目录

N 创建高性能的索引

N.1 索引的类型

N.1.1 B-Tree 索引

B-Tree 索引上的所有数据都是按顺序存储在叶子页上的，并且每一个叶子页到根的距离相同。B-Tree 索引能够加快访问数据的速度，因为存储引擎不再需要进行全表扫描来获取需要的数据，取而代之的是从索引的根节点开始搜索。通过比较节点页的值和要查找的值可以找到合适的指针进入下层子节点，这些指针实际上定义了子节点页中值的上限和下限。叶子节点的指针指向的是被索引的数据。

可以使用 B-Tree 索引的查询类型：

- 全值匹配：和索引中的所有列进行匹配。
- 匹配最左前缀：只使用索引的第一列。
- 匹配列前缀：只使用索引第一列的值的开头部分。
- 匹配范围值：用于查找索引第一列的值的某个范围内的数据。
- 精确匹配前 N 列并范围匹配 N+1 列
- 只访问索引的查询：查询只需要访问索引，而无需访问数据行。
- 用于查询中的 ORDER BY 操作（按顺序查找）。

B-Tree 索引的限制：

- 如果不是按照索引的最左列开始查找，则无法使用索引。
- 不能跳过索引中的列。
- 如果查询中有某个列的范围查询，则其右边所有列都无法使用索引优化查找。

N.1.2 哈希索引

哈希索引基于哈希表实现，只有精确匹配索引所有列的查询才有效。对于每一行数据，

存储引擎都会对所有的索引列计算一个哈希码，哈希索引将所有的哈希码存储在索引中，同时在哈希表中保存指向每个数据行的指针。如果多个行的哈希值相同，索引会以链表的方式存放多个记录指针到同一个哈希条目中。

哈希索引的限制：

- 哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。
- 哈希索引数据并不是按照索引值顺序存储的，所以也就无法用于排序。
- 哈希索引也不支持部分索引列查找，因为哈希索引始终是使用索引列的全部内容来计算哈希值的。
- 哈希索引只支持等值比较查询，不支持任何范围查询。
- 当出现哈希冲突时，存储引擎必须遍历链表中所有的行指针，逐行进行比较，直到找到所有符合条件的行。

创建自定义哈希索引：

在 B-Tree 基础上创建一个伪哈希索引，还是使用 B-Tree 进行查找，但是它使用哈希值而不是键本身进行索引查找。你需要做的就是新增一个列用来存放键的哈希值，然后在查询的 WHERE 子句中手动指定使用的哈希函数。这样实现的缺陷是需要维护哈希值，一般使用触发器实现。为了处理哈希冲突，当使用伪哈希索引进行查询时，必须在 WHERE 子句中包含常量值。

例如需要存储大量的 URL，并根据 URL 进行搜索查找。如果使用 B-Tree 来存储 URL，存储的内容就会很大。于是可以删除原来 URL 列上的索引，而新增一个被索引的 url_crc 列用于存储 URL 的哈希值，就可以使用下面的方式查询：

```
SELECT id FROM url WHERE url="http://www.mysql.com"
AND url_crc=CRC32("http://www.mysql.com");
```

N.1.3 空间数据索引

N.1.4 全文索引

N.2 高性能索引策略

N.2.1 独立的列

“独立的列”是指索引列不能是表达式的一部分，也不能是函数的参数。例如下面这个查询无法使用 actor_id 列的索引：

```
SELECT actor_id FROM actor WHERE actor_id+1=5;
```

N.2.2 前缀索引

有时候需要索引很长的字符列，这会让索引变得大且慢。一个策略是使用自定义哈希索引。还可以索引开始的部分字符，这样可以大大节约索引空间，从而提高索引效率。但也会降低索引的选择性，而且 MYSQL 无法使用前缀索引做 ORDER BY 和 GROUP BY，也无法使用前缀索引做覆盖扫描。

有时候后缀索引也有用途（例如，找到某个域名的所有电子邮件地址）。MySQL 原生并不支持反向索引，但是可以把字符串反转后存储，并基于此建立前缀索引。可以通过触发器来实现这种索引。

N.2.3 多列索引

在多个列上建立独立的单列索引大部分情况下并不能提高 MySQL 的查询性能。例如，对于下面这个查询 WHERE 条件，这两个单列索引都不是好的选择，在老的 MySQL 版本中，会对这个查询使用全表扫描。

```
SELECT film_id, actor_id FROM film_actor
WHERE actor_id=1 OR film_id=1;
```

MySQL5.0 和更新版本引入了一种叫“索引合并”的策略，能够同时使用多个单列索引进

行扫描，并将结果进行合并。索引合并策略有时候是一种优化的结果，但实际上更多时候说明了表上的索引建得很糟糕。

N.2.4 选择合适的索引列顺序（适用于 B-Tree 索引）

对于如何选择索引的列顺序有一个经验法则：将选择性最高的列放到索引最前列。

N.2.5 聚簇索引

聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点是指向对应数据块的指针。因为无法将数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。

InnoDB 数据引擎通过主键聚集数据，如果没有定义主键，InnoDB 会选择一个唯一的非空索引代替，如果没有这样的索引，InnoDB 会隐式定义一个主键来作为聚簇索引。InnoDB 只聚集在同一个页面中的记录，包含相邻键值的页面可能会相距甚远。InnoDB 的二级索引的叶子保存的是行的主键值，而不是行指针，这减小了移动数据或者数据页面分裂时维护二级索引的开销，因为 InnoDB 不需要更新索引的行指针。但这也意味着通过二级索引查找行，存储引擎需要先找到二级索引的叶子节点获得对应的主键值，然后根据这个值去聚簇索引中查找到对应的行，即进行了两次 B-Tree 查找。

聚簇索引的优点：

- 可以把相关数据保存在一起，最大限度地提高了 I/O 密集型应用的性能。
- 聚簇索引将索引和数据保存在同一个 B-Tree 中，因此数据访问更快。
- 使用覆盖索引扫描的查询可以直接使用叶节点中的主键值。

聚簇索引的缺点：

- 插入速度严重依赖于插入顺序。按照主键的顺序插入是加载数据到 InnoDB 表中速度最快的方式。
- 更新聚簇索引列的代价很高，因为会强制 InnoDB 将每个被更新的行移动到新的位置。

- 基于聚簇索引的表在插入新行，或者主键被更新导致需要移动行的时候，可能面临“页分裂”的问题。
- 聚簇索引可能导致全表扫描变慢，尤其是行比较稀疏，或者由于页分裂导致数据存储不连续的时候。

N.2.6 覆盖索引

如果一个索引包含(或者说覆盖)所有需要查询的字段的值，我们就称之为“覆盖索引”，索引不会覆盖所有的列。覆盖索引必须要存储索引列的值，因此 MySQL 只能使用 B-Tree 索引做覆盖索引，而且 MySQL 只能在索引中执行最左前缀匹配的 LIKE 比较，而不能执行以通配符开头的 LIKE 查询。

N.2.7 使用索引扫描来做排序

MySQL 有两种方式可以生成有序的结果：通过排序操作、按索引顺序扫描。除非使用聚簇索引，否则按索引顺序读取数据通常要比顺序地全表扫描慢，但这样做可以省略后续的排序操作。ORDER BY 子句和查找型查询的限制是一样的：需要满足索引的最左前缀的要求。

N.2.8 压缩（前缀压缩）索引

MyISAM 使用前缀压缩来减少索引的大小，从而让更多的索引可以放入内存中，代价是某些操作可能更慢。MyISAM 查找时无法在索引块使用二分查找而只能从头开始扫描，因此对于随机查找和倒序扫描速度要慢很多，尤其是倒序扫描，所有在块中查找某一行的操作平均都需要扫描半个索引块。

N.2.9 冗余和重复索引

重复索引是指在相同的列上按照相同的顺序创建的相同类型的索引。MySQL 需要单独维护重复的索引，并且优化器在优化查询的时候也需要逐个地进行考虑，这会影响性能。应

该避免创建重复索引，发现以后也应该立即移除。

如果创建了索引(A, B)，再创建索引(A)就是冗余索引，因为索引(A, B)可以当做索引(A)来使用；还有一种情况是将索引(A)扩展为(A, ID)，其中 ID 是主键，对于 InnoDB 来说主键列已经包含在二级索引中了，所以这也是冗余的。大多数情况下都不需要冗余索引，应该尽量扩展已有的索引而不是创建新的索引，但也有时候出于性能的考虑需要冗余索引，因为扩展已有的索引会导致其变得太大，从而影响其他使用该索引的查询的性能。

N.2.10 索引和锁

InnoDB 在访问行的时候会对其加锁，而索引能减少 InnoDB 访问的行数，从而减少锁的数量。但即使使用了索引，InnoDB 也可能锁住一些不需要的数据。InnoDB 在二级索引上使用共享（读）锁，但访问主键索引需要排他（写）锁。

1 MySQL 简介

- 在相同的数据库中不能两次使用相同的表名，在不同的数据库中可以使用相同的表名。
- 主键列不允许为 NULL。
- 最好不要在主键列中使用可能会改变的值。

2 检索数据

2.1 检索单个列

例如：SELECT prod_name FROM products;

从 products 表中检索一个名为 prod_name 的列。

2.2 检索多个列

例如：SELECT prod_id,prod_name,prod_price FROM products;

2.3 检索所有列

*例如：SELECT * FROM products；*

除非你确实需要表中的每个列，否则最好不要使用*通配符，检索不需要的列通常会降低检索和应用程序的性能。

2.4 检索不同的行（DISTINCT）

> *例如：SELECT DISTINCT vend_id FROM products;*

只返回不同（唯一）的 vend_id 行。

如果使用 DISTINCT 关键字，它必须直接放在列名的前面。

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相同，否则所有行都将被检索出来。

2.5 限制结果（LIMIT）

*SELECT prod_name FROM products LIMIT 5;*返回从第 0 行开始的 5 行结果。

*SELECT prod_name FROM products LIMIT 3,4;*返回从第 3 行开始的 4 行结果。

MySQL 检索出来的行从 0 开始算起。

在行数不够时，即指定要检索的行数大于剩余的总行数，则只返回剩余的行数。

2.6 使用完全限定的列名或表名

> *例如：SELECT products.prod_name FROM products;*
>
> *例如：SELECT products.prod_name FROM crashcourse.products;*

3 排序检索数据（ORDER BY）

3.1 排序数据

> *例如：SELECT prod_name FROM products ORDER BY prod_name;*

通常 ORDER BY 子句使用的列是 SELECT 子句中的列，但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。

3.2 按多个列排序

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。例如：

```
SELECT prod_id,prod_price,prod_name
FROM products
ORDER BY prod_price,prod_name;
```

上面的代码检索 3 个列，并按其中两个列对结果进行排序——首先按价格，然后再按名称排序。

3.3 指定排序方向（DESC、ASC）

默认的排序顺序是升序排序，如果要使用降序排序，需要指定 DESC 关键字。

下面的例子按价格以降序排序产品（最贵的排在最前面）：

```
SELECT prod_id,prod_price,prod_name
FROM products
ORDER BY prod_price DESC;
```

但是如果打算用多个列排序怎么办？

下面的例子以降序排序产品，然后再对产品名排序：

```
SELECT prod_id,prod_price,prod_name
FROM products
ORDER BY prod_price DESC,prod_name;
```

DESC 关键字只应用到直接位于其前面的列名，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。

使用 ORDER BY 和 LIMIT 组合，能够找出一个列中最高或最低的值。

下面的例子演示如何找出最昂贵物品的值：

```
SELECT prod_price
FROM products
ORDER BY prod_price DESC
LIMIT 1;
```

ORDER BY 子句位于 FROM 子句之后，LIMIT 子句位于 ORDER BY 子句之后。

4 过滤数据（WHERE）

SQL 子句顺序：FROM→WHERE→ORDER BY→LIMIT

4.1 检查单个值（=、>、<、<>、<=、>=、!=）

例如：SELECT prod_name,prod_price FROM products WHERE prod_name='fuses';

例如：SELECT prod_name,prod_price FROM products WHERE prod_price<10;

如果将值与串类型的列进行比较，则需要限定引号，如果用来与数值列进行比较的值不需要引号。

4.2 范围值检查（BETWEEN … AND …）

```
SELECT prod_name,prod_price
FROM products
WHERE prod_price BETWEEN 5 AND 10;
```

BETWEEN 匹配范围中所有的值，包括指定的开始值和结束值。

4.3 空值检查（IS NULL）

SELECT 语句有一个特殊的 WHERE 子句，可用来检查具有 NULL 值的列。这个 WHERE 子句就是 IS NULL 子句。其语法如下：

```
SELECT prod_name
FROM products
WHERE prod_price IS NULL;
```

4.4 AND/OR 操作符

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

```
SELECT prod_id,prod_price,prod_name
FROM products
WHERE ven_id=1003 AND prod_price<=10;
```

检索由 1002 或 1003 供应商制造的所有产品的产品名和价格。

```
SELECT prod_name，prod_price
FROM products
WHERE ven_id=1003 OR ven_id=1002;
```

AND 操作符的优先级高于 OR 操作符。

4.5 IN 操作符

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。例如：

```
SELECT prod_name,prod_price
FROM products
WHERE vend_id IN (1002,1003)
ORDER BY prod_name;
```

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。

4.6 NOT 操作符

```
SELECT prod_name,prod_price
FROM products
WHERE vend_id NOT IN (1002,1003)
ORDER BY prod_name;
```

MySQL 仅支持使用 NOT 对 IN、BETWEEN 和 EXISTS 子句取反。

5 用通配符进行过滤（LIKE）

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。

tips：

1) 不要过度使用通配符。通配符搜索的处理一般用时更长。

2) 一定要使用通配符时，最好把它们用在搜索条件的最后。

5.1 %通配符

%表示任何字符出现任意次数（包括 0 次）。

找出所有以 jet 开头的产品

```
SELECT prod_id,prod_name
FROM products
WHERE prod_name LIKE 'jet%';
```

找出所有名字中包含 anvil 的产品

```
SELECT prod_id,prod_name
FROM products
WHERE prod_name LIKE '%anvil%';
```

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。

5.2 下划线（_）通配符

下划线只匹配单个字符而不是任意个字符。

```
SELECT prod_id,prod_name
FROM products
WHERE prod_name LIKE '_ ton anvil';
```

6 使用正则表达式进行搜索（REGEXP）

MySQL 仅支持多数正则表达式实现的一个很小的子集。

6.1 基本字符匹配

检索列 prod_name 包含文本 1000 的所有行：

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '1000'
ORDER BY prod_name;
```

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '.000'
ORDER BY prod_name;
```

.表示匹配任意一个字符。

MySQL 中的正则表达式匹配不区分大小写。为区分大小写，可使用 BINARY 关键字，如 *WHERE prod_name REGEXP BINARY 'JetPack .000'*。

6.2 进行 OR 匹配

使用|从功能上类似于在 SELECT 语句中使用 OR 语句，多个 OR 条件可并入单个正则表达式。

返回名字中包含 1000 或 2000 的产品名称：

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '1000|2000'
ORDER BY prod_name;
```

6.3 匹配几个字符之一

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '[123] Ton'
ORDER BY prod_name;
```

[123]定义一组字符，它的意思是匹配 1 或 2 或 3

字符集合也可以被否定，为否定一个字符集，在集合的开始处放置一个^即可。因此，尽管[123]匹配字符 1、2 或 3，但[^123]却匹配除这些字符外的任何东西。

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '[1-5] Ton'
ORDER BY prod_name;
```

使用-来定义一个范围，[1-5]的意思是匹配 1 到 5 中任意一个字符。

6.4 匹配特殊字符（\\）

为了匹配特殊字符，必须用\\为前导。例如：

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '\\.'
ORDER BY prod_name;
```

6.5 匹配字符类

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]）
[:graph:]	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]）

6.6 使用元字符

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）

6.7 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，需要使用定位符。

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

```
SELECT prod_name
FROM products
WHERE prod_name REGEXP '^[0-9\\.]'
ORDER BY prod_name;
```

^有两种用法，在[ ]外用来指串的开始处，在[ ]内用来否定该集合。

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。

7 创建计算字段

7.1 拼接字段

使用 concat 来拼接多个列。

```
SELECT concat(vend_name,')',vend_country,')')
FROM vendors
ORDER BY vend_name;
```

使用 RTrim()函数来删除数据右侧多余的空格，LTrim 去除左边的空格，Trim 去除左右两边的空格。

```
SELECT concat(RTrime(vend_name),')',vend_country,')')
FROM vendors
ORDER BY vend_name;
```

7.2 使用别名

别名用 AS 关键字赋予。

```
SELECT concat(RTrime(vend_name),')',vend_country,')') AS vend_title
FROM vendors
ORDER BY vend_name;
```

7.3 执行算数计算

汇总物品的价格（单价乘以订购数量）

```
SELECT prod_id,quantity,item_price,quantity*item_price AS expanded_price
FROM orderitems
WHERE order_num=20005;
```

8 使用数据处理函数

8.1 文本处理函数

函　数	说　明
Left()	返回串左边的字符
Length()	返回串的长度
Locate()	找出串的一个子串
Lower()	将串转换为小写
LTrim()	去掉串左边的空格
Right()	返回串右边的字符

RTrim()	去掉串右边的空格
Soundex()	返回串的SOUNDEX值
SubString()	返回子串的字符
Upper()	将串转换为大写

8.2 日期和时间处理函数

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

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

```
SELECT cust_id,order_num
FROM orders
WHERE Date(order_date)='2005-09-01';
```

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这样。

检索出 2005 年 9 月下的所有订单

```
SELECT cust_id,order_num
FROM orders
WHERE Date(order_date) BETWEEN '2005-09-01' AND '2005-09-30';
```

或

```
SELECT cust_id,order_num
FROM orders
WHERE Year(order_date)=2005 AND Month(order_date)=9;
```

8.3 数值处理函数

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

9 汇总数据

9.1 AVG 函数

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。

AVG()函数忽略列值为 NULL 的行。

返回特定供应商所提供产品的平均价格：

```
SELECT AVG(prod_price) AS avg_price
FROM products
WHERE vend_id=1003;
```

9.2 COUNT 函数

COUNT 函数有两种使用方式：

1. 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。

```
SELECT COUNT(*) AS num_cust
FROM customers;
```

2. 使用 COUNT(column)对特定列中具有值的行进行计数，忽略 NULL 值。

```
SELECT COUNT(cust_email) AS num_cust
FROM customers;
```

9.3 MAX 函数，MIN 函数

MAX()一般用来找出最大的数值或日期值,但 MYSQL 也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。

```
SELECT MAX(prod_price) AS max_price
FROM products;
```

MIN()的功能正好与 MAX()功能相反。

```
SELECT MIN(prod_price) AS max_price
FROM products;
```

MIN 和 MAX 都忽略列值为 NULL 的行。

9.4 SUM 函数

合计每项物品的 item_price*quantity，得出总的订单金额

```
SELECT SUM(item_price*quantity) AS total_price
FROM orderitems
WHERE order_num=20005;
```

SUM 函数也忽略列值为 NULL 的行。

9.5 聚集不同值

对以上 5 个聚集函数，它们的参数都可以指定为 ALL（默认）或者 DISTINCT。

下面的例子使用 AVG()函数返回特定供应商提供的产品的平均价格。它使用了 DISTINCT 参数，因此平均值只考虑各个不同的价格：

```
SELECT AVG(DISTINCT prod_price) AS avg_price
FROM products
WHERE vend_id=1003;
```

DISTINCT 不能用于 COUNT(*)，DISTINCT 必须使用列名。

9.6 组合聚集函数

```
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;
```

10 分组数据

10.1 创建分组（GROUP BY）

- 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。
- 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)。
- 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。

```
SELECT vend_id,COUNT(*) AS num_prods
FROM products
GROUP BY vend_id;
```

GROUP BY 子句指示 MySQL 按 vend_id 排序并分组数据。这导致对每个 vend_id 而不是整个表计算 num_prods 一次。

10.2 过滤分组（HAVING）

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 子句中不能使用别名。

```
SELECT cust_id,COUNT(*) AS orders
FROM orders
GROUP BY cust_id
HAVING COUNT(*)>=2;
```

可以在一条语句中同时使用 WHERE 和 HAVING 子句。

```
SELECT vend_id,COUNT(*) AS num_prods
```

```
FROM products
WHERE prod_price>=10
GROUP BY vend_id
HAVING COUNT(*)>=2;
```

10.3 SELECT 子句顺序

SELECT→FROM→WHERE→GROUP BY→HAVING→ORDER BY→LIMIT

11 使用子查询

11.1 利用子查询进行过滤

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。

需要列出订购物品 TNT2 的所有客户。

```
SELECT cust_name,cust_contact
FROM customers
WHERE cust_id IN (SELECT cust_id
                  FROM orders
                  WHERE order_num IN (SELECT order_num
                                      FROM orderitems
                                      WHERE prod_id='TNT2'));
```

对于能嵌套的子查询数目没有限制，但在实际使用时由于性能的限制，不能嵌套太多子查询。

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。

11.2 作为计算字段使用子查询

需要显示 customers 表中每个客户的订单总数。订单与相应的客户 ID 存储在 orders 表中。

```
SELECT cust_name,
       cust_state,
       (SELECT COUNT(*)
        FROM orders
        WHERE orders.cust_id=customers.cust_id) AS orders
```

```
FROM customers
ORDER BY cust_name;
```

子查询中的 WHERE 子句应使用完全限定列名。

12 联结表

12.1 内部联结/等值联结

```
SELECT vend_name,prod_name,prod_price
FROM vendors,products
WHERE vendors.vend_id=products.vend_id
ORDER BY vend_name,prod_name;
```

或

```
SELECT vend_name,prod_name,prod_price
FROM vendors INNER JOIN products
ON vendors.vend_id=products.vend_id;
```

12.2 联结多个表

下面两个查询结果一致：

```
SELECT cust_name,cust_contact
FROM customers
WHERE cust_id IN (SELECT cust_id
                  FROM orders
                  WHERE order_num IN (SELECT order_num
                                      FROM orderitems
                                      WHERE prod_id='TNT2'));
```

```
SELECT cust_name,cust_contact
FROM customers,orders,orderitems
WHERE customers.cust_id=orders.cust_id
    AND orderitems.order_num=orders.order_num
    AND prod_id='TNT2';
```

12.3 使用表别名

使用表别名主要有两个目的：

1) 缩短 SQL 语句；

2) 允许在单条 SELECT 语句中多次使用相同的表。

表别名可以用在各种子句中。

表别名只在查询执行中使用，与列别名不一样，表别名不会返回到客户机。

12.4 自联结

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。

方法一：使用子查询

```
SELECT prod_id,prod_name
FROM products
WHERE vend_id=(SELECT vend_id
                FROM products
                WHERE prod_id='DTNTR');
```

方法二：使用自联结

```
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';
```

12.5 自然联结

标准的联结返回所有数据，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。

12.6 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：

检索所有客户及其下的订单，包括那些至今尚未下订单的客户。

```
SELECT customers.cust_id,orders.order_num
FROM customers LEFT OUTER JOIN orders
ON customers.cust_id=orders.cust_id;
```

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。

12.7 使用带聚集函数的联结

检索所有客户及每个客户所下的订单数

```
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;
```

如果不使用 GROUP BY 子句，则 COUNT 函数得出的将是所有客户的订单总数

13 组合查询（UNION）

- 任何使用 OR 操作符的过滤查询都可以使用 UNION 来完成；
- UNION 必须由两条或两条以上的 SELECT 语句组成，语句之间用 UNION 分隔；
- UNION 中的每个查询必须包含相同的列、表达式或聚集函数；
- UNION 默认会从查询结果中去掉重复的行，如果想返回所有的匹配行，可使用 UNION ALL 而不是 UNION；
- 在用 UNION 组合查询时，只能使用一条 ORDER BY 子句，放在最后一条 SELECT 语句之后。

例如，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。

```
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)
ORDER BY vend_id,prod_price;
```

14 全文本搜索

两个最常使用的引擎为 MyISAM 和 InnoDB，前者支持全文本搜索，而后者不支持。

为了进行全文本搜索，必须索引被搜索的列，而且要随着数据的改变不断地重新索引。

14.1 进行全文本搜索

在索引之后，使用两个函数 Match()和 Against()执行全文本搜索，其中 Match()指定被搜索的列，Against()指定要使用的搜索表达式。

```
SELECT note_text
FROM productnotes
WHERE Match(note_text) Against('rabbit');
```

14.2 使用查询扩展

查询扩展用来设法放宽所返回的全文本搜索结果的范围，找出可能相关的结果，即使它们并不精确包含所查找的词。

```
SELECT note_text
FROM productnotes
WHERE Match(note_text) Against('rabbit' WITH QUERY EXPANSION);
```

14.3 布尔文本搜索

布尔操作符	说　明
+	包含，词必须存在
-	排除，词必须不出现
>	包含，而且增加等级值
<	包含，且减少等级值
()	把词组成子表达式（允许这些子表达式作为一个组被包含、排除、排列等）
~	取消一个词的排序值
*	词尾的通配符
""	定义一个短语（与单个词的列表不一样，它匹配整个短语以便包含或排除这个短语）

搜索匹配包含词 rabbit 和 bait 的行

```
SELECT note_text
FROM productnotes
```

```
WHERE Match(note_text) Against('+rabbit +bait' IN BOOLEAN MODE);
```

没有指定操作符，这个搜索匹配包含 rabbit 和 bait 中的至少一个词的行。

```
SELECT note_text
FROM productnotes
WHERE Match(note_text) Against('rabbit bait' IN BOOLEAN MODE);
```

这个搜索匹配短语 rabbit bait 而不是匹配两个词 rabbit 和 bait。

```
SELECT note_text
FROM productnotes
WHERE Match(note_text) Against('"rabbit bait"' IN BOOLEAN MODE);
```

匹配 rabbit 和 carrot，增加前者的等级，降低后者的等级。

```
SELECT note_text
FROM productnotes
WHERE Match(note_text) Against('>rabbit <carrot' IN BOOLEAN MODE);
```

这个搜索匹配词 safe 和 combination，降低后者的等级。

```
SELECT note_text
FROM productnotes
WHERE Match(note_text) Against('+safe +(<combination)' IN BOOLEAN MODE);
```

15 插入数据

15.1 插入完整的行

```
INSERT INTO customers(cust_name,
      cust_address,
      cust_city,
      cust_state,
      cust_zip,
      cust_country,
      cust_contact,
      cust_email)
VALUES('Pep E. LaPew',
      '100 Main Street',
```

```
        'Los Angeles',
        'CA',
        '90046',
        'USA',
        NULL,
        NULL);
```

如果表的定义允许，则可以在 INSERT 操作中省略某些列。省略的列必须满足一下某个条件：

- 该列定义为允许 NULL 值（无值或空值）。
- 在表定义中给出默认值。这表示如果不给出值，将使用默认值。

可以通过在 INSERT 和 INTO 之间添加关键字 LOW_PRIORITY，指示 MySQL 降低 INSERT 语句的优先级，如下所示：

INSERT LOW_PRIORITY INTO

这也适用于 UPDATE 和 DELETE 语句。

15.2 插入多个行

```
INSERT INTO customers(cust_name,
        cust_address,
        cust_city,
        cust_state,
        cust_zip,
        cust_country)
VALUES(
        'Pep E. LaPew',
        '100 Main Street',
        'Los Angeles',
        'CA',
        '90046',
        'USA',
        NULL,
        NULL
),
        (
        'M. Martian',
        '42 Galaxy Way',
        'New York',
        'NY',
```

```
        '11213',
        'USA'
);
```

此技术可以提高数据库处理的性能，因为 MySQL 用单条 INSERT 语句处理多个插入比使用多条 INSERT 语句快。

15.3 插入检索出的数据

```
INSERT INTO customers(cust_name,
      cust_address,
      cust_city,
      cust_state,
      cust_zip,
      cust_country)
SELECT cust_name,
      cust_address,
      cust_city,
      cust_state,
      cust_zip,
      cust_country
FROM custnew;
```

16 更新和删除数据

16.1 更新数据

更新客户 1005 的电子邮件地址

```
UPDATE customers
SET cust_email='elmer@fudd.com'
WHERE cust_id=10005;
```

更新多个列

```
UPDATE customers
SET cust_name='The Fudds',
      cust_email='elmer@fudd.com'
WHERE cust_id=10005;
```

16.2 删除数据

删除客户 10006 的记录

```
DELETE FROM customers
WHERE cust_id=10006;
```

如果想从表中删除所有行，不要使用 DELETE。可使用 TRUNCATE TABLE 语句，它完成相同的工作，但速度更快（TRUNCATE 实际是删除原来的表并重新创建一个表，而不是逐行删除表中的数据）。

不允许删除具有与其他表相关联的数据的行。

17 创建和操纵表

17.1 创建表

```
CREATE TABLE customers
(
  cust_id        int          NOT NULL AUTO_INCREMENT,
  cust_name      char(50)     NOT NULL ,
  cust_address   char(50)     NULL ,
  cust_city      char(50)     NULL ,
  cust_state     char(5)      NULL ,
  cust_zip       char(10)     NULL ,
  cust_country   char(50)     NULL DEFAULT China,
  cust_contact   char(50)     NULL ,
  cust_email     char(255)    NULL ,
  PRIMARY KEY (cust_id)
) ENGINE=InnoDB;
```

- 主键中只能使用不允许 NULL 值的列。
- 每个表只允许一个 AUTO_INCREMENT 列，而且它必须被索引（如，通过使它成为主键）。
- 可以使用 last_insert_id()函数来返回最后一个 AUTO_INCREMENT 值。
- MySQL 不允许使用函数作为默认值，它只支持常量。
- 常用的引擎有三个：InnoDB（支持事务处理，不支持全文本搜索）、MyISAM（支持全文本搜索，不支持事务处理）、MEMORY（支持哈希索引，数据存储在内存中，速度很快，适合临时表）。引擎类型可以混用

17.2 更新表

为更新表定义，可使用 ALTER TABLE 语句。但是，在理想状态下，当表中存储数据以后，该表就不应该再被更新。

给 vendors 添加一个列

```
ALTER TABLE vendors
ADD vend_phone CHAR(20);
```

删除列

```
ALTER TABLE vendors
DROP COLUMN vend_phone;
```

ALTER TABLE 的一种常见用途是定义外键

```
ALTER TABLE orderitems
ADD CONSTRAINT fk_orderitems_products FOREIGN KEY(prod_id)
REFERENCES products(prod_id);
```

17.3 删除表

```
DROP TABLE customers2;
```

17.4 重名名表

重命名一个表

```
RENAME TABLE customers2 TO customers;
```

重命名多个表

```
RENAME TABLE backup_customers TO customers,
             backup_vendors TO vendors,
             backup_products TO products;
```

18 视图（VIEW）

视图的一些常见应用：

- 重用 SQL 语句。
- 简化复杂的 SQL 操作。在编写查询后，可以方便地重用它而不必知道其查询细节。
- 使用表的组成部分而不是整个表。
- 保护数据。可以给用户授予表的特定部分的访问权限而不是整个表的访问权限。
- 更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据。

创建一个视图

```
CREAT VIEW productcustomers AS
SELECT cust_name, cust_contact,prod_id
FROM customers,orders,orderitems
WHERE customers.cust_id=orders.cust_id
      AND orderitems.order_num=order.order_num;
```

通常视图是可更新的，更新一个视图将更新其基表。但如果视图定义中有以下操作，则不能进行视图的更新：

- 分组（使用 GROUP BY 和 HAVING）
- 联结
- 子查询
- 并
- 聚集函数
- DISTINCT
- 导出（计算）列

19 存储过程（PROCEDURE）

19.1 创建存储过程

```
CREATE PROCEDURE ordertotal(
      IN onumber INT,
      OUT ototal DECIMAL(8,2)
)
BEGIN
```

```
    SELECT Sum(item_price*quantity)
    FROM orderitems
    WHERE order_num=onumber
    INTO ototal;
END;
```

19.2 执行存储过程

```
CALL ordertotal(20005, @total);
SELECT @toal;
```

变量必须以@开头

19.3 删除存储过程

```
DROP PROCEDURE ordertoal;
```