结构化查询语言SQL介绍和基本操作

> 本章将通过丰富的实例对 SQL 语言的基础进行详细介绍,MySQL,使得读者不但能够学习到标准 SQL 的使用,又能够学习到 MySQL 中一些扩展 SQL 的使用方法。该教案根据《MySQL必知必会》撰写。

了解SQL

什么是SQL

> SQL 简介

当面对一个陌生的数据库时,通常需要一种方式与它进行交互,以完成用户所需要的各种工作,这个时候,就要用到 SQL 语言了。

SQL 是 Structure Query Language(结构化查询语言)的缩写,它是使用关系模型的数据库应用语言,由 IBM 在 20 世纪 70 年代开发出来,作为 IBM 关系数据库原型 System R 的原型关系语言,实现了关系数据库中的信息检索。

20 世纪 80 年代初,美国国家标准局(ANSI)开始着手制定 SQL 标准,最早的 ANSI 标准于1986 年完成,就被叫作 SQL-86。标准的出台使 SQL 作为标准关系数据库语言的地位得到了加强。SQL 标准目前已几经修改更趋完善。

正是由于 SQL 语言的标准化,所以大多数关系型数据库系统都支持 SQL 语言,它已经发展成为多种平台进行交互操作的底层会话语言。

这里用了(My)SQL 这样的标题,目的是在介绍标准 SQL 的同时,也将一些 MySQL 在标准 SQL上的扩展一同介绍给大家。希望读者看完本节后,能够对标准 SQL 的基本语法和 MySQL 的部分扩展语法有所了解。

> SQL 分类

SQL 语句主要可以划分为以下 4 个类别。

- DDL(Data Definition Languages)语句:数据定义语言,这些语句定义了不同的数据段、数据库、表、列、索引等

数据库对象的定义。常用的语句关键字主要包括 create、drop、alter等。

- DML(Data Manipulation Language)语句:数据操纵语句,用于添加、删除、更新和查询数据库记录,并检查数据完整性,常用的语句关键字主要包括 insert、delete、udpate 和select 等。
- DCL(Data Control Language)语句:数据控制语句,用于控制不同数据段直接的许可和访问级别的语句。这些语句定义了数据库、表、字段、用户的访问权限和安全级别。主要的语句关键字包括 grant、revoke 等。
- DQL(Data Query Language)语句:数据查询语句，用于从一个或多个表中检索信息。

> MySQL中的语法格式

1. SQL语句是由简单的英语单词构成的。这些单词称为关键字,每个SQL语句都是由一个或多个关键字构成的。
2. 结束SQL语句 多条SQL语句必须以分号(;)分隔。MySQL如同多数DBMS一样,不需要在单条SQL语句后加分号。但特定的DBMS可能必须在单条SQL语句后加上分号。当然,如果愿意可以总是加上分号。事实上,即使不一定需要,但加上分号肯定没有坏处。如果你使用的是 mysql命令行,必须加上分号来结束 SQL 语句。
3. SQL语句和大小写 请注意,SQL语句不区分大小写,因此SELECT 与 select 是相同的。同样,写成 Select 也没有关系。许多SQL开发人员喜欢对所有SQL关键字使用大写,而对所有列和表名使用小写,这样做使代码更易于阅读和调试。
4. 使用空格 在处理SQL语句时,其中所有空格都被忽略。SQL语句可以在一行上给出,也可以分成许多行。多数SQL开发人员认为将SQL语句分成多行更容易阅读和调试。

> 学习样例表获取方法

学习样例mysql_scripts.zip存放在uplooking教室共享路径为http://classroom.example.com/content/MYSQL/04-othersmysql_scripts.zip 。

可以在连接互联网的情况下，访问http://www.forta.com/books/0672327120/

```
[root@mastera0 ~]# yum install -y wget vim net-tools unzip
[root@mastera0 ~]# wget http://classroom.example.com/content/MYSQL/04-othersmysql_scripts.zip
[root@mastera0 ~]# ls
anaconda-ks.cfg  mysql_scripts.zip
[root@mastera0 ~]# unzip mysql_scripts.zip
[root@mastera0 ~]# ls
anaconda-ks.cfg  create.sql  mysql_scripts.zip  populate.sql
[root@mastera0 ~]# mysql -uroot -puplooking
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 4
Server version: 5.5.44-MariaDB MariaDB Server

Copyright (c) 2000, 2015, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
MariaDB [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
4 rows in set (0.00 sec)

MariaDB [(none)]> exit
Bye
# 导入create.sql到test库，将会创建一些表
[root@mastera0 ~]# mysql -uroot -puplooking test < create.sql
[root@mastera0 ~]# mysql -uroot -puplooking -e "show tables from test";
+----------------+
| Tables_in_test |
+----------------+
| customers      |
| orderitems     |
| orders         |
| productnotes   |
| products       |
| vendors        |
+----------------+
# 导入populate.sql到test库，将会向表中新增数据
[root@mastera0 ~]# mysql -uroot -puplooking test < populate.sql
[root@mastera0 ~]# mysql -uroot -puplooking -e "select * from test.vendors";
+---------+----------------+-----------------+-------------+------------+----------+-------------
-+
| vend_id | vend_name      | vend_address    | vend_city   | vend_state | vend_zip | vend_country
|
+---------+----------------+-----------------+-------------+------------+----------+-------------
-+
|    1001 | Anvils R Us    | 123 Main Street | Southfield  | MI         | 48075    | USA
|
|    1002 | LT Supplies    | 500 Park Street | Anytown     | OH         | 44333    | USA
```

```
|
|    1003 | ACME           | 555 High Street | Los Angeles | CA         | 90046    | USA
|
|    1004 | Furball Inc.   | 1000 5th Avenue | New York    | NY         | 11111    | USA
|
|    1005 | Jet Set        | 42 Galaxy Road  | London      | NULL       | N16 6PS  | England
|
|    1006 | Jouets Et Ours | 1 Rue Amusement | Paris       | NULL       | 45678    | France
|
+---------+----------------+-----------------+-------------+------------+----------+-------------
-+

[root@mastera0 ~]# mysql -uroot -puplooking
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 15
Server version: 5.5.44-MariaDB MariaDB Server

Copyright (c) 2000, 2015, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
# 查询所有的数据库
MariaDB [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
4 rows in set (0.00 sec)
# 使用test库
MariaDB [(none)]> use test;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
# 查询test库中所有表
MariaDB [test]> show tables;
+----------------+
| Tables_in_test |
+----------------+
| customers      |
| orderitems     |
| orders         |
| productnotes   |
| products       |
| vendors        |
+----------------+
6 rows in set (0.00 sec)
# 查询test库中products表的结构，  describe是show columns from的别名
# DESCRIBE tbl_name

# SHOW [FULL] COLUMNS FROM tbl_name [FROM db_name] [like_or_where]
```

```
MariaDB [test]> desc products;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| prod_id    | char(10)     | NO   | PRI | NULL    |       |
| vend_id    | int(11)      | NO   | MUL | NULL    |       |
| prod_name  | char(255)    | NO   |     | NULL    |       |
| prod_price | decimal(8,2) | NO   |     | NULL    |       |
| prod_desc  | text         | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)

MariaDB [test]> show columns from test.products;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| prod_id    | char(10)     | NO   | PRI | NULL    |       |
| vend_id    | int(11)      | NO   | MUL | NULL    |       |
| prod_name  | char(255)    | NO   |     | NULL    |       |
| prod_price | decimal(8,2) | NO   |     | NULL    |       |
| prod_desc  | text         | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

> 本教案中所有章节中使用的数据库表都是关系表，关于每个表及关系的描述,如下所示：

样例表为一个想象的随身物品推销商使用的订单录入系统,这些随身物品可能是你喜欢的卡通人物需要的(是的,卡通人物,没人规定学习MySQL必须沉闷地学)。这些表用来完成以下几个任务:

- 管理供应商;
- 管理产品目录;
- 管理顾客列表;
- 录入顾客订单。

要完成这几个任务需要作为关系数据库设计成分的紧密联系的6个表。

vendors 表 存储销售产品的供应商。每个供应商在这个表中有一个记录,供应商ID(vend_id)列用来匹配产品和供应商。

vendors 表的列	说明
vend_id	唯一的供应商ID
vend_name	供应商名
vend_address	供应商的地址
vend_city	供应商的城市
vend_state	供应商的州
vend_zip	供应商的邮政编码
vend_country	供应商的国家

products 表 包含产品目录,每行一个产品。每个产品有唯一的ID(prod_id 列),通过 vend_id (供应商的唯一ID)关联到它的供应商。

products 表的列	说明
prod_id	唯一的产品ID
vend_id	产品供应商ID(关联到vendors表中的vend_id)
prod_name	产品名

|prod_price| 产品价格
|prod_desc |产品描述|

customers 表 存储所有顾客的信息。每个顾客有唯一的ID(cust_id列)。

customers 表的列	说明
cust_id	唯一的顾客ID
cust_name	顾客名
cust_address	顾客的地址
cust_city	顾客的城市
cust_state	顾客的州
cust_zip	顾客的邮政编码
cust_country	顾客的国家
cust_contact	顾客的联系名
cust_email	顾客的联系email地址

orderitems 表 存储每个订单中的实际物品,每个订单的每个物品占一行。对 orders 中的每一行, orderitems 中有一行或多行。每个订单物品由订单号加订单物品(第一个物品、第二个物品等)唯一标识。订单物品通过 order_num 列(关联到 orders 中订单的唯一ID)与它们相应的订单相关联。此外,每个订单项包含订单物品的产品ID(它关联物品到 products 表)。

orderitems 表的列	说明
order_num	订单号(关联到orders表的order_num)
order_item	订单物品号(在某个订单中的顺序)
prod_id	产品ID(关联到products表的prod_id)
quantity	物品数量
item_price	物品价格

productnotes 表存储与特定产品有关的注释。并非所有产品都有相
关的注释,而有的产品可能有许多相关的注释。

productnotes 表的列	说明
note_id	唯一注释ID
prod_id	产品ID(对应于products表中的prod_id)
note_date	增加注释的日期
note_text	注释文本

DQL语言

DQL(Data Query Language)语句:数据查询语句，用于从一个或多个表中检索信息。本章介绍如何使用 SELECT 语句从表中检索一个或多个数据列。

检索数据

- 数据插入到数据库中后,就可以用 SELECT 命令进行各种各样的查询,使得输出的结果符合我们的要求。由于 SELECT 的语法很复杂,所有这里只介绍最基本的语法
- 大概,最经常使用的SQL语句就是 SELECT 语句了。它的用途是从一个或多个表中检索信息。
- 为了使用 SELECT 检索表数据,必须至少给出两条信息——想选择什么,以及从什么地方选择。

我们将从简单的SQL SELECT 语句开始介绍,此语句如下所示:

> 检索单个列
>
> > 利用 SELECT 语句从 products 表中检索一个名为prod_name 的列。所需的列名在 SELECT 关键字之后给出, FROM关键字指出从其中检索数据的表名。

```
MariaDB [test]> select prod_name from products;
+----------------+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| Detonator      |
| Bird seed      |
| Carrots        |
| Fuses          |
| JetPack 1000   |
| JetPack 2000   |
| Oil can        |
| Safe           |
| Sling          |
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
14 rows in set (0.00 sec)
```

未排序数据 如果读者自己试验这个查询,可能会发现显示输出的数据顺序与这里的不同。出现这种情况很正常。如果没有明确排序查询结果(下一章介绍),则返回的数据的顺序没有特殊意义。返回数据的顺序可能是数据被添加到表中的顺序,也可能不是。只要返回相同数目的行,就是正常的。

使用空格 在处理SQL语句时,其中所有空格都被忽略。SQL语句可以在一行上给出,也可以分成许多行。多数SQL开发人员认为将SQL语句分成多行更容易阅读和调试。

> 检索多个列
>
> > 使用 SELECT 语句从表 products中选择数据,指定3个列名prod_id,prod_name,prod_price,列名之间用逗号分隔。

```
MariaDB [test]> select prod_id,prod_name,prod_price from products;
+---------+----------------+------------+
| prod_id | prod_name      | prod_price |
+---------+----------------+------------+
| ANV01   | .5 ton anvil   |       5.99 |
| ANV02   | 1 ton anvil    |       9.99 |
| ANV03   | 2 ton anvil    |      14.99 |
| DTNTR   | Detonator      |      13.00 |
| FB      | Bird seed      |      10.00 |
| FC      | Carrots        |       2.50 |
| FU1     | Fuses          |       3.42 |
| JP1000  | JetPack 1000   |      35.00 |
| JP2000  | JetPack 2000   |      55.00 |
| OL1     | Oil can        |       8.99 |
| SAFE    | Safe           |      50.00 |
| SLING   | Sling          |       4.49 |
| TNT1    | TNT (1 stick)  |       2.50 |
| TNT2    | TNT (5 sticks) |      10.00 |
+---------+----------------+------------+
14 rows in set (0.00 sec)
```

当心逗号 在选择多个列时,一定要在列名之间加上逗号,但最后一个列名后不加。如果在最后一个列名后加了逗号,将出现错误。

数据表示 从上述输出可以看到,SQL语句一般返回原始的、无格式的数据。数据的格式化是一个表示问题,而不是一个检索问题。因此,表示(对齐和显示上面的价格值,用货币符号和逗号表示其金额)一般在显示该数据的应用程序中规定。一般很少使用实际检索出的原始数据(没有应用程序提供的格式)。

> 检索所有列
>
> > 除了指定所需的列外(如上所述,一个或多个列), SELECT 语句还可以检索所有的列而不必逐个列出它们。这可以通过在实际列名的位置使用星号(*)通配符来达到,使用 SELECT 语句从表 products中选择所有数据。

```
MariaDB [test]> select * from products;
+---------+---------+----------------+------------+-----------------------------------------
------------------+
| prod_id | vend_id | prod_name      | prod_price | prod_desc
                 |
+---------+---------+----------------+------------+-----------------------------------------
------------------+
| ANV01   |    1001 | .5 ton anvil   |       5.99 | .5 ton anvil, black, complete with handy hook
                 |
| ANV02   |    1001 | 1 ton anvil    |       9.99 | 1 ton anvil, black, complete with handy hook
and carrying case |
| ANV03   |    1001 | 2 ton anvil    |      14.99 | 2 ton anvil, black, complete with handy hook
and carrying case |
| DTNTR   |    1003 | Detonator      |      13.00 | Detonator (plunger powered), fuses not
included               |
| FB      |    1003 | Bird seed      |      10.00 | Large bag (suitable for road runners)
                 |
| FC      |    1003 | Carrots        |       2.50 | Carrots (rabbit hunting season only)
                 |
| FU1     |    1002 | Fuses          |       3.42 | 1 dozen, extra long
                 |
| JP1000  |    1005 | JetPack 1000   |      35.00 | JetPack 1000, intended for single use
                 |
| JP2000  |    1005 | JetPack 2000   |      55.00 | JetPack 2000, multi-use
                 |
| OL1     |    1002 | Oil can        |       8.99 | Oil can, red
                 |
| SAFE    |    1003 | Safe           |      50.00 | Safe with combination lock
                 |
| SLING   |    1003 | Sling          |       4.49 | Sling, one size fits all
                 |
| TNT1    |    1003 | TNT (1 stick)  |       2.50 | TNT, red, single stick
                 |
| TNT2    |    1003 | TNT (5 sticks) |      10.00 | TNT, red, pack of 10 sticks
                 |
+---------+---------+----------------+------------+-----------------------------------------
------------------+
14 rows in set (0.00 sec)
```

使用通配符 一般,除非你确实需要表中的每个列,否则最好别使用 * 通配符。虽然使用通配符可能会使你自己省事,不用明确列出所需列,但检索不需要的列通常会降低检索和应用程序的性能。

检索未知列 使用通配符有一个大优点。由于不明确指定列名(因为星号检索每个列),所以能检索出名字未知的列。

> 检索不同的行
>
> > 正如所见, SELECT 返回所有匹配的行。但是,如果你不想要每个值每次都出现,怎么办?例如,假如你想得出 products 表中产品的所有供应商ID:

```
MariaDB [test]> select vend_id from products;
+---------+
| vend_id |
+---------+
|    1001 |
|    1001 |
|    1001 |
|    1002 |
|    1002 |
|    1003 |
|    1003 |
|    1003 |
|    1003 |
|    1003 |
|    1003 |
|    1003 |
|    1005 |
|    1005 |
+---------+
14 rows in set (0.00 sec)
```

SELECT 语句返回14行(即使表中只有4个供应商),因为 products 表中列出了14个产品。那么,如何检索出有不同值的列表呢?

DISTINCT 关键字 顾名思义,此关键字指示MySQL只返回不同的值。

`SELECT DISTINCT vend_id` 告诉MySQL只返回不同(唯一)的`vend_id`行,因此只返回4行,如下面的输出所示。如果使用`DISTINCT` 关键字,它必须直接放在列名的前面。

```
MariaDB [test]> select distinct vend_id from products;
+---------+
| vend_id |
+---------+
|    1001 |
|    1002 |
|    1003 |
|    1005 |
+---------+
4 rows in set (0.01 sec)
```

不能部分使用 **DISTINCT** `DISTINCT` 关键字应用于所有列而不仅是前置它的列。如果给出 `SELECT DISTINCT vend_id,prod_price` ,除非指定的两个列都不同,否则所有行都将被检索出来。

> 限制结果
>
> > SELECT 语句返回所有匹配的行,它们可能是指定表中的每个行。为了返回第一行或前几行,可使用 `LIMIT` 子句。下面举一个例子:用 SELECT 语句检索单个列返回不多于5行。

```
MariaDB [test]> select prod_name from products;
+----------------+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| Detonator      |
| Bird seed      |
| Carrots        |
| Fuses          |
| JetPack 1000   |
| JetPack 2000   |
| Oil can        |
| Safe           |
| Sling          |
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
14 rows in set (0.00 sec)

MariaDB [test]> select prod_name from products limit 5;
+--------------+
| prod_name    |
+--------------+
| .5 ton anvil |
| 1 ton anvil  |
| 2 ton anvil  |
| Detonator    |
| Bird seed    |
+--------------+
5 rows in set (0.00 sec)
```

LIMIT 5, 5 指示MySQL返回从行5开始的5行。第一个数为开始位置,第二个数为要检索的行数。

```
MariaDB [test]> select prod_name from products limit 5,5;
+--------------+
| prod_name    |
+--------------+
| Carrots      |
| Fuses        |
| JetPack 1000 |
| JetPack 2000 |
| Oil can      |
+--------------+
5 rows in set (0.00 sec)
```

所以,带一个值的 LIMIT 总是从第一行开始,给出的数为返回的行数。带两个值的 LIMIT 可以指定从行号为第一个值的位置开始。

行 **0** 检索出来的第一行为行0而不是行1。因此,`LIMIT 1, 1`将检索出第二行而不是第一行

```
MariaDB [test]> select prod_name from products limit 1,1;
+-------------+
| prod_name   |
+-------------+
| 1 ton anvil |
+-------------+
1 row in set (0.00 sec)

MariaDB [test]> select prod_name from products limit 0,1;
+--------------+
| prod_name    |
+--------------+
| .5 ton anvil |
+--------------+
1 row in set (0.00 sec)

MariaDB [test]> select prod_name from products limit 1;
+--------------+
| prod_name    |
+--------------+
| .5 ton anvil |
+--------------+
1 row in set (0.00 sec)
```

在行数不够时 LIMIT 中指定要检索的行数为检索的最大行数。如果没有足够的行(例如,给出 `LIMIT 10, 5`,但只有13行),MySQL将只返回它能返回的那么多行。

```
MariaDB [test]> select prod_name from products limit 5,10;
+----------------+
| prod_name      |
+----------------+
| Carrots        |
| Fuses          |
| JetPack 1000   |
| JetPack 2000   |
| Oil can        |
| Safe           |
| Sling          |
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
9 rows in set (0.00 sec)
```

MySQL 5的 LIMIT 语法 `LIMIT 3, 4` 的含义是从行4开始的3行还是从行3开始的4行?如前所述,它的意思是从行3开始的4行,这容易把人搞糊涂。由于这个原因, MySQL 5支持 LIMIT 的另一种替代语法。 `LIMIT4 OFFSET 3` 意为从行3开始取4行,就像 `LIMIT 3, 4` 一样。

```
MariaDB [test]> select prod_name from products limit 3,4;
+-----------+
| prod_name |
+-----------+
| Detonator |
| Bird seed |
| Carrots   |
| Fuses     |
+-----------+
4 rows in set (0.00 sec)

MariaDB [test]> select prod_name from products limit 4 offset 3;
+-----------+
| prod_name |
+-----------+
| Detonator |
| Bird seed |
| Carrots   |
| Fuses     |
+-----------+
4 rows in set (0.00 sec)
```

> 使用完全限定的表名
>
> > 迄今为止使用的SQL例子只通过列名引用列。也可能会使用完全限定的名字来引用列(同时使用表名和列字)
> > 请看以下例子:通过完全限定的表名和列名查询products表的prod_name列的所有值

```
MariaDB [test]> select products.prod_name from products;
+----------------+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| Detonator      |
| Bird seed      |
| Carrots        |
| Fuses          |
| JetPack 1000   |
| JetPack 2000   |
| Oil can        |
| Safe           |
| Sling          |
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
14 rows in set (0.00 sec)
```

这条SQL语句在功能上等于本章最开始使用的那一条语句,但这里指定了一个完全限定的列名。

表名也可以是完全限定的,如下所示:

```
MariaDB [test]> select products.prod_name from test.products;
+----------------+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| Detonator      |
| Bird seed      |
| Carrots        |
| Fuses          |
| JetPack 1000   |
| JetPack 2000   |
| Oil can        |
| Safe           |
| Sling          |
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
14 rows in set (0.00 sec)
```

这条语句在功能上也等于刚使用的那条语句。

正如以后章节所介绍的那样,有一些情形需要完全限定名。现在,需要注意这个语法,以便在遇到时知道它的作用。

> 小结
>
> > 本章学习了如何使用SQL的 SELECT 语句来检索单个表列、多个表列以及所有表列。下一章将讲授如何排序检索出来的数据。

排序检索数据

本章将讲授如何使用 SELECT 语句的 ORDER BY 子句,根据需要排序检索出的数据。

> 排序数据
>
> > 利用 SELECT 语句从 products 表中检索一个名为prod_name 的列。对 prod_name 列以字母顺序排序。

```
MariaDB [test]> select prod_name from products order by prod_name;
+----------------+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| Bird seed      |
| Carrots        |
| Detonator      |
| Fuses          |
| JetPack 1000   |
| JetPack 2000   |
| Oil can        |
| Safe           |
| Sling          |
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
14 rows in set (0.00 sec)
```

如果不排序 数据一般将以它在底层表中出现的顺序显示。这可以是数据最初添加到表中的顺序。但是,如果数据后来进行过更新或删除,则此顺序将会受到MySQL重用回收存储空间的影响。因此,如果不明确控制的话,不能(也不应该)依赖该排序顺序。关系数据库设计理论认为,如果不明确规定排序顺序,则不应该假定检索出的数据的顺序有意义。

子句**(clause)** SQL语句由子句构成,有些子句是必需的,而有的是可选的。一个子句通常由一个关键字和所提供的数据组成。子句的例子有 SELECT 语句的 FROM 子句,我们在前一章看到过这个子句。

通过非选择列进行排序 通常, ORDER BY 子句中使用的列将是为显示所选择的列。但是,实际上并不一定要这样,用非检索的列排序数据是完全合法的。

```
MariaDB [test]> select prod_name from products order by vend_id;
+----------------+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| Fuses          |
| Oil can        |
| TNT (1 stick)  |
| Sling          |
| Safe           |
| TNT (5 sticks) |
| Carrots        |
| Bird seed      |
| Detonator      |
| JetPack 2000   |
| JetPack 1000   |
+----------------+

14 rows in set (0.00 sec)
```

经常需要按不止一个列进行数据排序。例如,如果要显示雇员清单,可能希望按姓和名排序(首先按姓排序,然后在每个姓中再按名排序)。如果多个雇员具有相同的姓,这样做很有用。

为了按多个列排序,只要指定列名,列名之间用逗号分开即可(就像选择多个列时所做的那样)。

> 按多个列排序
>
> > 利用SELECT语句从products表中检索3个列(prod_id,prod_name,prod_price)，并按其中两个列对结果进行排序——首先按价格prod_price,然后再按名称prod_name排序。

```
MariaDB [test]> select prod_id,prod_name,prod_price from products order by prod_price,prod_name;
+---------+----------------+------------+
| prod_id | prod_name      | prod_price |
+---------+----------------+------------+
| FC      | Carrots        |       2.50 |
| TNT1    | TNT (1 stick)  |       2.50 |
| FU1     | Fuses          |       3.42 |
| SLING   | Sling          |       4.49 |
| ANV01   | .5 ton anvil   |       5.99 |
| OL1     | Oil can        |       8.99 |
| ANV02   | 1 ton anvil    |       9.99 |
| FB      | Bird seed      |      10.00 |
| TNT2    | TNT (5 sticks) |      10.00 |
| DTNTR   | Detonator      |      13.00 |
| ANV03   | 2 ton anvil    |      14.99 |
| JP1000  | JetPack 1000   |      35.00 |
| SAFE    | Safe           |      50.00 |
| JP2000  | JetPack 2000   |      55.00 |
+---------+----------------+------------+
14 rows in set (0.00 sec)
```

重要的是理解在按多个列排序时,排序完全按所规定的顺序进行。换句话说,对于上述例子中的输出,仅在多个行具有相同的 prod_price值时才对产品按 prod_name 进行排序。如果 prod_price 列中所有的值都是唯一的,则不会按 prod_name 排序。

数据排序不限于升序排序(从 A 到 Z)。这只是默认的排序顺序,还可以使用 ORDER BY 子句以降序(从 Z 到 A)顺序排序。为了进行降序排序,必须指定 DESC 关键字。

> 指定排序方向
>
> > 利用SELECT语句从products表中检索3个列(prod_id,prod_name,prod_price)，按价格prod_price以降序排序产品(最贵的排在最前面)

```
MariaDB [test]> select prod_id,prod_name,prod_price from products order by prod_price desc;
+---------+----------------+------------+
| prod_id | prod_name      | prod_price |
+---------+----------------+------------+
| JP2000  | JetPack 2000   |      55.00 |
| SAFE    | Safe           |      50.00 |
| JP1000  | JetPack 1000   |      35.00 |
| ANV03   | 2 ton anvil    |      14.99 |
| DTNTR   | Detonator      |      13.00 |
| TNT2    | TNT (5 sticks) |      10.00 |
| FB      | Bird seed      |      10.00 |
| ANV02   | 1 ton anvil    |       9.99 |
| OL1     | Oil can        |       8.99 |
| ANV01   | .5 ton anvil   |       5.99 |
| SLING   | Sling          |       4.49 |
| FU1     | Fuses          |       3.42 |
| FC      | Carrots        |       2.50 |
| TNT1    | TNT (1 stick)  |       2.50 |
+---------+----------------+------------+
14 rows in set (0.00 sec)
```

但是,如果打算用多个列排序怎么办?

> 利用SELECT语句从products表中检索3个列(prod_id,prod_name,prod_price)，按价格prod_price以降序排序产品(最贵的排在最前面)，然后再对产品名排序prod_name

```
MariaDB [test]> select prod_id,prod_name,prod_price from products order by prod_price
desc,prod_name;
+---------+----------------+------------+
| prod_id | prod_name      | prod_price |
+---------+----------------+------------+
| JP2000  | JetPack 2000   |      55.00 |
| SAFE    | Safe           |      50.00 |
| JP1000  | JetPack 1000   |      35.00 |
| ANV03   | 2 ton anvil    |      14.99 |
| DTNTR   | Detonator      |      13.00 |
| FB      | Bird seed      |      10.00 |
| TNT2    | TNT (5 sticks) |      10.00 |
| ANV02   | 1 ton anvil    |       9.99 |
| OL1     | Oil can        |       8.99 |
| ANV01   | .5 ton anvil   |       5.99 |
| SLING   | Sling          |       4.49 |
| FU1     | Fuses          |       3.42 |
| FC      | Carrots        |       2.50 |
| TNT1    | TNT (1 stick)  |       2.50 |
+---------+----------------+------------+
14 rows in set (0.00 sec)
```

在多个列上降序排序 如果想在多个列上进行降序排序,必须对每个列指定 DESC 关键字。与 DESC 相反的关键字是 ASC (ASCENDING),在升序排序时可以指定它。但实际上, ASC 没有多大用处,因为升序是默认的(如果既不指定 ASC 也不指定 DESC ,则假定为 ASC)。

区分大小写和排序顺序 在对文本性的数据进行排序时,A与a相同吗?a位于B之前还是位于Z之后?这些问题不是理论问题,其答案取决于数据库如何设置。在字典(dictionary)排序顺序中, A被视为与a相同,这是MySQL(和大多数数据库管理系统)的默认行为。但是,许多数据库管理员能够在需要时改变这种行为(如果你的数据库包含大量外语字符,可能必须这样做)。这里,关键的问题是,如果确实需要改变这种排序顺序,用简单的 ORDER BY 子句做不到。你必须请求数据库管理员的帮助。

> 利用SELECT语句从products表中检索出最昂贵物品价格prod_price

```
MariaDB [test]> select prod_price from products order by prod_price desc limit 1;
+------------+
| prod_price |
+------------+
|      55.00 |
+------------+
1 row in set (0.00 sec)

MariaDB [test]> select prod_price from products order by prod_price desc limit 0,1;
+------------+
| prod_price |
+------------+
|      55.00 |
+------------+
1 row in set (0.00 sec)

MariaDB [test]> select prod_price from products order by prod_price desc limit 1 offset 0;
+------------+
| prod_price |
+------------+
|      55.00 |
+------------+
1 row in set (0.00 sec)
```

ORDER BY 子句的位置 在给出 ORDER BY 子句时,应该保证它位于 FROM 子句之后。如果使用 LIMIT ,它必须位于 ORDER BY之后。使用子句的次序不对将产生错误消息。

```
MariaDB [test]> select prod_price from products limit 1 order by prod_price desc;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
your MariaDB server version for the right syntax to use near 'order by prod_price desc' at line 1
```

> 小结
>
> > 本章学习了如何用 SELECT 语句的 ORDER BY 子句对检索出的数据进行排序。这个子句必须位于 FROM 子句之后,必须是 SELECT 语句中的最后一条子句。可根据需要,利用它在一个或多个列上对数据进行排序。

过滤数据

本章将讲授如何使用 SELECT 语句的 WHERE 子句指定搜索条件。

> 使用 WHERE 子句

数据库表一般包含大量的数据,很少需要检索表中所有行。通常只会根据特定操作或报告的需要提取表数据的子集。只检索所需数据需要指定搜索条件(search criteria),搜索条件也称为过滤条件(filtercondition)。

在 SELECT 语句中,数据根据 WHERE 子句中指定的搜索条件进行过滤。WHERE 子句在表名(FROM 子句)之后给出。

> 从 products 表中检索两个列(prod_name,prod_price),但不返回所有行,只返回 prod_price 值为 2.50 的行。

```
MariaDB [test]> select prod_name,prod_price  from products where prod_price=2.50;
+---------------+------------+
| prod_name     | prod_price |
+---------------+------------+
| Carrots       |       2.50 |
| TNT (1 stick) |       2.50 |
+---------------+------------+
2 rows in set (0.00 sec)
```

SQL过滤与应用过滤 数据也可以在应用层过滤。为此目的,SQL的 SELECT 语句为客户机应用检索出超过实际所需的数据,然后客户机代码对返回数据进行循环,以提取出需要的行。

通常,这种实现并不令人满意。因此,对数据库进行了优化,以便快速有效地对数据进行过滤。让客户机应用(或开发语言)处理数据库的工作将会极大地影响应用的性能,并且使所创建的应用完全不具备可伸缩性。此外,如果在客户机上过滤数据,服务器不得不通过网络发送多余的数据,这将导致网络带宽的浪费。

WHERE 子句的位置 在同时使用 ORDER BY 和 WHERE 子句时,应该让 ORDER BY 位于 WHERE 之后,否则将会产生错误。

```
MariaDB [test]> select prod_name,prod_price  from products where prod_name like "T%" order by
prod_price desc;
+----------------+------------+
| prod_name      | prod_price |
+----------------+------------+
| TNT (5 sticks) |      10.00 |
| TNT (1 stick)  |       2.50 |
+----------------+------------+
2 rows in set (0.01 sec)

MariaDB [test]> select prod_name,prod_price  from products order by prod_price desc where
prod_name like "T%";
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
your MariaDB server version for the right syntax to use near 'where prod_name like "T%"' at line
1
```

> WHERE 子句操作符

我们在关于相等的测试时看到了第一个 WHERE 子句,它确定一个列是否包含特定的值。MySQL支持下表列出的所有条件操作符。

操作符	说明
=	等于
<>	不等于
!=	不等于
<	小于
<=	小于等于
>	大于
>=	大于等于
between	在指定的两个值之间

1.检查单个值

> 从 products 表中检索两个列(prod_name,prod_price),但不返回所有行,只返回 prod_name 的值为 Fuses 的一行。

```
MariaDB [test]> select prod_name,prod_price from products where prod_name='fuses';
+-----------+------------+
| prod_name | prod_price |
+-----------+------------+
| Fuses     |       3.42 |
+-----------+------------+
1 row in set (0.00 sec)
```

检查 `WHERE prod_name='fuses'` 语句,它返回 prod_name 的值为 Fuses 的一行。MySQL在执行匹配时默认不区分大小写,所以 fuses 与 Fuses 匹配。

> 列出价格小于10美元的所有产品

```
MariaDB [test]> select prod_name,prod_price from products where prod_price < 10;
+---------------+------------+
| prod_name     | prod_price |
+---------------+------------+
| .5 ton anvil  |       5.99 |
| 1 ton anvil   |       9.99 |
| Carrots       |       2.50 |
| Fuses         |       3.42 |
| Oil can       |       8.99 |
| Sling         |       4.49 |
| TNT (1 stick) |       2.50 |
+---------------+------------+
7 rows in set (0.00 sec)
```

> 检索价格小于等于10美元的所有产品

```
MariaDB [test]> select prod_name,prod_price from products where prod_price <= 10;
+----------------+------------+
| prod_name      | prod_price |
+----------------+------------+
| .5 ton anvil   |       5.99 |
| 1 ton anvil    |       9.99 |
| Bird seed      |      10.00 |
| Carrots        |       2.50 |
| Fuses          |       3.42 |
| Oil can        |       8.99 |
| Sling          |       4.49 |
| TNT (1 stick)  |       2.50 |
| TNT (5 sticks) |      10.00 |
+----------------+------------+
9 rows in set (0.00 sec)
```

> 实践思考

```
 MariaDB [test]> select prod_name,prod_price from products where prod_name <= "b";
+--------------+------------+
| prod_name    | prod_price |
+--------------+------------+
| .5 ton anvil |       5.99 |
| 1 ton anvil  |       9.99 |
| 2 ton anvil  |      14.99 |
+--------------+------------+
3 rows in set (0.00 sec)

MariaDB [test]> select prod_name,prod_price from products where prod_name <= b;
ERROR 1054 (42S22): Unknown column 'b' in 'where clause'
MariaDB [test]> select prod_name,prod_price from products where prod_name < "F";
+--------------+------------+
| prod_name    | prod_price |
+--------------+------------+
| .5 ton anvil |       5.99 |
| 1 ton anvil  |       9.99 |
| 2 ton anvil  |      14.99 |
| Detonator    |      13.00 |
| Bird seed    |      10.00 |
| Carrots      |       2.50 |
+--------------+------------+
6 rows in set (0.00 sec)

MariaDB [test]> select prod_name,prod_price from products where prod_name < "f";
+--------------+------------+
| prod_name    | prod_price |
+--------------+------------+
| .5 ton anvil |       5.99 |
| 1 ton anvil  |       9.99 |
| 2 ton anvil  |      14.99 |
| Detonator    |      13.00 |
| Bird seed    |      10.00 |
| Carrots      |       2.50 |
+--------------+------------+
6 rows in set (0.00 sec)
```

何时使用引号 如果仔细观察上述 WHERE 子句中使用的条件,会看到有的值括在单引号内(如前面使用的 'fuses'),而有的值未括起来。单引号用来限定字符串。如果将值与串类型的列进行比较,则需要限定引号。用来与数值列进行比较的值不用引号。

2.不匹配检查

> 不是由供应商 1003 制造的所有产品

```
MariaDB [test]> select vend_id,prod_name from products where vend_id <> 1003;
+---------+--------------+
| vend_id | prod_name    |
+---------+--------------+
|    1001 | .5 ton anvil |
|    1001 | 1 ton anvil  |
|    1001 | 2 ton anvil  |
|    1002 | Fuses        |
|    1005 | JetPack 1000 |
|    1005 | JetPack 2000 |
|    1002 | Oil can      |
+---------+--------------+
7 rows in set (0.00 sec)

MariaDB [test]> select vend_id,prod_name from products where vend_id != 1003;
+---------+--------------+
| vend_id | prod_name    |
+---------+--------------+
|    1001 | .5 ton anvil |
|    1001 | 1 ton anvil  |
|    1001 | 2 ton anvil  |
|    1002 | Fuses        |
|    1005 | JetPack 1000 |
|    1005 | JetPack 2000 |
|    1002 | Oil can      |
+---------+--------------+
7 rows in set (0.00 sec)
```

3.范围值检查

为了检查某个范围的值,可使用 BETWEEN 操作符。其语法与其他 WHERE 子句的操作符稍有不同,因为它需要两个值,即范围的开始值和结束值。

例如, BETWEEN 操作符可用来检索价格在5美元和10美元之间或日期在指定的开始日期和结束日期之间的所有产品。

> 检索价格在5美元和10美元之间的所有产品

```
MariaDB [test]> select prod_name,prod_price from products where prod_price between 5 and 10;
+----------------+------------+
| prod_name      | prod_price |
+----------------+------------+
| .5 ton anvil   |       5.99 |
| 1 ton anvil    |       9.99 |
| Bird seed      |      10.00 |
| Oil can        |       8.99 |
| TNT (5 sticks) |      10.00 |
+----------------+------------+
5 rows in set (0.00 sec)

MariaDB [test]> select prod_name,prod_price from products where prod_price between 5 and 10 order
by prod_price;
+----------------+------------+
| prod_name      | prod_price |
+----------------+------------+
| .5 ton anvil   |       5.99 |
| Oil can        |       8.99 |
| 1 ton anvil    |       9.99 |
| Bird seed      |      10.00 |
| TNT (5 sticks) |      10.00 |
+----------------+------------+
5 rows in set (0.00 sec)
```

4.空值检查

在创建表时,表设计人员可以指定其中的列是否可以不包含值。在一个列不包含值时,称其为包含空值 NULL 。

NULL 无值**(no value)** 它与字段包含 0 、空字符串或仅仅包含空格不同。

> 检查products表中prod_price列具有 NULL 值的行

```
MariaDB [test]> select prod_name,prod_price from products where prod_price is null;
Empty set (0.00 sec)
```

> 检查customers表中cust_email列具有 NULL 值的行

```
MariaDB [test]> select cust_name,cust_email from customers where cust_email is null;
+-------------+------------+
| cust_name   | cust_email |
+-------------+------------+
| Mouse House | NULL       |
| E Fudd      | NULL       |
+-------------+------------+
2 rows in set (0.00 sec)
```

> 检查customers表中cust_email列不具有 NULL 值的行

```
MariaDB [test]> select cust_name,cust_email from customers where cust_email is not null;
+----------------+---------------------+
| cust_name      | cust_email          |
+----------------+---------------------+
| Coyote Inc.    | ylee@coyote.com     |
| Wascals        | rabbit@wascally.com |
| Yosemite Place | sam@yosemite.com    |
+----------------+---------------------+
3 rows in set (0.00 sec)
```

> 小结
>
> > 本章介绍了如何用 SELECT 语句的 WHERE 子句过滤返回的数据。我们学习了如何对相等、不相等、大于、小于、值的范围以及 NULL 值等进行测试。

数据过滤

本章讲授如何组合 WHERE 子句以建立功能更强的更高级的搜索条件。我们还将学习如何使用 NOT 和 IN 操作符。

前面章节中介绍的所有 WHERE 子句在过滤数据时使用的都是单一的条件。为了进行更强的过滤控制,MySQL允许给出多个 WHERE 子句。这些子句可以两种方式使用:以 AND 子句的方式或 OR 子句的方式使用。

操作符**(operator)** 用来联结或改变 WHERE 子句中的子句的关键字。也称为逻辑操作符(logical operator)。

> 组合 WHERE 子句

1. AND 操作符

`AND` 用在 WHERE 子句中的关键字,用来指示检索满足所有给定条件的行。

> > 检索products表由供应商 1003 制造且价格小于等于10美元的所有产品的名称和价格。

```
MariaDB [test]> select prod_name,prod_price from products where vend_id=1003 and prod_price <=
10;
+----------------+------------+
| prod_name      | prod_price |
+----------------+------------+
| Bird seed      |      10.00 |
| Carrots        |       2.50 |
| Sling          |       4.49 |
| TNT (1 stick)  |       2.50 |
| TNT (5 sticks) |      10.00 |
+----------------+------------+
5 rows in set (0.00 sec)
```

1. OR 操作符

`OR` 操作符与 AND 操作符不同,它指示MySQL检索匹配任一条件的行。

> > 检索products表由供应商 1002或1003 制造的所有产品的名称和价格。

```
MariaDB [test]> select prod_name,prod_price from products where vend_id=1002 or vend_id=1003;
+----------------+------------+
| prod_name      | prod_price |
+----------------+------------+
| Detonator      |      13.00 |
| Bird seed      |      10.00 |
| Carrots        |       2.50 |
| Fuses          |       3.42 |
| Oil can        |       8.99 |
| Safe           |      50.00 |
| Sling          |       4.49 |
| TNT (1 stick)  |       2.50 |
| TNT (5 sticks) |      10.00 |
+----------------+------------+
9 rows in set (0.00 sec)
```

1. 计算次序

WHERE 可包含任意数目的 AND 和 OR 操作符。允许两者结合以进行复杂和高级的过滤。

> 检索products表，列出价格为10美元(含)以上且由 1002 或 1003 制造的所有产品。

```
MariaDB [test]> select prod_name,prod_price,vend_id from products where vend_id=1002 or
vend_id=1003 and prod_price >=10;
+----------------+------------+---------+
| prod_name      | prod_price | vend_id |
+----------------+------------+---------+
| Detonator      |      13.00 |    1003 |
| Bird seed      |      10.00 |    1003 |
| Fuses          |       3.42 |    1002 |
| Oil can        |       8.99 |    1002 |
| Safe           |      50.00 |    1003 |
| TNT (5 sticks) |      10.00 |    1003 |
+----------------+------------+---------+
6 rows in set (0.00 sec)
```

请看上面的结果。返回的行中有两行价格小于10美元,显然,返回的行未按预期的进行过滤。为什么会这样呢?原因在于计算的次序。SQL(像多数语言一样)在处理 OR 操作符前,优先处理 AND 操作符。当SQL看到上述 WHERE 子句时,它理解为由供应商 1003 制造的任何价格为10美元(含)以上的产品,或者由供应商 1002 制造的任何产品,而不管其价格如何。换句话说,由于 AND 在计算次序中优先级更高,操作符被错误地组合了。

此问题的解决方法是使用圆括号明确地分组相应的操作符。请看下面的 SELECT 语句及输出:

```
MariaDB [test]> select prod_name,prod_price,vend_id from products where (vend_id=1002 or
vend_id=1003) and prod_price >=10;
+----------------+------------+---------+
| prod_name      | prod_price | vend_id |
+----------------+------------+---------+
| Detonator      |      13.00 |    1003 |
| Bird seed      |      10.00 |    1003 |
| Safe           |      50.00 |    1003 |
| TNT (5 sticks) |      10.00 |    1003 |
+----------------+------------+---------+
4 rows in set (0.00 sec)
```

其他方法：

```
MariaDB [test]> select prod_name,prod_price,vend_id from products where prod_price >= 10 and
vend_id=1002 or prod_price >=10 and vend_id=1003 ;
+----------------+------------+---------+
| prod_name      | prod_price | vend_id |
+----------------+------------+---------+
| Detonator      |      13.00 |    1003 |
| Bird seed      |      10.00 |    1003 |
| Safe           |      50.00 |    1003 |
| TNT (5 sticks) |      10.00 |    1003 |
+----------------+------------+---------+
4 rows in set (0.00 sec)
```

在**WHERE**子句中使用圆括号 任何时候使用具有 AND 和 OR 操作符的 WHERE 子句,都应该使用圆括号明确地分组操作符。不要过分依赖默认计算次序,即使它确实是你想要的东西也是如此。使用圆括号没有什么坏处,它能消除歧义。

> IN 操作符

圆括号在 WHERE 子句中还有另外一种用法。 IN 操作符用来指定条件范围,范围中的每个条件都可以进行匹配。 IN 取合法值的由逗号分隔的清单,全都括在圆括号中。

> 检索products表，列出由 1002 或 1003 制造的所有产品。

```
MariaDB [test]> select prod_name,prod_price,vend_id from products where vend_id in (1002,1003);
+----------------+------------+---------+
| prod_name      | prod_price | vend_id |
+----------------+------------+---------+
| Detonator      |      13.00 |    1003 |
| Bird seed      |      10.00 |    1003 |
| Carrots        |       2.50 |    1003 |
| Fuses          |       3.42 |    1002 |
| Oil can        |       8.99 |    1002 |
| Safe           |      50.00 |    1003 |
| Sling          |       4.49 |    1003 |
| TNT (1 stick)  |       2.50 |    1003 |
| TNT (5 sticks) |      10.00 |    1003 |
+----------------+------------+---------+

9 rows in set (0.00 sec)
```

`IN` WHERE 子句中用来指定要匹配值的清单的关键字,功能与 OR相当 。

IN 操作符与 OR 相同的功能，下面一些实例完成之前的一些操作：

```
MariaDB [test]> select prod_name,prod_price,vend_id from products where vend_id in (1002,1003)
and prod_price >= 10;
+----------------+------------+---------+
| prod_name      | prod_price | vend_id |
+----------------+------------+---------+
| Detonator      |      13.00 |    1003 |
| Bird seed      |      10.00 |    1003 |
| Safe           |      50.00 |    1003 |
| TNT (5 sticks) |      10.00 |    1003 |
+----------------+------------+---------+
4 rows in set (0.00 sec)

MariaDB [test]> select prod_name,prod_price,vend_id from products where vend_id in (1002,1003)
order by prod_price;
+----------------+------------+---------+
| prod_name      | prod_price | vend_id |
+----------------+------------+---------+
| Carrots        |       2.50 |    1003 |
| TNT (1 stick)  |       2.50 |    1003 |
| Fuses          |       3.42 |    1002 |
| Sling          |       4.49 |    1003 |
| Oil can        |       8.99 |    1002 |
| Bird seed      |      10.00 |    1003 |
| TNT (5 sticks) |      10.00 |    1003 |
| Detonator      |      13.00 |    1003 |
| Safe           |      50.00 |    1003 |
+----------------+------------+---------+
9 rows in set (0.00 sec)
```

为什么要使用 **IN** 操作符**?** 其优点具体如下。

- 在使用长的合法选项清单时, IN 操作符的语法更清楚且更直观。
- 在使用 IN 时,计算的次序更容易管理(因为使用的操作符更少)。
- IN 操作符一般比 OR 操作符清单执行更快。
- IN 的最大优点是可以包含其他 SELECT 语句,使得能够更动态地建立 WHERE 子句。之后章节将对此进行详细介绍。

> NOT 操作符

WHERE 子句中的 NOT 操作符有且只有一个功能,那就是否定它之后所跟的任何条件。

`NOT` WHERE 子句中用来否定后跟条件的关键字。

> 列出除 1002和1003 之外的所有供应商制造的产品

```
MariaDB [test]> select prod_name,prod_price,vend_id from products where vend_id not in
(1002,1003);
+--------------+------------+---------+
| prod_name    | prod_price | vend_id |
+--------------+------------+---------+
| .5 ton anvil |       5.99 |    1001 |
| 1 ton anvil  |       9.99 |    1001 |
| 2 ton anvil  |      14.99 |    1001 |
| JetPack 1000 |      35.00 |    1005 |
| JetPack 2000 |      55.00 |    1005 |
+--------------+------------+---------+
5 rows in set (0.00 sec)

MariaDB [test]> select prod_name,prod_price,vend_id from products where vend_id != 1002 and
vend_id != 1003;
+--------------+------------+---------+
| prod_name    | prod_price | vend_id |
+--------------+------------+---------+
| .5 ton anvil |       5.99 |    1001 |
| 1 ton anvil  |       9.99 |    1001 |
| 2 ton anvil  |      14.99 |    1001 |
| JetPack 1000 |      35.00 |    1005 |
| JetPack 2000 |      55.00 |    1005 |
+--------------+------------+---------+
5 rows in set (0.00 sec)

MariaDB [test]> select prod_name,prod_price,vend_id from products where vend_id <> 1002 and
vend_id <> 1003;
+--------------+------------+---------+
| prod_name    | prod_price | vend_id |
+--------------+------------+---------+
| .5 ton anvil |       5.99 |    1001 |
| 1 ton anvil  |       9.99 |    1001 |
| 2 ton anvil  |      14.99 |    1001 |
| JetPack 1000 |      35.00 |    1005 |
| JetPack 2000 |      55.00 |    1005 |
+--------------+------------+---------+
5 rows in set (0.00 sec)
```

为什么使用 **NOT ?** 对于简单的 WHERE 子句,使用 NOT 确实没有什么优势。但在更复杂的子句中, NOT 是非常有用的。

例如,在与 IN 操作符联合使用时, NOT 使找出与条件列表不匹配的行非常简单。

MySQL 中的**NOT** MySQL 支 持 使 用 NOT 对 IN 、 BETWEEN 和EXISTS子句取反,这与多数其他 DBMS允许使用 NOT 对各种条件取反有很大的差别。

> 小结
>
> > 本章讲授如何用 AND 和 OR 操作符组合成 WHERE 子句,而且还讲授了如何明确地管理计算的次序,如何使用 IN 和 NOT 操作符。

用通配符进行过滤

本章介绍什么是通配符、如何使用通配符以及怎样使用 LIKE 操作符进行通配搜索,以便对数据进行复杂过滤。

> LIKE 操作符

前面介绍的所有操作符都是针对已知值进行过滤的。不管是匹配一个还是多个值,测试大于还是小于已知值,或者检查某个范围的值,共同点是过滤中使用的值都是已知的。但是,这种过滤方法并不是任何时候都好用。例如,怎样搜索产品名中包含文本anvil的所有产品?用简单的比较操作符肯定不行,必须使用通配符。利用通配符可创建比较特定数据的搜索模式。

通配符**(wildcard)** 用来匹配值的一部分的特殊字符。

搜索模式**(search pattern)** 由字面值、通配符或两者组合构成的搜索条件。

通配符本身实际是SQL的 WHERE 子句中有特殊含义的字符,SQL支持几种通配符(%,_)。

为在搜索子句中使用通配符,必须使用 LIKE 操作符。 LIKE 指示MySQL,后跟的搜索模式利用通配符匹配而不是直接相等匹配进行比较。

谓词 操作符何时不是操作符?答案是在它作为谓词(predi-cate)时。从技术上说, LIKE 是谓词而不是操作符。虽然最终的结果是相同的,但应该对此术语有所了解,以免在SQL文档中遇到此术语时不知道。

1.百分号(%)通配符

最常使用的通配符是百分号(%)。在搜索串中, % 表示任何字符出现任意次数。

> 找出所有以词 jet 起头的产品

```
MariaDB [test]> select prod_name,prod_price from products where prod_name like "jet%" ;
+--------------+------------+
| prod_name    | prod_price |
+--------------+------------+
| JetPack 1000 |      35.00 |
| JetPack 2000 |      55.00 |
+--------------+------------+
2 rows in set (0.00 sec)
```

通配符可在搜索模式中任意位置使用,并且可以使用多个通配符。

> 找出所有包含anvil的产品

```
MariaDB [test]> select prod_name,prod_price from products where prod_name like "%anvil%" ;
+--------------+------------+
| prod_name    | prod_price |
+--------------+------------+
| .5 ton anvil |       5.99 |
| 1 ton anvil  |       9.99 |
| 2 ton anvil  |      14.99 |
+--------------+------------+
3 rows in set (0.00 sec)
```

通配符也可以出现在搜索模式的中间,虽然这样做不太有用。

> 找出以 s 起头以 e 结尾的所有产品

```
MariaDB [test]> select prod_name,prod_price from products where prod_name like "s%e" ;
+-----------+------------+
| prod_name | prod_price |
+-----------+------------+
| Safe      |      50.00 |
+-----------+------------+
1 row in set (0.00 sec)
```

重要的是要注意到,除了一个或多个字符外, % 还能匹配0个字符。 %代表搜索模式中给定位置的0个、1个或多个字符。

注意尾空格 尾空格可能会干扰通配符匹配。例如,在保存词anvil 时 , 如 果 它 后 面 有 一 个 或 多 个 空 格 , 则 子句 WHERE prod_name LIKE '%anvil' 将不会匹配它们,因为在最后的 l后有多余的字符。解决这个问题的一个简单的办法是在搜索模式最后附加一个 % 。一个更好的办法是使用函数(之后讲解将会介绍)去掉首尾空格。

注意**NULL** 虽然似乎 % 通配符可以匹配任何东西,但有一个例外,即 NULL 。即使是 WHERE prod_name LIKE '%' 也不能匹配用值 NULL 作为产品名的行。

2.下划线(_)通配符

另一个有用的通配符是下划线(_)。下划线的用途与 % 一样,但下划线只匹配单个字符而不是多个字符。

> 找出以 开头有一位，之后是空格，然后为ton anvil的所有产品

```
MariaDB [test]> select prod_name,prod_price from products where prod_name like "_ ton anvil" ;
+-------------+------------+
| prod_name   | prod_price |
+-------------+------------+
| 1 ton anvil |       9.99 |
| 2 ton anvil |      14.99 |
+-------------+------------+
2 rows in set (0.00 sec)
```

> 使用通配符的技巧

正如所见,MySQL的通配符很有用。但这种功能是有代价的:通配符搜索的处理一般要比前面讨论的其他搜索所花时间更长。

这里给出一些使用通配符要记住的技巧。

- 不要过度使用通配符。如果其他操作符能达到相同的目的,应该使用其他操作符。
- 在确实需要使用通配符时,除非绝对有必要,否则不要把它们用在搜索模式的开始处。把通配符置于搜索模式的开始处,搜索起来是最慢的。
- 仔细注意通配符的位置。如果放错地方,可能不会返回想要的数据。

总之,通配符是一种极重要和有用的搜索工具,以后我们经常会用到它。

> 小结
>
> > 本章介绍了什么是通配符以及如何在 WHERE 子句中使用SQL通配符,并且还说明了通配符应该细心使用,不要过度使用。

用正则表达式进行搜索

本章将学习如何在MySQL WHERE 子句内使用正则表达式来更好地控制数据过滤。

> 正则表达式介绍

前两章中的过滤例子允许用匹配、比较和通配操作符寻找数据。对于基本的过滤(或者甚至是某些不那么基本的过滤),这样就足够了。但随着过滤条件的复杂性的增加, WHERE 子句本身的复杂性也有必要增加。这也就是正则表达式变得有用的地方。正则表达式是用来匹配文本的特殊的串(字符集合)。

如果你想从一个文本文件中提取电话号码,可以使用正则表达式。如果你需要查找名字中间有数字的所有文件,可以使用一个正则表达式。如果你想在一个文本块中找到所有重复的单词,可以使用一个正则表达式。如果你想替换一个页面中的所有URL为这些URL的实际HTML链接,也可以使用一个正则表达式(对于最后这个例子,或者是两个正则表达式)。

所有种类的程序设计语言、文本编辑器、操作系统等都支持正则表达式。有见识的程序员和网络管理员已经关注作为他们技术工具重要内容的正则表达式很长时间了。正则表达式用正则表达式语言来建立,正则表达式语言是用来完成刚讨论的所有工作以及更多工作的一种特殊语言。与任意语言一样,正则表达式具有你必须学习的特殊的语法和指令。

完全覆盖正则表达式的内容超出了MySQL数据库教学的范围。虽然基础知识都在这里做了介绍,但对正则表达式更为透彻的介绍我们在前面shell课程中有。此处不再赘述。

> 使用MySQL正则表达式

那么,正则表达式与MySQL有何关系?已经说过,正则表达式的作用是匹配文本,将一个模式(正则表达式)与一个文本串进行比较。MySQL用 WHERE 子句对正则表达式提供了初步的支持,允许你指定正则表达式,过滤 SELECT 检索出的数据。

仅为正则表达式语言的一个子集 如果你熟悉正则表达式,需要注意:MySQL仅支持多数正则表达式实现的一个很小的子集。本章介绍MySQL支持的大多数内容。

1.基本字符匹配

> 产品名称中匹配1000的

```
MariaDB [test]> select prod_name from products where prod_name regexp '1000';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
+--------------+
1 row in set (0.00 sec)
```

> 产品名称中匹配000的

```
MariaDB [test]> select prod_name from products where prod_name regexp '.000';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
| JetPack 2000 |
+--------------+
2 rows in set (0.00 sec)

MariaDB [test]> select prod_name from products where prod_name like '%000%';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
| JetPack 2000 |
+--------------+
2 rows in set (0.00 sec)
```

LIKE 匹配整个列。如果被匹配的文本在列值中出现, LIKE 将不会找到它,相应的行也不被返回(除非使用通配符)。而 REGEXP 在列值内进行匹配,如果被匹配的文本在列值中出现, REGEXP 将会找到它,相应的行将被返回。这是一个非常重要的差别。

那么, REGEXP 能不能用来匹配整个列值(从而起与 LIKE 相同的作用)?答案是肯定的,使用 ^ 和 $ 定位符(anchor)即可

匹配不区分大小写 MySQL中的正则表达式匹配(自版本3.23.4后)不区分大小写(即,大写和小写都匹配)。

为区分大小写,可使用 BINARY 关键字,如 `WHERE prod_name REGEXP BINARY 'JetPack .000'` 。

```
MariaDB [test]> select prod_name from products where prod_name regexp 'jetpack';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
| JetPack 2000 |
+--------------+
2 rows in set (0.00 sec)

MariaDB [test]> select prod_name from products where prod_name regexp binary 'jetpack';
Empty set (0.00 sec)

MariaDB [test]> select prod_name from products where prod_name regexp binary 'JetPack';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
| JetPack 2000 |
+--------------+
2 rows in set (0.00 sec)
```

2.进行OR匹配

为搜索两个串之一(或者为这个串,或者为另一个串),使用 |

> 产品名称中匹配1000或者2000的

```
MariaDB [test]> select prod_name from products where prod_name regexp '1000|2000';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
| JetPack 2000 |
+--------------+
2 rows in set (0.00 sec)
```

> 产品名称中匹配 1000 或 2000 或 3000

```
MariaDB [test]> select prod_name from products where prod_name regexp '1000|2000|3000';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
| JetPack 2000 |
+--------------+
2 rows in set (0.00 sec)
```

两个以上的 **OR** 条件 可以给出两个以上的 OR 条件。

3.匹配几个字符之一

匹配任何单一字符。但是,如果你只想匹配特定的字符,怎么办?可通过指定一组用 [和] 括起来的字符来完成

> 产品名称中匹配 1 或 2 或 3后面空格ton

```
MariaDB [test]> select prod_name from products where prod_name regexp '[123] ton';
+-------------+
| prod_name   |
+-------------+
| 1 ton anvil |
| 2 ton anvil |
+-------------+
2 rows in set (0.00 sec)

MariaDB [test]> select prod_name from products where prod_name regexp '[1|2|3] ton';
+-------------+
| prod_name   |
+-------------+
| 1 ton anvil |
| 2 ton anvil |
+-------------+
2 rows in set (0.00 sec)

MariaDB [test]> select prod_name from products where prod_name regexp '[1,2,3] ton';
+-------------+
| prod_name   |
+-------------+
| 1 ton anvil |
| 2 ton anvil |
+-------------+
2 rows in set (0.00 sec)
```

4.匹配范围

集合可用来定义要匹配的一个或多个字符。例如,下面的集合将匹配数字0到9:[0123456789]

为简化这种类型的集合,可使用 - 来定义一个范围。

下面的式子功能上等同于上述数字列表:

范围不限于完整的集合, [1-3] 和 [6-9] 也是合法的范围。

此外,范围不一定只是数值的, [a-z] 匹配任意字母字符。

> 匹配 1 到 5 后面空格接ton

```
MariaDB [test]> select prod_name from products where prod_name regexp '[1-5] ton';
+--------------+
| prod_name    |
+--------------+
| .5 ton anvil |
| 1 ton anvil  |
| 2 ton anvil  |
+--------------+
3 rows in set (0.00 sec)
```

5.匹配特殊字符

正则表达式语言由具有特定含义的特殊字符构成。

我们已经看到 . 、[] 、| 和 - 等,还有其他一些字符。请问,如果你需要匹配这些字符,应该怎么办呢?

例如,如果要找出包含 . 字符的值,怎样搜索?

```
MariaDB [test]> select vend_name from vendors where vend_name regexp '.';
+----------------+
| vend_name      |
+----------------+
| Anvils R Us    |
| LT Supplies    |
| ACME           |
| Furball Inc.   |
| Jet Set        |
| Jouets Et Ours |
+----------------+
6 rows in set (0.00 sec)

MariaDB [test]> select vend_name from vendors where vend_name regexp '\.';
+----------------+
| vend_name      |
+----------------+
| Anvils R Us    |
| LT Supplies    |
| ACME           |
| Furball Inc.   |
| Jet Set        |
| Jouets Et Ours |
+----------------+
6 rows in set (0.00 sec)

MariaDB [test]> select vend_name from vendors where vend_name regexp '\\.';
+--------------+
| vend_name    |
+--------------+
| Furball Inc. |
+--------------+
1 row in set (0.00 sec)
```

这才是期望的输出。 `\\.` 匹配 `.` ,所以只检索出一行。这种处理就是所谓的 转义**(escaping)**,

正则表达式内具有特殊意义的所有字符都必须以这种方式转义。

这包括 . 、 | 、 [] 以及迄今为止使用过的其他特殊字符。

为了匹配特殊字符,必须用 `\\` 为前导。 `\\-` 表示查找 `-` , `\\.` 表示查找 `.` 。

匹配 `\` 为了匹配反斜杠(`\`)字符本身,需要使用 `\\\` 。

`\\` 也用来引用元字符(具有特殊含义的字符),如下表所列。

元 字 符	说 明
\f	换页
\n	换行
\r	回车
\t	制表
\v	纵向制表

`\` 或 `\\` **?** 多数正则表达式实现使用单个反斜杠转义特殊字符,以便能使用这些字符本身。

但MySQL要求两个反斜杠(MySQL自己解释一个,正则表达式库解释另一个)。

6.匹配字符类

存在找出你自己经常使用的数字、所有字母字符或所有数字字母字符等的匹配。

为更方便工作,可以使用预定义的字符集,称为 字符类**(character class)**

下表列出字符类以及它们的含义：

类	说 明
[:alnum:]	任意字母和数字(同[a-zA-Z0-9])
[:alpha:]	任意字符(同[a-zA-Z])
[:blank:]	空格和制表(同[\t])
[:cntrl:]	ASCII控制字符(ASCII 0到31和127)
[:digit:]	任意数字(同[0-9])
[□]	与[:print:]相同,但不包括空格
[:lower:]	任意小写字母(同[a-z])
[:print:]	任意可打印字符
[:punct:]	既不在[:alnum:]又不在[:cntrl:]中的任意字符
[:space:]	包括空格在内的任意空白字符(同[\f\n\r\t\v])
[:upper:]	任意大写字母(同[A-Z])
[:xdigit:]	任意十六进制数字(同[a-fA-F0-9]

7.匹配多个实例

目前为止使用的所有正则表达式都试图匹配单次出现。

如果存在一个匹配,该行被检索出来,如果不存在,检索不出任何行。

但有时需要对匹配的数目进行更强的控制。

例如,你可能需要寻找所有的数,不管数中包含多少数字,或者你可能想寻找一个单词并且还能够适应一个尾随的 s (如果存在),等等。

这可以用下表列出的正则表达式重复元字符来完成。

重复元字符	说明
*	0个或多个匹配
+	1个或多个匹配(等于{1,})
?	0个或1个匹配(等于{0,1})
{n}	指定数目的匹配
{n,}	不少于指定数目的匹配
{n,m}	匹配数目的范围(m不超过255)

> prod_name列中匹配包括括号，并且括号中的内容依次为 一位数字，一个空格，stick，s有或者没有

```
MariaDB [test]> select prod_name from products where prod_name regexp '\\([0-9] sticks?\\)';
+----------------+
| prod_name      |
+----------------+
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
2 rows in set (0.00 sec)
```

> prod_name列中匹配4个数字

```
MariaDB [test]> select prod_name from products where prod_name regexp '[[:digit:]]{4}';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
| JetPack 2000 |
+--------------+
2 rows in set (0.00 sec）
```

需要注意的是,在使用正则表达式时,编写某个特殊的表达式几乎总是有不止一种方法。上面的例子也可以如下编写:

```
MariaDB [test]> select prod_name from products where prod_name regexp '[0-9][0-9][0-9][0-9]';
+--------------+
| prod_name    |
+--------------+
| JetPack 1000 |
| JetPack 2000 |
+--------------+
2 rows in set (0.00 sec)
```

8.定位符

目前为止的所有例子都是匹配一个串中任意位置的文本。为了匹配特定位置的文本,需要使用下表列出的定位符。

定位元字符	说 明
^	文本的开始
$	文本的结尾
[[:<:]]	词的开始
[[:>:]]	词的结尾

> 找出以一个数(包括以小数点开始的数)开始的所有产品

```
MariaDB [test]> select prod_name from products where prod_name regexp '[0-9\\.]';+---------------
-+
| prod_name      |
+----------------+
| .5 ton anvil   |
| 1 ton anvil    |
| 2 ton anvil    |
| JetPack 1000   |
| JetPack 2000   |
| TNT (1 stick)  |
| TNT (5 sticks) |
+----------------+
7 rows in set (0.00 sec)

MariaDB [test]> select prod_name from products where prod_name regexp '^[0-9\\.]';
+--------------+
| prod_name    |
+--------------+
| .5 ton anvil |
| 1 ton anvil  |
| 2 ton anvil  |
+--------------+
3 rows in set (0.00 sec)
```

简单搜索 [0-9\.] (或 [[:digit:]\.])不行,因为它将在文本内任意位置查找匹配。解决办法是使用 ^ 定位符

`^` 的双重用途 `^` 有两种用法。在集合中(用 [和] 定义),用它来否定该集合,否则,用来指串的开始处。

使 **REGEXP** 起类似 **LIKE** 的作用 本章前面说过, LIKE 和 REGEXP的不同在于, LIKE 匹配整个串而 REGEXP 匹配子串。

利用定位符,通过用 ^ 开始每个表达式,用 $ 结束每个表达式,可以使REGEXP 的作用与 LIKE 一样。

简单的正则表达式测试 可以在不使用数据库表的情况下用SELECT 来测试正则表达式。

REGEXP 检查总是返回 0 (没有匹配)或 1 (匹配)。可以用带文字串的 REGEXP 来测试表达式,并试验它们。相应的语法如下:

> 测试文本 hello 中是否能匹配到数字

```
MariaDB [test]> select 'hello' regexp '[0-9]';
+------------------------+
| 'hello' regexp '[0-9]' |
+------------------------+
|                      0 |
+------------------------+
1 row in set (0.00 sec)
```

> 测试文本中是否能匹配到制定数量的o

```
MariaDB [test]> select 'booboo' regexp '[o]{3,}';
+---------------------------+
| 'booboo' regexp '[o]{3,}' |
+---------------------------+
|                         0 |
+---------------------------+
1 row in set (0.00 sec)

MariaDB [test]> select 'boobooo' regexp '[o]{3,}';
+----------------------------+
| 'boobooo' regexp '[o]{3,}' |
+----------------------------+
|                          1 |
+----------------------------+
1 row in set (0.00 sec)
```

> 小结
>
> > 本章介绍了正则表达式的基础知识,学习了如何在MySQL的 SELECT语句中通过 REGEXP 关键字使用它们。

创建计算字段

本章介绍什么是计算字段,如何创建计算字段以及怎样从应用程序中使用别名引用它们。

> 计算字段

存储在数据库表中的数据一般不是应用程序所需要的格式。下面举几个例子。

- 如果想在一个字段中既显示公司名,又显示公司的地址,但这两个信息一般包含在不同的表列中。
- 城市、州和邮政编码存储在不同的列中(应该这样),但邮件标签打印程序却需要把它们作为一个恰当格式的字段检索出来。
- 列数据是大小写混合的,但报表程序需要把所有数据按大写表示出来。
- 物品订单表存储物品的价格和数量,但不需要存储每个物品的总价格(用价格乘以数量即可)。为打印发票,需要物品的总价格。
- 需要根据表数据进行总数、平均数计算或其他计算。

在上述每个例子中,存储在表中的数据都不是应用程序所需要的。

我们需要直接从数据库中检索出转换、计算或格式化过的数据;而不是检索出数据,然后再在客户机应用程序或报告程序中重新格式化。

这就是计算字段发挥作用的所在了。与前面各章介绍过的列不同,计算字段并不实际存在于数据库表中。计算字段是运行时在 SELECT 语句内创建的。

字段**(field)** 基本上与列(column)的意思相同,经常互换使用,不过数据库列一般称为列,而术语字段通常用在计算字段的连接上。重要的是要注意到,只有数据库知道 SELECT 语句中哪些列是实际的表列,哪些列是计算字段。从客户机(如应用程序)的角度来看,计算字段的数据是以与其他列的数据相同的方式返回的。

客户机与服务器的格式 可在SQL语句内完成的许多转换和格式化工作都可以直接在客户机应用程序内完成。但一般来说,在数据库服务器上完成这些操作比在客户机中完成要快得多,因为DBMS是设计来快速有效地完成这种处理的。

> 拼接字段

为了说明如何使用计算字段,举一个创建由两列组成的标题的简单例子。

> > `vendors` 表包含供应商名和位置信息。假如要生成一个供应商报表,需要在供应商的名字中按照 `name (location)` 这样的格式列出供应商的位置。

此报表需要单个值,而表中数据存储在两个列 `vend_name` 和 `vend_country` 中。

此外,需要用括号将 `vend_country` 括起来,这些东西都没有明确存储在数据库表中。我们来看看怎样编写返回供应商名和位置的SELECT 语句。

拼接**(concatenate)** 将值联结到一起构成单个值。

解决办法是把两个列拼接起来。在MySQL的 SELECT 语句中,可使用`Concat()` 函数来拼接两个列。

MySQL的不同之处 多数DBMS使用 + 或 || 来实现拼接,MySQL则使用 `Concat()` 函数来实现。当把SQL语句转换成MySQL语句时一定要把这个区别铭记在心。

```
MariaDB [test]> select concat(vend_name,vend_country) from vendors;
+--------------------------------+
| concat(vend_name,vend_country) |
+--------------------------------+
| Anvils R UsUSA                 |
| LT SuppliesUSA                 |
| ACMEUSA                        |
| Furball Inc.USA                |
| Jet SetEngland                 |
| Jouets Et OursFrance           |
+--------------------------------+
6 rows in set (0.01 sec)

MariaDB [test]> select concat(vend_name,' (',vend_country,')') from vendors;
+----------------------------------------+
| concat(vend_name,' (',vend_country,')') |
+----------------------------------------+
| Anvils R Us (USA)                      |
| LT Supplies (USA)                      |
| ACME (USA)                             |
| Furball Inc. (USA)                     |
| Jet Set (England)                      |
| Jouets Et Ours (France)                |
+----------------------------------------+

6 rows in set (0.00 sec)
```

Concat() 拼接串,即把多个串连接起来形成一个较长的串。

Concat() 需要一个或多个指定的串,各个串之间用逗号分隔。

上面的 SELECT 语句连接以下4个元素:

- 存储在 vend_name 列中的名字;
- 包含一个空格和一个左圆括号的串;
- 存储在 vend_country 列中的国家;
- 包含一个右圆括号的串。

从上述输出中可以看到, SELECT 语句返回包含上述4个元素的单个列(计算字段)。

> 在前面曾提到通过删除数据右侧多余的空格来整理数据,这可以使用MySQL的 `RTrim()` 函数来完成,如下所示:

```
MariaDB [test]> select concat(rtrim(vend_name),' (',rtrim(vend_country),')') from vendors;
+----------------------------------------------------+
| concat(rtrim(vend_name),' (',rtrim(vend_country),')') |
+----------------------------------------------------+
| Anvils R Us (USA)                                  |
| LT Supplies (USA)                                  |
| ACME (USA)                                         |
| Furball Inc. (USA)                                 |
| Jet Set (England)                                  |
| Jouets Et Ours (France)                            |
+----------------------------------------------------+
6 rows in set (0.00 sec)
```

Trim 函数 MySQL除了支持 `RTrim()` (正如刚才所见,它去掉串右边的空格),还支持 `LTrim()` (去掉串左边的空格)以及 `Trim()` (去掉串左右两边的空格)。

使用别名 从前面的输出中可以看到, SELECT 语句拼接地址字段工作得很好。但此新计算列的名字是什么呢?实际上它没有名字,它只是一个值。如果仅在SQL查询工具中查看一下结果,这样没有什么不好。但是,一个未命名的列不能用于客户机应用中,因为客户机没有办法引用它。为了解决这个问题,SQL支持列别名。

别名**(alias)** 是一个字段或值的替换名。别名用 AS 关键字赋予。

> 创建一个包含指定计算的名为 vend_title 的计算字段

```
MariaDB [test]> select concat(rtrim(vend_name),' (',rtrim(vend_country),')') as vend_title from
vendors order by vend_name;
+--------------------------+
| vend_title               |
+--------------------------+
| ACME (USA)               |
| Anvils R Us (USA)        |
| Furball Inc. (USA)       |
| Jet Set (England)        |
| Jouets Et Ours (France)  |
| LT Supplies (USA)        |
+--------------------------+
6 rows in set (0.00 sec)
```

别名的其他用途 别名还有其他用途。常见的用途包括在实际的表列名包含不符合规定的字符(如空格)时重新命名它,在原来的名字含混或容易误解时扩充它,等等。

导出列 别名有时也称为导出列(derived column),不管称为什么,它们所代表的都是相同的东西。

> 执行算术计算

计算字段的另一常见用途是对检索出的数据进行算术计算。

> `orders` 表包含收到的所有订单, `orderitems` 表包含每个订单中的各项物品。
> SQL语句检索订单号 20005 中的所有物品，并汇总物品的价格(单价乘以订购数量)

```
MariaDB [test]> select *,quantity*item_price as expanded_price from orderitems where order_num =
20005;
+-----------+------------+---------+----------+------------+----------------+
| order_num | order_item | prod_id | quantity | item_price | expanded_price |
+-----------+------------+---------+----------+------------+----------------+
|     20005 |          1 | ANV01   |       10 |       5.99 |          59.90 |
|     20005 |          2 | ANV02   |        3 |       9.99 |          29.97 |
|     20005 |          3 | TNT2    |        5 |      10.00 |          50.00 |
|     20005 |          4 | FB      |        1 |      10.00 |          10.00 |
+-----------+------------+---------+----------+------------+----------------+
4 rows in set (0.00 sec)

MariaDB [test]> select *,quantity*item_price  expanded_price from orderitems where order_num =
20005;
+-----------+------------+---------+----------+------------+----------------+
| order_num | order_item | prod_id | quantity | item_price | expanded_price |
+-----------+------------+---------+----------+------------+----------------+
|     20005 |          1 | ANV01   |       10 |       5.99 |          59.90 |
|     20005 |          2 | ANV02   |        3 |       9.99 |          29.97 |
|     20005 |          3 | TNT2    |        5 |      10.00 |          50.00 |
|     20005 |          4 | FB      |        1 |      10.00 |          10.00 |
+-----------+------------+---------+----------+------------+----------------+
4 rows in set (0.00 sec)
```

MySQL支持下表中列出的基本算术操作符。此外,圆括号可用来区分优先顺序。

MySQL算术操作符	说 明
+	加
-	减
*	乘
/	除

如何测试计算 SELECT 提供了测试和试验函数与计算的一个很好的办法。虽然 SELECT 通常用来从表中检索数据,但可以省略 FROM 子句以便简单地访问和处理表达式。通过下面一些例子,可以明白如何根据需要使用 SELECT 进行试验。

> SELECT3*2; 将返回 6

```
MariaDB [(none)]> select 3*2;
+-----+
| 3*2 |
+-----+
|   6 |
+-----+
1 row in set (0.00 sec)
```

> SELECT Trim(' abc '); 将返回 abc

```
MariaDB [(none)]> select trim(' abc ');
+---------------+
| trim(' abc ') |
+---------------+
| abc           |
+---------------+
1 row in set (0.00 sec)
```

> SELECT Now() 利用 Now() 函数返回当前日期和时间。

```
MariaDB [(none)]> select now();
+---------------------+
| now()               |
+---------------------+
| 2016-09-17 21:25:08 |
+---------------------+
1 row in set (0.00 sec)
```

> 小结
>
> > 本章介绍了计算字段以及如何创建计算字段。我们用例子说明了计算字段在串拼接和算术计算的用途。此外,还学习了如何创建和使用别名,以便应用程序能引用计算字段。

使用数据处理函数

本章介绍什么是函数, MySQL支持何种函数,以及如何使用这些函数。

> 函数

与其他大多数计算机语言一样, SQL支持利用函数来处理数据。函数一般是在数据上执行的,它给数据的转换和处理提供了方便。

在前一章中用来去掉串尾空格的 RTrim() 就是一个函数的例子。

函数没有**SQL**的可移植性强 能运行在多个系统上的代码称为可移植的(portable)。相对来说,多数SQL语句是可移植的,在SQL实现之间有差异时,这些差异通常不那么难处理。而函数的可移植性却不强。几乎每种主要的DBMS的实现都支持其他实现不支持的函数,而且有时差异还很大。
为了代码的可移植,许多SQL程序员不赞成使用特殊实现的功能。虽然这样做很有好处,但不总是利于应用程序的性能。如果不使用这些函数,编写某些应用程序代码会很艰难。必须利用其他方法来实现DBMS非常有效地完成的工作。如果你决定使用函数,应该保证做好代码注释,以便以后你(或其他人)能确切地知道所编写SQL代码的含义。

> 使用函数

大多数SQL实现支持以下类型的函数。

- 用于处理文本串(如删除或填充值,转换值为大写或小写)的文本函数。
- 用于在数值数据上进行算术操作(如返回绝对值,进行代数运算)的数值函数。
- 用于处理日期和时间值并从这些值中提取特定成分(例如,返回两个日期之差,检查日期有效性等)的日期和时间函数。
- 返回DBMS正使用的特殊信息(如返回用户登录信息,检查版本细节)的系统函数。

1.文本处理函数

上一章中我们已经看过一个文本处理函数的例子,其中使用 `RTrim()` 函数来去除列值右边的空格。下面是另一个例子,这次使用 `Upper()` 函数Upper()：将文本转换为大写。

> 检索供应商表vendors中的供应商名vend_name，并都转换为大写，排序。

```
MariaDB [test]> select vend_name,upper(vend_name) as vend_name_upcase from vendors order by
vend_name;
+----------------+------------------+
| vend_name      | vend_name_upcase |
+----------------+------------------+
| ACME           | ACME             |
| Anvils R Us    | ANVILS R US      |
| Furball Inc.   | FURBALL INC.     |
| Jet Set        | JET SET          |
| Jouets Et Ours | JOUETS ET OURS   |
| LT Supplies    | LT SUPPLIES      |
+----------------+------------------+
6 rows in set (0.00 sec)
```

下表列出了某些常用的文本处理函数。

常用的文本处理函数	说 明
Left()	返回串左边的字符
Right()	返回串右边的字符
Lower()	将串转换为小写
Upper()	将串转换为大写
LTrim()	去掉串左边的空格
RTrim()	去掉串右边的空格
Length()	返回串的长度
Soundex()	返回串的SOUNDEX值
Locate()	找出串的一个子串
SubString()	返回子串的字符

上表中的 SOUNDEX 需要做进一步的解释。 SOUNDEX 是一个将任何文本串转换为描述其语音表示的字母数字模式的算法。 SOUNDEX 考虑了类似的发音字符和音节,使得能对串进行发音比较而不是字母比较。虽然SOUNDEX不是SQL概念,但MySQL(就像多数DBMS一样)都提供对SOUNDEX 的支持。

下面给出一个使用 Soundex() 函数的例子。

> `customers` 表中有一个顾客 `Coyote Inc.` ,其联系名为`Y Lee`。但如果这是输入错误,此联系名实际应该是 `Y Lie` ,怎么办?显然,按正确的联系名搜索不会返回数据。

```
MariaDB [test]> select cust_name,cust_contact from customers where cust_contact = 'Y Lie';
Empty set (0.00 sec)

MariaDB [test]> select cust_name,cust_contact from customers where soundex(cust_contact) =
soundex('Y Lie');
+-------------+--------------+
| cust_name   | cust_contact |
+-------------+--------------+
| Coyote Inc. | Y Lee        |
+-------------+--------------+
1 row in set (0.00 sec)
```

2.日期和时间处理函数

日期和时间采用相应的数据类型和特殊的格式存储,以便能快速和有效地排序或过滤,并且节省物理存储空间。

一般,应用程序不使用用来存储日期和时间的格式,因此日期和时间函数总是被用来读取、统计和处理这些值。由于这个原因,日期和时间函数在MySQL语言中具有重要的作用。

下表列出了某些常用的日期和时间处理函数。

常用日期和时间处理函数	说 明
AddDate()	增加一个日期(天、周等)
AddTime()	增加一个时间(时、分等)
CurDate()	返回当前日期
CurTime()	返回当前时间
Date()	返回日期时间的日期部分
DateDiff()	计算两个日期之差
Date_Add()	高度灵活的日期运算函数
Date_Format()	返回一个格式化的日期或时间串
Day()	返回一个日期的天数部分
DayOfWeek()	对于一个日期,返回对应的星期几
Hour()	返回一个时间的小时部分
Minute()	返回一个时间的分钟部分
Month()	返回一个日期的月份部分
Now()	返回当前日期和时间
Second()	返回一个时间的秒部分
Time()	返回一个日期时间的时间部分
Year()	返回一个日期的年份部分

这是重新复习用 WHERE 进行数据过滤的一个好时机。迄今为止,我们都是用比较数值和文本的 WHERE 子句过滤数据,但数据经常需要用日期进行过滤。用日期进行过滤需要注意一些别的问题和使用特殊的MySQL函数。

首先需要注意的是MySQL使用的日期格式。无论你什么时候指定一个日期,不管是插入或更新表值还是用 WHERE 子句进行过滤,日期必须为格式yyyy-mm-dd。因此,2005年9月1日,给出为2005-09-01。虽然其他的日期格式可能也行,但这是首选的日期格式,因为它排除了多义性(如,04/05/06是2006年5月4日或2006年4月5日或2004年5月6日或……)。

应该总是使用**4**位数字的年份 支持2位数字的年份,MySQL处理00-69为2000-2069,处理70-99为1970-1999。虽然它们可能是打算要的年份,但使用完整的4位数字年份更可靠,因为MySQL不必做出任何假定。

> 检索出一个订单记录,该订单记录的 `order_date` 为 `2005-09-01`

```
MariaDB [test]> select * from orders where order_date='2005-09-01';
+-----------+---------------------+---------+
| order_num | order_date          | cust_id |
+-----------+---------------------+---------+
|     20005 | 2005-09-01 00:00:00 |   10001 |
+-----------+---------------------+---------+
1 row in set (0.00 sec)
```

但是,使用 `WHERE order_date = '2005-09-01'` 可靠吗? `order_date` 的数据类型为 `datetime` 。这种类型存储日期及时间值。样例表中的值全都具有时间值 00:00:00 ,但实际中很可能并不总是这样。如果用当前日期和时间存储订单日期(因此你不仅知道订单日期,还知道下订单当天的时间),怎么办?

比如,存储的 `order_date` 值为 `2005-09-01 11:30:05` ,则 `WHERE order_date = '2005-09-01'` 失败。即使给出具有该日期的一行,也不会把它检索出来,因为 WHERE 匹配失败。

解决办法是指示MySQL仅将给出的日期与列中的日期部分进行比较,而不是将给出的日期与整个列值进行比较。为此,必须使用 `Date()` 函数。 `Date(order_date)` 指示MySQL仅提取列的日期部分,更可靠的SELECT 语句为:

```
MariaDB [test]> select * from orders where date(order_date)='2005-09-01';
+-----------+---------------------+---------+
| order_num | order_date          | cust_id |
+-----------+---------------------+---------+
|     20005 | 2005-09-01 00:00:00 |   10001 |
+-----------+---------------------+---------+
1 row in set (0.00 sec)
```

如果要的是日期,请使用 **Date()** 如果你想要的仅是日期,则使用 Date() 是一个良好的习惯,即使你知道相应的列只包含日期也是如此。这样,如果由于某种原因表中以后有日期和时间值,你的SQL代码也不用改变。当然,也存在一个Time()函数,在你只想要时间时应该使用它。Date() 和 Time() 都是在MySQL 4.1.1中第一次引入的。

在你知道了如何用日期进行相等测试后,其他操作符的使用也就很清楚了。不过,还有一种日期比较需要说明。

> 检索出2005年9月下的所有订单

```
MariaDB [test]> select * from orders where date(order_date) between '2005-09-01' and '2005-09-
30';
+-----------+---------------------+---------+
| order_num | order_date          | cust_id |
+-----------+---------------------+---------+
|     20005 | 2005-09-01 00:00:00 |   10001 |
|     20006 | 2005-09-12 00:00:00 |   10003 |
|     20007 | 2005-09-30 00:00:00 |   10004 |
+-----------+---------------------+---------+
3 rows in set (0.00 sec)
```

还有另外一种办法(一种不需要记住每个月中有多少天或不需要操心闰年2月的办法)

```
MariaDB [test]> select * from orders where year(order_date) = 2005 and month(order_date)= 9 ;
+-----------+---------------------+---------+
| order_num | order_date          | cust_id |
+-----------+---------------------+---------+
|     20005 | 2005-09-01 00:00:00 |   10001 |
|     20006 | 2005-09-12 00:00:00 |   10003 |
|     20007 | 2005-09-30 00:00:00 |   10004 |
+-----------+---------------------+---------+
3 rows in set (0.01 sec)
```

MySQL的版本差异 MySQL 4.1.1中增加了许多日期和时间函数。如果你使用的是更早的MySQL版本,应该查阅具体的文档以确定可以使用哪些函数。

3.数值处理函数

数值处理函数仅处理数值数据。这些函数一般主要用于代数、三角或几何运算,因此没有串或日期—时间处理函数的使用那么频繁。

具有讽刺意味的是,在主要DBMS的函数中,数值函数是最一致最统一的函数。下表列出一些常用的数值处理函数。

常用数值处理函数	说 明
Abs()	返回一个数的绝对值
Cos()	返回一个角度的余弦
Exp()	返回一个数的指数值
Mod()	返回除操作的余数
Pi()	返回圆周率
Rand()	返回一个随机数
Sin()	返回一个角度的正弦
Sqrt()	返回一个数的平方根
Tan()	返回一个角度的正切

> 测试函数功能

```
MariaDB [test]> select pi();
+----------+
| pi()     |
+----------+
| 3.141593 |
+----------+
1 row in set (0.00 sec)

MariaDB [test]> select rand();
+--------------------+
| rand()             |
+--------------------+
| 0.7680412305221346 |
+--------------------+
1 row in set (0.00 sec)
```

> 小结
>
> > 本章介绍了如何使用SQL的数据处理函数,并着重介绍了日期处理函数。

汇总数据

本章介绍什么是SQL的聚集函数以及如何利用它们汇总表的数据。

> 聚集函数

我们经常需要汇总数据而不用把它们实际检索出来,为此MySQL提供了专门的函数。使用这些函数,MySQL查询可用于检索数据,以便分析和报表生成。这种类型的检索例子有以下几种。

- 确定表中行数(或者满足某个条件或包含某个特定值的行数)。
- 获得表中行组的和。
- 找出表列(或所有行或某些特定的行)的最大值、最小值和平均值。

上述例子都需要对表中数据(而不是实际数据本身)汇总。因此,返回实际表数据是对时间和处理资源的一种浪费(更不用说带宽了)。重复一遍,实际想要的是汇总信息。

为方便这种类型的检索,MySQL给出了5个聚集函数，见下表，这些函数能进行上述罗列的检索。

聚集函数**(aggregate function)** 运行在行组上,计算和返回单个值的函数。

SQL聚集函数	说明
AVG()	返回某列的平均值
COUNT()	返回某列的行数
MAX()	返回某列的最大值
MIN()	返回某列的最小值
SUM()	返回某列值之和

标准偏差 MySQL还支持一系列的标准偏差聚集函数,但该教案并未涉及这些内容。

1.AVG() 函数

AVG() 通过对表中行数计数并计算特定列值之和,求得该列的平均值。 AVG() 可用来返回所有列的平均值,也可以用来返回特定列或行的平均值。

> 使用 `AVG()` 返回 `products` 表中所有产品的平均价格 `avg_Price`

```
MariaDB [test]> select avg(prod_price) as avg_price from products;
+-----------+
| avg_price |
+-----------+
| 16.133571 |
+-----------+
1 row in set (0.00 sec)
```

> 使用 `AVG()` 返回 `products` 表中产品供应商1003提供的商品的平均价格 `avg_Price`

```
MariaDB [test]> select avg(prod_price) as avg_price from products where vend_id=1003;
+-----------+
| avg_price |
+-----------+
| 13.212857 |
+-----------+
1 row in set (0.00 sec)
```

只用于单个列 AVG() 只能用来确定特定数值列的平均值,而且列名必须作为函数参数给出。为了获得多个列的平均值,必须使用多个 AVG() 函数。

NULL 值 AVG() 函数忽略列值为 NULL 的行

2.COUNT() 函数

COUNT() 函数进行计数。可利用 COUNT() 确定表中行的数目或符合特定条件的行的数目。

COUNT() 函数有两种使用方式。

- 使用 COUNT(*) 对表中行的数目进行计数,不管表列中包含的是空值(NULL)还是非空值。
- 使用 COUNT(column) 对特定列中具有值的行进行计数,忽略NULL 值。

> 返回 customers 表中客户的总数

```
MariaDB [test]> select count(prod_name) from products;
+------------------+
| count(prod_name) |
+------------------+
|               14 |
+------------------+
1 row in set (0.00 sec)

MariaDB [test]> select count(*) from products;
+----------+
| count(*) |
+----------+
|       14 |
+----------+
1 row in set (0.00 sec)
```

> 对具有电子邮件地址的客户计数,别名num_cust

```
MariaDB [test]> select cust_email from customers;
+---------------------+
| cust_email          |
+---------------------+
| ylee@coyote.com     |
| NULL                |
| rabbit@wascally.com |
| sam@yosemite.com    |
| NULL                |
+---------------------+
5 rows in set (0.00 sec)

MariaDB [test]> select count(cust_email) from customers;
+-------------------+
| count(cust_email) |
+-------------------+
|                 3 |
+-------------------+
1 row in set (0.00 sec)

MariaDB [test]> select count(*) from customers;
+----------+
| count(*) |
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)

MariaDB [test]> select count(cust_email) as num_cust from customers;
+----------+
| num_cust |
+----------+
|        3 |
+----------+
1 row in set (0.00 sec)
```

NULL 值 如果指定列名,则指定列的值为空的行被 COUNT()函数忽略,但如果 COUNT() 函数中用的是星号(*),则不忽略。

3.MAX() 函数

MAX() 返回指定列中的最大值。 MAX() 要求指定列名。

> MAX() 返回 products 表中最贵的物品的价格

```
MariaDB [test]> select prod_name,max(prod_price) as max_price from products;
+--------------+-----------+
| prod_name    | max_price |
+--------------+-----------+
| .5 ton anvil |     55.00 |
+--------------+-----------+
1 row in set (0.00 sec)
```

对非数值数据使用 **MAX()** 虽然 MAX() 一般用来找出最大的数值或日期值,但MySQL允许将它用来返回任意列中的最大值,包括返回文本列中的最大值。在用于文本数据时,如果数据按相应的列排序,则 MAX() 返回最后一行。

```
MariaDB [test]> select prod_name,max(prod_name) as max_name from products;
+--------------+----------------+
| prod_name    | max_name       |
+--------------+----------------+
| .5 ton anvil | TNT (5 sticks) |
+--------------+----------------+
1 row in set (0.00 sec)
```

NULL 值 MAX() 函数忽略列值为 NULL 的行。

4.MIN() 函数

MIN() 的功能正好与 MAX() 功能相反,它返回指定列的最小值。与MAX() 一样, MIN() 要求指定列名。

> MIN() 返回 products 表中最便宜物品的价格

```
MariaDB [test]> select prod_name,min(prod_price) as min_price from products;
+--------------+-----------+
| prod_name    | min_price |
+--------------+-----------+
| .5 ton anvil |      2.50 |
+--------------+-----------+
1 row in set (0.00 sec)
```

对非数值数据使用 **MIN()** MIN() 函数与 MAX() 函数类似,MySQL允许将它用来返回任意列中的最小值,包括返回文本列中的最小值。在用于文本数据时,如果数据按相应的列排序,则 MIN() 返回最前面的行。

NULL 值 MIN() 函数忽略列值为 NULL 的行。

5.SUM() 函数

SUM() 用来返回指定列值的和(总计)。

> `orderitems`表 包含订单中实际的物品,每个物品有相应的数量(`quantity`)。
> 检索所订购物品的总数(所有`quantity` 值之和)

```
MariaDB [test]> desc orderitems;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| order_num  | int(11)      | NO   | PRI | NULL    |       |
| order_item | int(11)      | NO   | PRI | NULL    |       |
| prod_id    | char(10)     | NO   | MUL | NULL    |       |
| quantity   | int(11)      | NO   |     | NULL    |       |
| item_price | decimal(8,2) | NO   |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)

MariaDB [test]> select sum(quantity) from orderitems;
+---------------+
| sum(quantity) |
+---------------+
|           174 |
+---------------+
1 row in set (0.00 sec)

MariaDB [test]> select sum(quantity) as items_ordered  from orderitems where order_num = 20005;
+---------------+
| items_ordered |
+---------------+
|            19 |
+---------------+
1 row in set (0.00 sec)
```

> `SUM()` 也可以用来合计计算值。在下面的例子中,合计每项物品的`item_price*quantity`,得出总的订单金额`total_price`

```
MariaDB [test]> select item_price*quantity from orderitems;
+---------------------+
| item_price*quantity |
+---------------------+
|               59.90 |
|               29.97 |
|               50.00 |
|               10.00 |
|               55.00 |
|             1000.00 |
|              125.00 |
|               10.00 |
|                8.99 |
|                4.49 |
|               14.99 |
+---------------------+
11 rows in set (0.00 sec)

MariaDB [test]> select sum(item_price*quantity) as total_price from orderitems;
+-------------+
| total_price |
+-------------+
|     1368.34 |
+-------------+
1 row in set (0.01 sec)
```

在多个列上进行计算 如本例所示,利用标准的算术操作符,所有聚集函数都可用来执行多个列上的计算。

NULL 值 SUM() 函数忽略列值为 NULL 的行。

> 聚集不同值

MySQL 5 及 后 期 版 本 下面将要介绍的聚集函数的DISTINCT 的使用,已经被添加到MySQL 5.0.3中。下面所述内容在MySQL 4.x中不能正常运行。

以上5个聚集函数都可以如下使用:

- 对所有的行执行计算,指定 ALL 参数或不给参数(因为 ALL 是默认行为);
- 只包含不同的值,指定 DISTINCT 参数。

ALL 为默认 ALL 参数不需要指定,因为它是默认行为。如果不指定 DISTINCT ,则假定为 ALL 。

> 使用 AVG() 函数返回特定供应商提供的产品的平均价格。它与上面的 SELECT 语句相同,但使用了 DISTINCT 参数,因此平均值只考虑各个不同的价格:

```
MariaDB [test]> select prod_price from products order by prod_price;
+------------+
| prod_price |
+------------+
|       2.50 |
|       2.50 |
|       3.42 |
|       4.49 |
|       5.99 |
|       8.99 |
|       9.99 |
|      10.00 |
|      10.00 |
|      13.00 |
|      14.99 |
|      35.00 |
|      50.00 |
|      55.00 |
+------------+
14 rows in set (0.00 sec)

MariaDB [test]> select distinct prod_price from products order by prod_price;
+------------+
| prod_price |
+------------+
|       2.50 |
|       3.42 |
|       4.49 |
|       5.99 |
|       8.99 |
|       9.99 |
|      10.00 |
|      13.00 |
|      14.99 |
|      35.00 |
|      50.00 |
|      55.00 |
+------------+
12 rows in set (0.00 sec)

MariaDB [test]> select avg(distinct prod_price) from products where vend_id = 1003;
+---------------------------+
| avg(distinct prod_price) |
+---------------------------+
|                15.998000 |
+---------------------------+
1 row in set (0.00 sec)

MariaDB [test]> select avg(prod_price) from products where vend_id = 1003;
+-----------------+
| avg(prod_price) |
+-----------------+
|       13.212857 |
```

```
+-----------------+
1 row in set (0.00 sec)
```

可以看到,在使用了 `DISTINCT` 后,此例子中的 `avg_price` 比较高,因为有多个物品具有相同的较低价格。排除它们提升了平均价格。

注意 如果指定列名,则 DISTINCT 只能用于 COUNT() 。 DISTINCT不能用于 COUNT(*),因此不允许使用 COUNT(DISTINCT),否则会产生错误 。类似地, DISTINCT 必须使用列名,不能用于计算或表达式。

将 **DISTINCT** 用于 **MIN()** 和 **MAX()** 虽然 DISTINCT 从技术上可用于 MIN() 和 MAX() ,但这样做实际上没有价值。一个列中的最小值和最大值不管是否包含不同值都是相同的。

> 组合聚集函数

目前为止的所有聚集函数例子都只涉及单个函数。但实际上 SELECT语句可根据需要包含多个聚集函数。

> 检索products 表中物品的数目 `num_items` ,产品价格的最高 `price_max` 、最低 `price_min` 以及平均值 `price_avg`

```
MariaDB [test]> select count(*) as num_items,min(prod_price)as price_min,max(prod_price) as
price_max,avg(prod_price) as price_avg from products;
+-----------+-----------+-----------+-----------+
| num_items | price_min | price_max | price_avg |
+-----------+-----------+-----------+-----------+
|        14 |      2.50 |     55.00 | 16.133571 |
+-----------+-----------+-----------+-----------+
1 row in set (0.00 sec)
```

取别名 在指定别名以包含某个聚集函数的结果时,不应该使用表中实际的列名。虽然这样做并非不合法,但使用唯一的名字会使你的SQL更易于理解和使用(以及将来容易排除故障)。

> 小结
>
> > 聚集函数用来汇总数据。MySQL支持一系列聚集函数,可以用多种方法使用它们以返回所需的结果。这些函数是高效设计的,它们返回结果一般比你在自己的客户机应用程序中计算要快得多。

分组数据

本章将介绍如何分组数据,以便能汇总表内容的子集。这涉及两个新 SELECT 语句子句,分别是 GROUP BY 子句和 HAVING 子句。

> 数据分组

从上一章知道, SQL聚集函数可用来汇总数据。这使我们能够对行进行计数,计算和与平均数,获得最大和最小值而不用检索所有数据。目前为止的所有计算都是在表的所有数据或匹配特定的 WHERE 子句的数据上进行的。

> > 下面的例子返回供应商 `vend_id` 1003 提供的产品数目

```
MariaDB [test]> select vend_id,count(vend_id) as num_prods from products where vend_id=1003;
+---------+----------+
| vend_id | num_prods |
+---------+----------+
|    1003 |        7 |
+---------+----------+
1 row in set (0.00 sec)
```

但如果要返回每个供应商提供的产品数目怎么办?或者返回只提供单项产品的供应商所提供的产品,或返回提供10个以上产品的供应商怎么办?

这就是分组显身手的时候了。分组允许把数据分为多个逻辑组,以便能对每个组进行聚集计算。

> 创建分组

分组是在 SELECT 语句的 GROUP BY 子句中建立的。理解分组的最好办法是看一个例子

> 按 `vend_id` 排序并分组数据计算每个供应商的商品总数 `num_prods`

```
MariaDB [test]> select vend_id,count(vend_id) as num_prods from products group by vend_id;
+---------+-----------+
| vend_id | num_prods |
+---------+-----------+
|    1001 |         3 |
|    1002 |         2 |
|    1003 |         7 |
|    1005 |         2 |
+---------+-----------+
4 rows in set (0.00 sec)
```

因为使用了 GROUP BY ,就不必指定要计算和估值的每个组了。系统会自动完成。 GROUP BY 子句指示MySQL分组数据,然后对每个组而不是整个结果集进行聚集。

在具体使用 GROUP BY 子句前,需要知道一些重要的规定。

- GROUP BY 子句可以包含任意数目的列。这使得能对分组进行嵌套,为数据分组提供更细致的控制。
- 如果在 GROUP BY 子句中嵌套了分组,数据将在最后规定的分组上进行汇总。换句话说,在建立分组时,指定的所有列都一起计算(所以不能从个别的列取回数据)。
- GROUP BY 子句中列出的每个列都必须是检索列或有效的表达式(但不能是聚集函数)。如果在 SELECT 中使用表达式,则必须在GROUP BY 子句中指定相同的表达式。不能使用别名。
- 除聚集计算语句外, SELECT 语句中的每个列都必须在 GROUP BY 子句中给出。
- 如果分组列中具有 NULL 值,则 NULL 将作为一个分组返回。如果列中有多行 NULL 值,它们将分为一组。
- GROUP BY 子句必须出现在 WHERE 子句之后, ORDER BY 子句之前。

使用 **ROLLUP** 使用 WITH ROLLUP 关键字,可以得到每个分组以及每个分组汇总级别(针对每个分组)的值,如下所示:

```
MariaDB [test]> select vend_id,count(vend_id) as num_prods from products group by vend_id with
rollup;
+---------+-----------+
| vend_id | num_prods |
+---------+-----------+
|    1001 |         3 |
|    1002 |         2 |
|    1003 |         7 |
|    1005 |         2 |
|    NULL |        14 |
+---------+-----------+
5 rows in set (0.00 sec)
```

> 过滤分组

除了能用 GROUP BY 分组数据外,MySQL还允许过滤分组,规定包括哪些分组,排除哪些分组。例如,可能想要列出至少有两个订单的所有 113顾客。为得出这种数据,必须基于完整的分组而不是个别的行进行过滤。。但是,在这个例我们已经看到了 WHERE 子句的作用(第6章中引入)子中 WHERE 不能完成任务,因为 WHERE 过滤指定的是行而不是分组。事实上, WHERE 没有分组的概念。那么,不使用 WHERE 使用什么呢?MySQL为此目的提供了另外的子句,那就是 HAVING 子句。 HAVING 非常类似于 WHERE 。事实上,目前为止所学过的所有类型的 WHERE 子句都可以用 HAVING 来替代。唯一的差别是WHERE 过滤行,而 HAVING 过滤分组。HAVING 支持所有 WHERE 操作符在第6章和第7章中,我们学习了 WHERE 子句的条件(包括通配符条件和带多个操作符的子句)。所学过的有关 WHERE 的所有这些技术和选项都适用于HAVING 。它们的句法是相同的,只是关键字有差别。那么,怎么过滤分组呢?请看以下的例子:

> > 过滤两个以上的订单的那些分组

```
MariaDB [test]> select cust_id,count(*) as orders from orders group by cust_id;
+---------+--------+
| cust_id | orders |
+---------+--------+
|   10001 |      2 |
|   10003 |      1 |
|   10004 |      1 |
|   10005 |      1 |
+---------+--------+
4 rows in set (0.00 sec)

MariaDB [test]> select cust_id,count(*) as orders from orders group by cust_id having count(*) >=
2;
+---------+--------+
| cust_id | orders |
+---------+--------+
|   10001 |      2 |
+---------+--------+
1 row in set (0.00 sec)
```

正如所见,这里 WHERE 子句不起作用,因为过滤是基于分组聚集值而不是特定行值的。

HAVING 和 **WHERE** 的差别 这里有另一种理解方法, WHERE 在数据分组前进行过滤, HAVING 在数据分组后进行过滤。这是一个重要的区别, WHERE 排除的行不包括在分组中。这可能会改变计算值,从而影响 HAVING 子句中基于这些值过滤掉的分组。

那么,有没有在一条语句中同时使用 WHERE 和 HAVING 子句的需要呢?事实上,确实有。假如想进一步过滤上面的语句,使它返回过去12个月内具有两个以上订单的顾客。为达到这一点,可增加一条 WHERE 子句,过滤出过去12个月内下过的订单。然后再增加 HAVING 子句过滤出具有两个以上订单的分组。

为更好地理解,请看下面的例子。

> 列出具有 2 个(含)以上、价格为 10 (含)以上的产品的供应商:

```
MariaDB [test]> select vend_id,count(vend_id) from products where prod_price >= 10 group by
vend_id ;
+---------+----------------+
| vend_id | count(vend_id) |
+---------+----------------+
|    1001 |              1 |
|    1003 |              4 |
|    1005 |              2 |
+---------+----------------+
3 rows in set (0.00 sec)

MariaDB [test]> select vend_id,count(vend_id) from products where prod_price >= 10 group by
vend_id having count(vend_id) >= 2 ;
+---------+----------------+
| vend_id | count(vend_id) |
+---------+----------------+
|    1003 |              4 |
|    1005 |              2 |
+---------+----------------+
2 rows in set (0.01 sec)
```

这条语句中,使用了聚集函数的基本 `SELECT` ,它与前面的例子很相像。 `WHERE` 子句过滤所有 `prod_price` 至少为 10 的行。然后按 `vend_id` 分组数据, `HAVING` 子句过滤计数为 2 或 2 以上的分组。如果没有 `WHERE` 子句,将会多检索出两行(供应商 1002 ,销售的所有产品价格都在 10 以下;供应商 1001 ,销售3个产品,但只有一个产品的价格大于等于 10):

```
MariaDB [test]> select vend_id,count(vend_id) from products group by vend_id having
count(vend_id) >= 2 ;
+---------+----------------+
| vend_id | count(vend_id) |
+---------+----------------+
|    1001 |              3 |
|    1002 |              2 |
|    1003 |              7 |
|    1005 |              2 |
+---------+----------------+
4 rows in set (0.00 sec)
```

> 分组和排序

虽然 GROUP BY 和 ORDER BY 经常完成相同的工作,但它们是非常不同的。

下表汇总了它们之间的差别

ORDER BY	GROUP BY
排序产生的输出	分组行。但输出可能不是分组的顺序
任意列都可以使用(甚至非选择的列也可以使用)	只可能使用选择列或表达式列,而且必须使用每个选择列表达式
不一定需要	如果与聚集函数一起使用列(或表达式),则必须使用

表中列出的第一项差别极为重要。我们经常发现用 GROUP BY 分组的数据确实是以分组顺序输出的。但情况并不总是这样,它并不是SQL规范所要求的。此外,用户也可能会要求以不同于分组的顺序排序。仅因为你以某种方式分组数据(获得特定的分组聚集值),并不表示你需要以相同的方式排序输出。应该提供明确的 ORDER BY 子句,即使其效果等同于 GROUP BY 子句也是如此。

不要忘记 **ORDER BY** 一般在使用 GROUP BY 子句时,应该也给出 ORDER BY 子句。这是保证数据正确排序的唯一方法。千万不要仅依赖 GROUP BY 排序数据。

为说明 GROUP BY 和 ORDER BY 的使用方法,请看一个例子。下面的SELECT 语句类似于前面那些例子。

> 检索总计订单价格大于等于 50 的订单的订单号和总计订单价格

```
MariaDB [test]> select * from orderitems;
+-----------+------------+---------+----------+------------+
| order_num | order_item | prod_id | quantity | item_price |
+-----------+------------+---------+----------+------------+
|     20005 |          1 | ANV01   |       10 |       5.99 |
|     20005 |          2 | ANV02   |        3 |       9.99 |
|     20005 |          3 | TNT2    |        5 |      10.00 |
|     20005 |          4 | FB      |        1 |      10.00 |
|     20006 |          1 | JP2000  |        1 |      55.00 |
|     20007 |          1 | TNT2    |      100 |      10.00 |
|     20008 |          1 | FC      |       50 |       2.50 |
|     20009 |          1 | FB      |        1 |      10.00 |
|     20009 |          2 | OL1     |        1 |       8.99 |
|     20009 |          3 | SLING   |        1 |       4.49 |
|     20009 |          4 | ANV03   |        1 |      14.99 |
+-----------+------------+---------+----------+------------+
11 rows in set (0.00 sec)

MariaDB [test]> select order_num,sum(quantity*item_price) from orderitems group by order_num
order by sum(quantity*item_price);
+-----------+---------------------------+
| order_num | sum(quantity*item_price) |
+-----------+---------------------------+
|     20009 |                     38.47 |
|     20006 |                     55.00 |
|     20008 |                    125.00 |
|     20005 |                    149.87 |
|     20007 |                   1000.00 |
+-----------+---------------------------+
5 rows in set (0.00 sec)

MariaDB [test]> select order_num,sum(quantity*item_price) as ordertotal from orderitems group by
order_num order by ordertotal;
+-----------+------------+
| order_num | ordertotal |
+-----------+------------+
|     20009 |      38.47 |
|     20006 |      55.00 |
|     20008 |     125.00 |
|     20005 |     149.87 |
|     20007 |    1000.00 |
+-----------+------------+
5 rows in set (0.00 sec)

MariaDB [test]> select order_num,sum(quantity*item_price) as ordertotal from orderitems group by
order_num having ordertotal >= 50 order by ordertotal;
+-----------+------------+
| order_num | ordertotal |
+-----------+------------+
|     20006 |      55.00 |
|     20008 |     125.00 |
|     20005 |     149.87 |
|     20007 |    1000.00 |
```

```
+-----------+------------+
4 rows in set (0.00 sec)

MariaDB [test]> select order_num,sum(quantity*item_price) as ordertotal from orderitems group by
order_num having sum(quantity*item_price) >= 50 order by sum(quantity*item_price);
+-----------+------------+
| order_num | ordertotal |
+-----------+------------+
|     20006 |      55.00 |
|     20008 |     125.00 |
|     20005 |     149.87 |
|     20007 |    1000.00 |
+-----------+------------+
4 rows in set (0.00 sec)
```

在这个例子中, `GROUP BY` 子句用来按订单号(`order_num` 列)分组数据,以便 `SUM(*)` 函数能够返回总计订单价格。 `HAVING` 子句过滤数据,使得只返回总计订单价格大于等于 50 的订单。最后,用 `ORDER BY` 子句排序输出。

> SELECT 子句顺序

下面回顾一下 SELECT 语句中子句的顺序。下表以在 SELECT 语句中使用时必须遵循的次序,列出迄今为止所学过的子句。

SELECT子句	说明	是否必须使用
SELECT	要返回的列或表达式	是
FROM	从中检索数据的表	仅在从表选择数据时使用
WHERE	行级过滤	否
GROUP BY	分组说明	仅在按组计算聚集时使用
HAVING	组级过滤	否
ORDER BY	输出排序顺序	否
LIMIT	要检索的行数	否

> 小结
>
> > 在本章中,我们学习了如何用SQL聚集函数对数据进行汇总计算。本章讲授了如何使用 GROUP BY 子句对数据组进行这些汇总计算,返回每个组的结果。我们看到了如何使用 HAVING 子句过滤特定的组,还知道了 ORDER BY 和 GROUP BY 之间以及 WHERE 和 HAVING 之间的差异。

使用子查询

本章介绍什么是子查询以及如何使用它们。

> 子查询
> 版本要求 MySQL 4.1引入了对子查询的支持,所以要想使用本章描述的SQL,必须使用MySQL 4.1或更高级的版本。

SELECT语句 是SQL的查询。迄今为止我们所看到的所有 SELECT 语句都是简单查询,即从单个数据库表中检索数据的单条语句。

查询**(query)** 任何SQL语句都是查询。但此术语一般指 SELECT语句。

SQL还允许创建子查询(subquery),即嵌套在其他查询中的查询。为什么要这样做呢?理解这个概念的最好方法是考察几个例子。

> 利用子查询进行过滤

订单信息存储在两个表中。对于包含订单号、客户ID、订单日期的每个订单, orders 表存储一行。
各订单的物品存储在相关的orderitems 表中。 orders 表不存储客户信息。它只存储客户的ID。
实际的客户信息存储在 customers 表中。

> > 现在,假如需要列出订购物品 TNT2 的所有客户,应该怎样检索?下面列出具体的步骤。

(1) 检索包含物品 TNT2 的所有订单的编号。

(2) 检索具有前一步骤列出的订单编号的所有客户的ID。

(3) 检索前一步骤返回的所有客户ID的客户信息。

上述每个步骤都可以单独作为一个查询来执行。可以把一条 SELECT语句返回的结果用于另一条 SELECT 语句的 WHERE 子句。也可以使用子查询来把3个查询组合成一条语句

```
MariaDB [test]> select order_num from orderitems where prod_id='TNT2';
+-----------+
| order_num |
+-----------+
|     20005 |
|     20007 |
+-----------+
2 rows in set (0.00 sec)

MariaDB [test]> select cust_id from orders where order_num = 20005 or order_num = 20007;
+---------+
| cust_id |
+---------+
|   10001 |
|   10004 |
+---------+
2 rows in set (0.00 sec)

MariaDB [test]> select cust_name from customers where cust_id in (10001,10004);
+----------------+
| cust_name      |
+----------------+
| Coyote Inc.    |
| Yosemite Place |
+----------------+
2 rows in set (0.00 sec)

MariaDB [test]> select cust_name from customers where cust_id in
> (
> select cust_id from orders where order_num in
> (select order_num from orderitems where prod_id='TNT2')
> );
+----------------+
| cust_name      |
+----------------+
| Coyote Inc.    |
| Yosemite Place |
+----------------+
2 rows in set (0.00 sec)
```

可见,在 WHERE 子句中使用子查询能够编写出功能很强并且很灵活的SQL语句。对于能嵌套的子查询的数目没有限制,不过在实际使用时由于性能的限制,不能嵌套太多的子查询。

列必须匹配 在 WHERE 子句中使用子查询(如这里所示),应该保证 SELECT 语句具有与 WHERE 子句中相同数目的列。通常,子查询将返回单个列并且与单个列匹配,但如果需要也可以使用多个列。

虽然子查询一般与 `IN` 操作符结合使用,但也可以用于测试等于(`=`)、不等于(`<>`)等。

子查询和性能 这里给出的代码有效并获得所需的结果。但是,使用子查询并不总是执行这种类型的数据检索的最有效的方法。更多的论述,请参阅下一章,其中将再次给出这个例子。

> 作为计算字段使用子查询

使用子查询的另一方法是创建计算字段。

> 查询每个客户的姓名`cust_name`和状态`cust_state`以及每个客户的订单总数`orders`。订单与相应的客户ID存储在 orders 表中，客户信息存储在customers表中。

为了执行这个操作,遵循下面的步骤。

(1) 从 customers 表中检索客户列表。

(2) 对于检索出的每个客户,统计其在 orders 表中的订单数目。

正如前两章所述,可使用 SELECT COUNT (*) 对表中的行进行计数,并且通过提供一条 WHERE 子句来过滤某个特定的客户ID,可仅对该客户的订单进行计数。

```
MariaDB [test]> select cust_name,cust_state,(select count(*) from orders where
orders.cust_id=customers.cust_id) as orders from customers order by cust_name;
+----------------+------------+--------+
| cust_name      | cust_state | orders |
+----------------+------------+--------+
| Coyote Inc.    | MI         |      2 |
| E Fudd         | IL         |      1 |
| Mouse House    | OH         |      0 |
| Wascals        | IN         |      1 |
| Yosemite Place | AZ         |      1 |
+----------------+------------+--------+
5 rows in set (0.00 sec)
```

`cust_name` 、`cust_state` 和 `orders` 。 `orders` 是一个计算字段,它是由圆括号中的子查询建立的。该子查询对检索出的每个客户执行一次。在此例子中,该子查询执行了5次,因为检索出了5个客户。

分析子查询中的 WHERE 子句与前面使用的 WHERE 子句稍有不同,因为它使用了完全限定列名(在前面提到过)。

相关子查询**(correlated subquery)** 涉及外部查询的子查询。这种类型的子查询称为相关子查询。任何时候只要列名可能有多义性,就必须使用这种语法(表名和列名由一个句点分隔)。为什么这样?

> 我们来看看如果不使用完全限定的列名会发生什么情况

```
MariaDB [test]> select cust_name,cust_state,(select count(*) from orders where cust_id=cust_id)
as orders from customers order by cust_name;+----------------+------------+--------+
| cust_name      | cust_state | orders |
+----------------+------------+--------+
| Coyote Inc.    | MI         |      5 |
| E Fudd         | IL         |      5 |
| Mouse House    | OH         |      5 |
| Wascals        | IN         |      5 |
| Yosemite Place | AZ         |      5 |
+----------------+------------+--------+
5 rows in set (0.00 sec)
```

显然,返回的结果不正确(请比较前面的结果),那么,为什么会这样呢?有两个 `cust_id` 列,一个在 `customers` 中,另一个在`orders` 中,需要比较这两个列以正确地把订单与它们相应的顾客匹配。如果不完全限定列名,MySQL将假定你是对 `orders` 表中的 `cust_id` 进行自身比较。而 `SELECT COUNT(*) FROM orders WHERE cust_id = cust_id;`总是返回 `orders` 表中的订单总数(因为MySQL查看每个订单的 `cust_id`是否与本身匹配,当然,它们总是匹配的)。

虽然子查询在构造这种 SELECT 语句时极有用,但必须注意限制有歧义性的列名。

不止一种解决方案 正如本章前面所述,虽然这里给出的样例代码运行良好,但它并不是解决这种数据检索的最有效的方法。在后面的章节中我们还要遇到这个例子。

逐渐增加子查询来建立查询 用子查询测试和调试查询很有技巧性,特别是在这些语句的复杂性不断增加的情况下更是如此。用子查询建立(和测试)查询的最可靠的方法是逐渐进行,这与MySQL处理它们的方法非常相同。首先,建立和测试最内层的查询。然后,用硬编码数据建立和测试外层查询,并且仅在确认它正常后才嵌入子查询。这时,再次测试它。对于要增加的每个查询,重复这些步骤。这样做仅给构造查询增加了一点点时间,但节省了以后(找出查询为什么不正常)的大量时间,并且极大地提高了查询一开始就正常工作的可能性。

> 小结
>
> > 本章学习了什么是子查询以及如何使用它们。子查询最常见的使用是在 WHERE 子句的 IN 操作符中,以及用来填充计算列。我们举了这两种操作类型的例子。

联结表

本章将介绍什么是联结,为什么要使用联结,如何编写使用联结的SELECT 语句。

> 联结

SQL最强大的功能之一就是能在数据检索查询的执行中联结(join)表。联结是利用SQL的 SELECT 能执行的最重要的操作,很好地理解联结及其语法是学习SQL的一个极为重要的组成部分。

在能够有效地使用联结前,必须了解关系表以及关系数据库设计的一些基础知识。下面的介绍并不是这个内容的全部知识,但作为入门已经足够了。

1.关系表

理解关系表的最好方法是来看一个现实世界中的例子。假如有一个包含产品目录的数据库表,其中每种类别的物品占一行。对于每种物品要存储的信息包括产品描述和价格,以及生产该产品的供应商信息。

现在,假如有由同一供应商生产的多种物品,那么在何处存储供应商信息(如,供应商名、地址、联系方法等)呢?将这些数据与产品信息分开存储的理由如下。

- 因为同一供应商生产的每个产品的供应商信息都是相同的,对每个产品重复此信息既浪费时间又浪费存储空间。
- 如果供应商信息改变(例如,供应商搬家或电话号码变动),只需改动一次即可。
- 如果有重复数据(即每种产品都存储供应商信息),很难保证每次输入该数据的方式都相同。不一致的数据在报表中很难利用。

关键是,相同数据出现多次决不是一件好事,此因素是关系数据库设计的基础。关系表的设计就是要保证把信息分解成多个表,一类数据一个表。各表通过某些常用的值(即关系设计中的关系(relational))互相关联。

在这个例子中,可建立两个表,一个存储供应商信息,另一个存储产品信息。 vendors 表包含所有供应商信息,每个供应商占一行,每个供应商具有唯一的标识。此标识称为主键(primary key)(在第1章中首次提到),可以是供应商ID或任何其他唯一值。

products 表只存储产品信息,它除了存储供应商ID(vendors 表的主键)外不存储其他供应商信息。 vendors 表的主键又叫作 products 的外键,它将 vendors 表与 products 表关联,利用供应商ID能从 vendors 表中找出相应供应商的详细信息。

外键**(foreign key)** 外键为某个表中的一列,它包含另一个表的主键值,定义了两个表之间的关系。

这样做的好处如下:

- 供应商信息不重复,从而不浪费时间和空间;
- 如果供应商信息变动,可以只更新 vendors 表中的单个记录,相关表中的数据不用改动;
- 由于数据无重复,显然数据是一致的,这使得处理数据更简单。

总之,关系数据可以有效地存储和方便地处理。因此,关系数据库的可伸缩性远比非关系数据库要好。

可伸缩性**(scale)** 能够适应不断增加的工作量而不失败。设计良好的数据库或应用程序称之为可伸缩性好(scale well)。

2.为什么要使用联结

正如所述,分解数据为多个表能更有效地存储,更方便地处理,并且具有更大的可伸缩性。但这些好处是有代价的。

如果数据存储在多个表中,怎样用单条 SELECT 语句检索出数据?

答案是使用联结。简单地说,联结是一种机制,用来在一条 SELECT语句中关联表,因此称之为联结。使用特殊的语法,可以联结多个表返回一组输出,联结在运行时关联表中正确的行。

维护引用完整性 重要的是,要理解联结不是物理实体。换句话说,它在实际的数据库表中不存在。联结由MySQL根据需要建立,它存在于查询的执行当中。在使用关系表时,仅在关系列中插入合法的数据非常重要。回到这里的例子,如果在 products 表中插入拥有非法供应商ID(即没有在 vendors 表中出现)的供应商生产的产品,则这些产品是不可访问的,因为它们没有关联到某个供应商。为防止这种情况发生,可指示MySQL只允许在 products 表的供应商ID列中出现合法值(即出现在 vendors 表中的供应商)。这就是维护引用完整性,它是通过在表的定义中指定主键和外键来实现的。

> 创建联结

联结的创建非常简单,规定要联结的所有表以及它们如何关联即可。

> 检索出所有的供应商名`vend_name`，以及供应商提供的商品名`prod_name`和价格`prod_price`。

```
MariaDB [test]> describe vendors;
+--------------+----------+------+-----+---------+----------------+
| Field        | Type     | Null | Key | Default | Extra          |
+--------------+----------+------+-----+---------+----------------+
| vend_id      | int(11)  | NO   | PRI | NULL    | auto_increment |
| vend_name    | char(50) | NO   |     | NULL    |                |
| vend_address | char(50) | YES  |     | NULL    |                |
| vend_city    | char(50) | YES  |     | NULL    |                |
| vend_state   | char(5)  | YES  |     | NULL    |                |
| vend_zip     | char(10) | YES  |     | NULL    |                |
| vend_country | char(50) | YES  |     | NULL    |                |
+--------------+----------+------+-----+---------+----------------+
7 rows in set (0.00 sec)

MariaDB [test]> describe products;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| prod_id    | char(10)     | NO   | PRI | NULL    |       |
| vend_id    | int(11)      | NO   | MUL | NULL    |       |
| prod_name  | char(255)    | NO   |     | NULL    |       |
| prod_price | decimal(8,2) | NO   |     | NULL    |       |
| prod_desc  | text         | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)

MariaDB [test]> select vend_name,prod_name,prod_price from vendors,products where vendors.vend_id
= products.vend_id order by vend_name,prod_name;
+-------------+----------------+------------+
| vend_name   | prod_name      | prod_price |
+-------------+----------------+------------+
| ACME        | Bird seed      |      10.00 |
| ACME        | Carrots        |       2.50 |
| ACME        | Detonator      |      13.00 |
| ACME        | Safe           |      50.00 |
| ACME        | Sling          |       4.49 |
| ACME        | TNT (1 stick)  |       2.50 |
| ACME        | TNT (5 sticks) |      10.00 |
| Anvils R Us | .5 ton anvil   |       5.99 |
| Anvils R Us | 1 ton anvil    |       9.99 |
| Anvils R Us | 2 ton anvil    |      14.99 |
| Jet Set     | JetPack 1000   |      35.00 |
| Jet Set     | JetPack 2000   |      55.00 |
| LT Supplies | Fuses          |       3.42 |
| LT Supplies | Oil can        |       8.99 |
+-------------+----------------+------------+
14 rows in set (0.00 sec)
```

完全限定列名 在引用的列可能出现二义性时,必须使用完全限定列名(用一个点分隔的表名和列名)。如果引用一个没有用表名限制的具有二义性的列名,MySQL将返回错误。

1.WHERE子句的重要性

利用 WHERE 子句建立联结关系似乎有点奇怪,但实际上,有一个很充分的理由。请记住,在一条 SELECT 语句中联结几个表时,相应的关系是在运行中构造的。在数据库表的定义中不存在能指示MySQL如何对表进行联结的东西。你必须自己做这件事情。在联结两个表时,你实际上做的是将第一个表中的每一行与第二个表中的每一行配对。WHERE 子句作为过滤条件,它只包含那些匹配给定条件(这里是联结条件)的行。没有WHERE 子句,第一个表中的每个行将与第二个表中的每个行配对,而不管它们逻辑上是否可以配在一起。

笛卡儿积**(cartesian product)** 由没有联结条件的表关系返回的结果为笛卡儿积。检索出的行的数目将是第一个表中的行数乘以第二个表中的行数。

为理解这一点,请看下面的 SELECT 语句及其输出:

```
MariaDB [test]> select vend_name,prod_name,prod_price from vendors,products order by
vend_name,prod_name;
+----------------+----------------+------------+
| vend_name      | prod_name      | prod_price |
+----------------+----------------+------------+
| ACME           | .5 ton anvil   |       5.99 |
| ACME           | 1 ton anvil    |       9.99 |
| ACME           | 2 ton anvil    |      14.99 |
| ACME           | Bird seed      |      10.00 |
| ACME           | Carrots        |       2.50 |
| ACME           | Detonator      |      13.00 |
| ACME           | Fuses          |       3.42 |
| ACME           | JetPack 1000   |      35.00 |
| ACME           | JetPack 2000   |      55.00 |
| ACME           | Oil can        |       8.99 |
| ACME           | Safe           |      50.00 |
| ACME           | Sling          |       4.49 |
| ACME           | TNT (1 stick)  |       2.50 |
| ACME           | TNT (5 sticks) |      10.00 |
| Anvils R Us    | .5 ton anvil   |       5.99 |
| Anvils R Us    | 1 ton anvil    |       9.99 |
| Anvils R Us    | 2 ton anvil    |      14.99 |
| Anvils R Us    | Bird seed      |      10.00 |
| Anvils R Us    | Carrots        |       2.50 |
| Anvils R Us    | Detonator      |      13.00 |
| Anvils R Us    | Fuses          |       3.42 |
| Anvils R Us    | JetPack 1000   |      35.00 |
| Anvils R Us    | JetPack 2000   |      55.00 |
| Anvils R Us    | Oil can        |       8.99 |
| Anvils R Us    | Safe           |      50.00 |
| Anvils R Us    | Sling          |       4.49 |
| Anvils R Us    | TNT (1 stick)  |       2.50 |
| Anvils R Us    | TNT (5 sticks) |      10.00 |
| Furball Inc.   | .5 ton anvil   |       5.99 |
| Furball Inc.   | 1 ton anvil    |       9.99 |
| Furball Inc.   | 2 ton anvil    |      14.99 |
| Furball Inc.   | Bird seed      |      10.00 |
| Furball Inc.   | Carrots        |       2.50 |
| Furball Inc.   | Detonator      |      13.00 |
| Furball Inc.   | Fuses          |       3.42 |
| Furball Inc.   | JetPack 1000   |      35.00 |
| Furball Inc.   | JetPack 2000   |      55.00 |
| Furball Inc.   | Oil can        |       8.99 |
| Furball Inc.   | Safe           |      50.00 |
| Furball Inc.   | Sling          |       4.49 |
| Furball Inc.   | TNT (1 stick)  |       2.50 |
| Furball Inc.   | TNT (5 sticks) |      10.00 |
| Jet Set        | .5 ton anvil   |       5.99 |
| Jet Set        | 1 ton anvil    |       9.99 |
| Jet Set        | 2 ton anvil    |      14.99 |
| Jet Set        | Bird seed      |      10.00 |
| Jet Set        | Carrots        |       2.50 |
| Jet Set        | Detonator      |      13.00 |
```

```
| Jet Set        | Fuses          |       3.42 |
| Jet Set        | JetPack 1000   |      35.00 |
| Jet Set        | JetPack 2000   |      55.00 |
| Jet Set        | Oil can        |       8.99 |
| Jet Set        | Safe           |      50.00 |
| Jet Set        | Sling          |       4.49 |
| Jet Set        | TNT (1 stick)  |       2.50 |
| Jet Set        | TNT (5 sticks) |      10.00 |
| Jouets Et Ours | .5 ton anvil   |       5.99 |
| Jouets Et Ours | 1 ton anvil    |       9.99 |
| Jouets Et Ours | 2 ton anvil    |      14.99 |
| Jouets Et Ours | Bird seed      |      10.00 |
| Jouets Et Ours | Carrots        |       2.50 |
| Jouets Et Ours | Detonator      |      13.00 |
| Jouets Et Ours | Fuses          |       3.42 |
| Jouets Et Ours | JetPack 1000   |      35.00 |
| Jouets Et Ours | JetPack 2000   |      55.00 |
| Jouets Et Ours | Oil can        |       8.99 |
| Jouets Et Ours | Safe           |      50.00 |
| Jouets Et Ours | Sling          |       4.49 |
| Jouets Et Ours | TNT (1 stick)  |       2.50 |
| Jouets Et Ours | TNT (5 sticks) |      10.00 |
| LT Supplies    | .5 ton anvil   |       5.99 |
| LT Supplies    | 1 ton anvil    |       9.99 |
| LT Supplies    | 2 ton anvil    |      14.99 |
| LT Supplies    | Bird seed      |      10.00 |
| LT Supplies    | Carrots        |       2.50 |
| LT Supplies    | Detonator      |      13.00 |
| LT Supplies    | Fuses          |       3.42 |
| LT Supplies    | JetPack 1000   |      35.00 |
| LT Supplies    | JetPack 2000   |      55.00 |
| LT Supplies    | Oil can        |       8.99 |
| LT Supplies    | Safe           |      50.00 |
| LT Supplies    | Sling          |       4.49 |
| LT Supplies    | TNT (1 stick)  |       2.50 |
| LT Supplies    | TNT (5 sticks) |      10.00 |
+----------------+----------------+------------+
84 rows in set (0.00 sec)
```

从上面的输出中可以看到,相应的笛卡儿积不是我们所想要的。这里返回的数据用每个供应商匹配了每个产品,它包括了供应商不正确的产品。实际上有的供应商根本就没有产品。

分析

不要忘了 **WHERE** 子句 应该保证所有联结都有 WHERE 子句,否则MySQL将返回比想要的数据多得多的数据。同理,应该保证 WHERE 子句的正确性。不正确的过滤条件将导致MySQL返回不正确的数据。

叉联结 有时我们会听到返回称为叉联结(cross join)的笛卡儿积的联结类型

2.内部联结

目前为止所用的联结称为等值联结(equijoin),它基于两个表之间的
相等测试。这种联结也称为内部联结。其实,对于这种联结可以使用稍
微不同的语法来明确指定联结的类型。下面的 SELECT 语句返回与前面例
子完全相同的数据:

```
MariaDB [test]> select vend_name,prod_name,prod_price from vendors inner join products on
vendors.vend_id = products.vend_id;
+-------------+----------------+------------+
| vend_name   | prod_name      | prod_price |
+-------------+----------------+------------+
| Anvils R Us | .5 ton anvil   |       5.99 |
| Anvils R Us | 1 ton anvil    |       9.99 |
| Anvils R Us | 2 ton anvil    |      14.99 |
| LT Supplies | Fuses          |       3.42 |
| LT Supplies | Oil can        |       8.99 |
| ACME        | Detonator      |      13.00 |
| ACME        | Bird seed      |      10.00 |
| ACME        | Carrots        |       2.50 |
| ACME        | Safe           |      50.00 |
| ACME        | Sling          |       4.49 |
| ACME        | TNT (1 stick)  |       2.50 |
| ACME        | TNT (5 sticks) |      10.00 |
| Jet Set     | JetPack 1000   |      35.00 |
| Jet Set     | JetPack 2000   |      55.00 |
+-------------+----------------+------------+
14 rows in set (0.00 sec)
```

此语句中的 `SELECT` 与前面的`SELECT` 语句相同,但 `FROM` 子句不同。这里,两个表之间的关系是 `FROM` 子句的组成部分,以 `INNER JOIN` 指定。在使用这种语法时,联结条件用特定的 `ON` 子句而不是 `WHERE`子句给出。传递给 `ON` 的实际条件与传递给 `WHERE` 的相同。

使用哪种语法 ANSI SQL规范首选 `INNER JOIN` 语法。此外,尽管使用 `WHERE` 子句定义联结的确比较简单,但是使用明确的联结语法能够确保不会忘记联结条件,有时候这样做也能影响性能。

3.联结多个表

SQL对一条 SELECT 语句中可以联结的表的数目没有限制。创建联结的基本规则也相同。首先列出所有表,然后定义表之间的关系。例如:

> 检索出订单号为20005的所有商品的商品名称，商品的供应商名称，商品的价格，商品的数量（`prod_name,vend_name,prod_price,quantit`）。

```
MariaDB [test]> select prod_name,vend_name,prod_price,quantity from orderitems,products,vendors
where products.vend_id = vendors.vend_id and orderitems.prod_id  = products.prod_id and order_num
= 20005;
+----------------+-------------+------------+----------+
| prod_name      | vend_name   | prod_price | quantity |
+----------------+-------------+------------+----------+
| .5 ton anvil   | Anvils R Us |       5.99 |       10 |
| 1 ton anvil    | Anvils R Us |       9.99 |        3 |
| TNT (5 sticks) | ACME        |      10.00 |        5 |
| Bird seed      | ACME        |      10.00 |        1 |
+----------------+-------------+------------+----------+
4 rows in set (0.00 sec)
```

此例子显示编号为 20005 的订单中的物品。订单物品存储在orderitems 表中。每个产品按其产品ID存储,它引用products表中的产品。这些产品通过供应商ID联结到 vendors 表中相应的供应商,供应商ID存储在每个产品的记录中。这里的 FROM 子句列出了3个表,而WHERE 子句定义了这两个联结条件,而第三个联结条件用来过滤出订单 20005 中的物品。

性能考虑 MySQL在运行时关联指定的每个表以处理联结。这种处理可能是非常耗费资源的,因此应该仔细,不要联结不必要的表。联结的表越多,性能下降越厉害。

现在可以回顾一下前面章中的例子了。该例子如下所示

> 返回订购产品 TNT2 的客户列表

```
MariaDB [test]> select cust_name from customers where cust_id in (select cust_id from orders
where order_num in (select order_num from orderitems where prod_id='TNT2'));
+----------------+
| cust_name      |
+----------------+
| Coyote Inc.    |
| Yosemite Place |
+----------------+
2 rows in set (0.00 sec)
```

子查询并不总是执行复杂 SELECT 操作的最有效的方法,下面是使用联结的相同查询:

```
MariaDB [test]> select cust_name from customers where cust_id in (select cust_id from orders
where order_num in (select order_num from orderitems where prod_id='TNT2'));
+----------------+
| cust_name      |
+----------------+
| Coyote Inc.    |
| Yosemite Place |
+----------------+
2 rows in set (0.00 sec)
```

这个查询中返回数据需要使用3个表。但这里我们没有在嵌套子查询中使用它们,而是使用了两个联结。这里有3个WHERE 子句条件。前两个关联联结中的表,后一个过滤产品 TNT2的数据。

多做实验 正如所见,为执行任一给定的SQL操作,一般存在不止一种方法。很少有绝对正确或绝对错误的方法。性能可能会受操作类型、表中数据量、是否存在索引或键以及其他一些条件的影响。因此,有必要对不同的选择机制进行实验,以找出最适合具体情况的方法。

> 小结
>
> > 联结是SQL中最重要最强大的特性,有效地使用联结需要对关系数据库设计有基本的了解。本章随着对联结的介绍讲述了关系数据库设计的一些基本知识,包括等值联结(也称为内部联结)这种最经常使用的联结形式。下一章将介绍如何创建其他类型的联结。

创建高级联结

本章将讲解另外一些联结类型(包括它们的含义和使用方法),介绍如何对被联结的表使用表别名和聚集函数。

> 使用表别名

前面章节中介绍了如何使用别名引用被检索的表列。给列起别名的语法如下:

> 检索出供应商的名字和国家，并以指定格式显示【`vend_name`(`vend_country`)】

```
MariaDB [test]> select concat(rtrim(vend_name),'(',rtrim(vend_country),')') as vend_tile from
vendors order by vend_name;
+-------------------------+
| vend_tile               |
+-------------------------+
| ACME(USA)               |
| Anvils R Us(USA)        |
| Furball Inc.(USA)       |
| Jet Set(England)        |
| Jouets Et Ours(France)  |
| LT Supplies(USA)        |
+-------------------------+
6 rows in set (0.00 sec)
```

别名除了用于列名和计算字段外, SQL还允许给表名起别名。这样做有两个主要理由:

- 缩短SQL语句;
- 允许在单条 SELECT 语句中多次使用相同的表。

请看下面的 SELECT 语句。它与前一章的例子中所用的语句基本相同,
但改成了使用别名:

> 返回订购产品 TNT2 的客户列表

```
MariaDB [test]> select cust_name,cust_contact from customers as c,orders as o,orderitems as oi
where c.cust_id = o.cust_id and oi.order_num = o.order_num and prod_id = 'TNT2';
+----------------+--------------+
| cust_name      | cust_contact |
+----------------+--------------+
| Coyote Inc.    | Y Lee        |
| Yosemite Place | Y Sam        |
+----------------+--------------+

2 rows in set (0.00 sec)
```

可以看到, FROM 子句中3个表全都具有别名。 customers AS c 建立 c 作为 customers 的别名,等等。这使得能使用省写的 c 而不是全名 customers 。在此例子中,表别名只用于 WHERE 子句。但是,表别名不仅能用于 WHERE 子句,它还可以用于 SELECT 的列表、 ORDER BY 子句以及语句的其他部分。

应该注意,表别名只在查询执行中使用。与列别名不一样,表别名不返回到客户机。

> 使用不同类型的联结

迄今为止,我们使用的只是称为内部联结或等值联结(equijoin)的简单联结。现在来看3种其他联结,它们分别是自联结、自然联结和外部联结。

1.自联结

如前所述,使用表别名的主要原因之一是能在单条 SELECT 语句中不止一次引用相同的表。下面举一个例子。

假如你发现某物品(其ID为 DTNTR)存在问题,因此想知道生产该物品的供应商生产的其他物品是否也存在这些问题。此查询要求首先找到生产ID为 DTNTR 的物品的供应商,然后找出这个供应商生产的其他物品。

下面是解决此问题的一种方法:

> 查找商品 `prod_id ='DTNTR'` 的供应商生产的商品名和id号`prod_id,prod_name`

```
MariaDB [test]> select prod_id,prod_name from products where vend_id = (select vend_id from
products where prod_id ='DTNTR');
+---------+----------------+
| prod_id | prod_name      |
+---------+----------------+
| DTNTR   | Detonator      |
| FB      | Bird seed      |
| FC      | Carrots        |
| SAFE    | Safe           |
| SLING   | Sling          |
| TNT1    | TNT (1 stick)  |
| TNT2    | TNT (5 sticks) |
+---------+----------------+
7 rows in set (0.00 sec)
```

这是第一种解决方案,它使用了子查询。内部的 `SELECT` 语句做了一个 简单的检索 ,返回生产 ID`prod_id` 为 `DTNTR` 的 物品供应商 的`vend_id` 。该ID用于外部查询的 `WHERE` 子句中,以便检索出这个供应商生产的所有物品

现在来看使用联结的相同查询:

```
MariaDB [test]> select p1.prod_id,p1.prod_name from products as p1,products as p2 where
p1.vend_id = p2.vend_id and p2.prod_id = 'DTNTR';
+---------+----------------+
| prod_id | prod_name      |
+---------+----------------+
| DTNTR   | Detonator      |
| FB      | Bird seed      |
| FC      | Carrots        |
| SAFE    | Safe           |
| SLING   | Sling          |
| TNT1    | TNT (1 stick)  |
| TNT2    | TNT (5 sticks) |
+---------+----------------+
7 rows in set (0.00 sec)
```

此查询中需要的两个表实际上是相同的表,因此 `products` 表在 `FROM` 子句中出现了两次。虽然这是完全合法的,但对 `products` 的引用具有二义性,因为MySQL不知道你引用的是 `products` 表中的哪个实例。

为解决此问题,使用了表别名。 `products` 的第一次出现为别名 `p1` ,第二次出现为别名 `p2` 。现在可以将这些别名用作表名。例如, `SELECT` 语句使用 `p1` 前缀明确地给出所需列的全名。如果不这样, `MySQL` 将返回错误,因为分别存在两个名为 `prod_id` 、 `prod_name` 的列。MySQL不知道想要的是哪一个列(即使它们事实上是同一个列)。 `WHERE` (通过匹配 `p1` 中的 `vend_id` 和 `p2` 中的 `vend_id`)首先联结两个表,然后按第二个表中的 `prod_id` 过滤数据,返回所需的数据。

用自联结而不用子查询 自联结通常作为外部语句用来替代从相同表中检索数据时使用的子查询语句。虽然最终的结果是相同的,但有时候处理联结远比处理子查询快得多。应该试一下两种方法,以确定哪一种的性能更好。

2.自然联结

无论何时对表进行联结,应该至少有一个列出现在不止一个表中(被联结的列)。标准的联结(前一章中介绍的内部联结)返回所有数据,甚至相同的列多次出现。自然联结排除多次出现,使每个列只返回一次。

怎样完成这项工作呢?答案是,系统不完成这项工作,由你自己完成它。自然联结是这样一种联结,其中你只能选择那些唯一的列。这一般是通过对表使用通配符(SELECT *),对所有其他表的列使用明确的子集来完成的。下面举一个例子:

```
MariaDB [test]> select c.*,o.order_num,o.order_date,oi.prod_id,oi.quantity,oi.item_price from
customers as c,orders as o,orderitems as oi where c.cust_id=o.cust_id and
oi.order_num=o.order_num and prod_id='FB';
+---------+-------------+----------------+-----------+------------+----------+--------------+----
---------+-----------------+-----------+---------------------+---------+----------+------------+
| cust_id | cust_name   | cust_address   | cust_city | cust_state | cust_zip | cust_country |
cust_contact | cust_email      | order_num | order_date          | prod_id | quantity |
item_price |
+---------+-------------+----------------+-----------+------------+----------+--------------+----
---------+-----------------+-----------+---------------------+---------+----------+------------+
|   10001 | Coyote Inc. | 200 Maple Lane | Detroit   | MI         | 44444    | USA          | Y
Lee         | ylee@coyote.com |     20005 | 2005-09-01 00:00:00 | FB      |        1 |      10.00
|
|   10001 | Coyote Inc. | 200 Maple Lane | Detroit   | MI         | 44444    | USA          | Y
Lee         | ylee@coyote.com |     20009 | 2005-10-08 00:00:00 | FB      |        1 |      10.00
|
+---------+-------------+----------------+-----------+------------+----------+--------------+----
---------+-----------------+-----------+---------------------+---------+----------+------------+
2 rows in set (0.00 sec)
```

在这个例子中,通配符只对第一个表使用。所有其他列明确列出,所以没有重复的列被检索出来。

事实上,迄今为止我们建立的每个内部联结都是自然联结,很可能我们永远都不会用到不是自然联结的内部联结。

3.外部联结

许多联结将一个表中的行与另一个表中的行相关联。但有时候会需要包含没有关联行的那些行。例如,可能需要使用联结来完成以下工作:

- 对每个客户下了多少订单进行计数,包括那些至今尚未下订单的客户;
- 列出所有产品以及订购数量,包括没有人订购的产品;
- 计算平均销售规模,包括那些至今尚未下订单的客户。

在上述例子中,联结包含了那些在相关表中没有关联行的行。这种类型的联结称为外部联结。

下面的 SELECT 语句给出一个简单的内部联结。

> 它检索所有客户及其订单

```
MariaDB [test]> select customers.cust_id,orders.order_num from customers inner join orders on
customers.cust_id=orders.cust_id;
+---------+-----------+
| cust_id | order_num |
+---------+-----------+
|   10001 |     20005 |
|   10001 |     20009 |
|   10003 |     20006 |
|   10004 |     20007 |
|   10005 |     20008 |
+---------+-----------+
5 rows in set (0.00 sec)
```

> 外部联结语法类似。为了检索所有客户,包括那些没有订单的客户,可如下进行:

```
MariaDB [test]> select customers.cust_id,orders.order_num from customers left outer join orders
on customers.cust_id=orders.cust_id;
+---------+-----------+
| cust_id | order_num |
+---------+-----------+
|   10001 |     20005 |
|   10001 |     20009 |
|   10002 |      NULL |
|   10003 |     20006 |
|   10004 |     20007 |
|   10005 |     20008 |
+---------+-----------+
6 rows in set (0.00 sec)
```

类似于上一章中所看到的内部联结,这条 `SELECT` 语句使用了关键字 `OUTER JOIN` 来指定联结的类型(而不是在 `WHERE` 子句中指定)。但是,与内部联结关联两个表中的行不同的是,外部联结还包括没有关联行的行。在使用 `OUTER JOIN` 语法时,必须使用 `RIGHT` 或 `LEFT` 关键字指定包括其所有行的表(`RIGHT` 指出的是 `OUTER JOIN` 右边的表,而 `LEFT` 指出的是 `OUTER JOIN` 左边的表)。

上面的例子使用 `LEFT OUTER JOIN` 从 `FROM` 子句的左边表(`customers` 表)中选择所有行。为了从右边的表中选择所有行,应该使用 `RIGHT OUTER JOIN` ,如下例所示:

```
MariaDB [test]> select customers.cust_id,orders.order_num from customers right outer join orders
on orders.cust_id = customers.cust_id;
+---------+-----------+
| cust_id | order_num |
+---------+-----------+
|   10001 |     20005 |
|   10001 |     20009 |
|   10003 |     20006 |
|   10004 |     20007 |
|   10005 |     20008 |
+---------+-----------+
5 rows in set (0.00 sec)
```

没有 `*=` 操作符 MySQL不支持简化字符 `*=` 和 `=*` 的使用,这两种操作符在其他DBMS中是很流行的。

外部联结的类型 存在两种基本的外部联结形式:左外部联结和右外部联结。它们之间的唯一差别是所关联的表的顺序不同。换句话说,左外部联结可通过颠倒 FROM 或 WHERE 子句中表的顺序转换为右外部联结。因此,两种类型的外部联结可互换使用,而究竟使用哪一种纯粹是根据方便而定。

> 使用带聚集函数的联结

正如第12章所述,聚集函数用来汇总数据。虽然至今为止聚集函数的所有例子只是从单个表汇总数据,但这些函数也可以与联结一起使用。

为说明这一点,请看一个例子。

> > 如果要检索所有客户及每个客户所下的订单数,下面使用了 COUNT() 函数的代码可完成此工作:

```
MariaDB [test]> select customers.cust_name,customers.cust_id,count(orders.order_num) as num_ord
from customers inner join orders on customers.cust_id = orders.cust_id group by
customers.cust_id;
+----------------+---------+---------+
| cust_name      | cust_id | num_ord |
+----------------+---------+---------+
| Coyote Inc.    |   10001 |       2 |
| Wascals        |   10003 |       1 |
| Yosemite Place |   10004 |       1 |
| E Fudd         |   10005 |       1 |
+----------------+---------+---------+
4 rows in set (0.00 sec)
```

此`SELECT`语句使用`INNER JOIN`将`customers`和`orders`表互相关联。`GROUP BY`子句按客户分组数据,因此,函数调用`COUNT (orders.order_num)`对每个客户的订单计数,将它作为`num_ord`返回。

```
MariaDB [test]> select * from customers;
+---------+----------------+---------------------+-----------+------------+----------+-----------
---+--------------+----------------------+
| cust_id | cust_name      | cust_address        | cust_city | cust_state | cust_zip |
cust_country | cust_contact | cust_email           |
+---------+----------------+---------------------+-----------+------------+----------+-----------
---+--------------+----------------------+
|   10001 | Coyote Inc.    | 200 Maple Lane      | Detroit   | MI         | 44444    | USA
   | Y Lee        | ylee@coyote.com     |
|   10002 | Mouse House    | 333 Fromage Lane    | Columbus  | OH         | 43333    | USA
   | Jerry Mouse  | NULL                |
|   10003 | Wascals        | 1 Sunny Place       | Muncie    | IN         | 42222    | USA
   | Jim Jones    | rabbit@wascally.com |
|   10004 | Yosemite Place | 829 Riverside Drive | Phoenix   | AZ         | 88888    | USA
   | Y Sam        | sam@yosemite.com    |
|   10005 | E Fudd         | 4545 53rd Street    | Chicago   | IL         | 54545    | USA
   | E Fudd       | NULL                |
+---------+----------------+---------------------+-----------+------------+----------+-----------
---+--------------+----------------------+
5 rows in set (0.00 sec)

MariaDB [test]> select  * from orders;
+-----------+---------------------+---------+
| order_num | order_date          | cust_id |
+-----------+---------------------+---------+
|     20005 | 2005-09-01 00:00:00 |   10001 |
|     20006 | 2005-09-12 00:00:00 |   10003 |
|     20007 | 2005-09-30 00:00:00 |   10004 |
|     20008 | 2005-10-03 00:00:00 |   10005 |
|     20009 | 2005-10-08 00:00:00 |   10001 |
+-----------+---------------------+---------+
5 rows in set (0.00 sec)
# 内部联结或等值联结(equijoin)的简单联结
MariaDB [test]> select orders.*,customers.* from customers inner join orders on customers.cust_id
= orders.cust_id;
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+----------------------+
| order_num | order_date          | cust_id | cust_id | cust_name      | cust_address        |
cust_city | cust_state | cust_zip | cust_country | cust_contact | cust_email           |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+----------------------+
|     20005 | 2005-09-01 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |
|     20006 | 2005-09-12 00:00:00 |   10003 |   10003 | Wascals        | 1 Sunny Place       |
Muncie    | IN         | 42222    | USA          | Jim Jones    | rabbit@wascally.com |
|     20007 | 2005-09-30 00:00:00 |   10004 |   10004 | Yosemite Place | 829 Riverside Drive |
Phoenix   | AZ         | 88888    | USA          | Y Sam        | sam@yosemite.com    |
|     20008 | 2005-10-03 00:00:00 |   10005 |   10005 | E Fudd         | 4545 53rd Street    |
Chicago   | IL         | 54545    | USA          | E Fudd       | NULL                |
|     20009 | 2005-10-08 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |


+-----------+---------------------+---------+---------+----------------+---------------------+---
```

```
--------+------------+----------+--------------+--------------+---------------------+
5 rows in set (0.00 sec)
# 内部联结或等值联结(equijoin)的简单联结
MariaDB [test]> select orders.*,customers.* from orders inner join customers on customers.cust_id
= orders.cust_id;
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
| order_num | order_date          | cust_id | cust_id | cust_name      | cust_address        |
cust_city | cust_state | cust_zip | cust_country | cust_contact | cust_email          |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
|     20005 | 2005-09-01 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |
|     20009 | 2005-10-08 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |
|     20006 | 2005-09-12 00:00:00 |   10003 |   10003 | Wascals        | 1 Sunny Place       |
Muncie    | IN         | 42222    | USA          | Jim Jones    | rabbit@wascally.com |
|     20007 | 2005-09-30 00:00:00 |   10004 |   10004 | Yosemite Place | 829 Riverside Drive |
Phoenix   | AZ         | 88888    | USA          | Y Sam        | sam@yosemite.com    |
|     20008 | 2005-10-03 00:00:00 |   10005 |   10005 | E Fudd         | 4545 53rd Street    |
Chicago   | IL         | 54545    | USA          | E Fudd       | NULL                |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
5 rows in set (0.00 sec)
# 左外部联结
MariaDB [test]> select orders.*,customers.* from customers left outer join orders on
customers.cust_id = orders.cust_id;
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
| order_num | order_date          | cust_id | cust_id | cust_name      | cust_address        |
cust_city | cust_state | cust_zip | cust_country | cust_contact | cust_email          |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
|     20005 | 2005-09-01 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |
|     20006 | 2005-09-12 00:00:00 |   10003 |   10003 | Wascals        | 1 Sunny Place       |
Muncie    | IN         | 42222    | USA          | Jim Jones    | rabbit@wascally.com |
|     20007 | 2005-09-30 00:00:00 |   10004 |   10004 | Yosemite Place | 829 Riverside Drive |
Phoenix   | AZ         | 88888    | USA          | Y Sam        | sam@yosemite.com    |
|     20008 | 2005-10-03 00:00:00 |   10005 |   10005 | E Fudd         | 4545 53rd Street    |
Chicago   | IL         | 54545    | USA          | E Fudd       | NULL                |
|     20009 | 2005-10-08 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |
|      NULL | NULL                |    NULL |   10002 | Mouse House    | 333 Fromage Lane    |
Columbus  | OH         | 43333    | USA          | Jerry Mouse  | NULL                |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
6 rows in set (0.00 sec)
# 左外部联结
MariaDB [test]> select orders.*,customers.* from orders left outer join customers on
customers.cust_id = orders.cust_id;


+-----------+---------------------+---------+---------+----------------+---------------------+---
```

```
--------+------------+----------+--------------+--------------+---------------------+
| order_num | order_date          | cust_id | cust_id | cust_name      | cust_address        |
cust_city | cust_state | cust_zip | cust_country | cust_contact | cust_email          |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
|     20005 | 2005-09-01 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |
|     20006 | 2005-09-12 00:00:00 |   10003 |   10003 | Wascals        | 1 Sunny Place       |
Muncie    | IN         | 42222    | USA          | Jim Jones    | rabbit@wascally.com |
|     20007 | 2005-09-30 00:00:00 |   10004 |   10004 | Yosemite Place | 829 Riverside Drive |
Phoenix   | AZ         | 88888    | USA          | Y Sam        | sam@yosemite.com    |
|     20008 | 2005-10-03 00:00:00 |   10005 |   10005 | E Fudd         | 4545 53rd Street    |
Chicago   | IL         | 54545    | USA          | E Fudd       | NULL                |
|     20009 | 2005-10-08 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
5 rows in set (0.00 sec)
# 右外部联结
MariaDB [test]> select orders.*,customers.* from customers right outer join orders on
customers.cust_id = orders.cust_id;
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
| order_num | order_date          | cust_id | cust_id | cust_name      | cust_address        |
cust_city | cust_state | cust_zip | cust_country | cust_contact | cust_email          |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
|     20005 | 2005-09-01 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |
|     20006 | 2005-09-12 00:00:00 |   10003 |   10003 | Wascals        | 1 Sunny Place       |
Muncie    | IN         | 42222    | USA          | Jim Jones    | rabbit@wascally.com |
|     20007 | 2005-09-30 00:00:00 |   10004 |   10004 | Yosemite Place | 829 Riverside Drive |
Phoenix   | AZ         | 88888    | USA          | Y Sam        | sam@yosemite.com    |
|     20008 | 2005-10-03 00:00:00 |   10005 |   10005 | E Fudd         | 4545 53rd Street    |
Chicago   | IL         | 54545    | USA          | E Fudd       | NULL                |
|     20009 | 2005-10-08 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
5 rows in set (0.00 sec)
# 右外部联结
MariaDB [test]> select orders.*,customers.* from orders right outer join customers on
customers.cust_id = orders.cust_id;
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
| order_num | order_date          | cust_id | cust_id | cust_name      | cust_address        |
cust_city | cust_state | cust_zip | cust_country | cust_contact | cust_email          |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
|     20005 | 2005-09-01 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA          | Y Lee        | ylee@coyote.com     |


|     20006 | 2005-09-12 00:00:00 |   10003 |   10003 | Wascals        | 1 Sunny Place       |
```

```
Muncie    | IN         | 42222    | USA           | Jim Jones     | rabbit@wascally.com |
|     20007 | 2005-09-30 00:00:00 |   10004 |   10004 | Yosemite Place | 829 Riverside Drive |
Phoenix   | AZ         | 88888    | USA           | Y Sam         | sam@yosemite.com    |
|     20008 | 2005-10-03 00:00:00 |   10005 |   10005 | E Fudd         | 4545 53rd Street    |
Chicago   | IL         | 54545    | USA           | E Fudd        | NULL                |
|     20009 | 2005-10-08 00:00:00 |   10001 |   10001 | Coyote Inc.    | 200 Maple Lane      |
Detroit   | MI         | 44444    | USA           | Y Lee         | ylee@coyote.com     |
|      NULL | NULL                |    NULL |   10002 | Mouse House    | 333 Fromage Lane    |
Columbus  | OH         | 43333    | USA           | Jerry Mouse   | NULL                |
+-----------+---------------------+---------+---------+----------------+---------------------+---
--------+------------+----------+--------------+--------------+---------------------+
6 rows in set (0.00 sec)
```

聚集函数也可以方便地与其他联结一起使用。请看下面的例子:

> 检索所有客户及每个客户所下的订单数(包括没有订单的客户)

```
MariaDB [test]> select customers.cust_name,customers.cust_id,count(orders.order_num) as num_ord
from customers left outer join orders on customers.cust_id = orders.cust_id group by
customers.cust_id;
+----------------+---------+---------+
| cust_name      | cust_id | num_ord |
+----------------+---------+---------+
| Coyote Inc.    |   10001 |       2 |
| Mouse House    |   10002 |       0 |
| Wascals        |   10003 |       1 |
| Yosemite Place |   10004 |       1 |
| E Fudd         |   10005 |       1 |
+----------------+---------+---------+
5 rows in set (0.00 sec)
```

这个例子使用左外部联结来包含所有客户,甚至包含那些没有任何下订单的客户。结果显示也包含了客户 `Mouse House` ,它有 0 个订单。

> 使用联结和联结条件

在总结关于联结的这两章前,有必要汇总一下关于联结及其使用的某些要点。

- 注意所使用的联结类型。一般我们使用内部联结,但使用外部联结也是有效的。
- 保证使用正确的联结条件,否则将返回不正确的数据。
- 应该总是提供联结条件,否则会得出笛卡儿积。
- 在一个联结中可以包含多个表,甚至对于每个联结可以采用不同的联结类型。虽然这样做是合法的,一般也很有用,但应该在一起测试它们前,分别测试每个联结。这将使故障排除更为简单。

> 小结
>
> > 本章是上一章关于联结的继续。本章从讲授如何以及为什么要使用别名开始,然后讨论不同的联结类型及对每种类型的联结使用的各种语法形式。我们还介绍了如何与联结一起使用聚集函数,以及在使用联结时应该注意的某些问题。

组合查询

本章讲述如何利用 UNION 操作符将多条 SELECT 语句组合成一个结果集。

> 组合查询

多数SQL查询都只包含从一个或多个表中返回数据的单条 SELECT 语句。MySQL也允许执行多个查询(多条SELECT 语句),并将结果作为单个查询结果集返回。这些组合查询通常称为并**(union)**或复合查询**(compound query)**。

有两种基本情况,其中需要使用组合查询:

- 在单个查询中从不同的表返回类似结构的数据;
- 对单个表执行多个查询,按单个查询返回数据。

组合查询和多个 WHERE 条件 多数情况下,组合相同表的两个查询完成的工作与具有多个 WHERE 子句条件的单条查询完成的工作相同。换句话说,任何具有多个 WHERE 子句的 SELECT 语句都可以作为一个组合查询给出,在以下段落中可以看到这一点。这两种技术在不同的查询中性能也不同。因此,应该试一下这两种技术,以确定对特定的查询哪一种性能更好。

> 创建组合查询

可用 UNION 操作符来组合数条SQL查询。利用 UNION ,可给出多条SELECT 语句,将它们的结果组合成单个结果集。

1.使用 UNION

UNION 的使用很简单。所需做的只是给出每条 SELECT 语句,在各条语句之间放上关键字 UNION 。

> 举一个例子,假如需要价格小于等于 5 的所有物品的一个列表,而且还想包括供应商 1001 和 1002 生产的所有物品(不考虑价格)。

当然,可以利用 WHERE 子句来完成此工作,不过这次我们将使用 UNION 。正如所述,创建 UNION 涉及编写多条 SELECT 语句。首先来看单条语句:

```
MariaDB [test]> select vend_id,prod_id,prod_price from products where prod_price <=5;
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1003 | FC      |       2.50 |
|    1002 | FU1     |       3.42 |
|    1003 | SLING   |       4.49 |
|    1003 | TNT1    |       2.50 |
+---------+---------+------------+
4 rows in set (0.00 sec)

MariaDB [test]> select vend_id,prod_id,prod_price from products where vend_id in (1001,1002);
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1001 | ANV01   |       5.99 |
|    1001 | ANV02   |       9.99 |
|    1001 | ANV03   |      14.99 |
|    1002 | FU1     |       3.42 |
|    1002 | OL1     |       8.99 |
+---------+---------+------------+
5 rows in set (0.00 sec)
```

第一条 SELECT 检索价格不高于 5 的所有物品。第二条 SELECT 使用 IN 找出供应商 1001 和 1002 生产的所有物品。为了组合这两条语句,按如下进行:

```
MariaDB [test]>  select vend_id,prod_id,prod_price from products where prod_price <=5 union
select vend_id,prod_id,prod_price from products where vend_id in (1001,1002);
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1003 | FC      |       2.50 |
|    1002 | FU1     |       3.42 |
|    1003 | SLING   |       4.49 |
|    1003 | TNT1    |       2.50 |
|    1001 | ANV01   |       5.99 |
|    1001 | ANV02   |       9.99 |
|    1001 | ANV03   |      14.99 |
|    1002 | OL1     |       8.99 |
+---------+---------+------------+
8 rows in set (0.00 sec)
```

这条语句由前面的两条 SELECT 语句组成,语句中用 UNION 关键字分隔。 UNION 指示MySQL执行两条 SELECT 语句,并把输出组合成单个查询结果集。

作为参考,这里给出使用多条 WHERE 子句而不是使用 UNION 的相同查询:

```
MariaDB [test]> select vend_id,prod_id,prod_price from products where prod_price <=5 or vend_id
in (1001,1002);
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1001 | ANV01   |       5.99 |
|    1001 | ANV02   |       9.99 |
|    1001 | ANV03   |      14.99 |
|    1003 | FC      |       2.50 |
|    1002 | FU1     |       3.42 |
|    1002 | OL1     |       8.99 |
|    1003 | SLING   |       4.49 |
|    1003 | TNT1    |       2.50 |
+---------+---------+------------+
8 rows in set (0.00 sec)
```

在这个简单的例子中,使用 UNION 可能比使用 WHERE 子句更为复杂。但对于更复杂的过滤条件,或者从多个表(而不是单个表)中检索数据的情形,使用 UNION 可能会使处理更简单。

> 检索客户信息表中客户名包含Mouse的客户id，以及订单表中，订单号为20005的客户id。

```
MariaDB [test]> select cust_id from orders where order_num=20005 union select cust_id from
customers where cust_name regexp 'Mouse';
+---------+
| cust_id |
+---------+
|   10001 |
|   10002 |
+---------+
2 rows in set (0.00 sec)
```

2.UNION 规则

正如所见,并是非常容易使用的。但在进行并时有几条规则需要注意。

- UNION 必须由两条或两条以上的 SELECT 语句组成,语句之间用关键字 UNION 分隔(因此,如果组合4条 SELECT 语句,将要使用3个UNION 关键字)。
- UNION 中的每个查询必须包含相同的列、表达式或聚集函数(不过各个列不需要以相同的次序列出)。
- 列数据类型必须兼容:类型不必完全相同,但必须是DBMS可以隐含地转换的类型(例如,不同的数值类型或不同的日期类型)。

如果遵守了这些基本规则或限制,则可以将并用于任何数据检索任务。

3.包含或取消重复的行

我们注意到,在分别执行时,第一条 SELECT 语句返回4行,第二条 SELECT 语句返回5行。但在用 UNION 组合两条 SELECT 语句后,只返回了8行而不是9行。

UNION 从查询结果集中自动去除了重复的行(换句话说,它的行为与。因为供应商 1002 生产单条 SELECT 语句中使用多个 WHERE 子句条件一样)的一种物品的价格也低于 5 ,所以两条 SELECT 语句都返回该行。在使用UNION 时,重复的行被自动取消。

这是 UNION 的默认行为,但是如果需要,可以改变它。事实上,如果想返回所有匹配行,可使用 UNION ALL 而不是 UNION 。

请看下面的例子:

```
MariaDB [test]>  select vend_id,prod_id,prod_price from products where prod_price <=5 union all
select vend_id,prod_id,prod_price from products where vend_id in (1001,1002);
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1003 | FC      |       2.50 |
|    1002 | FU1     |       3.42 |
|    1003 | SLING   |       4.49 |
|    1003 | TNT1    |       2.50 |
|    1001 | ANV01   |       5.99 |
|    1001 | ANV02   |       9.99 |
|    1001 | ANV03   |      14.99 |
|    1002 | FU1     |       3.42 |
|    1002 | OL1     |       8.99 |
+---------+---------+------------+
9 rows in set (0.00 sec)
```

使用 `UNION ALL` ,MySQL不取消重复的行。因此这里的例子返回9行,其中有一行出现两次。

UNION 与 WHERE 本章开始时说过, `UNION` 几乎总是完成与多个`WHERE` 条件相同的工作。 `UNION ALL` 为 `UNION` 的一种形式,它完成`WHERE` 子句完成不了的工作。如果确实需要每个条件的匹配行全部出现(包括重复行),则必须使用 `UNION ALL` 而不是 `WHERE` 。

4.对组合查询结果排序

`SELECT` 语句的输出用 `ORDER BY` 子句排序。在用 `UNION` 组合查询时,只能使用一条 `ORDER BY` 子句,它必须出现在最后一条 `SELECT` 语句之后。对于结果集,不存在用一种方式排序一部分,而又用另一种方式排序另一部分的情况,因此不允许使用多条 `ORDER BY` 子句。

> 下面的例子排序前面 UNION 返回的结果:

```
MariaDB [test]> select vend_id,prod_id,prod_price from products where prod_price <=5 union all
select vend_id,prod_id,prod_price from products where vend_id in (1001,1002) order by
vend_id,prod_price;
+---------+---------+------------+
| vend_id | prod_id | prod_price |
+---------+---------+------------+
|    1001 | ANV01   |       5.99 |
|    1001 | ANV02   |       9.99 |
|    1001 | ANV03   |      14.99 |
|    1002 | FU1     |       3.42 |
|    1002 | FU1     |       3.42 |
|    1002 | OL1     |       8.99 |
|    1003 | FC      |       2.50 |
|    1003 | TNT1    |       2.50 |
|    1003 | SLING   |       4.49 |
+---------+---------+------------+
9 rows in set (0.00 sec)
```

这条 `UNION` 在最后一条 `SELECT` 语句后使用了 `ORDER BY` 子句。虽然 `ORDER BY` 子句似乎只是最后一条 `SELECT` 语句的组成部分,但实际上MySQL将用它来排序所有 `SELECT` 语句返回的所有结果。

组合不同的表 为使表述比较简单,本章例子中的组合查询使用的均是相同的表。但是其中使用 UNION 的组合查询可以应用不同的表。

> 小结
>
> > 本章讲授如何用 UNION 操作符来组合 SELECT 语句。利用 UNION ,可把多条查询的结果作为一条组合查询返回,不管它们的结果中包含还是不包含重复。使用 UNION 可极大地简化复杂的 WHERE 子句,简化从多个表中检索数据的工作。