
    UNION ALL
    SELECT CHILD.FId, CHILD.FNumber,CHILD.FName,CHILD.FManagerId
    FROM RPL PARENT, T_Person CHILD
    WHERE PARENT.FId = CHILD.FManagerId
)
SELECT DISTINCT FId, FNumber,FName,FManagerId
```

```
FROM RPL
ORDER BY FManagerId,FId, FNumber,FName
```

执行完毕我们就能在输出结果中看到下面的执行结果：

FId	FNumber	FName	FManagerId
00001	1	Robert	<NULL>
00002	2	John	00001
00003	3	Tom	00001
00005	5	Lily	00002
00004	4	Jim	00003
00006	6	Merry	00003

这个递归查询SQL语句的核心部分在子查询定义中，子查询中使用UNION ALL操作符将两个SELECT语句联合起来，第一个SELECT语句是递归查询的起始条件，而第二个SELECT语句将RPL本身和CHILD使用“PARENT.FId = CHILD.FManagerId”这个递归条件连接起来，第二个SELECT语句会递归调用RPL，从而根据“PARENT.FId = CHILD.FManagerId”条件列出T_Person表中的所有层次信息。