第 1 章

DBA 五分钟速成

用户经常抱怨他们的应用程序运行缓慢。通过检查系统和数据库性能，你找到了一个数据库中运行缓慢的 SQL 查询语句。但

如果你不知道如何在 MySQL 中对这个 SQL 语句进行调优，那又该怎么办呢？本书旨在通过对优化 SQL 语句的理想方法和最佳原则的讨论来满足优化 SQL 语句的需要。本章将为读者提供一些可以直接应用到工作中的速成技巧。

本章中我们将会用到以下方法：

- 确认运行缓慢的查询
- 识别缺失的索引
- 应用新的索引
- 验证新的索引

1.1 识别性能问题

用户报告说你的应用程序太慢。在确定不存在物理系统资源瓶颈之后，就应该把注意力转向 MySQL 数据库。

1.1.1 寻找运行缓慢的 SQL 语句

通过在当前运行的 MySQL 系统中运行 SHOW FULL PROCESSLIST 命令，可以查看以下细节信息：

```
mysql> SHOW FULL PROCESSLIST\G...
*********************** 6. row ***********************
     Id: 42
   User: appl
   Host: localhost
     db: NULL
Command: Query
   Time: 3
  State: Query
```

```
Info: SELECT * FROM inventory WHERE item_id = 16102176
```

从以上代码的 Time 列可以看出，Info 列中列出的 SELECT 语句的运行时间为 3 秒。

现在该做些什么？

1.1.2　确认低效查询

当发现一个潜在的低效查询问题之后，首先要做的就是确认这个查询是否每次重复执行都很缓慢。我们需要验证这个低效查询并不是由于锁定或者系统瓶颈等其他因素导致的个别现象。

1. 运行 SQL 语句并记录执行时间

使用 MySQL 命令行客户端或者其他客户端工具重复运行那条 SQL 语句是一个简单的验证方法。

```
mysql> SELECT * FROM inventory WHERE item_id =
16102176;
Empty set (3.19 sec)
```

这样就可以确认这条查询的执行时间超过了 3 秒。对于执行时间大于 10 毫秒的查询，MySQL 命令行客户端返回的查询结果已足够。第 7 章将详细介绍其他确定查询响应时间的方法。

警告

重复运行的方法只能用于 SELECT 语句，因为它不会修改现有的数据。如果低效查询语句是 UPDATE 或者 DELETE 语句，那么可以简单地将它们重写为 SELECT 语句以完成验证。例如，如果 SQL 查询是 DELETE FROM inventory WHERE item_id=16102176，可以将该查询重写为本例中的 SELECT 语句来模拟原来语句的性

能，且不改变任何信息。

2. 生成一个查询执行计划(Query Execution Plan，QEP)

当 MySQL 要执行一个 SQL 查询的时候，它首先会对该 SQL 语句进行语法检查，然后构造一个 QEP，QEP 决定了 MySQL 从底层存储引擎中获取信息的方式。如果想要查看 MySQL 查询优化器为 SQL 语句构造的 QEP，只需要在 SELECT 语句前加上如下的 EXPLAIN 关键字前缀：

```
mysql> EXPLAIN SELECT * FROM inventory WHERE item_id
= 16102176\G
********************** 1. row **********************
            id: 1
  select_ type: SIMPLE
         table: inventory
          type: ALL
possible_ keys: NULL
           key: NULL
       key_len: NULL
           ref: NULL
          rows: 787338
         Extra: Using where
```

在 MySQL 命令行客户端添加 \G 语句终止符可以让返回的结果垂直显示。这可以帮助我们更好地分析那些通过自动操作返回的结果，同时也对打印的表单有好处，比如本书中的那些需要打印的表单。如果使用(;)终止符，则返回结果是按列组织的，这种格式更易于阅读多行的输出结果。

注意

在大多数情况下，EXPLAIN 并不运行实际的 SQL 语句；然而当优化器需要执行这条 SQL 语句的一部分来决定如何构造 QEP 时

也有例外情况，例如在 FROM 子句中使用派生表，这时你会看到在 select_type 列中显示的单词 DERIVED。可以在 MySQL 参考手册 (http://dev.mysql.com/doc/refman/5.5/en/from-clause-subqueries.html) 中找到更多有关这些限制的信息。

如果你对如何阅读 QEP 一无所知，那么应该先看看使用到的索引和受影响的行数这两列信息。EXPLAIN 的结果中 key 列可以显示查询语句使用的索引，任何没有使用索引的查询语句都可以认为是没有被足够调优的 SQL 查询。受影响的行数显示在 rows 列中，这个数值可以用来估计查询需要读取的数据量，而这和查询所需要的执行时间是直接相关的。结果中 type 列显示的 ALL 值也是潜在性能问题的一个标志，我们将会在第 4 章和第 9 章详细讨论。

注意

根据底层使用的不同存储引擎，受影响的行数这个指标可能是一个估计值，也可能是一个精确值。即使受影响的行数是一个估计值(例如当使用 InnoDB 存储引擎管理表存储时)，通常情况下这个估计值也足以使优化器做出一个有充分依据的决定。

在这个 EXPLAIN 例子中，key 列未显示索引值。由于这是一个单表 SELECT 语句，这个语句可以被理解成对整个表进行扫描并从中寻找那些符合 WHERE 子句判定的行。为了找到与 item_id=16102176 相匹配的事件，结果中的 rows 列的值可以被认为是一个近似值。

1.2 优化查询

识别一个运行缓慢的 SQL 查询是任何优化过程的先决条件。本书将详细介绍一些工具和原理，这些工具和原理将有助于在各种选项中做出选择，从而创建一个理想的解决方案。

1.2.1 不应该做的事情

当你毫无头绪无规则可循的时候，可能会想到在这个基于 WHERE 语句的表上添加一个索引，如下所示：

```
mysql> ALTER TABLE inventory ADD INDEX (item_id);
Query OK, 734787 rows affected (54.22 sec)
Records: 734787 Duplicates: 0 Warnings: 0
```

警告

在没有进一步验证的情况下，千万不要在生产环境中这样做。

决定是否添加一个新的索引并部署它需要考虑很多因素。该语句仅强调其对生产环境一个潜在的影响。完成这条数据定义语言(DDL)语句花了 55 秒。在此期间，由于 ALTER 语句是阻塞操作，因此所有为表添加和修改数据的额外请求都被阻塞了。根据其他数据操作语言(DML)的执行顺序，此时 SELECT 语句也会被阻塞因而无法完成。如果表更大一些的话，一个 ALTER 语句可能需要几小时甚至几天才能执行完成！另一个需要考虑的因素是在一个表有多个索引的情况下 DML 语句的性能开销。

1.2.2　确认优化

重新运行 SQL 查询可以看到性能得到明显改善，这个查询现在的执行时间少于 10 毫秒。

```
mysql> SELECT * FROM inventory WHERE item_id =
16102176;
Empty set (0.00 sec)
```

也可以通过查看修正了的 QEP 来确认新索引的效率：

```
mysql> EXPLAIN SELECT * FROM inventory WHERE item_id
= 16102176\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: inventory
         type: ref
possible_keys: item_id
          key: item_id
      key_len: 4
          ref: const
         rows: 1
        Extra:
```

MySQL 优化器现在选择了一个 key 列值指定的索引。预计执行 SQL 语句时受影响的列数变成了 1，而之前的值为 787 338。

1.2.3　正确的方式

为表添加索引有很多优点，包括性能优化。然而，添加索引同时还有许多其他含义。第 4 章和第 5 章将讨论在表的列上添加索引的优缺点。在决定添加索引之前，通常应该至少做两项检查：首先验证表现有的结构，然后确认表的大小。可以通过执行以下这些 SQL 命令来获得上述信息：

```
mysql> SHOW CREATE TABLE inventory\G
*********************** 1. row ***********************
Create Table: CREATE TABLE 'inventory' (
  'id' int(10) unsigned NOT NULL AUTO_INCREMENT,
  'supp_id' int(10) unsigned NOT NULL DEFAULT '0',
  'item_id' int(10) unsigned NOT NULL DEFAULT '0',
  'qty' int(11) NOT NULL,
  'created' timestamp NOT NULL DEFAULT
  CURRENT_TIMESTAMP
  PRIMARY KEY ('id'),
  UNIQUE KEY 'supp_id' ('supp_id','item_id'),
  KEY 'created' ('created'),
) ENGINE=InnoDB DEFAULT CHARSET=latin1

mysql> SHOW TABLE STATUS LIKE 'inventory'\G
*********************** 1. row ***********************
           Name: inventory
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 679890
 Avg_row_length: 371
    Data_length: 252395520
Max_data_length: 0
   Index_length: 40861696
      Data_free: 0
 Auto_increment: 1612406
    Create_time: 2010-08-17 20:16:13
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
        Comment: InnoDB free: 644096 Kb
```

从这些命令的返回结果中可以看出，当前的表结构包括一系列索引，也包括一个使用了 item_id 列的索引。然而这个索引并没

有被用到，因为之前的查询不能满足索引中最左边的列。也可以通过 SHOW TABLE STATUS 命令的 Data_length 和 Rows 信息来获得表大小的一个近似值。第 4 章和第 5 章将进一步讨论这些信息在确定添加索引带来的影响方面的重要性，这些影响主要是添加索引带来的时间影响以及在同一列上添加多个索引的影响。

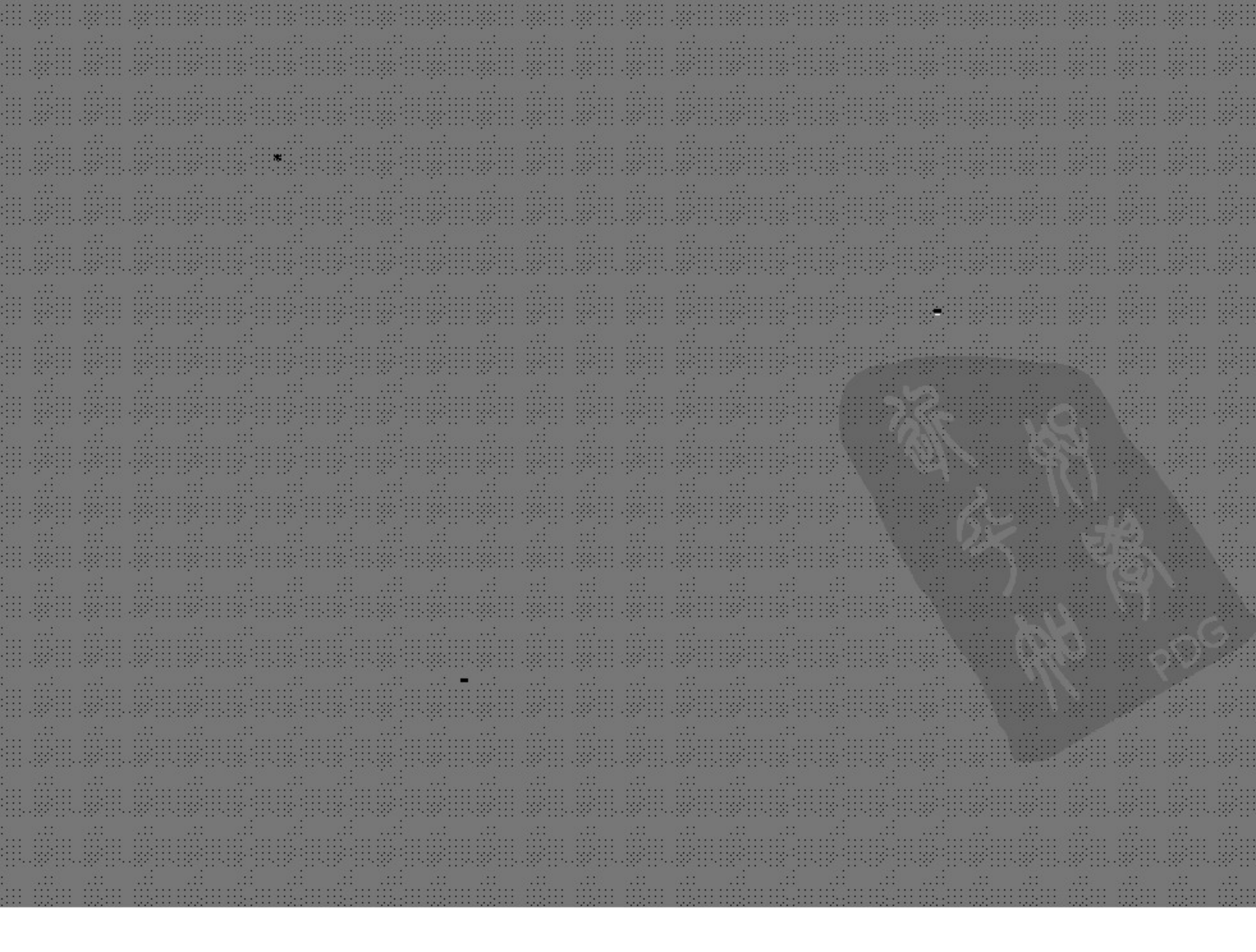
1.2.4　备选的解决方案

通过单独查看 SQL 语句，DBA 和架构师可以选择是否创建一个前面所讨论的索引。优化 SQL 的正确方法包括理解和验证此 SQL 语句以及与表相关的 SQL 语句的目的。通过执行这种分析，你将会发现执行该 SQL 语句的应用程序代码已经含有改进查询的补充信息。supp_id 的值在执行这句 SQL 语句的时候才能确定。可以在 SQL 语句的 WHERE 子句中添加这一列作为条件，这样现有的索引就可以被使用，为了改进 SQL 语句没有必要改变任何模式。

本例中，通过添加索引并不是解决查询速度慢的理想方法。如果不做进一步的分析，该表将会因为存在额外的不必要的索引而导致开销。

1.2　本章小结

优化 SQL 语句绝不仅仅是添加索引。本章介绍了几种用于优化语句的分析工具，包括 EXPLAIN 和 SHOW CREATE TABLE。我们重点关注那些能够识别性能问题的属性，并大致描述出原始的重要信息。我们详细讨论了一些需要考虑的添加索引时会影响到其他操作的点，我们还通过提供另一个可选的方案强调了一些从业务角度出发需要考虑到的因素。



第 2 章

基本的分析命令

在 MySQL 中，单一的工具或命令无法完成 SQL 语句的优化功能。本章将要介绍的 SQL 命令能够基本涵盖所有可能用到的 SQL 优化技术，包括创建索引；但如果单纯学习这些命令我们还是不

知道在什么场合应该使用什么样的优化技术。SQL 优化是一门需要数据库管理员来分析事实并推断出一个或多个可选方案的艺术。这些命令中有一些读者已经在第 1 章接触过了。

以下就是本章将要介绍的命令：

- EXPLAIN 命令
- SHOW CREATE TABLE 命令
- SHOW INDEXES 命令
- SHOW TABLE STATUS 命令
- SHOW STATUS 命令
- SHOW VARIABLES 命令

2.1 EXPLAIN 命令

要确定一条想要运行的 SQL 语句的 QEP，EXPLAIN 命令是必不可少的工具。用户可以使用这条命令深入了解 MySQL 的基于开销的优化器，还可以获得很多可能被优化器考虑到的访问策略的细节，以及当运行 SQL 语句时哪种策略预计会被优化器采用。需要注意的是，生成的 QEP 并不确定，它可能会根据很多因素而发生改变。MySQL 不会将一个 QEP 和某个给定查询绑定，QEP 将由 SQL 语句每次执行时的实际情况确定，即便使用存储过程也是如此。尽管在存储过程中 SQL 语句都是预先解析过的，但 QEP 仍然会在每次调用存储过程的时候才被确定。

下面列出的两个 QEP 的例子展示了添加一个 WHERE 条件对 SELECT 语句的查询执行计划会产生怎样的影响：

```
mysql> EXPLAIN SELECT host,user,password
    -> FROM mysql.user
```

```
    -> WHERE user LIKE 'r%'\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: user
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 39
        Extra: Using where

mysql> EXPLAIN SELECT host,user,password
    -> FROM mysql.user
    -> WHERE host='localhost'
    -> AND user LIKE 'r%'\G

********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: user
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 228
          ref: NULL
         rows: 5
        Extra: Using where
```

在本书中，我们将会阅读许多不同的 QEP 的例子，并详细介绍各种技术。第 9 章将会详细介绍这些例子中每一列代表的意义。下面的一些简单提示可以帮助我们快速找到大致的问题所在：

- 没有使用索引(key 列的值为 NULL)
- 很多处理过的行(rows 列)

- 很多被评估的索引(possible_keys 列)

提示

理想情况下，读者应该对系统中的每条 SQL 语句都运行 EXPLAIN 命令。所有 SELECT 语句前都可以直接加上 EXPLAIN 关键字。而对于 UPDATE 和 DELETE 语句，需要把查询重写成 SELECT 语句以确保有效使用索引。

有两个可选的关键字可以和 EXPLAIN 命令一起使用：PARTITIONS 和 EXTENDED 关键字。

2.1.1 EXPLAIN PARTITIONS 命令

EXPLAIN 命令的 PARTITIONS 关键字(MySQL 5.1 之后)对用于满足在 partitions 列中的查询的特定表分区提供附加信息，请看下面的例子：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM audit _log
WHERE yr IN (2011,2012)\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: audit_log
   partitions: p2,p3
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 2
        Extra: Using where
```

2.1.2　EXPLAIN EXTENDED 命令

EXPLAIN 命令的 EXTENDED 关键字提供了额外的 filtered 列，下面是一个例子：

```
mysql> EXPLAIN EXTENDED SELECT t1.name -> FROM test1
t1 INNER JOIN test2 t2 USING(uid)\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra:
********************** 2. row **********************
           id: 1
  select_type: SIMPLE
        table: t2
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 98
          ref: func
         rows: 1
     filtered: 100.00
        Extra: Using where; Using index
2 rows in set, 1 warning (0.00 sec)
```

此语法也可以深入了解实际执行的 SQL 语句。MySQL 优化

器可能会在运行时简化或重写用户的 SQL 命令，例如在用户使用视图(view)或者一个总是满足的搜索条件(例如 WHERE 1=1)时。

在个别情况下，EXTENDED 关键字还可以解释一些其他问题，例如为什么一个显然存在的索引最后却没有被用到。此时最重要的输出结果不是来自 EXPLAIN EXTENDED 命令，而是由后面执行的 SHOW WARNINGS 命令产生的。请看下面的例子：

```
mysql> SHOW WARNINGS\G
********************** 1. row **********************
  Level: Note
   Code: 1003
Message: select 'book'.'t1'.'name' AS 'name' from
'book'.'test1' 't1' join 'book'.'test2' 't2' where
(convert('book'.'t1'.'uid' using utf8) =
'book'.'t2'.'uid')
```

在这个例子中，两个表都有一个主键 uid，而 EXPLAIN 计划并没有显示预期的连接。进一步分析 EXPLAIN EXTENDED 表明，由于字符集转换的缘故，实际上查询被重写了。

2.2 SHOW CREATE TABLE 命令

SHOW CREATE TABLE 命令以一种便于阅读和操作的格式向用户呈现基本表中的当前列和索引定义的全部细节。该命令告诉用户如何用准确的语法来重新创建数据库表，并且用户可以很容易地在给定表上针对新的或更改的索引、数据类型、是否为 null 限制条件、字符集以及存储引擎创建优化。

```
mysql> SHOW CREATE TABLE wp_options\G
*********************** 1. row ***********************
       Table: wp_options
Create Table: CREATE TABLE 'wp_options' (
  'option_id' bigint(20) unsigned NOT NULL
  AUTO_INCREMENT,
  'blog_id' int(11) NOT NULL DEFAULT '0',
  'option_name' varchar(64) NOT NULL DEFAULT '',
  'option_value' longtext NOT NULL,
  'autoload' varchar(20) NOT NULL DEFAULT 'yes',
  PRIMARY KEY ('option_id'),
  UNIQUE KEY 'option_name' ('option_name')
) ENGINE=MyISAM AUTO_INCREMENT=4138 DEFAULT
CHARSET=utf8
1 row in set (0.00 sec)
```

注意

在引用数据库对象时，'符号一般情况下并不是必需的。只有当使用的模式对象名是一个保留字或者其中包含空格时才必须使用这个符号。而这两种情况是架构师和开发人员需要避免的，所以这个反向符号一般不会用到。但是在 SHOW CREATE TABLE 命令生成的语法中，这个符号是不可避免的。

Mysql dump 工具能够快速生成用户模式或数据库实例中所有表的定义。请看下面的例子：

```
$ mysqldump -u user -p --no-data [name-of-schema] >
schema.sql
$ more schema.sql
```

用户也可以使用 INFORMATION_SCHEMA 表获取表结构的组件，这些 INFORMATION_SCHEMA 的表包括：TABLES、COLUMNS、TABLE_CONSTRAINTS、KEY_COLUMN_USAGE、REFERENTIAL_CONSTRAINTS(从 MySQL 5.1 起开始支持)以及

PARTITIONS(从 MySQL 5.1 起开始支持)。

2.3 SHOW INDEXES 命令

用户可以用 SHOW INDEXES 命令查看索引的信息，这些信息包括索引的类型和当前报告的 MySQL 索引的基数。

```
mysql> SHOW INDEXES FROM wp_posts;
+-----+-------------------+---+------------+-----+--------------+   +-----
| Non.| Key_name          |Seq|Column_name |Col. | Cardinality |...| Index
+-----+-------------------+---+------------+-----+--------------+   +-----
|   0 | PRIMARY           | 1 | ID         | A   |         2689 |   | BTREE
|   1 | post_name         | 1 | post_name  | A   |         2689 |   | BTREE
|   1 | type_status_date| 1 | post_type  | A   |            4 |   | BTREE
|   1 | type_status_date| 2 | post_status | A   |            6 |   | BTREE
|   1 | type_status_date| 3 | post_date  | A   |         2689 |   | BTREE
|   1 | type_status_date| 4 | ID         | A   |         2689 |   | BTREE
|   1 | post_parent       | 1 | post_parent | A   |          224 |   | BTREE
|   1 | post_author       | 1 | post_author | A   |            1 |   | BTREE
+-----+-------------------+---+------------+-----+--------------+   +-----
```

下面是所有输出结果列的列表：

```
mysql> SHOW INDEXES FROM wp_posts\G
*********************** 1. row ***********************
        Table: wp_posts
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: ID
    Collation: A
  Cardinality: 2689
     Sub_part: NULL
       Packed: NULL
         Null:
```

```
   Index_type: BTREE
      Comment:
Index_comment:
********************** 2. row **********************
...
```

也可以从 INFORMATION_SCHEMA.STATISTICS 表中获得同样的信息。

Cardinality 列的值非常重要，需要重点关注。该值代表在索引中每一列唯一值的数量的估计值。本例中，主键的 cardinality 表示 2 689 个唯一值，而 post_type 列只报告了 4 个唯一值。统计 cardinality 值的目的将在第 4 章讨论。

注意

可以阅读 MySQL 参考手册(http://dev.mysql.com/doc/refman/5.5/en/show-index.html)了解更多关于 SHOW INDEXES 的信息。

2.4　SHOW TABLE STATUS 命令

用户可以使用 SHOW TABLE STATUS 命令查看数据库表的底层大小以及表结构，其中包括存储引擎类型、版本、数据和索引大小、行的平均长度以及行数。

```
mysql> SHOW TABLE STATUS LIKE 'wp_posts'\G
********************** 1. row **********************
           Name: wp_posts
         Engine: MyISAM
        Version: 10
     Row_format: Dynamic
           Rows: 2689
 Avg_row_length: 2847
```

```
     Data_length: 7657456
Max_data_length: 281474976710655
   Index_length: 269312
      Data_free: 0
 Auto_increment: 3627
    Create_time: 2011-02-02 12:04:21
    Update_time: 2011-04-17 00:20:20
     Check_time: 2011-02-02 12:04:21
      Collation: utf8_general_ci
       Checksum: NULL
 Create_options:
        Comment:
```

这条命令的返回值的准确度取决于数据库使用的存储引擎。例如，如果使用 MyISAM、MEMORY、ARCHIVE 或者 BLACKHOLE 存储引擎，那么行的平均长度和行数就都是精确值。而如果是 InnoDB 存储引擎，上述值都将是估计值并且可能有很大误差。从下面的例子可以看出 MyISAM 和 InnoDB 存储引擎的值的差异。

```
mysql> ALTER TABLE wp_posts ENGINE=InnoDB;
mysql> SELECT COUNT(*) FROM wp_posts;
+-----------+
| COUNT(*)  |
+-----------+
| 2689      |
+-----------+

mysql> SHOW TABLE STATUS LIKE 'wp_posts'\G
*********************** 1. row ***********************
           Name: wp_posts
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 2532
 Avg_row_length: 5597
```

福建师范大学
图书馆
藏书印记


```
     Data_length: 14172160
Max_data_length: 0
   Index_length: 589824
      Data_free: 745537536
...

mysql> SHOW TABLE STATUS LIKE 'wp_posts'\G
********************* 1. row ************************
...
           Rows: 1842
...
mysql> SHOW TABLE STATUS LIKE 'wp_posts'\G
...
           Rows: 3794
...

mysql> SELECT COUNT(*) FROM wp_posts;
+-----------+
| COUNT(*)  |
+-----------+
|      2689 |
+-----------+
```

用户也可以从 INFORMATION_SCHEMA.TABLES 表中获取底层表的信息和与 SHOW TABLE STATUS 命令类似的信息。使用 INFORMATION_SCHEMA 表来获取这些信息的好处是，用户可以用 SQL SELECT 语法来对输出信息进行限制、排序并指定输出格式。请看下面的例子：

```
SET @schema = IFNULL(@schema,DATABASE());
SET @table='inventory';
SELECT  @schema AS table_schema, CURDATE() AS today;
SELECT    table_name,
          engine,row_format AS format, table_rows,
          avg_row_length AS avg_row,
          round((data_length+index_length)/1024/10
```

1040625

```
            24,2) AS total_mb,
            round((data_length)/1024/1024,2) AS
            data_mb,
            round((index_length)/1024/1024,2) AS
            index_mb
      FROM INFORMATION_SCHEMA.tables
       WHERE table_schema=@schema
      AND table_name = @table
      \G
      ********************** 1. row **********************
       table_name: inventory
           engine: InnoDB
           format: Compact
      table_ rows: 609209
          avg_row: 414
        total_mb: 279.67
         data_mb: 240.70
        index_mb: 38.97
```

警告

对 INFORMATION_SCHEMA 表的 SELECT 查询可能需要执行很长时间。对于一个包含很多表并且每个表都很大的模式，本例中的查询执行时间高达 60 秒以上。

2.5 SHOW STATUS 命令

SHOW [GLOBAL | SESSION] STATUS 命令可以用来查看 MySQL 服务器的当前内部状态信息。这些信息是非常重要的，可以从全局角度帮助用户确定 MySQL 服务器的负载的各种指标。在任何特定的会话中，上述信息可以帮助用户从大量变量中找到有用的信息，从而进一步判断目标 SQL 语句可能对服务器产生的影响——例如一个内部临时表的使用、一次索引范围的扫描以及

读取的行的数量。如果 SHOW STATUS 没有给出范围(中括号中可选的关键字)，则默认为 SESSION。下面看一个例子：

```
mysql> SHOW GLOBAL STATUS LIKE 'Created_tmp_%tables';
+-------------------------+--------+
| Variable_name           | Value  |
+-------------------------+--------+
| Created_tmp_disk_tables | 551371 |
| Created_tmp_tables      | 674259 |
+-------------------------+--------+
2 rows in set (0.00 sec)

mysql> SHOW SESSION STATUS LIKE 'Created_tmp_%tables';
+-------------------------+--------+
| Variable_name           | Value  |
+-------------------------+--------+
| Created_tmp_disk_tables | 0      |
| Created_tmp_tables      | 2      |
+-------------------------+--------+
2 rows in set (0.00 sec)
```

用户也可以从INFORMATION_SCHEMA.GLOBAL_STATUS表和 INFORMATION_SCHEMA.SESSION_STATUS(从 MySQL 5.1 起支持)表中得出同样的信息。

对这些状态变量的理解可以帮助一个有经验的架构师更好地优化 MySQL 内部执行 SQL 语句的过程。下面就是一个对帮助提高索引效率的状态值进行比较例子：

```
mysql> FLUSH STATUS;
mysql> SELECT ...
mysql> SHOW SESSION STATUS LIKE 'handler_read%';
+---------------------------+--------+
| Variable_name             | Value  |
+---------------------------+--------+
| Handler_read_first        | 0      |
```

```
| Handler_read_key          | 0      |
| Handler_read_last         | 0      |
| Handler_read_next         | 0      |
| Handler_read_prev         | 0      |
| Handler_read_rnd          | 0      |
| Handler_read_rnd_next     | 479629 |
+---------------------------+--------+
```

这些信息表示没有使用任何关键字。

在以下的查询中，Handler_read_key 的值是 1，这表示使用了一个索引，Handler_read_next 的值是 71 表示这个索引被用来读取其他 71 行数据。

```
mysql> FLUSH STATUS;
mysql> SELECT ...
mysql> SHOW SESSION STATUS LIKE 'handler_read%';
+---------------------------+--------+
| Variable_name             | Value  |
+---------------------------+--------+
| Handler_read_first        | 0      |
| Handler_read_key          | 1      |
| Handler_read_last         | 0      |
| Handler_read_next         | 71     |
| Handler_read_prev         | 0      |
| Handler_read_rnd          | 0      |
| Handler_read_rnd_next     | 0      |
+---------------------------+--------+
```

在下一个查询中可以看到 Handler_read_prev 的值是 71，表示这个索引被用来从反方向读取了其他 71 行数据。

```
mysql> FLUSH STATUS;
mysql> SELECT ... ORDER BY 1 DESC
mysql> SHOW SESSION STATUS LIKE 'handler_read%';
+---------------------------+--------+
| Variable_name             | Value  |
```

```
+--------------------------+---------+
| Handler_read_first       | 0       |
| Handler_read_key         | 1       |
| Handler_read_last        | 0       |
| Handler_read_next        | 0       |
| Handler_read_prev        | 71      |
| Handler_read_rnd         | 0       |
| Handler_read_rnd_next    | 0       |
+--------------------------+---------+
```

2.6　SHOW VARIABLES 命令

SHOW [GLOBAL | SESSION] VARIABLES 命令可以用来查看 MySQL 系统变量的当前值。其中一些变量的值会影响到 SQL 语句执行的方式，例如，tmp_table_size 限制了内部创建的临时表的最大内存使用量。通过了解当前会话或者特殊设置的全局变量的值并用 SET 命令动态地更改这些变量的值，可以改变 SQL 语句的性能。当没有指定 SHOW VARIABLES 命令的范围时，默认在 SESSION 范围内执行。请看下面的例子：

```
mysql> SHOW SESSION VARIABLES LIKE 'tmp_table_size';
+-------------------------+----------+
| Variable_name           | Value    |
+-------------------------+----------+
| tmp_table_size          | 33554432 |
+-------------------------+----------+
```

用户也可以从 INFORMATION_SCHEMA.GLOBAL_VARIABLES 表和 INFORMATION_SCHEMA.SESSION_VARIABLES 表(从 MySQL 5.1 起支持)中获取同样的信息。当用户需要找出变量的 SESSION 的值和 GLOBAL 范围的差别时，上述信息将会很有帮助。请看下面的例子：

```
mysql> SELECT 'SESSION' AS scope,variable_name,variable_value
    -> FROM INFORMATION_SCHEMA.SESSION_VARIABLES
    -> WHERE variable_name IN
    ('tmp_table_size','max_heap_table_size')
    -> UNION
    -> SELECT 'GLOBAL',variable_name,variable_value
    -> FROM INFORMATION_SCHEMA.GLOBAL_VARIABLES
    -> WHERE variable_name IN
    ('tmp_table_size','max_heap_table_size')
+---------+----------------------+-----------------+
| scope   | variable_name        | variable_value |
+---------+----------------------+-----------------+
| SESSION | MAX_HEAP_TABLE_SIZE  | 16777216        |
| SESSION | TMP_TABLE_SIZE       | 33554432        |
| GLOBAL  | MAX_HEAP_TABLE_SIZE  | 16777216        |
| GLOBAL  | TMP_TABLE_SIZE       | 16777216        |
+---------+----------------------+-----------------+
```

2.7 INFORMATION_SCHEMA

INFORMATION_SCHEMA 为本章很多的 SHOW 命令提供了一个 ANSI SQL 语句接口。INFORMATION_SCHEMA 包含的表的数量取决于用户的 MySQL 版本。MySQL 5.5 目前有 37 个默认表。以下是本章介绍的一系列命令：

SHOW CREATE TABLE	TABLES, COLUMNS, KEY_COLUMN_USAGE, REFERENTIAL_CONSTRAINTS
SHOW INDEXES	STATISTICS
SHOW TABLE STATUS	TABLES
SHOW STATUS	GLOBAL_STATUS, SESSION_STATUS
SHOW VARIABLES	GLOBAL_VARIABLES, SESSION_VARIABLES

注意

想了解 INFORMATION_SCHEMA 中全部的表，请参考 MySQL 参考手册 http://dev.mysql.com/doc/refman/5.5/en/information-schema.html。

2.8 本章小结

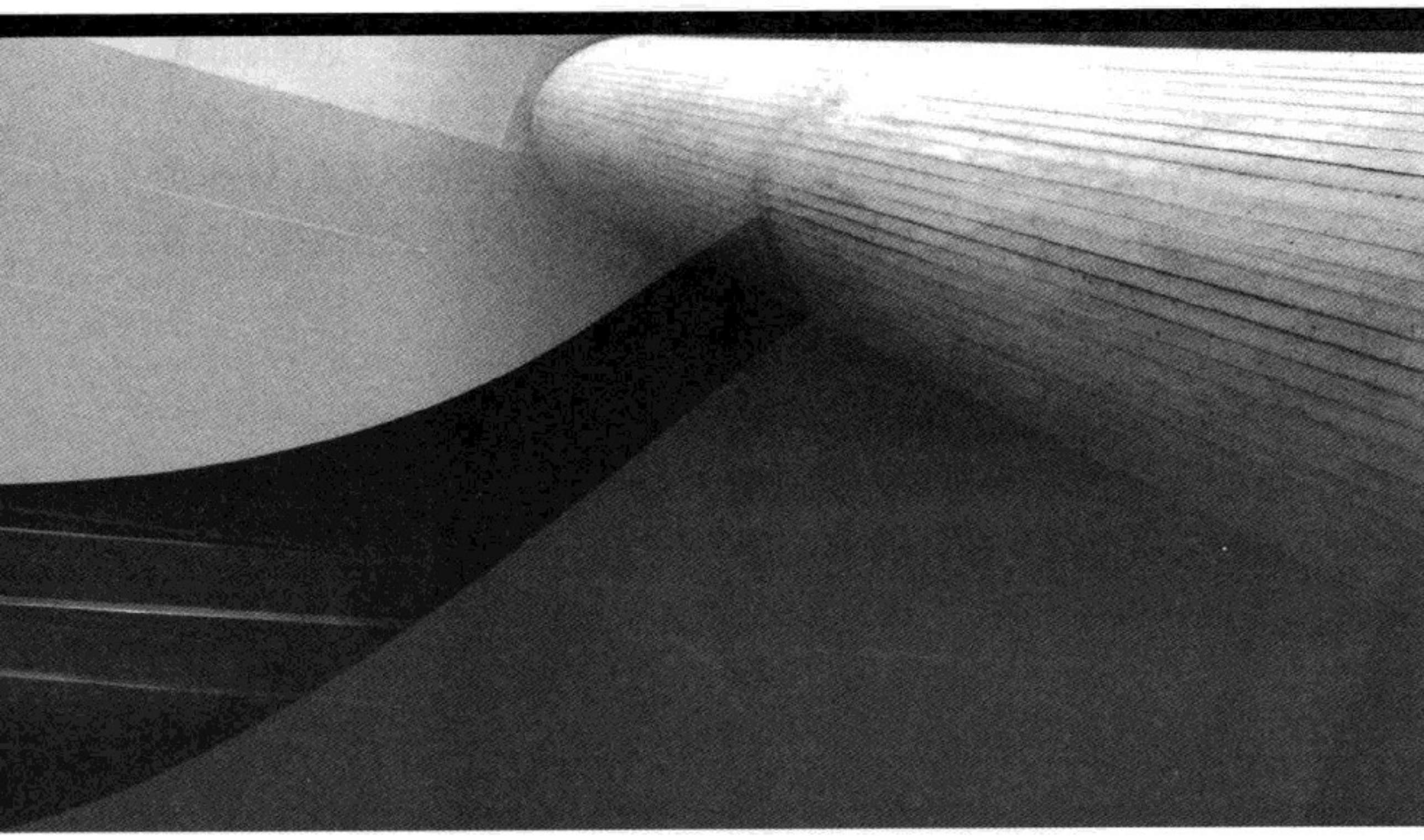
本章介绍的命令都是 SQL 优化实践中的必备技能，并且都是实践中的基本技能。尽管如此，包括时间、数据发生改变、存储引擎以及 MySQL 版本在内的很多因素都会影响 SQL 语句的执行。所以当优化 SQL 语句时，并不是一次性使用这些命令就可以解决问题。用户可能需要多次重复获取信息然后分析信息这样的过程才能最终解决问题。

技巧

掌握 MySQL EXPLAIN 语句需要多年不断地经验积累，这样才能达到在 SQL 优化中运用自如的程度。

本章介绍过的所有相关 SQL 语句都可以从以下网站下载：http://effectivemysql.com/book/optimizing-sql-statements。

PDG



第 3 章

深入理解 MySQL 的索引

创建合适的索引是 SQL 性能调优中最重要的技术之一。在稍后学习创建 MySQL 索引之前，读者需要首先了解 MySQL 的架构细节，包括索引在硬盘上是如何组织的、索引和内存用法和操作

方式，以及存储引擎的差异如何影响到索引的选择。这些知识都是在 MySQL 生产环境中精通优化索引创建的先决条件。读者可能会觉得本章的内容有些枯燥，但是索引的特性以及在 MySQL 中操作索引的详细知识会比其他 RDBMS 产品更显著地提升读者作为 DBA 的能力。

本章将介绍下列内容：

- MySQL 索引各种可能的用途
- 理解各种索引数据结构理论
- 各种存储引擎的索引实现方式
- 分区的 MySQL 索引

3.1 示例表

在本章中，我们会创建和引用一些表来演示各种 MySQL 索引的实现方式。下面的 SQL 语句就是用来创建许多单独的示例表并在这些表中插入适量数据的：

```
CREATE SCHEMA IF NOT EXISTS book
USE book;
CREATE TABLE source_words (
  word VARCHAR(50) NOT NULL,
  INDEX (word)
) ENGINE=MyISAM;
LOAD DATA LOCAL INFILE '/usr/share/dict/words'
INTO TABLE source_words(word);
CREATE TABLE million_words(
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  word VARCHAR(50) NOT NULL,
  PRIMARY KEY (id),
  UNIQUE INDEX (word)
) ENGINE=InnoDB;
```

```
INSERT INTO million_words(word)
SELECT DISTINCT word FROM source_words;
INSERT INTO million_words(word)
SELECT DISTINCT REVERSE(word) FROM source_words
WHERE REVERSE(word) NOT IN (SELECT word FROM
source_words);
SELECT @cnt := COUNT(*) FROM million_words;
SELECT @diff := 1000000 - @cnt;
-- We need to run dynamic SQL to support a variable LIMIT
SET @sql = CONCAT("
INSERT INTO million_words(word)
SELECT DISTINCT CONCAT(word,'X1Y') FROM source_words
LIMIT ",@diff);
PREPARE cmd FROM @sql;
EXECUTE cmd;
SELECT COUNT(*) FROM million_words;
```

本例中用到的输入文件是一个 Community Enterprise OS (CentOS) 5.5 Linux 发行版中的标准程序库文件。在不同的操作系统下文件的大小也会有变化。读者如果想精确地重现这个示例，可以从下面这个地址获得这个文件的副本：http://EffectiveMySQL.com/downloads/words。

3.2　MySQL 索引用法

MySQL 的索引有很多用途。索引并不仅仅是在读取数据时优化 MySQL 的性能。这些功能还包括：

- 保持数据完整性
- 优化数据访问性能
- 改进表的连接(join)操作
- 对结果进行排序
- 简化聚合数据操作

3.2.1 数据完整性

MySQL 用主键和唯一键(unique key)来执行每个表中存储数据的唯一性等级。主键和唯一键的差别有以下几点：

1. 主键

- 每个表只能有一个主键。
- 主键不能包含 NULL 值。
- 通过主键可以获取表中任意特定行。
- 如果定义了 AUTO INCREMENT 列，那么此列必须是主键的一部分。

2. 唯一键

- 表可以有多个唯一键
- 唯一键可以包含 NULL 值，并且每个 NULL 值都是唯一的(即 NULL != NULL)。

million_words 表的主键在 id 列上，这个限制条件确保没有重复值。请看下面的例子：

```
mysql> INSERT INTO million_words(id,word)
VALUES(1,'xxxxxxxxx');
ERROR 1062 (23000): Duplicate entry '1' for key
'PRIMARY'
```

同理，million_words 也包含一个唯一键在 word 列上，这个限制条件确保重复的值不会被再次添加。

```
mysql> INSERT INTO million_words(word)
VALUES('oracle');
```

```
ERROR 1062 (23000): Duplicate entry 'oracle' for key
'word'
```

第 4 章将介绍更多关于主键和唯一键索引的例子。

此外，有些 MySQL 的存储引擎可以创建外键来保证数据完整性。外键其实不是索引，它们属于约束。然而通常情况下大部分外键约束实现的先决条件就是外键所在的表和外键参照的表都必须有索引，这样才能管理外键约束。目前在 MySQL 的默认存储引擎中，只有 InnoDB 支持外键约束且不要求存在对应的索引；但考虑到性能因素，还是建议在添加外键约束时建立索引。

警告

尽管 MyISAM 引擎不支持外键约束，但仍可通过 REFERENCES 语法在 CREATE TABLE (...) ENGINE=MyISAM 语句中定义外键。

3.2.2　优化数据访问

索引可以让优化器在执行查询的时候不必检索表中所有数据。通过限制需要访问的行的数目，可以显著提高查询速度。这是索引最普遍的用途。

例如，在含一百万个词的范例表中，如果 word 列没有添加索引，那么每一句 SELECT 语句都要随机地逐条扫描一百万行数据，每次都要从中寻找 0 或更多匹配的行。虽然这些数据最初就是按顺序加载的，但 SQL 也不能理解这种顺序，它必须要处理所有行才能找到匹配的数据。

在下面的示例中，我们要创建一个没有索引的表：

```
mysql> CREATE TABLE no_index_words LIKE
million_words;
mysql> ALTER TABLE no_index_words DROP INDEX word;
mysql> INSERT INTO no_index_words SELECT * FROM
```

```
million_words;
mysql> SELECT * FROM no_index_words WHERE
word='oracle';
+---------+---------+
| id      | word    |
+---------+---------+
| 266877  | oracle  |
+---------+---------+
1 row in set (0.25 sec)
```

当表在 word 列有索引的时候，每个 SELECT 语句都会先扫描索引，索引是有序存储的并且以搜索为目标进行了优化，以确定对 0 或更多包含匹配信息的行的引用。当索引被定义为唯一索引时，SELECT 语句就知道查询结果最多包含一行匹配数据。下面是另一个用到了 million_words 表的示例：

```
mysql> SELECT * FROM million_words WHERE
word='oracle';
+---------+---------+
| id      | word    |
+---------+---------+
| 266877  | oracle  |
+---------+---------+
1 row in set (0.00 sec)
```

通过 MySQL 客户端的输出可以看到，在有索引列的例子中获取一行数据使用的时间小于 10 毫秒，而在没有索引的例子中获取一行数据花费了 250 毫秒。

警告

添加索引并不总是能够自动改善所有类型的 SQL 查询的性能。有时候执行全表扫描反而会更高效，这取决于所要求的行数。这就是两种不同访问方式的差异，即通过随机 I/O 操作来获取个别行的数据和使用索引查询及有序 I/O 操作来读取所有数据。

稍后我们将会举出更多更详细的示例来说明如何将索引用于查询限制。

3.2.3　表连接

除了在给定表上限制需要读取的数据外，索引的另一个主要用途就是快捷高效地在相关的表之间做连接操作。如前所述，在需要连接的列上使用索引可以显著提升性能，并可以在另一个表中快速找到一个匹配的值。掌握创建正确的索引来高效执行表连接的操作是所有关系型数据库 SQL 性能优化的基础。

3.2.4　结果排序

MySQL 索引把数据存储在一个有序的表格中。如果希望 SELECT 语句的结果是有序的，那么就应该适当地使用索引。通过 ORDER BY 关键字可以对任意 SELECT 查询的结果进行排序。如果在需要排序的列上没有找到索引，MySQL 一般会对获取的表行进行内部排序。在高并发系统中使用预先建立的索引可以带来巨大的性能改进，这些系统每秒需要对几百几千条单独的查询结果进行排序，此后这些查询结果顺理成章地按顺序存放在索引中了。但是仅仅简单地根据读者对查询结果的顺序要求创建一个索引并不总是意味着 MySQL 会选用它。第 4 章会讨论 MySQL 如何选用索引以及它的限制和不足。

3.2.5　聚合操作

索引还可以作为一种更方便的计算聚合结果的工具。例如在计算指定时期内所有账单的总和时，如果在日期和账单账户上添加合适的索引就可以更高效地执行。第 4 章将会通过例子介绍如何通过创建支持聚合操作的合适的索引。

3.3 关于存储引擎

如果读者对 MySQL 不熟悉，或者更了解其他关系型数据库系统，那么可能需要花一些时间来理解存储引擎这个概念。总体来说，尽管 MySQL 是通过结构化查询语言(SQL)来与数据交互并管理数据，但是在 MySQL 内部可以有多种不同的机制来管理存储并且获取底层数据。MySQL 的存储引擎的灵活性是一把双刃剑。本书不打算介绍存储引擎的细节部分，但读者很有必要了解一些有关存储引擎的功能和特性的基本信息，主要包括以下几方面：

- 事务性和非事务性
- 持久性和非持久性
- 表和行级锁定
- 不同的索引方法，例如 B-树、B+树、散列以及 R-树
- 聚簇索引和非聚簇索引
- 主码索引和非主码索引
- 数据压缩
- 全文索引能力

MySQL 支持来自其他服务提供商的插件化存储引擎，这些引擎中有开源的也有商业的。作为一个开源产品，MySQL 有很多种衍生版本，这些衍生版本支持更多不同种类的存储引擎。

本书将会集中讨论三种主要的存储引擎，这三种也是 MySQL 默认包含的引擎：

- **MyISAM** 一种非事务性的存储引擎，是 MySQL 5.5 之前版本默认的存储引擎

- **InnoDB** 最流行的事务性存储引擎，从 5.5 版开始成为 MySQL 的默认引擎
- **Memory** 顾名思义，这是一种基于内存的、非事务性的以及非持久性的存储引擎

注意

从 5.5 版本开始，MySQL 表的默认存储引擎从 MyISAM 换成了 InnoDB。这将会在用户安装那些依赖默认设置或者专门为 MyISAM 编写的软件包时带来很大影响。

当前版本的 MySQL 也包括一些内置的存储引擎，例如 ARCHIVE、MERGE、BLACKHOLE 以及 CSV 等。其他一些由 MySQL 或者第三方提供的存储引擎还包括 Federated、ExtraDB、TokuDB、NDB、Maria、InfinDB 和 Infobright 等。

小技巧

用户可以使用 SHOW CREATE TABLE、SHOW TABLE STATUS 或者 INFORMATION_SCHEMA.TABLES 来查看任何给定表使用的存储引擎。第 2 章介绍过这些命令并给出了详细示例。

想了解更多关于 MySQL 存储引擎的信息及其详细列表，请访问 http://EffectiveMySQL.com/article/storage-engines/。

3.4　索引专业术语

读者可能容易混淆对索引的所有专业术语的理解。以下是对这些术语的介绍：

索引技术	这个词是关于不同数据结构如何用不同的方法访问底层信息的理论。这些技术包括 B-树、B+树、R-树以及散列。每一种技术都采用不同的概念来实现一种特定目标或数据结构的优势
索引实现	这个词是关于 MySQL 及各种存储引擎实现不同的数据结构技术的方法。例如，MyISAM 引擎实现 B-树的方法和 InnoDB 实现的方法就有所不同
索引类型	这个词是描述索引的普遍用法的。索引的类型包括主键，唯一键，非主码索引，全文本索引以及空间类型。每种类型支持在单一列、多列(也叫做混合列)或者列的一部分上定义特定的索引类型。最后，这些索列类型中的一个或多个会成为所谓的覆盖索引

在本章接下来的内容和第 4 章中将详细讨论这些术语。

3.5 MySQL 索引类型

MySQL 支持在所有关系型数据库表中创建主键、唯一键、不唯一的非主码索引等多种类型的索引。此外，在一些特定的存储引擎中 MySQL 还支持纯文本和空间索引类型。

MySQL 内置的存储引擎对各种索引技术有不同的实现方式，包括：B-树，B+树，R-树以及散列。在更多其他插件式的存储引擎中还支持使用其他的索引实现方法，例如 TokuDB(参考 http://tokutek.com/technology)引擎就使用了 Fractal Tree 索引，而 InfiniDB(参考 http://www.calpont.com)和 Infobright(参考 http://infobright.com)引擎则使用面向列的存储方式。

这些使用不同数据结构技术的索引实现方式是非常重要的，因为它们会直接影响到 SQL 查询在数据库中的执行方式。看一个简单的例子，散列实现对直接查找方式能提供最优的性能，但对一定范围的查找却效率低下；而一个 B-树的索引实现则是专门为给定范围的查询设计的。我们可以使用数学方法评估一种特定数据结构的有效性和效率。而且，InfiniDB 和 Infobright 引擎提供的面向列的索引实现则完全不需要用户定义的索引策略。

3.5.1　索引数据结构理论

在我们介绍不同的索引类型如何在 MySQL 中实现之前，先了解一些主要用到的数据结构的基本理论将会对读者很有帮助。

1. B-树

B-树数据结构支持数据插入、控制操作以及通过管理一系列树根状结构的彼此联通的节点中来做选择。B-树中有两种节点类型：索引节点和叶子节点。叶子节点是用来存储数据的，而索引节点则用来告诉用户存储在叶子节点中的数据的顺序，并帮助用户找到相应数据。注意不要把 B-树和二叉树混淆了，二叉树只是一种简单的节点层次结构的实现。

B-树数据结构如图 3-1 所示。如果想了解关于 B-tree 数据结构的概念、数学证明以及技术描述的信息，参见 http://en.wikipedia.org/wiki/Btree。

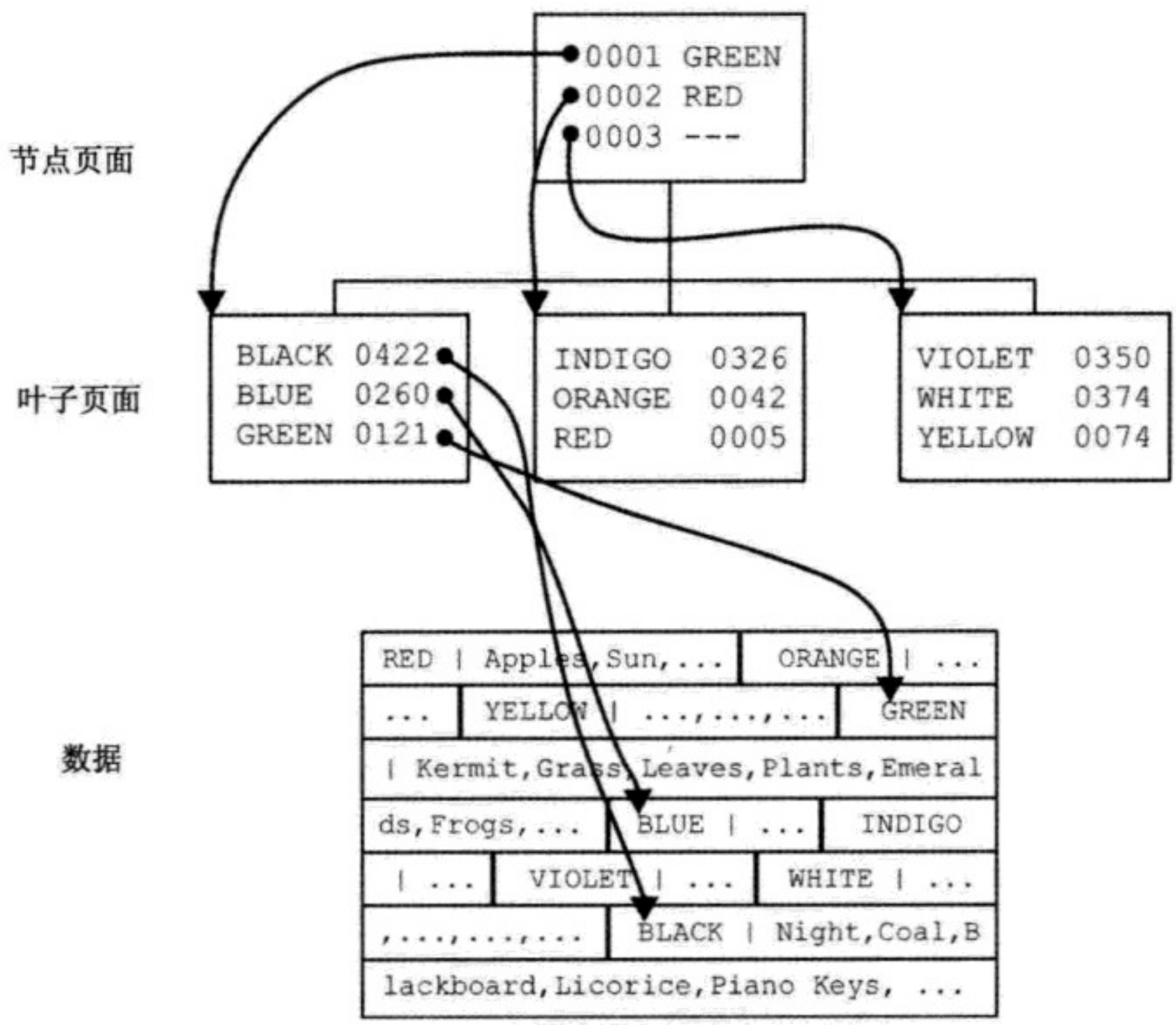


图 3-1 B-树数据结构

2. B+树

B+树数据结构是 B-树实现的增强版本。尽管 B+树支持 B-树索引的所有特性，它们之间的最显著不同点在于 B+树中底层数据是根据被提及的索引列进行排序的。B+树还通过在叶子节点(没有子节点的节点)之间的附加引用来优化扫描性能。

B+树数据结构如图 3-2 所示。想了解更多 B+树数据结构的概念和技术细节可以访问 http://en.wikipedia.org/wiki/B%2B_tree。

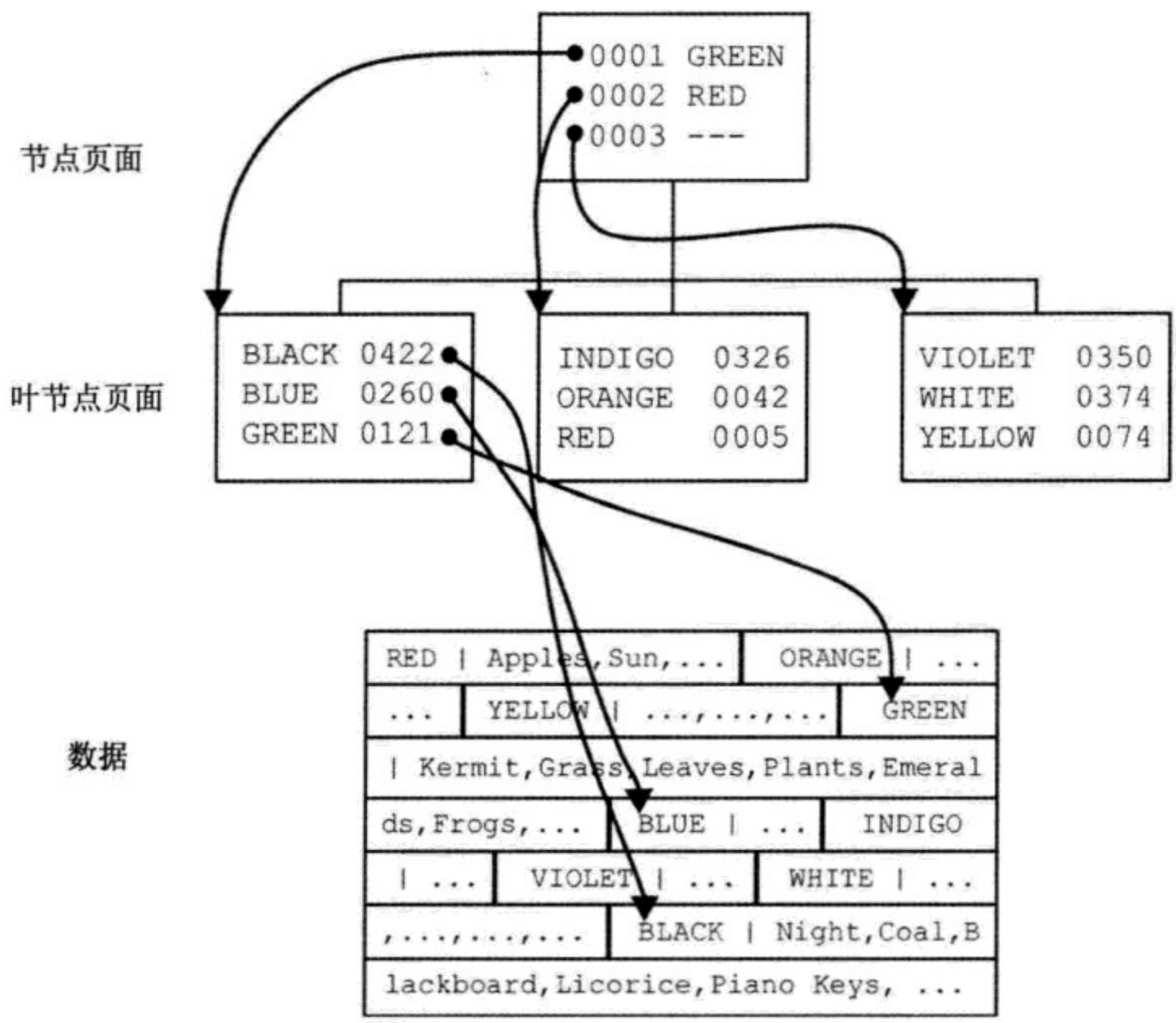


图 3-2　B+树数据结构

3. 散列

散列表数据结构是一种很简单的概念，它将一种算法应用到给定值中以在底层数据存储系统中返回一个唯一的指针或位置。散列表的优点是始终以线性时间复杂度找到需要读取的行的位置，而不像 B-树那样需要横跨多层节点来确定位置。

散列树数据结构如图 3-3 所示。想了解更多散列表的概念和技术细节可以访问 http://en.wikipedia.org/wiki/Hash_table。

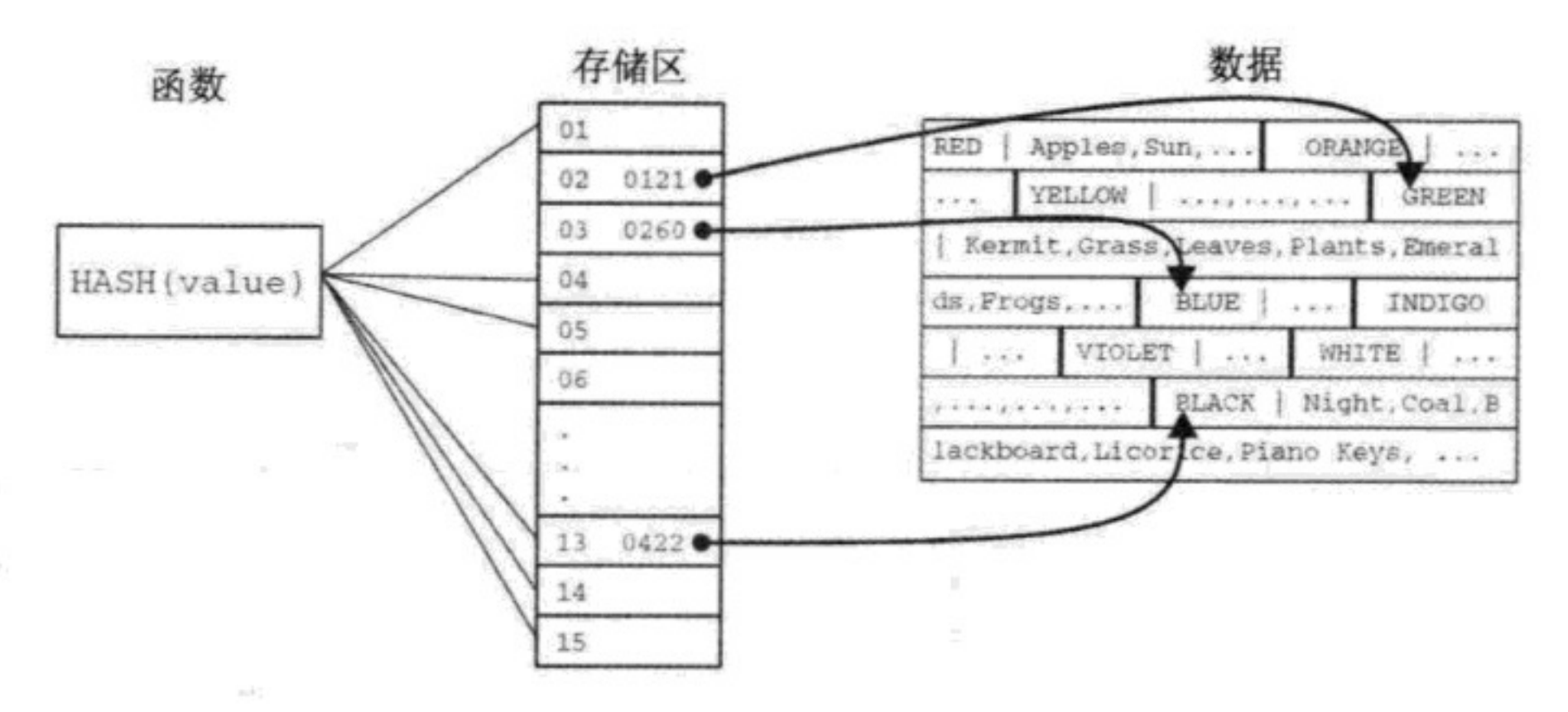


图 3-3 散列树的数据结构

4.通信 R-树

R-树数据结构支持基于数据类型对几何数据进行管理。目前只有 MyISAM 使用 R-树实现支持空间索引。使用空间索引也有很多限制，比如只支持唯一的 NOT NULL 列等。由于空间索引并不常用，本书将不再讨论空间索引的更多细节。

想了解 R-树的概念和技术细节请访问 http://en.wikipedia.org/wiki/R-tree。

5. 全文本

全文本结构也是一种 MySQL 采用的基本数据结构。这种数据结构目前只有当前版本 MySQL 中的 MyISAM 存储引擎支持。在 MySQL 5.6 版本中将要在 InnoDB 存储引擎中加入全文本功能。全文本索引在大规模系统中并没有什么实用价值，因为在这些大规模系统中有很多专门用于文本检索的产品。由于全文本索引在大规模系统中的实用性有限，本书将不再进一步讨论全文本索引的细节。

想了解全文本索引的概念和技术细节请访问 http://en.wikipedia.org/wiki/Full_text_search。

技巧

Sphinx 搜索服务器可以以插件存储引擎的方式被整合以支持许多全文本特性并且还能和 MySQL 中的可用数据进行整合，更多信息请参考 http://sphinxsearch.com/。

3.5.2 MySQL 实现

对 B-树、B+树和散列等数据结构的基本概念有了一些了解之后，我们就可以开始讨论 MySQL 通过支持它们的存储引擎如何实现不同的算法。每种实现都在 SQL 性能方面有一些和某些查询类型相关的优点。同时每种实现也对磁盘和内存使用情况有不同的影响，这一点在大型数据库系统中是非常重要的考虑因素。

1. MyISAM 的 B-树

MyISAM 存储引擎使用 B-树数据结构来实现主码索引、唯一索引以及非主码索引。在 MySQL 实例数据目录和数据库模式子目录中，用户可以找到和每个 MySQL 表对应的.MYD 和 MYI 文件。数据库表上定义的索引信息就存储在 MYI 文件中，该文件的块大小是 1 024 字节。这个大小是可以通过 myisam-block-size 系统变量配置的。

```
$ ls -lh /var/lib/mysql/book/source_words.MY*
-rw-rw---- 1 mysql mysql 9.2M 2011-04-30 19:08
source_words.MYD
-rw-rw---- 1 mysql mysql 7.8M 2011-04-30 19:08
source_words.MYI
```

这些文件结构的内部格式可以从MySQL免费的源代码中找到，

也可以查看 MySQL 内部手册 http://forge.mysql.com/wiki/MySQL_Internals_MyISAM#The_.MYI_file。

在 MyISAM 中，非主码索引的 B-树结构存储索引值和一个指向主码数据的指针，这是 MyISAM 和 InnoDB 的一个显著区别。这一点导致了两种存储引擎中的索引的不同工作方式。第 4 章将通过一个例子专门描述这个问题。

MyISAM 索引是在内存的一个公共键缓存中管理的，这个缓存的大小可以通过 key_buffer_size 或者其他命名键缓存来定义。这是在根据统计和规划的表索引的大小来设定缓存大小时主要的考虑因素。上述配置的选项将在第 6 章介绍。

实际上通过简单地分析底层文件结构可以知道 MyISAM 是如何实现 B-树数据结构的。下面的代码创建了一个小的示例表和相应的索引：

```
CREATE TABLE colors (
  name VARCHAR(20) NOT NULL,
  items VARCHAR(255) NOT NULL
) ENGINE=MyISAM;INSERT
INTO colors(name, items) VALUES
('RED','Apples,Sun,Blood,...'),
('ORANGE','Oranges,Sand,...'),
('YELLOW','...'),
('GREEN','Kermit,Grass,Leaves,Plants,Emeralds,Frogs,
Seaweed,Spinach,Money,Jade,Go Traffic Light'),
('BLUE','Sky,Water,Blueberries,Earth'),
('INDIGO','...'),
('VIOLET','...'),
('WHITE','...'),
('BLACK','Night,Coal,Blackboard,Licorice,Piano
Keys,...');
ALTER TABLE colors ADD index (name);
```

读者可以通过直接查看 MyISAM 数据文件来了解数据是根据 INSERT 语句的指定来有序存放的。下面就是一个例子：

```
$ od -c colors.MYD0000000 001\0 031 003R  E  D
024  A  p  p  l  e  s  ,   S
0000020un, Blood, . . . 003\0 030\0
0000040 006  O  R  A  N  G  E
020  O  r  a  n  g  e  s  ,
0000060Sand, . . . 003\0\v 005 006Y  E  L
0000100  L  O  W 003. . . \0\0\0\0\0 003\0\ \
00000120 005  G  R  E  E  N  U  K  e  r  m  i  t  ,
G  r
0000140  a  s  s  ,L  e  a  v  e  s  ,
P  l  a  n  t
0000160  s  ,E  m  e  r  a  l  d  s  ,F  r
o  g  s
0000200  ,S  e  a  w  e  e  d  ,
S  p  i  n  a  c  h
0000220  ,M  o  n  e  y  ,J  a  d  e  ,
G  o     T
0000240rafficLight 001\0! 004
0000260  B  L  U  E 033  S  k  y  ,W  a  t  e  r  ,B
0000300  l  u  e  b  e  r  r  i  e  s  ,
E  a  r  t  h
0000320 003\0\f 004\aI  N  D  I  G  O 003  . . . .
0000340\0\0\0\0 003\0\v 005 006VIOLET 003
0000360. . . \0\0\0\0\0 003\0\n 006 005W  H  I
0000400  T  E 003. . . \0\0\0\0\0\0 003\04\
00000420 005  B  L  A  C  K  -N  i  g  h  t  ,
C  o  a
0000440  l  ,B  l  a  c  k  b  o  a  r  d  ,
L  i  c
0000460  o  r  i  c  e  ,P  i  a  n  o     K  e  y  s
0000500  , . .
```

也可以通过查看 MyISAM 索引来观察升序排列的值，例如：

```
$ od -c colors.MYI...0002000 \0 y \0 005 B L A C K \0
\0 \0 \0 001 \f \0
0002020 004 B L U E \0 \0 \0 \0 \0 254 \0 005 G R E
0002040 E N \0 \0 \0 \0 \0 L \0 \a I N D I G O
```

```
0002060 \0 \0 \0 \0 \0 \0 320 \0 006 O R A N G E \0
0002100 \0 \0 \0 \0 034 \0 003 R E D \0 \0 \0 \0 \0
\0
0002120 \0 006 V I O L E T \0 \0 \0 \0 \0 344 \0 005
0002140 W H I T E \0 \0 \0 \0 \0 370 \0 006 Y E L
0002160 L O W \0 \0 \0 \0 \0 8 003 \0 \0 \0 \0 \0 \0...
```

2. InnoDB 的 B+树聚簇主码

InnoDB 存储引擎在它的主码索引(也被称为聚簇主码)中使用了 B+树。这种结构把所有数据都和对应的主码组织在一起，并且在叶子节点这一层上添加额外的向前和向后的指针，这样就可以更方便地进行范围扫描操作。

注意

在其他数据库产品比如 Oracle 中，这种类型的结构叫做索引组织的表。

在文件系统层面，所有 InnoDB 数据和索引信息都默认在公共 InnoDB 表空间中管理，否则管理员就通过 innodb_data_file_path 这个变量指定文件路径。这是一个叫做 ibdata1 的文件，可以在 MySQL 数据目录中找到。请看下面的例子：

```
$ ls -lh /var/lib/mysql/ibdata1
-rw-rw---- 1 mysql mysql 92.5G 2011-04-30 20:28
/var/lib/mysql/ibdata1
```

可以用 SHOW TABLE STATUS 命令或者 INFORMATION_SCHEMA.TABLE 表中的信息来确定每个表的大小。用户也可以通过指定 innodb_file_per_table 选项来定义 InnoDB 为每个表使用单独的表空间，这样用户就可以查看每个文件中合并的表数据和索引的大小了。这种每个表单独使用的表空间可以在对应模式的

子目录中找到。请看下面的例子：

```
$ ls -lh /var/lib/mysql/data/million_words.ibd
-rw-rw---- 1 mysql mysql 80M 2011-04-30 19:09
million_words.ibd
```

由于 InnoDB 用聚簇主码存储数据，底层信息占用的磁盘空间的大小很大程度上取决于页面的填充因子。对于按序排列的主码，InnoDB 会用 16K 页面的 15/16 作为填充因子。对于不是按序排列的主码，默认情况下 InnoDB 会在插入初始数据的时候为每个页面分配 50%作为填充因子。了解这个填充因子对于估计潜在的磁盘存储空间需求是至关重要的。例如，我们可以查看已有表的大小，并创建一个基于非排序主码的新表作为对比。

```
$ cat tablesize.sql
SET @schema = IFNULL(@schema,DATABASE());
SELECT @schema AS table_schema, CURDATE() AS today;
SELECT  table_name,
        engine,row_format AS format, table_rows,
        avg_row_length AS avg_row,
        round((data_length+index_length)/1024/1024
        ,2) AS
        total_mb,
        round((data_length)/1024/1024,2) AS data_mb,
        round((index_length)/1024/1024,2) AS
        index_mb
FROM    INFORMATION_SCHEMA.tables
WHERE   table_schema=@schema
AND     table_name = @table
\G
mysql> SET @table='million_words';
mysql> SOURCE tablesize.sql
*********************** 1. row ************************
table_ name: million_words
     engine: InnoDB
```

```
        format: Compacttable_rows: 985821
      avg_row: 36
     total_mb: 70.14
      data_mb: 34.56
     index_mb: 35.58
mysql> CREATE TABLE million_words2 (
    -> id INT UNSIGNED NOT NULL,
    -> word VARCHAR(50) NOT NULL,
    -> PRIMARY KEY (word),
    -> UNIQUE KEY(id))
    -> ENGINE=InnoDB;mysql> SELECT word,id FROM
million_words ORDER BY id
    -> INTO OUTFILE '/tmp/million_words.tsv';
mysql> LOAD DATA LOCAL INFILE
'/tmp/million_words.tsv'
    -> INTO TABLE million_words2(word,id);
mysql> SET @table='million_words2';
mysql> SOURCE tablesize.sql
*********************** 1. row ***********************
table_name: million_words2
    engine: InnoDB
    format: Compact
table_rows: 1294686
   avg_row: 53
  total_mb: 86.19
   data_mb: 66.63
  index_mb: 19.56
```

在原来那张表上使用 AUTO_INCREMENT 和序列主码可以看到底层数据大小是 34MB。而同样的数据集合使用无序的主码产生的数据大小是 66MB。尽管这并不是一种精确的衡量手段，但也可以说明默认填充因子对于存储空间的影响。

标准的 MySQL 产品目前不提供调整这些存储因子计算量的方法。此外，理解存储因子底层实现的重要性在于所有 InnoDB 数据和索引都是在内存中通过 innodb_buffer_pool_size 设置选项

定义的 InnoDB 缓存池进行管理。随着系统不断变大，这种对磁盘布局和内存占用将会直接影响到 SQL 语句的性能。

这些文件结构的内部格式可以从 MySQL 源代码(是免费的)中找到，也可以查看 MySQL 内部手册 http://forge.mysql.com/wiki/MySQL_Internals_InnoDB。

想要在文件系统中查看文件内容并试图验证 InnoDB 比验证 MyISAM 的示例复杂很多。如果我们使用前面描述过的颜色表示例，那么在 InnoDB 中将看不到任何有序数据，如下所示：

```
$ od -c colors.ibd
0140200   R   E   D  \0  \0  \0  \0   "   x 200  \0 002   @ 002 001 020
0140220   A   p   p   l   e   s   ,   S   u   n   ,   B   l   o   o   d
0140240   ,   .   .   . 020 006 037 005  \0 030 377 324   O   R   A   N
0140260   G   E  \0  \0  \0  \0   "   x 200  \0 002   @ 002 001 034   O
0140300   r   a   n   g   e   s   ,   S   a   n   d   ,   .   .   . 003
0140320 006 037  \0  \0     377 231   Y   E   L   L   O   W  \0  \0  \0
```

InnoDB 把数据存储在 16K 的数据页面中，这里不介绍其技术细节。每个页面中 B+树结构相对于所有页面而言是有序的，而每个页面的底层数据可以是无序的。想要在 InnoDB 中看到有序的数据，需要增加每一行的大小以减少每个数据页中的行数。请看下面的例子：

```
CREATE TABLE colors_wide (
name VARCHAR(20) NOT NULL,
items VARCHAR(255) NOT NULL, filler1 VARCHAR(500) NULL,
PRIMARY KEY (name)
) ENGINE=InnoDB;
INSERT INTO colors_wide(name, items) VALUES
('RED','Apples,Sun,Blood,...'),
('ORANGE','Oranges,Sand,...'),
('YELLOW','...'),
('GREEN','Kermit,Grass,Leaves,Plants,Emeralds,Frogs,
Seaweed,Spinach,Money,Jade,Go Traffic Light'),
```

```
('BLUE','Sky,Water,Blueberries,Earth'),
('INDIGO','...'),
('VIOLET','...'),
('WHITE','...'),
('BLACK','Night,Coal,Blackboard,Licorice,Piano
Keys,...');
UPDATE colors_wide SET filler1=REPEAT('x',500);
$ od -c colors_wide.ibd
*
 0147520   x   x   x   x   x 364 201 003 006  \0  \0  \0 210 002 024   v
 0147540   I   O   L   E   T  \0  \0  \0  \0   " 204  \0  \0  \0  \0   -
 0147560   !   )   .   .   .   x   x   x   x   x   x   x   x   x   x   x
 0147600   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
*
 0150540   x   x   x   x   x   x   x   x   x 364 201 003 005  \0  \0  \0
 0150560 220 002 023   W   H   I   T   E  \0  \0  \0  \0   " 204  \0  \0
 0150600  \0  \0   -   !   M   .   .   .   x   x   x   x   x   x   x   x
 0150620   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   X
*
 0151560   x   x   x   x   x   x   x   x   x   x   x   x 364 201 003 006
 0151600  \0  \0  \0  \0 230 354 352   Y   E   L   L   O   W  \0  \0  \0
```

注意

笔者不建议在任何重要的分析过程中使用这种方法。这种分析只是让读者有个基本的概念，相对于 MyISAM，数据在 InnoDB 中是按照有序的方式存储的。

3. InnoDB 的 B-树非主码

InnoDB 中的非主码索引使用了 B-树数据结构；但 InnoDB 中的 B-树结构实现和 MyISAM 中的并不一样。在 InnoDB 中，非主码索引存储的是主码的实际值。而在 MyISAM 中，非主码索引存储的是包含主码值的数据的指针。

这一点是很重要的。首先，当定义了很大的主码的时候——例

如当读者的主码长度是 40 字节的时候，InnoDB 的非主码索引可能会更大。随着非主码索引数量的增加，索引之间大小差别可能会变得很大。另一个不同点在于非主码索引当前可以包含主键的值，并且可以不是索引必须有的部分。在做表连接操作或者使用覆盖索引的时候，这将会带来很大的性能改进。以上将在第 5 章详细介绍。

4. 内存散列索引

在默认 MySQL 的引擎索引中，只有 MEMORY 引擎支持散列数据结构，散列也是主码索引和非主码索引的默认结构。散列结构的强度可以表示为直接键查找的简单性。以下示例展示了不同查询的对比情况：

```
SET SESSION max_heap_table_size = 1024 *1024 * 100;
INSERT INTO memory_words(id,word)
SELECT id,word FROM million_words;
SELECT COUNT(*) FROM memory_words;
SET PROFILING=1;
SELECT * FROM memory_words WHERE word = 'apple';
SELECT * FROM memory_words WHERE word = 'orange';
SELECT * FROM memory_words WHERE word = 'lemon';
SELECT * FROM memory_words WHERE word =
'wordnotfound';
SELECT * FROM memory_words WHERE word LIKE 'apple%';
// Matches 91 rows
SHOW PROFILES;
+-----------+-----------+-------------------------
| Query_ID  | Duration  | Query
+-----------+-----------+-------------------------
|         1 | 0.00020900 | SELECT * FROM memory_words
WHERE word = 'apple'
|         2 | 0.00021100 | SELECT * FROM memory_words
WHERE word = 'orange'
```

```
|          3  | 0.00021600 | SELECT * FROM memory_words
WHERE word = 'lemon'
|          4  | 0.00022500 | SELECT * FROM memory_words
WHERE word = 'wordn...
|          5  | 0.08243100 | SELECT * FROM memory_words
WHERE word LIKE 'apple%'
+-----------+------------+-----------------------------
```

在这个例子中，散列索引的相似范围模式匹配查询比直接值查询慢 1 000 倍。

也可以为 MEMORY 存储引擎指定一个 B-树索引实现。从下面的信息可以看出表数据和索引空间的大小，这可以和稍后给出的 MEMORY 使用 B-树结构的情况做个比较。

```
mysql> SET @table='memory_words';
mysql> SOURCE tablesize.sql
*********************** 1. row ***********************
table_name: memory_words
    engine: MEMORY
    format: Fixed
table_rows: 1000000
   avg_row: 55
  total_mb: 84.61
   data_mb: 53.85
  index_mb: 30.76
```

5. 内存 B-树索引

对大型 MEMORY 表来说，使用散列索引进行索引范围搜索的效率很低，这一点在之前的例子中也可以看出。使用 B-树索引重复运行同样的 SQL 命令，结果如下：

```
SET SESSION max_heap_table_size = 1024 *1024 * 150;
ALTER TABLE memory_words DROP INDEX word,ADD INDEX
USING BTREE (word);
```

```
SHOW PROFILES;
+------------+-----------+---------------------------
| Query_ID   | Duration  | Query
+------------+-----------+---------------------------
|          1 | 0.00023900 | SELECT * FROM memory_words
WHERE word = 'apple'
|          2 | 0.00011000 | SELECT * FROM memory_words
WHERE word = 'orange'
|          3 | 0.00010600 | SELECT * FROM memory_words
WHERE word = 'lemon'
|          4 | 0.00010500 | SELECT * FROM memory_words
WHERE word = 'wordn...
|          5 | 0.00014300 | SELECT * FROM memory_words
WHERE word LIKE 'apple%'
```

从下面输出结果的最后一行可以看出索引空间容量的增加：

```
mysql> SET @table='memory_words';
mysql> SOURCE tablesize.sql
********************** 1. row **********************
table_name: memory_words
    engine: MEMORY
    format: Fixed
table_rows: 1000000
   avg_row: 55
  total_mb: 109.92
   data_mb: 53.85
  index_mb: 56.07
```

可以看出， B-树索引的大小是散列索引的两倍之多。

警告

在这个例子中，读者会发现 B-树索引在执行直接键查询时确实比使用默认的散列索引快。根据 B-树的不同深度，B-树索引在个别操作中的确可能比散列算法快。这个例子强调的是测试索引的可变性在寻找特殊用例的最优解决方案时是十分重要的。

6. InnoDB 内部散列索引

InnoDB 存储引擎在聚簇 B+树索引中存储主码；但在 InnoDB 内部还是使用内存中的散列表来更高效地进行主码查找。这个机制由 InnoDB 存储引擎来管理，用户只能通过 innodb_adaptive_hash_index 配置项来选择是否启用这个唯一的配置选项。

注意

想了解更多关于 InnoDB 自适应散列的信息请访问：http://dev.mysql.com/doc/refman/5.5/en/innodb-adaptive-hash.html 和 http://dev.mysql.com/doc/innodb/1.1/en/innodb-performanceadaptive_hash_index.html

3.6 MySQL 分区

从 5.1 版本起 MySQL 开始支持通过范围、列表、散列、键以及列等对表分区的方式。分区的实现对索引的使用和优化也有一些帮助。

一个已分区的表不支持全文本索引、空间索引以及外键索引。分区表上的主索引和唯一索引必须包含分区表达式中用到的所有列。

分区的一个优势就是使得执行 SQL 语句时启用分区精简。就像第 2 章对 EXPLAIN PARTIONS 命令介绍的那样，MySQL 可以通过控制分区来实现只扫描一些用到的索引，而不是扫描所有索引。

注意

想了解关于分区索引的限制和不足的更多信息请访问 http://dev.mysql.com/doc/mysql-reslimits-excerpt/5.5/en/partitioning-limitations.html。

3.7 本章小结

本章讨论了几种常用数据结构，以及不同的 MySQL 存储引擎如何实现这些数据结构来支持不同的索引类型。虽然本章并没有详细介绍如何创建能优化 SQL 语句的索引，但了解在 MySQL 中索引的用途以及索引的不同表现方式是非常重要的。接下来我们将详细介绍的大量示例都会参考到本章介绍的信息。

读者可以从下面这个地址下载所有本章中用到的 SQL 语句：http://effectivemysql.com/book/optimizing-sql-statements。

第4章

创建 MySQL 索引

本章我们将会讨论索引的各种类型、MySQL 如何选择索引以及添加索引对系统整体性能的影响。我们还会介绍如何为特殊的给定查询创建索引。这并不是说如果你的环境中有类似的表就一

定要自动创建相同的索引；这些索引示例只是说明如何针对特定情况创建和评估索引。

创建索引并不是优化 SQL 语句的唯一方式。选择一种好的方式对数据库模式或者 SQL 语句中数据的应用程序的用法进行优化，通常可以对系统整体性能产生更加深远的影响。而创建索引通常能够在尽量不影响系统整体的情况下获得很大的性能提升，并且在一些特定环境中索引是最容易实现的优化方式。

本章将介绍下列内容：

- 基本索引类型，包括单列和多列索引
- MySQL 是如何使用索引的
- 添加索引对性能造成的影响
- 各种 MySQL 索引的限制和不足

虽然 MyISAM 存储引擎支持空间索引和全文本索引，但本章不讨论这两类索引的示例。

4.1 本章范例中用到的表

本章将用到一些示例表以及示例数据。下面就是主要用到的表：

```
CREATE TABLE artist (
  artist_id INT UNSIGNED NOT NULL,
  type ENUM('Band','Person','Unknown','Combination')
  NOT NULL,
  name VARCHAR(255) NOT NULL,
  gender ENUM('Male','Female') DEFAULT NULL,
  founded YEAR DEFAULT NULL,
  country_id SMALLINT UNSIGNED DEFAULT NULL,
  PRIMARY KEY (artist_id)
) ENGINE=InnoDB;
```

```
CREATE TABLE album (
  album_id INT UNSIGNED NOT NULL,
  artist_id INT UNSIGNED NOT NULL,
  album_type_id INT UNSIGNED NOT NULL,
  name VARCHAR(255) NOT NULL,
  first_released YEAR NOT NULL,
  country_id SMALLINT UNSIGNED DEFAULT NULL,
  PRIMARY KEY (album_id)
) ENGINE=InnoDB;
```

读者可以从以下地址下载本章中所有模式和用到的示例数据：http://EffectiveMySQL.com/downloads/music_example.tar.gz。表中的数据是从 MusicBrainz——一个地址为 http://musicbrainz.org 的开放音乐百科全书获取的，这个服务提供公开数据许可证。想了解更多有关信息可以访问 http://musicbrainz.org/doc/MusicBrainz_License。

注意

这些表的结构是专门为解释 MySQL 索引的可能用法而定义的，并用于描述不同的示例。它们并不代表在生产环境中应该像这样定义数据模型。

4.2　已有的索引

在大多数情况下，你可以在现有的表上使用已有的索引优化 SQL 语句。读者可以回顾第 2 章介绍过的工具。在本章中我们将会进一步通过示例说明如何利用这些工具帮助我们优化 SQL 语句。下面是一个很普通的查询语句，用来获取指定艺人的相关信息：

```
SELECT artist_id, type, founded
```

```
FROM artist
WHERE name = 'Coldplay';
```

首先要做的一定是在 SELECT 语句前添加 EXPLAIN 关键字来查看 QEP。这并不会真正执行 SQL 语句。

```
mysql> EXPLAIN SELECT artist_id, type, founded
    -> FROM artist
    -> WHERE name = 'Coldplay'\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ref
possible_keys: name
          key: name
      key_len: 257
          ref: const
         rows: 1
        Extra: Using where
```

注意

尽管 EXPLAIN 命令不会执行 SQL 语句，但当执行计划确定时它会执行 FROM 语句中的子查询。

在重新检查 SQL 语句时，一定不要忘了使用下面的命令验证表的结构、索引以及使用的存储引擎：

```
mysql> SHOW CREATE TABLE artist\G
*********************** 1. row ***********************
       Table: artist
Create Table: CREATE TABLE
  `artist` ( `artist_id` int(10) unsigned NOT NULL,
  `type` enum('Band','Person','Unknown','Combination')
  NOT NULL,
  `name` varchar(255) NOT NULL,
```

```
`gender` enum('Male','Female') DEFAULT NULL,
`founded` year(4) DEFAULT NULL,
`country_id` smallint(5) unsigned DEFAULT NULL,
PRIMARY KEY (`artist_id`),
KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

从输出结果可以得知，在 name 列上已经存在索引，并且该索引在执行查询的时候会被用到。

4.3 单列索引

单列索引是最基础的索引，这是一种建立在数据库表中特定列上的索引。MySQL 并没有限制在一个表的索引的数量，然而创建索引还是会对性能有影响，我们会在后面的 4.5 节“添加索引造成的影响”中讨论这个问题。MySQL 一般会为一个表选择唯一一个索引。从 MySQL 5.0 开始，在个别执行过程中优化器可能会用到不止一个索引。我们将会在 4.4.6 小节“MySQL 优化器的特性”中讨论这个问题。

4.3.1 创建单列索引的语法

可以使用下面的语法在一个现有的表上创建单列索引：

```
ALTER TABLE <table>
        ADD PRIMARY KEY [index-name] (<column>);

ALTER TABLE <table>
        ADD [UNIQUE] KEY|INDEX [index-name]
        (<column>);
```

注意

当创建非主码索引时，KEY 和 INDEX 关键字可以互换。但创建主码索引时只能使用 KEY 关键字。

4.3.2 利用索引限制查询读取的行数

假设在艺人数据表中按照出道时间来获取艺人信息是一个常用的操作，那我们就可以通过创建索引来避免每次都扫描整张表。如果在 EXPLAIN 的结果中看到 type=ALL 或者 key=NULL，则可以判断这条查询扫描了整张表，如下所示：

```
mysql> EXPLAIN SELECT artist_id, type, founded
    -> FROM artist
    -> WHERE founded=1942\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 587328
        Extra: Using where
```

可以用下面的语句在 founded 列上创建一个索引：

```
mysql> ALTER TABLE artist ADD INDEX (founded);
```

现在重新执行 EXPLAIN SELECT 语句来查看到新添加的索引是否被使用了。下面的输出结果显示新建的索引确实被用到了，并且估计需要读取的行数比以前少了很多，这将使查询执行得更快。

```
mysql> EXPLAIN SELECT artist_id, type, founded
    -> FROM artist
    -> WHERE founded=1942\G
********************** 1. row **************************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ref
 possible_keys: founded
          key: founded
      key_len: 2
          ref: const
         rows: 499
        Extra: Using where
```

前面提到过 MySQL 并不限制在一个表上创建索引的数目，用户甚至可以创建重复的索引。如果用户无意中重复创建了同样的索引，将会看到下列信息：

```
mysql> ALTER TABLE artist ADD INDEX (founded);
mysql> EXPLAIN ...
********************** 1. row **************************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ref
 possible_keys: founded,founded_2
          key: founded
      key_len: 2
          ref: const
         rows: 499
        Extra: Using where
```

注意

用户并不需要指定索引的名称。MySQL 会根据索引所在的首列的名称自动为索引命名，并在名字后面添加可选的附加信息确

保唯一性。从上面的示例中可以看到，我们在用一列上再次添加了索引。重复的索引会产生性能开销。在接下来的关于多列索引的部分我们将讨论如何发现并删除重复的索引。

尽管目前并没有命名索引的标准化规定，但当用户指定一张表中所有的索引名时除创建重复索引外可能还会导致其他错误信息。

4.3.3 使用索引连接表

索引的另一个好处就是可以提高关系表连接操作的性能。例如，下列 SQL 语句要获取指定艺人的专辑信息：

```
mysql> EXPLAIN SELECT ar.name, ar.founded, al.name,
al.first_released
    -> FROM artist ar
    -> INNER JOIN album al USING (artist_id)
    -> WHERE ar.name = 'Queen'\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: ar
         type: ref
 possible_keys: PRIMARY,name
          key: name
      key_len: 257
          ref: const
         rows: 1
        Extra: Using where
*********************** 2. row ***********************
           id: 1
  select_type: SIMPLE
        table: al
         type: ALL
```

```
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 535450
        Extra: Using where; Using join buffer
```

这个示例的结果显示 album 表会执行全表查询。我们可以通过为连接条件添加索引并重复 EXPLAIN 命令来解决这个问题。

```
mysql> ALTER TABLE album ADD INDEX (artist_id);
mysql> EXPLAIN ...
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: ar
         type: ref
 possible_keys: PRIMARY,name
          key: name
      key_len: 257
          ref: const
         rows: 1
        Extra: Using where
*********************** 2. row ***********************
           id: 1
  select_type: SIMPLE
        table: al
         type: ref
 possible_keys: artist_id
          key: artist_id
      key_len: 4
          ref: book.ar.artist_id
         rows: 1
        Extra:
```

在 album 表中我们现在使用了由 key 值新创建的 artist_id 索

引，并且可以看到 ref 的值显示 album 表将要和 artist 表做连接操作。

4.3.4 理解索引的基数

当一个查询中使用不止一个索引的时候，MySQL 会试图找到一个最高效的索引。它通过分析每条索引内部数据分布的统计信息来做到这一点。本例中我们要查询创建于 1980 年的所有品牌，因此我们在 artist 表的 type 列上创建一个索引，因为我们要在其上执行搜索。

```
mysql> ALTER TABLE artist ADD INDEX (type);
```

为了在所有 MySQL 5.x 版本中更好地说明这一点，在本例中我们要禁用一个优化器设置：

```
mysql> SET @@session.optimizer_switch='index_merge
intersection=off';
mysql> EXPLAIN SELECT artist_id, name, country_id
    -> FROM artist
    -> WHERE type='Band'
    -> AND founded = 1980\G
********************** 1. row **********************
            id: 1
   select_type: SIMPLE
         table: artist
          type: ref
 possible_keys: founded,founded_2,type
           key: founded
       key_len: 2
           ref: const
          rows: 1216
         Extra: Using where
```

本例中，MySQL 必须在 possible_keys 列出的索引中做出选择。优化器会根据最少工作量的估算开销来选择索引，这往往和人们想到的选择顺序不一样。我们可以使用索引基数来确定最有可能被选中的索引。请看下面的示例：

```
mysql> SHOW INDEXES FROM artist\G...
*********************** 3. row ************************
        Table: artist
   Non_unique: 1
     Key_name: founded
  Seq_in_index: 1
  Column_name: founded
    Collation: A
  Cardinality: 846
...
*********************** 5. row ************************
        Table: artist
   Non_unique: 1
     Key_name: type
  Seq_in_index: 1
  Column_name: type
    Collation: A
  Cardinality: 10
...
```

这些信息表明 founded 列拥有更高的基数，也就是说该列中唯一值的数量越多，那么越有可能在选用这个索引时以更少的读操作中找到需要的记录。这些统计信息只是估计值。从数据分析中我们可以知道，artist 表中 type 只有 4 个唯一值，但在统计信息中则不是这样。

关于基数不得不提的一点就是选择性。仅仅知道索引中唯一值的数目意义并不大，重要的是将这个数值和索引中的总行数做比较。选择性就是表中明确值的数量和表中包含的记录的总数的

关系。理想情况下，选择性值为 1，且每一个值都是一个非空唯一值。一个有着优秀选择性的索引意味着有更少的相同值的行。当某一列中仅仅有少数不同的值的时候就会有较差的选择性——例如性别或者状态列。当查询需要用到所有列时，这些信息不但可以帮助我们判断索引是否高效，还可以告诉我们如何在多列索引中对列进行排序。

结果中显示的索引基数提供了一些简单的线索。下面的两个查询想要查找 20 世纪 80 年代的乐队和组合。

```
mysql> EXPLAIN SELECT artist_id, name, country_id
    -> FROM artist
    -> WHERE founded BETWEEN 1980 AND 1989 AND
    type='Band'\G
********************** 1. row **********************
...
possible keys: founded,founded_2,type
          key: founded
      key_len: 2
          ref: NULL
         rows: 18690
        Extra: Using where

mysql> EXPLAIN SELECT artist_id, name, country_id
    -> FROM artist
    -> WHERE founded BETWEEN 1980 AND 1989 AND
    type='Combination'\G
********************** 1. row **********************
..
possible keys: founded,founded_2,type
          key: type
      key_len: 1
          ref: const
         rows: 19414
        Extra: Using where
```

这两个查询看起来很简单，但它们却根据列信息分布的详细统计信息选择了不同的索引路径。

4.3.5　使用索引进行模式匹配

利用通配符可以通过索引来做模式匹配的工作。请看下面的示例：

```
mysql> EXPLAIN SELECT artist_id, type, founded
    -> FROM artist
    -> WHERE name LIKE 'Queen%'\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: range
 possible_keys: name
          key: name
      key_len: 257
          ref: NULL
         rows: 93
        Extra: Using where
```

如果你查找的词是以通配符开头，则 MySQL 不会使用索引。请看下面的示例：

```
mysql> EXPLAIN SELECT artist_id, type, founded
    -> FROM artist
    -> WHERE name LIKE '%Queen%'\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ALL
 possible_keys: NULL
          key: NULL
```

```
       key_len: NULL
           ref: NULL
          rows: 585230
         Extra: Using where
```

技巧

如果你经常需要一个以通配符开头的查询，常用的方法是在数据库中保存需要查询的值的反序值。例如，假设你想要找所有以.com 结尾的电子邮件地址，当搜索 email Like '%.com' 时 MySQL 不能使用索引；而搜索 reverse_email LIKE REVERSE('%.com')就可以使用定义在 reverse_email 列上的索引。

MySQL 不支持基于索引的函数。如果想创建一个带有列函数的索引将会导致语法错误。不同数据库产品背景知识的开发者在执行下面的语句遇到一个共同问题，希望在 name 列上的一个索引能够被用来满足这个查询：

```
mysql> EXPLAIN SELECT artist_id, type, founded
    -> FROM artist
    -> WHERE UPPER(name) = UPPER('Billy Joel')\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 585230
        Extra: Using where
```

因为使用了应用到 name 列上的 UPPER 函数，MySQL 不会使用 name 上的索引。

技巧

MySQL 默认使用大小写敏感的字符集存储文本信息。因此没有必要用特殊的格式存储数据并在 SQL 语句中启用事件特定比较。

4.3.6 选择唯一的行

如果我们想要保证每个艺人都有一个唯一的名字，可以创建唯一索引。唯一索引有两个目的：

- 提供数据完整性以保证在列中任何值都只出现一次
- 告知优化器对给定的记录最多只可能有一行结果返回；这一点很重要，因为有了这些信息就可以避免额外的索引扫描

可以使用第 2 章介绍的 SHOW STATUS 命令来查看一般索引和唯一索引在查询内部造成的不同影响。下面就是一个使用已有的非唯一索引的示例：

```
mysql> FLUSH STATUS;
mysql> SHOW SESSION STATUS LIKE 'Handler_read_next';
mysql> SELECT name FROM artist WHERE name ='Enya';
mysql> SHOW SESSION STATUS LIKE 'Handler_read_next';
```

输出结果如下：

```
+--------------------+---------+
| Variable_name      | Value   |
+--------------------+---------+
| Handler_read_next  | 0       |
+--------------------+---------+
+------+
| name |
+------+
| Enya |
```

```
+------+
+---------------------+---------+
| Variable_name       | Value   |
+---------------------+---------+
| Handler_read_next   | 1       |
+---------------------+---------+
```

在内部，MySQL 会去读索引中下一项记录来判断 name 索引的下一个值不是那个指定的值。创建一个唯一索引并再次运行同一个查询，我们可以看到以下结果：

```
mysql> ALTER TABLE artist DROP INDEX name,
    -> ADD UNIQUE INDEX(name);

+---------------------+---------+
| Variable_name       | Value   |
+---------------------+---------+
| Handler_read_next   | 0       |
+---------------------+---------+
+------+
| name |
+------+
| Enya |
+------+
+---------------------+---------+
| Variable_name       | Value   |
+---------------------+---------+
| Handler_read_next   | 0       |
+---------------------+---------+
```

对比两个结果可以发现，当使用唯一索引时 MySQL 知道最多只可能返回一行数据，当找到一个匹配结果之后就不需要继续扫描了。当数据确实是唯一的情况下，把索引定义为唯一索引是非常好的方式。

技巧

在可以为空的列上定义唯一索引也是可行的。这种情况下，NULL 的值被认为是一个未知的值，并且 NULL!=NULL。这就是三态逻辑的好处，它避免了使用默认值或者一个空字符串值。

4.3.7　结果排序

索引也可以用来对查询结果进行排序。如果没有索引，MySQL 会使用内部文件排序算法对返回的行按照指定顺序进行排序。请看下面的示例：

```
mysql> EXPLAIN SELECT name,founded
    -> FROM artist
    -> WHERE name like 'AUSTRALIA%'
    -> ORDER BY founded\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: range
 possible_keys: name
          key: name
      key_len: 257
          ref: NULL
         rows: 22
        Extra: Using where; Using filesort
```

可以看到，通过在 Extra 的属性中设置了 Using filesort 信息，MySQL 内部使用 sort_buffer 来对结果进行排序。

也可以通过下面的命令从内部确认上述结论：

```
mysql> FLUSH STATUS;
mysql> SELECT ...
mysql> SHOW SESSION STATUS LIKE '%sort%';
```

```
+---------------------+---------+
| Variable_name       | Value   |
+---------------------+---------+
| Sort_merge_passes   | 0       |
| Sort_range          | 1       |
| Sort_rows           | 22      |
| Sort_scan           | 0       |
+---------------------+---------+
```

通过使用基于索引的数据排序方法，就可以免去分类的过程了，如下所示：

```
mysql> EXPLAIN SELECT name,founded
    -> FROM artist
    -> WHERE name like 'AUSTRALIA%'
    -> ORDER BY name\G
********************** 1. row ************************
...
     key: name
key_len: 257
     ref: NULL
    rows: 22
  Extra: Using where

mysql> FLUSH STATUS;
mysql> SELECT ...
mysql> SHOW SESSION STATUS LIKE '%sort%';
+---------------------+---------+
| Variable_name       | Value   |
+---------------------+---------+
| Sort_merge_passes   | 0       |
| Sort_range          | 0       |
| Sort_rows           | 0       |
| Sort_scan           | 0       |
+---------------------+---------+
```

接下来我们将讨论如何使用索引来限制返回的行并使用多列

索引对结果排序。

4.4　多列索引

索引可以创建在两列或多列上。多列索引也被称为混合索引或者连接索引。

4.4.1　确定使用何种索引

我们来看一个查询，这个查询根据 WHERE 语句的限制在表上使用两个不同的索引。我们首先创建这些索引。

```
mysql> ALTER TABLE album
    -> ADD INDEX (country_id),
    -> ADD INDEX (album_type_id);
Query OK, 553875 rows affected (18.89 sec)
```

尽可能地合并给定表 DML 语句会获得更高的效率。如果选择以两条独立语句的方式分别运行这些 ALTER 语句，则会有下面这样的结果：

```
mysql> ALTER TABLE album DROP index country_id, drop
index album_type_id;
Query OK, 553875 rows affected (15.72 sec)
mysql> ALTER TABLE album ADD INDEX (country_id);
Query OK, 553875 rows affected (16.76 sec)
mysql> ALTER TABLE album ADD INDEX (album_type_id);
Query OK, 553875 rows affected (25.23 sec)
```

如果这是一张生产环境规模的表，而每条 ALTER 语句运行需要 60 分钟或者 6 小时，那么合并 ALTER 语句会显著地节省时间。

技巧

创建索引是一件非常耗时的工作，并且会阻塞其他操作。你可以使用一条 ALTER 语句将给定表上多个索引创建的语句合并起来。

```
mysql> EXPLAIN SELECT al.name, al.first_released,
al.album_type_id
    -> FROM album al
    -> WHERE al.country_id=221
    -> AND album_type_id=1\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: al
         type: ref
possible_keys: album_type_id,country_id
          key: country_id
      key_len: 3
          ref: const
         rows: 154638
        Extra: Using where
```

注意

在这个示例中，根据你使用的 MySQL 不同版本，优化器可能经过改进而提供不同的 QEP。在所有 MySQL 5.x 版本中使用以下的 MySQL 向后兼容性系统变量的设置，都可以在以下示例中获得一样的结果：

```
mysql> SET @@session.optimizer_switch='index_merge_
intersection=off';
```

但是如果我们对 album type 的不同值运行同一个查询，那么使用不同的索引：

```
mysql> EXPLAIN SELECT al.name, al.first_released,
```

```
al.album_type_id
    -> FROM album al
    -> WHERE al.country_id=221
    -> AND album_type_id=4\G
*********************** 1. row ************************
           id: 1
  select_type: SIMPLE
        table: al
         type: ref
possible_keys: album_type_id,country_id
          key: album_type_id
      key_len: 4
          ref: const
         rows: 58044
        Extra: Using where
```

为什么 MySQL 会这样决定？从 EXPLAIN 语句的 rows 列我们可以得出这样的结论：基于开销的优化器会选择开销更小的方式，也就是说相对于读取 154 000 行，优化器选择读取 58 000 行的方案。

```
mysql> SHOW INDEXES FROM album\G...

*********************** 4. row ************************
       Table: album
  Non_unique: 1
    Key_name: album_type_id
Seq_in_index: 1
 Column_name: album_type_id
   Collation: A
 Cardinality: 12
...
*********************** 6. row ************************
       Table: album
  Non_unique: 1
    Key_name: country_id
```

```
Seq_in_index: 1
 Column_name: country_id
   Collation: A
 Cardinality: 499
...
```

如果 MySQL 仅仅使用索引基数，那么你可能会认为 QEP 总是会使用 country_id 列，因为该列拥有更多的唯一值并且可以获取更少的行。尽管索引基数是唯一性的一个重要指标，但 MySQL 也会参考有关唯一值的范围和容量等统计信息。我们可以通过查看实际表分布来确定这些数目。

```
mysql> SELECT COUNT(*) FROM album where
country_id=221;
+-----------+
| count(*)  |
+-----------+
|   92544   |
+-----------+
mysql> SELECT COUNT(*) FROM album where
album_type_id=4;
+-----------+
| count(*)  |
+-----------+
|  111908   |
+-----------+
mysql> SELECT COUNT(*) FROM album where
album_type_id=1;
+-----------+
| count(*)  |
+-----------+
|  289923   |
+-----------+
```

第一个查询选择了 country_id 列上的索引。实际结果显示获取了 92 000 行，而选择 album_type_id 则会有 289 000 行被获取。

第二个查询选择了 album_type 列上的索引的情况下，实际结果显示获取了 111 000 行，与之前的 92 000 行做比较。如果你把实际数量与 QEP 估算的行数做比较，你会发现一个矛盾——例如对于第二个查询估计有 58 000 行数据，但实际数据有 111 000 行之多，几乎是实际行数的两倍。

4.4.2 多列索引的语法

创建多列索引的语法和之前相同，唯一不同的是需要指定该索引是要跨越多列的：

```
ALTER TABLE <table>
        ADD PRIMARY KEY [index-name]
        (<column1>,<column2>...);

ALTER TABLE <table>
        ADD [UNIQUE] KEY|INDEX [index-name]
        (<column1>,<column2>...);
```

4.4.3 创建更好的索引

我们可以在国家和专辑类型列上创建多列索引，这样优化器就可以得到更多信息。请看下面的示例：

```
mysql> ALTER TABLE album ADD INDEX m1
(country_id, album_type_id);
```

然后我们重新运行下面的 SQL 语句可以得到如下 QEP：

```
mysql> EXPLAIN SELECT al.name, al.first_released,
al.album_type_id
    -> FROM album al
    -> WHERE al.country_id=221
    -> AND album_type_id=4\G
```

```
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: al
         type: ref
possible_keys: album_type_id,country_id,m1
          key: m1
      key_len: 7
          ref: const,const
         rows: 23800
        Extra: Using where
```

这次优化器选择使用新的索引了。你可能还会注意到 key_len =7。这是用来确定索引所使用的列的效率的工具。我们将会在更多示例中讨论这个问题。

按照这样的顺序来创建索引看起来是合理的；但由于你的查询同时使用到两列，你可能会选择使用相反的列序：

```
mysql> ALTER TABLE album ADD INDEX m2
(album_type_id,country_id);
```

再次查看 QEP 可以发现使用了新的索引：

```
mysql> EXPLAIN SELECT al.name, al.first_released,
al.album_type_id
    -> FROM album al
    -> WHERE al.country_id=221
    -> AND album_type_id=4\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: al
         type: ref
possible_keys: album_type_id,country_id,m1,m2
          key: m2
      key_len: 7
```

```
          ref: const,const
         rows: 18264
        Extra: Using where
```

我们可以查看该表的索引基数来找出选择这条索引的原因：

```
mysql> SHOW INDEXES FROM album\G
...
*********************** 7. row ***********************
        Table: album
     Key name: m1
Seq_in_index: 1
 Column_name: country_id
 Cardinality: 487
...
*********************** 8. row ***********************
        Table: album
     Key_name: m1
Seq_in_index: 2
 Column_name: album_type_id
 Cardinality: 960
...
*********************** 9. row ***********************
        Table: album
     Key_name: m2
Seq_in_index: 1
 Column_name: album_type_id
 Cardinality: 16
...
*********************** 10. row ***********************
        Table: album
     Key_name: m2
Seq_in_index: 2
 Column_name: country_id
 Cardinality: 682
...
```

如果我们仅仅观察索引基数，我们可能会认为 m1 将会被选

中，因为它提供了唯一行的更高分布；然而，这条信息并没有提供足够的可以用于决定当前执行计划的细节信息。

技巧

当你在对一个交集表使用多列索引时，尤其是在每一列都有指定值时，交换列的顺序可能会创建出更好的索引。

多列索引除了优化限制返回的行之外还有更重要的用途。多列索引中最左边的列也可以被当做单一列索引来高效地使用。当这些列被频繁用于聚合操作(即 GROUP BY 操作)和排序操作(即 ORDER BY 操作)时，最左边的列也同样可以显著提升性能。

4.4.4 多个列上的索引

虽然索引可以包含多列，但实际上对索引的效率会有所限制。索引是用于改进性能的关系模型的一部分。索引的行的宽度应该尽可能的短，这样就可以在一个索引数据页面中包含更多的索引记录。这样做的好处是可以读取尽量少的数据，从而尽可能快地遍历索引。你可能还希望你的索引保持这样的高效，这样能使系统内存的使用最大化。EXPLAIN 命令结果中的 key_len 和 ref 两个属性的值可以用来判断选中的索引的列利用率。请看下面的示例：

```
mysql> ALTER TABLE artist ADD index
(type,gender,country_id);

mysql> EXPLAIN SELECT name FROM artist WHERE type=
'Person' AND gender='Male' AND country_id = 13\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: artist
```

```
          type: ref
possible_keys: type,country_id,type_2
           key: type_2
       key_len: 6
           ref: const,const,const
          rows: 40
         Extra: Using where
```

从结果中可以看到，ref 列的值为 3 个常量，正好和索引中的 3 列匹配。key_len 的值为 6 也同样证实了这一点：ENUM 长度 1 字节，SMALLINT 类型长度 2 字节，可以为空占用 1 字节，ENUM 类型 1 字节，可以为空类型占用 1 字节。

如果我们不用国家这个条件来限制查询，则会得到以下结果：

```
mysql> EXPLAIN SELECT name FROM artist WHERE type=
'Person' AND gender='Male'\G
*********************** 1. row ***********************
...
           key: type_2
       key_len: 3
           ref: const,const
...
```

这里强调了当使用索引时，多余的列没有被用到查询中。如果没有其他查询用到了第 3 列，那么这就是一个可优化的点以减少索引行的宽度。

4.4.5　合并 WHERE 和 ORDER BY 语句

我们已经通过示例演示了如何使用索引优化数据行的限制条件，以及如何使用索引优化排序结果。MySQL 还可以利用多列索引执行上述两种操作。

```
mysql> ALTER TABLE album ADD INDEX (name);
```

```
mysql> EXPLAIN SELECT a.name, ar.name,
a.first_released
    -> FROM album a
    -> INNER JOIN artist ar USING (artist_id)
    -> WHERE a.name = 'Greatest Hits'
    -> ORDER BY a.first_released\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: a
         type: ref
possible_keys: artist_id,name
          key: name
      key_len: 257
          ref: const
         rows: 904
        Extra: Using where; Using filesort
*********************** 2. row ***********************
...
```

我们可以建立一个能够同时满足 WHERE 语句和 ORDER BY 语句的索引：

```
mysql> ALTER TABLE album
    -> ADD INDEX name_release (name,first_released);
mysql> EXPLAIN SELECT a.name, ar.name,
a.first_released
    -> FROM album a INNER JOIN artist ar USING
    (artist_id)
    -> WHERE a.name = 'Greatest Hits'
    -> ORDER BY a.first_released\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: a
         type: ref
possible_keys: artist_id,name,name_release
```

```
        key: name_release
    key_len: 257
        ref: const
       rows: 904
      Extra: Using where
*********************** 2. row ***********************
...
```

注意

优化器也可能为 WHERE 条件和 ORDER BY 语句使用一个索引；而这一点是不能从 key_len 的值看出来的。

技巧

创建一个能够用于对结果排序时的索引是有难度的；然而在某些频繁地(例如每秒 100 次)对相同数据进行排序的应用程序中，这样做将会带来很多益处。从使用 PROCESSLIST 命令查看 sorting results 的症状中，明显可以看出对 CPU 的影响，以及对一个经过优化的模式和 SQL 设计的参考方案的强烈需求。

4.4.6　MySQL 优化器的特性

MySQL 可以在 WHERE、ORDER BY 以及 GROUP BY 列中使用索引；然而，一般来说 MySQL 在一个表上只选择一个索引。从 MySQL 5.0 开始，在个别例外情况中优化器可能会使用一个以上的索引，但是在早期的版本中这样做会导致查询运行更加缓慢。最常见的索引合并的操作是两个索引取并集，当用户对两个有很高基数的索引执行 OR 操作时会出现这种这种索引合并操作。请看下面的示例：

```
mysql> SET @@session.optimizer_switch='
index_merge_intersection=on';
```

```
mysql> EXPLAIN SELECT artist_id, name
    -> FROM artist
    -> WHERE name = 'Queen'
    -> OR founded = 1942\G
********************** 1. row **********************
             id: 1
    select_type: SIMPLE
          table: artist
           type: index_merge
  possible_keys: name,founded,founded_2
            key: name,founded
        key_len: 257,2
            ref: NULL
           rows: 500
          Extra: Using union(name,founded); Using where
```

注意

在 MySQL 5.1 中首次引入了 optimizer_switch 系统变量，可以通过启用或禁用这个变量来控制这些附加选项。想了解更多信息可以参考以下链接：http://dev.mysql.com/doc/refman/5.1/en/switchable-optimizations.html。

第二种类型的索引合并是对两个有少量唯一值的索引取交集，如下所示：

```
mysql> SET @@session.optimizer_switch='index_merge_
intersection=on';
mysql> EXPLAIN SELECT artist_id, name
    -> FROM artist
    -> WHERE type = 'Band'
    -> AND founded = 1942\G
...
          Extra: Using intersect(founded,type); Using
          where
```

第三种类型的索引合并操作和对两个索引取并集比较类似，

但它需要先经过排序：

```
mysql> EXPLAIN SELECT artist_id, name
    -> FROM artist
    -> WHERE name = 'Queen'
    -> OR (founded BETWEEN 1942 AND 1950)\G
...
        Extra: Using sort_union(name,founded); Using
        where
```

可以通过以下链接了解更多关于索引合并的信息：http://dev.mysql.com/doc/refman/5.5/en/index-merge-optimization.html。

在创建这些示例的过程中，笔者发现一种以前在任何客户端的查询中未曾出现过的新情况。以下是三个索引合并的示例：

```
mysql> EXPLAIN SELECT artist_id, name
    -> FROM artist
    -> WHERE name = 'Queen'
    -> OR (type = 'Band' AND founded = '1942')\G
...
        Extra: Using union(name,intersect(founded,
        type)); Using where
```

技巧

应该经常评估多列索引是否比让优化器合并索列效率更高。

多个单列索引和多个多列索引到底哪个更有优势？这个问题只有结合特定应用程序的查询类型和查询容量才能给出答案。在各种不同的查询条件下，将一些高基数列上的那些单列索引进行索引合并能够带来很高的灵活性。数据库写操作的性能参考因素也同样会影响到获取数据的最优的数据访问路径。

4.4.7 查询提示

MySQL 中少数几个查询提示会影响性能。这些查询提示有些会影响到整个查询，而有些会影响到每个表索引的用法。

1. 总查询提示

所有总查询提示都会在 SELECT 关键字之后立刻产生。这些选项包括 SQL_CACHE、SQL_NO_CACHE、SQL_SMALL_RESULT、SQL_BIG_RESULT、SQL_BUFFER_RESULT、SQL_CALC_FOUND_ROWS 以及 HIGH_PRIORITY。上述提示不会影响到任何表上索引的使用，所以这里我们就不详细讨论了。

只有 STRAIGHT_JOIN 查询提示会对查询执行中索引的使用有影响。这个提示会告诉优化器按照查询中指定的表的顺序来执行查询执行计划。请看下面的示例：

```
mysql> EXPLAIN SELECT album.name, artist.name,
album.first_released
    -> FROM artist INNER JOIN album USING (artist_id)
    -> WHERE album.name = 'Greatest Hits'\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: album
         type: ref
possible_keys: artist_id,name,name_release
          key: name
      key_len: 257
          ref: const
         rows: 904
        Extra: Using where
********************** 2. row **********************
           id: 1
  select_type: SIMPLE
```

```
              table: artist
               type: eq_ref
      possible_keys: PRIMARY
                key: PRIMARY
...
mysql> EXPLAIN SELECT STRAIGHT_JOIN
album.name,artist.name,album.first_released

    -> FROM artist INNER JOIN album USING (artist_id)
    -> WHERE album.name = 'Greatest Hits'\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: index
 possible_keys: PRIMARY
          key: name
      key_len: 257
          ref: NULL
         rows: 586756
        Extra: Using index
********************** 2. row **********************
           id: 1
  select_type: SIMPLE
        table: album
         type: ref
 possible_keys: artist_id,name,name_release
          key: artist_id
...
```

在第一个查询中可以看出，优化器选择先在 album 表上执行连接操作。在第二个有 STRAIGHT_JOIN 提示的查询中，优化器会强制按照表所指定的顺序对 name 字段做连接。尽管这个查询在两个表上都使用了索引，但第二个查询需要处理的行数比第一个查询多很多，因此本例中它的效率也更低。

2. 索引提示

除了 STRAIGHT_JOIN 查询提示以外，所有索引提示都会被连接语句中的表所使用。可以为每张表定义一个索引的 USE、IGNORE 或者 FORCE 列表。也可以选择限制索引在查询中 JOIN、ORDER BY 或者 GROUP BY 部分的使用。在查询的每个表后面都可以添加下面的语法：

```
USE {INDEX|KEY}
    [{FOR {JOIN|ORDER BY|GROUP BY}} ([index_list])
| IGNORE {INDEX|KEY}
    [{FOR {JOIN|ORDER BY|GROUP BY}} (index_list) |
 FORCE {INDEX|KEY}
    [{FOR {JOIN|ORDER BY|GROUP BY}} (index_list)
```

```
mysql> EXPLAIN SELECT artist_id, name, country_id
-> FROM artist WHERE founded = 1980 AND type='Band'\G
********************** 1. row **********************
             id: 1
    select_type: SIMPLE
          table: artist
           type: ref
  possible_keys: founded,founded_2,type,type_2
            key: founded
        key_len: 2
            ref: const
           rows: 1216
          Extra: Using where
1 row in set (0.01 sec)
```

在这个查询中优化器有多个索引可供选择，但它最终选择了 founded 索引。

下面的示例会提示优化器使用某个特定的索引：

```
mysql> EXPLAIN SELECT artist_id, name, country_id
    -> FROM artist USE INDEX (type)
    -> WHERE founded = 1980 AND type='Band'\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: refpossible_
         keys: type
          key: type
      key_len: 1
          ref: const
         rows: 186720
        Extra: Using where
```

可以看到在这个查询中使用了指定的索引。

同样也可以要求优化器忽略某个索引：

```
mysql> EXPLAIN SELECT artist_id, name, country_id
    -> FROM artist IGNORE INDEX (founded)
    -> WHERE founded = 1980 AND type='Band'\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ref
possible_keys: founded_2,type,type_2
          key: founded_2
      key_len: 2
          ref: const
         rows: 1216
        Extra: Using where
```

可以提供多个索引名或者多个索引提示：

```
mysql> EXPLAIN SELECT artist_id, name, country_id
    -> FROM artist IGNORE INDEX (founded,founded_2)
```

```
    ->                  USE INDEX (type_2)
    -> WHERE founded = 1980 AND type='Band'\G
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ref
  possible_keys: type_2
          key: type_2
      key_len: 1
          ref: const
         rows: 177016
        Extra: Using where
```

想了解更多信息可以访问 http://dev.mysql.com/doc/refman/5.5/en/index-hints.html。

使用 MySQL 提示对更改全部的执行路径不会产生影响，因此你可以指定多个提示。使用 USE INDEX 提示会让 MySQL 从指定的索引中选择一个。FORCE INDEX 会对基于开销的优化器产生影响，让优化器更倾向于索引扫描而不是全表扫描。

提示

在 SQL 语句中添加提示是有很大风险的。尽管这可能会对查询有帮助，然而数据量随着时间的推移而变化会改变查询的有效性。添加或者改变表上的索引并不会影响到一个在特定索引中指定的硬编码 SQL 语句，所以查询提示应该是你最后考虑的方案。

4.4.8 复杂查询

本章的示例并不包含十个表的连接操作或者更复杂的查询。但本章以及第 5 章和第 8 章介绍的规则同样可以依次应用到单表中更复杂的查询上去。对于更复杂的查询我们使用同样的分析工

具，所要做的就是把 SQL 语句拆开，分别对每个组成部分进行测试和验证以了解和验证最有可能的优化方法，然后逐步增加查询的复杂度直到满足要求为止。

4.5　添加索引造成的影响

尽管本章给出了很多示例说明添加索引可以优化 SQL 语句的性能，但是添加索引同时也会带来不小的开销。

4.5.1　DML 影响

在表上添加索引会影响写操作的性能。这一点可以很明显地从本章使用的 artist 表中看出。查看目前此表的定义可以看到此表上有很多索引：

```
mysql> SHOW CREATE TABLE album\G
*********************** 1. row ***********************
       Table: album
Create Table: CREATE TABLE `album` (
  `album_id` int(10) unsigned NOT NULL,
  `artist_id` int(10) unsigned NOT NULL,
  `album_type_id` int(10) unsigned NOT NULL,
  `name` varchar(255) NOT NULL,
  `first_released` year(4) NOT NULL,
  `country_id` smallint(5) unsigned DEFAULT NULL,
  PRIMARY KEY (`album_id`),
  KEY `artist_id` (`artist_id`),
  KEY `country_id` (`country_id`),
  KEY `album_type_id` (`album_type_id`),
  KEY `m1` (`country_id`,`album_type_id`),
  KEY `m2` (`album_type_id`,`country_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

通过运行一个简单的基准测试，我们可以测试当前 album 表在包含较少索引的原始状态的数据插入速率：

```
DROP TABLE IF EXISTS t1;
CREATE TABLE t1 LIKE album;
INSERT INTO t1 SELECT * FROM album;
DROP TABLE t1;
CREATE TABLE t1 LIKE album;
-- NOTE: Due to indexes created during this chapter,
this may fail.
--       Adjust dropped indexes appropriately
ALTER TABLE t1 DROP INDEX first_released, DROP INDEX
album_type_id,
DROP INDEX name, DROP INDEX country_id, DROP INDEX m1,
DROP INDEX m2;
INSERT INTO t1 SELECT * FROM album;
DROP TABLE t1;
```

下面是查询返回结果的时间对比：

```
# Insert with indexes
Query OK, 553875 rows affected (24.77 sec)
# Insert without indexes
Query OK, 553875 rows affected (7.14 sec)
```

在包含更多索引的表中插入数据比在原始表中插入慢了 4 倍。当然这只是一次粗略的测试，还有很多其他因素可以导致查询缓慢。然而这已经足够说明在表上添加索引对写操作性能有直接影响。

1. 重复索引

在各种索引优化的技术中最简单的就是删除重复的索引。虽然找到重复的索引很容易，但是还有其他情况发生，例如一个索引与主码或者某些其他索引的子集相匹配。任何包含在其他索引

的最左边部分中的索引都属于重复索引，且不会被使用。请看下面的示例：

```
CREATE TABLE `album` (
...
  PRIMARY KEY (`album_id`),
  KEY `artist_id` (`artist_id`),
  KEY `country_id` (`country_id`),
  KEY `m1` (`album_type_id`,`country_id`),
  KEY `m2` (`country_id`,`album_type_id`)
  ...
```

可以看出 country_id 实际上是重复索引，因为已存在 m2 索引。

Maatkit mk-duplicate-key-checker 是一个用来找出重复索引的开源工具。也可以对模式表进行桌面人工验证。

2. 索引的使用

MySQL 机制的一个缺点是不能确定索引的使用。只有分析完所有 SQL 语句才能知道哪些索引没有被使用。找出使用到的和没有使用到的索引是很重要的。索引会影响写操作的性能，并且会占用磁盘空间从而影响到你的备份和恢复策略。有些低效的索引还会占用很大的内存资源。

2008 年 Google 发布了 SHOW INDEX_STATISTICS 命令，这个命令使你可以通过一种更精确的方式获取索引使用情况的信息。很多 MySQL 发行版都包含了这一特性，但官方 MySQL 产品还不支持。

想了解更多信息请访问 http://code.google.com/p/google-mysql-tools/wiki/UserTableMonitoring。

不管用什么工具来检查是否有没有用到的索引，你都需要分析索引中定义的列的有效性，同时也需要找到索引中无效的部分，

这是非常重要的。

4.5.2 DDL 影响

随着表大小的不断增长，对性能的影响也不断加大。例如，在主表上添加索引平均需要 20～30 秒。

```
mysql> ALTER TABLE album ADD INDEX m1
(album_type_id,country_id);
Query OK, 553875 rows affected (21.05 sec)
```

在以往版本中，ALTER 语句的开销是阻塞其他语句，就像创建一个新版本的表那样。在这期间可以 SELECT 数据，但根据标准的升级法则，任何 DML 操作都会导致所有语句被阻塞。当表的大小有 1G 或者 100G，这个阻塞时间可能会非常长。但比较近期的版本在包括 MySQL 产品方面和创新的解决方案方面都有了很多改进。

添加索引带来的影响并不总是一样，也会有些例外情况。InnoDB 提供了快速创建索引的特性，从 MySQL 5.1 版本开始就可以在 InnoDB 的插件中使用了，并且在 MySQL 5.5 或更高版本中已经成为默认设置了。更多信息可参考 http://dev.mysql.com/doc/innodb/1.1/en/innodb-create-index.html。

其他搜索引擎也可以以不同方式来实现执行锁定的快速索引的创建，Tokutek 就是其中的一个。更多信息请参考 http://tokutek.com/2011/03/hot-column-addition-and-deletion-part-i-performance。

对磁盘空间的影响也是一个重要的考虑因素，尤其是当你在 InnoDB 中使用默认的公共表空间配置的时候。MySQL 会为你的表创建一份副本。如果表的大小有 200GB，那么在执行 ALTER TABLE 时你需要至少 200GB 额外的磁盘空间。使用 InnoDB 时，

在执行期间这些额外的磁盘空间会被添加到公共表空间中。这部分磁盘空间在命令完成后不会被文件系统回收，而是当 InnoDB 需要额外磁盘空间时在内部被重复利用。尽管你可以调整策略让每个表用单独的表空间，但对于写操作密集的系统，这也是有影响的。

技巧

有一些技巧可以让阻塞操作减少到最低限度。你可以选择使用一个高可用性的容错度高的主表复制技术来支持在线变更表结构。比如近期 Shlomi Noach 介绍的 oak-online-alter-table 工具。更多信息可参见 http://code.openark.org/blog/mysql/online-alter-table-now-available-in-openark-kit。Facebook 也发布了自己的在线模式变更(OSC)工具，也能够以类似的方式在线运行 ALTER 操作。更多信息可参见 http://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932。

4.5.3　磁盘空间影响

使用第 2 章介绍的 INFORMATION_SCHEMA.TABLES 查询可以查看本章使用的 album 表的大小。

在添加索引之前：

```
*********************** 1. row ***********************
table_name: album
    engine: InnoDB
    format: Compact
table_rows: 539927
   avg row: 65
  total_mb: 47.08
   data_mb: 33.56
  index_mb: 13.52
```

添加索引之后：

```
*********************** 1. row ***********************

table_name: album
    engine: InnoDB
    format: Compact
table_rows: 548882
   avg_row: 64
  total_mb: 129.27
   data_mb: 33.56
  index_mb: 95.70
```

可以看出添加索引后表中索引空间的使用量增加了 7 倍。根据你选择的备份和恢复程序，添加索引也会对备份和恢复这两个过程需要的时间产生直接影响。添加索引还会有其他方面的影响。所以在添加索引之前，一定要理解并且考虑到这些影响。

使用 InnoDB 也会对磁盘使用空间产生直接影响，这些影响取决于所选择的主码以及如何使用这些主码。对于非主码索引，总是有一个主码索引附在非主码索引的记录后面。因此对于 InnoDB 表，一定要在主码中使用尽可能小的数据类型。

也有一种例外情况存在，更大的磁盘空间占用从长期角度也会对性能有好的影响。在表容量极大的情况下(例如几百 GB)，当所有查询都使用主码顺序时，一个有序的但并不是按次序排列的主码可能产生更加有序的磁盘活动。尽管填充因子导致了庞大的数据总量，但在一个高并发系统中按照主码顺序获取大量行的总时间会提升磁盘性能和总体查询性能。这是个很罕见的示例，它强调了在考虑总体查询性能的长期益处时，详细的监控数据以及合适的产品空间占用量的测试是必不可少的。

1. 页面的填充因子

选择用现实中存在的属性做主码而不是用现实中无意义的编会对默认页面的填充因子产生直接影响。对于一个现实中无意义的编号，当 InnoDB 会把数据插入一个使用率已经达到 15/16 的页面时将填充页面，因为主码是自增加的编号。当主码是现实中存在的属性时，InnoDB 插入新数码时会尝试把页面分割开来尽量降低数据重新组织的次数。一般来说，InnoDB 只会使用一个数据页的 50%的容量。这就导致了更大的磁盘占用量，当数据容量超过了分配给 InnoDB 缓冲池的内存量时，可以通过把更多数据包装成 16K 大的数据页来改善性能。第 3 章给出过一个有关磁盘空间的示例，就是由连续存放并且现实中存在的属性做主码的填充因子导致的。

译者注

在这里原文使用 natural primary key 和 surrogate primary key，natural primary key 就是指主码是现实中存在的属性，例如居民身份证号。而 surrogate primary key 则是指现实中无意义的编号，仅仅在数据库内部使用，并通常是插入新的记录时系统自动添加的一个编号。

2. 非主码索引

在 InnoDB 中由 B-树内部实现的非主码索引和 MyISAM 中 B-树非主码索引有显著的不同。InnoDB 在非主码索引中使用了主码的值，而不是一个指向主码的指针。在每个索引记录后面都附上了一个可应用的主码的副本。当数据库表有一个长度为 40 字节的主码，并且你还拥有 15 个索引时，引入一个更短的主码可以大幅度减少索引的空间占用量。这种使用主码的值的实现方式与

InnoDB 内部的主码散列算法结合使用能够改善性能。

4.6 MySQL 的限制和不足

与其他关系型数据库产品相比，MySQL 在使用和管理索引方面也有一些限制和不足。

4.6.1 基于开销的优化器

MySQL 用基于开销的优化器来调整可能的查询树以创建最优的 SQL 执行路径。MySQL 通过生成的统计信息来辅助优化器的能力是很有限的。MySQL 支持数量有限的索引提示用于帮助优化器选择一个合适的路径。

4.6.2 指定 QEP

MySQL 不支持为给定查询指定 QEP。在 MySQL 中，没有办法为一个数据一直随时间变化的查询定义一个 QEP，这也影响了对 QEP 的选择。也导致了需要为每个查询的执行计划确定 QEP。

4.6.3 索引的统计信息

MySQL 支持有限的索引统计信息，这些统计信息因存储引擎不同而不同。使用 MyISAM 存储引擎的话，ANALYZE TABLE 命令会为数据库表生成统计信息。目前还没有办法指定采样率。当一个表第一次被打开，然后被某个方法修改了一定比例的行的时候，InnoDB 存储引擎会在一个数据页中执行随机采样以生成给定表的统计信息。

目前正在开发的 MySQL 5.6 版中包含了存储 InnoDB 统计信

息的能力。

4.6.4　基于函数的索引

目前 MySQL 不支持基于函数的索引。而且，在已有的索引中使用函数会导致性能下降。MySQL 支持部分列的索引，实际上就是索引左边的子串。我们将在第 5 章详细讨论这个问题。

你也不能指定一个索引的相反序列，因为事实上所有数据都是升序排列的。当需要对数据排序时，如果有 DESC 关键字，MySQL 会反向遍历一个已有的索引。

4.6.5　一个表上的多个索引

正如本章介绍过的，默认情况下 MySQL 会对一个表只使用一个索引，但是有五种例外情况。在设计表、索引以及 SQL 语句之前认识到这个限制是很有好处的。未来版本的 MySQL 优化器的改进也会有助于弥补这些限制和不足。

4.7　本章小结

尽管创建索引的原则很简单，但是要选择在合适的列上创建索引，并且要判断在当前情况下添加索引带来的读数据性能的改进是否超过了对写数据性能的影响则不是那么容易。不管你的 SQL 语句有多么复杂，你都可以将一个 SQL 语句简化成为包含用到的表的子集的更小的工作单元，然后分别分析这些小型组件，最后再一点一点重新构造成为你的 SQL 语句。

本章主要讨论了如何在已经存在的模式中针对已有的 SQL 语句创建索引。我们没有讨论 SQL 语句的结构的重要性，优秀的

SQL 语句结构能够最大限度地利用索引。改变你使用所选的列编写 SQL 语句的方法，改变连接表的方式以及结果的顺序如何，都会对 SQL 语句的优化产生重要影响。

注意

在读扩展性要求很高的环境中，MySQL 的副本可以通过分散读数据负载给MySQL附属节点上的读数据性能带来很大的益处。于是在不同的服务器上使用相同的数据库模式并且同时使用不同的索引策略变得可行，这些不同的索引可以是针对某一类查询特别优化过的。

第 5 章

创建更好的 MySQL 索引

创建合适的索引是一种重要的优化技术。在第 4 章我们讨论了创建 MySQL 索引来改进各种查询的性能的基本方法。创建多列索引能够带来比单列索引更显著的性能提升。以下两种其他的

索引技术也能够进一步提高查询性能。

本章将介绍这两个其他的索引技术：

- 创建覆盖索引
- 创建局部列的索引

本章将使用第 4 章提供的表和数据的示例，因此建议重新创建示例中用到的表和数据。

警告

前面几章曾介绍过，MySQL 存储引擎的使用会影响到定义最优索引的方法。我们的示例表使用 InnoDB 事务性存储引擎。

5.1 更好的索引

通过使用索引，查询的执行时间可以从秒的数量级减少到毫秒数量级，这样的性能改进能够为你的应用程序的性能带来飞跃。合理的调整你的索引对优化来说是非常重要的，尤其是对于高吞吐量的应用程序。即使对执行时间的改进仅仅是数毫秒，但对于一个每秒执行 1000 次的查询来说这也是非常有意义的性能提升。例如，把一个原本需要 20 毫秒执行的每秒运行 1 000 次的查询的执行之间缩短 4 毫秒，这对于优化 SQL 语句来说是至关重要的。

我们将使用第 4 章介绍的方法创建多列索引，并在这一基础上创建更好的覆盖索引。

5.1.1 覆盖索引

如果我们想查询所有在 1969 年出道的艺人的名字，可以运行下面的查询：

```
mysql> SELECT artist_id, name, founded
    -> FROM artist
    -> WHERE founded=1969;
```

我们的示例数据库比较小，这个表只有大约 500 000 行数据，然而我们还是可以借这个示例说明改进索引的影响。

在没有索引的情况下，这个查询耗时 190 毫秒。从查询执行计划中我们可以看出执行了一次全表扫描(使用第 2 章介绍的方法判断)。建议添加一个索引加以改进。请看下面的示例：

```
mysql> ALTER TABLE artist ADD INDEX (founded);
mysql> EXPLAIN SELECT
...
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ref
possible_keys: founded
          key: founded
      key_len: 2
          ref: const
         rows: 1035
        Extra: Using where
```

在 WHERE 条件用到的 founded 列上添加索引之后，查询耗时减少到了 5.9 毫秒。这样简单的一个改动就可以让查询执行速度比原来提高了 97%。然而，还可以创建一个使这条查询执行起来更快的索引：

```
mysql> ALTER TABLE artist
    -> DROP INDEX founded,
    -> ADD INDEX founded_name (founded,name);
mysql> EXPLAIN SELECT
...
```

```
*********************** 1. row ***********************
           id: 1 select_
         type: SIMPLE
        table: artist
         type: ref
possible_keys: founded_name
          key: founded_name
      key_len: 2
          ref: const
         rows: 3696
        Extra: Using where; Using index
```

使用多列索引之后，查询执行只需要 1.2 毫秒了。这比刚才的查询快了 4 倍，查询执行时间比第一次优化时又减少了 80%，从总体来看我们节省了 99%的查询执行时间。

尽管我们是通过多列索引来获得这样的性能提升的，但改善查询的真正因素并不是因为额外增加的列限制了访问的行数。使用第 4 章介绍的分析技术，我们可以看到这个多列索引只占用了 2 字节。可能你会认为这个多列索引中额外的列是无效的，但要注意在 Extra 这一列中显示了 Using index。

当 QEP 在 Extra 列中显示 Using index 时，这并不意味着在访问底层表数据时使用到了索引，这表示只有这个索引才是满足查询所有要求的。这种索引可以为大型查询或者频繁执行的查询带来显著的性能提升，它被称为覆盖索引。

覆盖索引得名于它满足了查询中给定表用到的所有的列。想要创建一个覆盖索引，这个索引必须包含指定表上包括 WHERE 语句、ORDER BY 语句、GROUP BY 语句(如果有的话)以及 SELECT 语句中的所有列。

看了覆盖索引的介绍之后，你可能会好奇为什么 artist_id 这一列没有在索引中？对那些选择跳过第 3 章中介绍的索引的理论

知识的读者来说，你们可能需要了解 InnoDB 的非主码索引的一个重要的特性。在 InnoDB 中，主码的值会被附加在非主码索引的每个对应记录后面，因此没有必要在非主码索引中指定主码。这一重要特性意味着 InnoDB 引擎中所有非主码索引都隐含主码列了。并且对于那些从 MyISAM 存储引擎转换过来的表，通常会在它们 InnoDB 表索引中将主码添加为最后一个元素。

技巧

*有很多理由可以说服用户不要在 SQL 查询中使用 SELECT *。上面提到的就是其中一个原因，说明如果在 select 语句中只包含那些真正需要的列，就能够通过创建合适的索引来获得更好的 SQL 优化。*

然而我们的索引仅能使特殊定义查询性能大幅提升。如果我们想要通过添加某一类特定的艺术家来进一步限制查询，则结果如下：

```
mysql> EXPLAIN SELECT artist_id, name, founded
    -> FROM artist
    -> WHERE founded=1969
    -> AND type='Person'\G
*********************** 1. row ***********************
           id: 1 select_
         type: SIMPLE
        table: artist
         type: ref
possible_keys: founded_name
          key: founded_name
      key_len: 2
          ref: const
         rows: 3696
        Extra: Using where
```

从结果可以看出我们使用了同样的索引，但却没有享受到覆盖索引带来的益处了。执行这个查询耗时 5.4 毫秒。我们可以根据新加的列来调整索引如下所示：

```
mysql> ALTER TABLE artist
    -> DROP INDEX founded_name,
    -> ADD INDEX founded_type_name(founded,type,name);
mysql> EXPLAIN SELECT
...
********************** 1. row **********************
           id: 1 select_
         type: SIMPLE
        table: artist
         type: ref
possible_keys: founded_type_name
          key: founded_type_name
      key_len: 3
          ref: const,const
         rows: 1860
        Extra: Using where; Using index
```

以上使用了修改之后的索引。我们还可以通过 key_len 的值为 3 来确定 type 列现在也是优化器限制的一部分，并且可以看出 founded_type_name 是覆盖索引。执行查询现在耗时 1.3 毫秒。

警告

创建这些索引只是用来描述确认覆盖索引的过程，但在生产环境中它们可能并不是理想的索引。由于数据集大小有限，我们在这些例子中使用了一个长字符列。随着数据容量的增加，尤其是超过内存和磁盘最大容量的时候，为一个大型列创建索引可能会对系统整体性能有影响。覆盖索引对于那些使用了很多较小长度的主码和外键约束的大型规范化模式来说是理想的优化方式。

5.1.2　存储引擎的含义

第 3 章中着重强调了对于 InnoDB 的非主码索引来说，索引使用了主码的实际值，而不是指向底层数据行的指针。MyISAM 使用了一种 B-树索引的不同的实现方式，从下面的示例可以看出不同之处：

```
mysql> ALTER TABLE artist ENGINE=MyISAM;
mysql> EXPLAIN SELECT
...
********************** 1. row **************************
           id: 1
  select_type: SIMPLE
        table: artist
         type: ref
possible_keys: founded_type_name
          key: founded_type_name
      key len: 3
          ref: const,const
         rows: 1511
        Extra: Using where
```

这个查询并没有使用之前 QEP 所给出的索引的全部。MyISAM 的平均查询时间大约为 3.3 毫秒，比优化过的 InnoDB 语句慢，但是比没有使用覆盖索引的查询快。为了确定这个重要的底层架构的不同点，我们修改这个索引来满足 MyISAM 引擎的全部要求：

```
mysql> ALTER TABLE artist
    -> DROP INDEX founded_type_name,
    -> ADD INDEX founded_myisam
   (founded,type,name,artist_id);
mysql> EXPLAIN SELECT ...
********************** 1. row **************************
           id: 1 select_
```

```
           type: SIMPLE
          table: artist
           type: ref
  possible_keys: founded_myisam
            key: founded_myisam
        key_len: 3
            ref: const,const
           rows: 520
          Extra: Using where; Using index
```

注意

本书不会讨论架构方面的考虑因素以及选择不同存储引擎间的优缺点。Effective MySQL 网站：http://EffectiveMySQL.com 提供了关于最佳数据架构的很有帮助的信息，它强调了基准测试在优化 SQL 语句时是一项重要的任务。

在以后的测试中，我们还是会重置表使用的存储引擎：

```
mysql> ALTER TABLE artist DROP INDEX founded_myisam,
ENGINE=InnoDB;
```

5.1.3 局部索引

尽管索引可以用来限制需要查询的行数，但如果 MySQL 需要获取大量行中的更多列的数据，那么创建具有更小行宽度的小型索引则会更加高效。就像在创建覆盖索引的示例里面看到的那样，使用一个索引来做更多的工作可以显著地改善 SQL 查询的性能。当数据大小超过物理内存资源时，你在选择使用索引的时候就要考虑到物理资源，而不是仅仅考虑查询执行计划。

INFORMATION_SCHEMA 查询会检查表数据和索引空间的大小，请看下面的示例：

```
$ cat tablesize.sql
```

```
SET @schema = IFNULL(@schema,DATABASE());
SELECT @schema AS table_schema, CURDATE() AS today;
SELECT table_name,
        engine,row_format AS format, table_rows,
        avg_row_length AS avg_row,
        round((data_length+index_length)/1024/1024
        ,2) AS total_mb,
        round((data_length)/1024/1024,2) AS data_mb,
        round((index_length)/1024/1024,2) AS
        index_mb
FROM    INFORMATION_SCHEMA.TABLES
WHERE  table_schema=@schema
AND     table_name = @table
\G
```

首先从 album 表中删除已有的索引：

```
mysql> ALTER TABLE album DROP INDEX artist_id;
mysql> SHOW CREATE TABLE album\G
********************** 1. row **********************
       Table: album
Create Table: CREATE TABLE `album` (
  `album_id` int(10) unsigned NOT NULL,
  `artist_id` int(10) unsigned NOT NULL,
  `album_type_id` int(10) unsigned NOT NULL,
  `name` varchar(255) NOT NULL,
  `first_released` year(4) NOT NULL,
  `country_id` smallint(5) unsigned DEFAULT NULL,
  PRIMARY KEY (`album_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

album 表应该只有一个定义好的主码索引。我们可以通过查询定义好的 INFORMATION_SCHEMA.TABLES 查询来确定表和索引空间的大小：

```
mysql> SET @table='album';
mysql> SOURCE tablesize.sql
```

```
*********************** 1. row ***********************
 table_name: album
     engine: InnoDB
     format: Compact
 table_rows: 541244
    avg_row: 65
  total_mb: 33.56
   data_mb: 33.56
  index_mb: 0.00
```

可以看出这个表的索引不占用空间，因为 InnoDB 的主码索引是聚集索引，并且与底层数据相结合。如果这个表使用的是 MyISAM 存储引擎，我们可以看到以下信息重点强调了数据和索引占用的空间：

```
*********************** 1. row ***********************
 table name: album
     engine: MyISAM
     format: Dynamic
 table rows: 553875
    avg row: 44
  total mb: 29.09
   data mb: 23.66
  index_mb: 5.43
```

现在要在 album 表的 name 列上添加一个索引：

```
mysql> ALTER TABLE album ADD INDEX (name);
mysql> SOURCE tablesize.sql
*********************** 1. row ***********************
 table_name: album
     engine: InnoDB
     format: Compact
 table_rows: 537820
    avg_row: 65
  total_mb: 64.17
   data_mb: 33.56
```

```
index_mb: 30.61
```

在 name 列上新的索引使用了大约 30MB 磁盘空间。现在让我们为 name 列创建一个更紧凑的索引：

```
mysql> ALTER TABLE album
    -> DROP INDEX name,
    -> ADD INDEX (name(20));
mysql> SOURCE tablesize.sql
********************** 1. row **********************
 table_name: album
     engine: InnoDB
     format: Compact
 table_rows: 552306 a
     vg_row: 63
   total_mb: 57.14
    data_mb: 33.56
index_mb: 23.58
```

这个更加紧凑的索引只占用了 23MB，较之前节省了 20%的空间。尽管这看起来很不起眼，但我们的示例表只包含 500 000 行数据。如果这个表包含 500 000 000 行数据的话，那么节约的空间将会是 6GB。

这里主要考虑的是如何减小索引占用的空间。一个更小的索引意味着更少的磁盘 I/O 开销，而这又意味着能更快地访问到需要访问的行，尤其是当磁盘上的索引和数据列远大于可用的系统内存时。这样获得的性能改进将会超过一个非唯一的并且拥有低基数的索引带来的影响。

局部索引是否适用取决于数据是如何访问的。之前介绍覆盖索引时，你可以看到记录一个短小版本的 name 列不会对执行过的 SQL 语句有任何好处。最大的益处只有当你在被索引的列上添加限制条件时才能体现出来。请看下面的示例：

```
mysql> ALTER TABLE artist
    -> DROP INDEX name,
    -> ADD INDEX name_part (name(20));
mysql> EXPLAIN SELECT artist_id,name,founded
    -> FROM artist
    -> WHERE name LIKE 'Queen%'\G
********************** 1. row **********************
           id: 1 select_
         type: SIMPLE
        table: artist
         type: range
possible_keys: name_
     part_key: name_
 part_key_len: 22
          ref: NULL
         rows: 92
        Extra: Using where
```

在这个示例中，在索引中记录全名并没有带来额外的益处。而所提供的局部列索引满足了 WHERE 条件。如何选择合适的长度取决于数据的分布以及访问路径。目前没有准确的方法计算索引的恰当长度。因此对给定范围的列长度内的唯一值数目的比较是必不可少的。

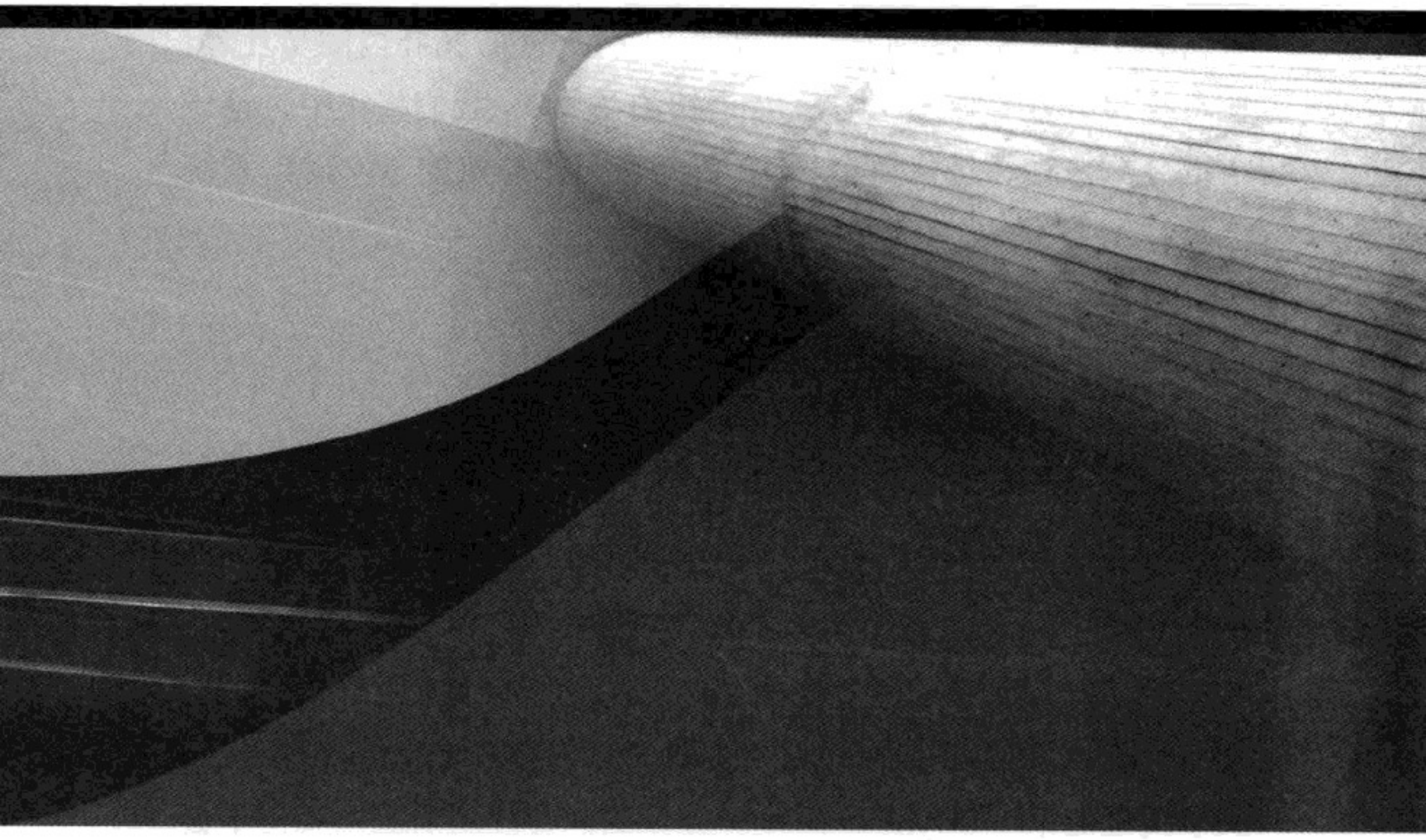
5.2 本章小结

在索引中正确的定义列(包括定义列的顺序和位置)能够改变索引的实际使用效果。好的索引能够为一个执行缓慢的查询带来巨大的性能提升。索引也可能使原来执行很快的查询的执行时间减少若干毫秒。在高并发系统中，将 1 000 000 条查询减少几毫秒将会显著改善性能，并且获得更大的容量和扩展性。为 SQL 查询创建最优索引可以认为是一项艺术。

写出好的 SQL 语句对查询优化和最大化当前以及以后的索引效果也是非常重要的。我们会在第 8 章进一步讨论如何改进 SQL 查询。

Effective MySQL 网站上可以下载到一个演示文档，里面提供很多详细的关于如何找到并创建更好的索引的示例。想了解更多信息可以访问：http://effectivemysql.com/presentation/improving-performance-with-better-indexes/。

可以从下列地址下载本章所有 SQL 语句：http://effectivemysql.com/book/optimizing-sql-statements。



第 6 章

MySQL 配置选项

MySQL 5.5 版本支持超过 300 种可配置的系统变量。其中很多系统变量如果得到正确设定，可以对整体系统性能改进起到很大作用。系统变量可以定义 MySQL 的配置方式、重要的文件和

数据的存放位置、如何管理兼容性以及 MySQL 拓扑内的交互等。在本章和本书中有关优化 SQL 语句的章节中，我们将集中讨论一小部分对执行和优化单个 SQL 语句有直接影响的 MySQL 系统变量。

本章将介绍下面的内容：

- 内存相关的系统变量
- 日志和工具系统变量
- 各种查询相关的系统变量

6.1 内存相关的系统变量

MySQL 内存系统变量能够影响全局内存的使用，也能够影响到多线程的单一 MySQL 进程中会话的内存使用情况。我们不会在本书中讨论有关系统调整对于优化 MySQL 中内存使用情况的重要性。一定要注意的是，MySQL 对于进程分配和内存存储引擎的分配是没有限制的。错误的系统调整会导致系统可用的内存资源耗尽。

这些MySQL系统变量可以在MySQL配置文件中定义并在启动时加载。很多 MySQL 变量是动态的——也就是说，它们的值可以在运行时通过 mysql 客户端的 SET 命令更改。可以参考 MySQL 参考手册来了解某个特定版本中所有动态变量的定义。

注意

在 MySQL 中，引用会话其实与一个给定的连接有关。默认情况下，一个连接也就是一个线程——即一个连接有且只有一个线程。

注意

对于熟悉 Oracle 的读者，全局内存缓冲区可以被理解成 MySQL 进程的系统全局区域(SGA)。这个缓冲区是在 mysqld 进程启动时被预先分配的。每个会话的内存缓冲区以及内存表可以被理解成当前运行的 mysqld 进程的进程全局区域(PGA)。

除非特别指出，否则下面所有变量都可以在 MySQL 5.0 或更高版本中使用。这些变量中的很多也存在于更早的MySQL版本中，如 MySQL 3.x 和 MySQL 4.x 中。

1. 全局内存缓冲区

MySQL 变量名	描　述
key_buffer_size	这个变量定义了 MyISAM 索引码的缓冲区的大小，通常这个变量也被称为码缓存。可以定义多个不同大小的已命名的码缓冲区
innodb_buffer_pool_size	这个变量定义了 InnoDB 缓冲池的大小，InnoDB 缓冲池是存储数据和索引页的主要场所
innodb_additional_mem_pool_size	这个变量定义了 InnoDB 的数据字典以及内部数据结构缓冲区的大小
query_cache_size	这个动态变量定义了查询缓存的大小。查询缓存可以用来缓存经常执行的 SELECT 语句

2. 全局/会话内存缓冲区

max_heap_table_size	这个动态变量定义了一个 MEMORY 存储引擎表的最大容量
tmp_table_size	这个动态变量定义了一个内部基于内存的临时表(在此表被修改并写入磁盘之前)的最大容量。这个变量和 max_heap_table_size 密切相关

3. 会话缓冲区

join_buffer_size	这个动态变量定义了当索引无法满足连接条件时在两个表之间做全表连接操作能够使用的内存缓冲区的大小
sort_buffer_size	这个动态变量定义了当索引无法满足需要的顺序信息时对结果排序能够使用的内存缓冲区的大小
read_buffer_size	这个动态变量定义了连续数据扫描时能够使用的内存缓冲区大小
read_rnd_buffer_size	这个动态变量定义了有序(也就是不连续的)数据扫描时能够使用的内存缓冲区大小

警告

一个常见的问题是管理员没有认识到这 4 个缓冲区是以每个线程为基础定义的。在 MySQL 5.1 以及更高版本中，sort_buffer_size 缓冲区的默认值可以在 128K/256K～2M 之间变化。如果一个缓冲区的大小定义为 10M 或者 100M 的话，就会对查询和系统性能产生相反的影响。在没有证据能证明性能提升的情况下，最好的方法是恢复这 4 个变量的默认值来保证总体内存利用率最大。

可以在 MySQL 参考手册中找到关于这些变量和其他变量的更多信息：http://dev.mysql.com/doc/refman/5.5/en/memory-use.html。

6.1.1 key_buffer_size

key_buffer_size 是只存储 MyISAM 索引信息的全局内存缓冲区。在对应的.MYI 文件中的索引数据从磁盘上被读取出来然后存入这个缓冲区。想要调整变量 key_buffer_size 的大小，只需要简单地统计所有 MyISAM 表中总索引的大小，然后随着数据随时间

增长不断调整即可。

当这个索引码缓冲区中没有足够的空间来存储新的索引数据时，将会用最近最少使用(LRU)的方法覆盖掉旧的页面。

注意

即使运行一个全部采用 InnoDB 的模式，仍然需要定义一个索引码缓冲区，因为 MySQL 元信息与 MyISAM 定义的相同。可以给这个小的缓冲区定义一个 8M 的值。

当观察单个 MySQL 连接时，一个没调整好的 MyISAM 索引码缓冲区能够呈现出不同状态。在 SHOW [FULL] PROCESSLIST 的 State 列中，值 Repairing the keycache 是一个明显的指标，它指出当前索引码缓冲区大小不足以执行当前运行的 SQL 语句。这将导致额外的磁盘 I/O 开销。

想要了解更高级调整信息可以使用以下变量：key_cache_age_threshold、key_cache_block_size、key_cache_division_limit 和 preload_buffer_size。

6.1.2　命名码缓冲区

除了默认的 MyISAM 码缓冲区，MySQL 支持其他的命名码缓冲区。这使得管理员能够为 MyISAM 索引定义专用的内存池，该 MyISAM 可以为命名码缓冲区分配和指定给定表。下面是创建一个已命名 64M 的缓存的示例：

```
mysql> SET GLOBAL hot.key_buffer_size=1024*1024*64;
```

也可以在 MySQL 配置文件 my.cnf 中定义这个缓存。

警告

在 MySQL 配置文件中，可以用字节或者 KB、MB 和 GB 这样的通用格式来定义变量的大小。当使用 SET 命令动态地定义同一个变量时，只有以字节为单位的数字值是合法的。可以用下面的语法把字节转化成 1K、1M 或者 64M：1024×1024×64，然而这并不是必需的。

可以使用 CACHE INDEX 语句把表索引分配到指定的命名缓存上：

```
mysql> CACHE INDEX table1, table2 IN hot;
```

有关哪些表被缓存到命名缓存中这样的信息是不会被持久化的。在 MySQL 重启的过程中，这些信息都会丢失。如果想在系统重启的时候完成这些功能，可以在服务器启动的时候使用 init_file 变量运行特定的 SQL 语句。

如果你能够预先填充包含给定的表中所有索引的码缓冲区，当码缓冲区能够存储所有的索引或者没有叶子节点页面的索引时将会获得额外的性能提升。请看下面的示例：

```
LOAD INDEX INTO CACHE table1, table2;
```

可以使用这个语法把索引加载到任何索引码缓存中，包括默认索引码缓存。

6.1.3 innodb_buffer_pool_size

innodb_buffer_pool_size 是用来存储所有 InnoDB 数据和索引的全局内存缓冲区。对完全使用 InnoDB 引擎的模式来说，这是最重要的缓冲区，一定要正确分配。不正确的分配这个缓冲区的空间可能导致额外的磁盘 I/O 开销并降低查询性能。

对小型系统来说，可以用所有 InnoDB 表的容量加上一个小的额外开销来计算出这个缓冲区的合适大小。如果你只有一个小的有限空间的 RAM 存储器时，这一点尤为重要。举例来说，当你在一个支持 1.7GB 的 RAM 存储器的 Amazon Web Service 小实例上分配 1GB 的 InnoDB 缓冲区，而总数据库大小仅有 200MB 时，你就等于分配了不必要的内存空间。

对于这个变量，没有最好的估计大小的方式，因为还有很多其他因素都会影响到 MySQL 的内存总使用量。常见的方法是把这个变量设定为 RAM 空间的 80%，但在很多情况下这样设定也是不合理的。

可以使用 SHOW GLOBAL STATUS 或者 SHOW ENGINE INNODB STATUS 命令来监控 InnoDB 缓冲池的使用情况。请看下面的示例：

```
mysql> SHOW GLOBAL STATUS LIKE 'innodb_buffer%';
+---------------------------------------+----------+
| Variable_name                         | Value    |
+---------------------------------------+----------+
| Innodb_buffer_pool_pages_data         | 196892   |
| Innodb_buffer_pool_pages_dirty        | 37799    |
| Innodb_buffer_pool_pages_flushed      | 456354   |
| Innodb_buffer_pool_pages_free         | 445552   |
| Innodb_buffer_pool_pages_misc         | 12908    |
| Innodb_buffer_pool_pages_total        | 655352   |
| Innodb_buffer_pool_read_ahead         | 122      |
| Innodb_buffer_pool_read_ahead_evicted | 0        |
| Innodb_buffer_pool_read_requests      | 442125161|
| Innodb_buffer_pool_reads              | 18283    |
| Innodb_buffer_pool_wait_free          | 0        |
| Innodb_buffer_pool_write_requests     | 175015163|
+---------------------------------------+----------+
mysql> SHOW ENGINE INNODB STATUS;
...
```

```
----------------------
BUFFER POOL AND MEMORY
----------------------
Total memory allocated 43168694272; in additional pool
allocated 0
Dictionary memory allocated 960147
Buffer pool size 2574464
Free buffers 1893581
Database pages 618715
Old database pages 228288
Modified db pages 52209
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 16276, not young 0
0.75 youngs/s, 0.00 non-youngs/s
Pages read 34431, created 584284, written 2473937
0.00 reads/s, 3.12 creates/s, 219.40 writes/s
Buffer pool hit rate 1000 / 1000, young-making rate
0 / 1000 not 0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s
LRU len: 618715, unzip_LRU len: 0
I/O sum[0]:cur[1104], unzip sum[0]:cur[0]
...
```

总共有超过 50 个 InnoDB 相关的系统变量。所有这些变量都以 innodb_前缀开头。那些会影响到整体系统性能并且应该在定义 innodb_key_buffer_size 变量时应该考虑到的 InnoDB 变量包括：innodb_buffer_pool_instances(从 5.5 开始)、innodb_max_dirty_pages_pct、innodb_thread_concurrency、innodb_spin_wait_delay、innodb_purge_threads(从 5.5 开始)、innodb_checksums、innodb_file_per_table 以及 innodb_log_file_size。

6.1.4 innodb_additional_mem_pool_size

innodb_additional_mem_pool_size 变量为 InnoDB 特定数据字

典信息定义了内存池。对于这个变量，没有什么好的方法来确定它的最优值，一般将其设定为 10M。在 MySQL 5.5 中默认大小为 8M。在早期版本中，默认值是 1M。如果这个缓冲区空间不足，MySQL 会在错误日志中报告一个警告信息。

6.1.5　query_cache_size

query_cache_size 变量是一个用来存储经常缓存过的查询全局内存缓冲区。当启用 query_cache_type 变量之后，SELECT 查询会被自动缓存起来。还可以通过 SQL_CACHE 和 SQL_NO_CACHE 来进一步确定哪些语句需要被缓存。启用查询缓存能够在读取操作频繁的系统中显著地改进性能，但它也可能降低读/写系统的整体性能。

使用 query_cache_type 变量可以总体启用和禁用查询缓存。启用时，query_cache_size 的值可能为 0，这表示没有查询需要被缓存；而 MySQL 实例可以通过动态地改变 query_cache_size 的值在某个时间仍然可以支持缓存。在不需要的时候从物理上禁用查询缓存可以带来一些小的性能开销。

可以使用下面的语句动态启用查询缓存：

```
mysql> SET GLOBAL query_cache_type = 1;
mysql> SET GLOBAL query_cache_size = 1024 * 1024 * 16;
```

要动态禁用查询缓存可以使用下面的语句：

```
mysql> SET GLOBAL query_cache_type = 0;
mysql> SET GLOBAL query_cache_size = 0;
```

可以通过SHOW GLOBAL STATUS命令或者INFORMATION_SCHEMA.GLOBAL_STATUS 表来监控查询缓存的有效性。请看下面的示例：

```
mysql> SHOW GLOBAL STATUS LIKE 'Qcache%';
+-------------------------------+-------------+
| Variable_name                 | Value       |
+-------------------------------+-------------+
| Qcache_queries_in_cache       | 8115        |
| Qcache_inserts                | 596110709   |
| Qcache_hits                   | 1015208171  |
| Qcache_lowmem_prunes          | 526595375   |
| Qcache_not_cached             | 1499575     |
| Qcache_free_memory            | 8400752     |
| Qcache_free_blocks            | 2465        |
| Qcache_total_blocks           | 18707       |
+-------------------------------+-------------+
```

可以用下面的公式来确定查询缓存的有效性：

```
((Qcache_hits/(Qcache_hits + Com_select + 1))*100)
```

在确定读写量时，一定要把 Qcache_queries_in_cache 状态变量考虑进去。这个变量将和 Com_select 状态变量一起决定你的 MySQL 实例的总体读操作的满意率。

查询缓存还可以通过很多不同的系统变量来进一步优化，这些系统变量会影响到个别查询的分配。这些变量包括：query_cache_limit、query_case_min_res_unit、query_cache_alloc_block_size、query_cache_wlock_invalidate 和 query_prealloc_size。

6.1.6 max_heap_table_size

这个变量定义了 MySQL ME MORY 存储引擎表的最大容量。当某个表容量超过最大值时，应用程序会收到下面的信息：

```
mysql> SET SESSION max_heap_table_size=1024*1024;
mysql> CREATE TABLE t1(
    -> i INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> c VARCHAR(1000)) ENGINE=MEMORY;
```

```
mysql> INSERT INTO t1(i) VALUES
    -> (NULL),(NULL),(NULL),(NULL),(NULL),
    -> (NULL),(NULL),(NULL),(NULL),(NULL);
mysql> INSERT INTO t1(i) SELECT NULL FROM t1 AS a, t1
AS b, t1 AS c;
ERROR 1114 (HY000): The table 't1' is full
```

如前所述，这个变量有一个全局默认值，而且在上例的每个线程上也可以指定这个变量的值。MySQL 并没有为所有 MEMORY 表的总容量做任何限制。这个变量仅用于单个表。

MEMORY 存储引擎表的总大小可以通过 SHOW TABLE STATUS 命令和 INFORMATION_SCHEMA.TABLES 表来确定。

另外，MEMORY 存储引擎表对任意字符列都使用固定的宽度。在前面的示例中，当在 MEMORY 表中定义时，那么 VARCHAR(1000)列实际上使用的是 CHAR(1000)。因为有了这个限制，一个 MEMORY 表无法包含任何 TEXT(即 CLOB)或者 BLOB 字段。这也是 tmp_table_size 变量中非常重要的一点。

6.1.7 tmp_table_size

max_heap_table_size 变量和 tmp_table_size 变量中的最小值定义了内部临时表的最大容量，内部临时表用于存储在内存中的查询执行过程。如果在 EXPLAIN SELECT 的结果中的 Extra 列中出现了 Using temporary，那么可以判断在查询执行过程中用到了内部临时表。一个 SQL 查询可以使用多个临时表。

MySQL 使用 MEMORY 存储引擎来支持这些内部临时表。当内部临时表的容量超过 max_heap_table_size 变量和 tmp_table_size 变量中的最小值时，MySQL 会在临时位置创建一个基于 MyISAM 磁盘的表。由于 MEMORY 存储引擎的限制，为临时表定义的任何 TEXT 和 BLOB 列都会导致一个基于磁盘的 MyISAM 临时表。

在临时表中定义大变量字符数据类型也会导致创建基于磁盘的临时表，因为这些字段是固定宽度的。

目前没有一种简便的方式来确定内部临时表的总容量。可以通过 MySQL 状态变量 created_tmp_tables 和 created_tmp_disk_tables 来分别确认创建了临时表或者基于磁盘的临时表，请看下面的示例：

```
mysql> SHOW SESSION STATUS LIKE 'create%tables';
+-------------------------------+-------------+
| Variable_name                 | Value       |
+-------------------------------+-------------+
| Created_tmp_disk_tables       | 0           |
| Created_tmp_tables            | 6           |
+-------------------------------+-------------+
mysql> SELECT ...
mysql> SHOW SESSION STATUS LIKE 'create%tables';
+-------------------------------+-------------+
| Variable_name                 | Value       |
+-------------------------------+-------------+
| Created_tmp_disk_tables       | 1           |
| Created_tmp_tables            | 8           |
+-------------------------------+-------------+
```

注意

衡量 MySQL 状态变量的值也会对返回的结果有影响。例如，执行 SHOW STATUS 命令生成一个临时表。在 MySQL 版本、全局范围以及会话范围内的变量的值也会有不同。所以一定不要忘记经常检查测量变量带来的影响。

了解 MySQL 中内部临时表的操作原理对系统性能是至关重要的。例如，超过容量限制并写入磁盘的高并发内部临时表会出现如下错误：

```
ERROR 126 (HY000): Incorrect key file for table '//
tmp/#sql_5b7_1.MYI'; try to repair it
```

这是由于 tmpdir 变量定义的磁盘空间已经满了。这个错误中提到的临时磁盘表没有办法修复了。在 MySQL 5.5 中，可以使用 PERFORMANCE_SCHEMA 来帮助统计基于磁盘的临时表的总大小。

tmp_table_size 变量有一个默认全局值，它也可基于每线程来指定，这对那些需要生产很大的临时表的查询很有好处。

可以在 MySQL 参考手册中找到更多关于如何使用内部临时表的信息：http://dev.mysql.com/doc/refman/5.5/en/internal-temporary-tables.html。

6.1.8 join_buffer_size

join_buffer_size 定义了每个线程的内存缓冲区，当查询必须连接两个表的数据集并且不能使用索引时，这个缓冲区会被用到。这个缓冲区是专门为每个线程的无索引连接操作准备的。可以通过查询计划中 Extra 列的值为 Using join buffer 来证实使用了这个缓冲区。建议将这个缓冲区设置为默认大小。增加这个缓冲区的大小也不会加快全连接操作的速度。为这个缓冲区设定太大的容量可能会对高连接的环境中的内存使用造成影响。

6.1.9 sort_buffer_size

这个变量定义了每个线程用于对结果集排序的每线程缓冲区。可以通过查询计划中 Extra 列的值为 Using file-sort 来确定使用了这个缓冲区。不推荐你增加这个缓冲区的大小，因为这个缓冲区是完全分配给每个请求的，而且当默认值太大时可能会降低查询的执行速度。

6.1.10 read_buffer_size

当 SQL 查询执行连续的表数据扫描时会用到这个缓冲区。只有在大量连续表数据扫描时才推荐增加这个缓冲区的大小。

6.1.11 read_rnd_buffer_size

这个缓冲区用来存储那些作为排序操作的结果被读取的数据。这个缓冲区和 read_buffer_size 的不同之处在于，它读取的连续数据是和数据在磁盘上的存储方式相关的。只有在执行大型 ORDER BY 语句时才推荐增加这个缓冲区的大小。

6.2 有关基础工具的变量

slow_query_log(从 5.1 版本开始)	这个动态布尔变量决定是否在日志中记录执行缓慢的查询
slow_query_log_file (从 5.1 版本开始) log_slow_queries (对 5.0 和更早版本，5.1 开始废弃)	当输出结果记录到文件中时，这个动态变量用来指定执行缓慢的查询日志的文件名
long_query_time	这个动态变量用来指定一个正常查询最大的执行时间，超过这个时间，查询就被认为是执行缓慢的查询
general_log (从 5.1 版本开始)	这个动态布尔变量决定是否把所有的数据库查询都记录在日志中
general_log_file (从 5.1 版本开始) log (对 5.0 和更早版本，5.1 开始废弃)	当输出结果记录到日志中时，这个动态变量指定全面日志的文件名
log_output(从 5.1 版本开始)	这个动态变量定义了执行缓慢的查询日志和全面日志的目标类型
profiling	这个动态变量定义了分析每个线程的语句

6.2.1　slow_query_log

这个布尔变量可以启用执行缓慢的查询的日志功能，日志将会报告所有执行时间超过 long_query_time 变量值的查询。这里强烈推荐一直定义这个选项，并且对那些运行时间长的可以优化的查询的结果经常分析。这是一个可以被动态改变的全局变量。

技巧

缓慢查询日志和通用查询日志是分别由 slow_query_log 和 general_log 系统变量动态控制的。在 MySQL 5.0 及之前的版本中，这些日志不是动态的，想要启用或者禁用这些功能都需要重启 MySQL 实例。

6.2.2　slow_query_log_file

这个变量定义了当慢查询日志功能开启时保存所有被记录的查询文件的文件名。这是个全局变量，可以动态改变它的值。

6.2.3　general_log

这个变量用来启用记录每条查询执行情况的全面查询日志。这个变量只能在每个服务器实例值上启用或者禁用。这是个全局变量，可以动态改变它的值。

6.2.4　general_log_file

这个变量定义了记录了当全面日志启用时所有 SQL 查询的文件名。这是个全局变量，可以动态改变它的值。

6.2.5 long_query_time

这个变量指定了一个查询执行时间的限制，当慢查询日志功能启用时，执行时间超过这个限制的查询都会被记录在慢查询日志中。从 MySQL 5.1.21 开始，这个变量接受以毫秒为单位的值。当指定日志输出为 TABLE 类型时，这个变量只支持以秒为单位的值。这是一个全局变量，可被动态定义。

6.2.6 log_output

这个变量定义了慢查询日志和全面查询日志的输出位置，有效的选项有 FILE、TABLE 和 NONE。当定义输出位置为 FILE 时，日志的输出文件分别由 slow_query_log_file 和 general_log_file 系统变量来定义。如果这个变量为 TABLE，日志输出将会分别记录在 mysql.slow_log 和 mysql.general_log 表中。这两个表是在内部以 CSV 存储引擎定义的，所以不支持任何索引。当需要频繁查询这些表时，建议创建表的副本并相应地优化你的查询。这是一个全局变量，可被动态定义。

6.2.7 profiling

MySQL 通过分析工具暴露一些语句执行的内部细节信息。这个布尔会话变量可以激活细节语句分析并且可以提供 SQL 的精确到毫秒级的准确执行时间。这个工具还可以计算出关键的内部组件的时间并逐步跟进查询计划进程中。

可以通过 profiling_history_size 变量控制在每个线程分析缓冲区中的 SQL 语句的数量。

6.3　其他优化变量

optimizer_switch (从 5.1 版本开始)	这个变量决定了 MySQL 优化器中哪个高级索引合并功能会被启用
default_storage_engine	当没有指定存储引擎时，这个变量定义了表使用的存储引擎
max_allowed_packet	这个变量定义了结果集的最大容量
sql_mode	这个变量定义了支持的各种服务器 SQL 模式
innodb_strict_mode(从 5.1 版本开始)	这个变量定义了一个专门为 InnoDB 插件提供的服务器 SQL 模式级别

6.3.1　optimizer_switch

这个选项定义了一系列MySQL查询优化器特性的高级开关，可以用来关闭(默认是激活状态)三种不同的索引合并条件以及引擎下推条件。

具体内容可以从以下系统变量的介绍中了解到：max_seeks_for_key、optimizer_prune_level、optimizer_search_depth 以及 engine_condition_pushdown。

6.3.2　default_storage_engine

当未指定 ENGINE 值时，这个变量用来为 CREATE TABLE 命令指定存储引擎。在 MySQL 5.5 中，默认存储引擎由 MyISAM 变成了 InnoDB。任何历史遗留的应用程序都可能经历过这样的问题，原本以为默认使用的 MyISAM 引擎现在变成了 InnoDB。

6.3.3 max_allowed_packet

可以用 max_allowed_packet 变量来定义 SQL 查询结果集的最大值。增大这个值会允许查询返回更大的结果集。通过限制这个变量的大小，你会得到一个潜在的指标来表明有一个查询返回了很大的结果集，而这可能是无意义的。

6.3.4 sql_mode

MySQL 支持一系列不同的服务器 SQL 模式。这些模式之间区别很大，并且提供了更加符合美国国家标准学会(ANSI)的 SQL 语法，也提供了默认预期的数据验证，并支持不同的 null 和零日期数据的管理方式。这些设置非常重要，因为它们能够改变 SQL 语句的运行方式所带来的影响。

6.3.5 innodb_strict_mode

这个变量为 InnoDB 插件提供了 sql_mode 功能，这个功能在 MySQL 5.1 中是可选的，但现在在 MySQL 5.5 中已经是默认的 InnoDB 实现了。这些选项扩展了原生的 sql_mode，包含对表对象的管理以及行检查功能。

6.4 其他变量

那些并没有在本章讨论过但你也应该考虑做深入分析的 MySQL 系统变量还包括下面这些：

concurrent_insert	foreign_key_checks	log_bin
max_join_size	max_seeks_for_key	min_examined_row_count
open_files_limit	optimizer_prune_level	optimizer_search_depth
sql_buffer_result	sql_select_limit	sync_binlog
thread_cache_size	thread_stack	tmpdir
tx_isolation	unique_checks	

6.5　本章小结

本章详细介绍了那些和本书的主要目的相关的最常用的 MySQL 系统变量，这些变量可以帮助进行全面的 SQL 优化。除此之外还有很多其他变量需要关注，尤其是那些能够影响到流行的 InnoDB 存储引擎的 DML 性能优化的。请查阅 MySQL 参考手册上给出的详细列表。

第 7 章

SQL 的生命周期

优化 SQL 语句的生命周期涉及 6 个独立的部分，这包括如何截取 SQL 语句、识别有问题的 SQL 语句以及在开始分析前如何确认 SQL 语句。

本章我们将讨论整个 SQL 优化生命周期中的 6 个阶段：

- 截取 SQL 语句
- 识别并分类有问题的 SQL 语句
- 确认 SQL 语句的当前操作
- 分析 SQL 语句和辅助信息
- 优化 SQL 语句
- 验证 SQL 优化的结果

7.1 截取 SQL 语句

在审查和优化 SQL 语句之前，需要收集那些需要审查的 SQL 语句。可以通过一个采样过程来收集所有执行过语句的一部分；或者也可以通过详细收集进程收集一段时间内所有执行过的语句。下面列出的是 MySQL 中各种流行的 SQL 语句截取技术：

- 全面查询日志
- 慢查询日志
- 二进制日志
- 进程列表
- 引擎状态
- MySQL 连接器
- 应用程序代码
- INFORMATION_SCHEMA
- PERFORMANCE_SCHEMA
- SQL 语句统计信息插件
- MySQL 代理

- TCP/IP

注意这里并没有列出所有可能的 SQL 截取技术。

7.1.1　全面查询日志

MySQL 全面查询日志允许你截取所有在这个数据库实例上运行的 SQL 语句。从 5.1 版本开始，可以配置这个日志输出到文件或者数据库表中。全面查询日志功能只有启用或者禁用两种状态，没有其他控制粒度，它可以通过下面的 MySQL 配置来启用：

```
[mysqld]
general_log=1
general_log_file=/path/to/file
log_output=FILE
```

注意

这些配置选项在 MySQL 5.1 及以后的版本中是合法的。对于 5.0 或更早的版本，请参考 MySQL 参考手册 http://dev.mysql.com/doc/refman/5.0/en/query-log.html。

也可以通过 SQL 语句动态地启用或者禁用全面日志，并且让日志输出到文件、数据库表或者两者同时写入，例如：

```
mysql> SET GLOBAL general_log=1;
mysql> SET GLOBAL log_output=TABLE;
...
mysql> SELECT * FROM mysql.general_log\G
...
********************** 61. row **********************
  event_time: 2011-04-20 01:32:53
   user_host: em_rw[em_rw] @ localhost []
   thread_id: 1355924
   server_id: 1
command_type: Query
```

```
argument: SELECT t.*, tt.* FROM wp_terms AS t INNER
JOIN wp_term_taxonomy AS tt ON t.term_id = tt.term_id
WHERE tt.taxonomy IN ('link_category') AND tt.count
> 0 ORDER BY t.name ASC
********************** 62. row **********************
  event_time: 2011-04-20 01:32:53
   user_host: em_rw[em_rw] @ localhost []
   thread_id: 1355924
   server_id: 1
command_type: Query
argument: SELECT * FROM wp_links INNER JOIN
wp_term_relationships AS tr ON (wp_links.link_id =
tr.object_id) INNER JOIN wp_term_taxonomy as tt ON
tt.term_taxonomy_id = tr.term_taxonomy_id WHERE 1=1 AND
link_visible = 'Y' AND ( tt.term_id = 2 )
AND taxonomy = 'link_category' ORDER BY link_name ASC
...
```

注意

全面查询日志很适合提供所有连续的 SQL 查询语句顺序，但它并不提供执行时间。在开发环境或者小容量数据的测试环境中启用全面日志功能是不错的审查 SQL 语句的方式，但永远不要在生产环境中启用这个功能。

7.1.2 慢查询日志

MySQL 的慢查询日志用于提供那些在指定数据库实例上执行时间超过一定限制的那些查询。慢查询日志可以通过下面的配置来启用：

```
[mysqld]
slow_query_log=1
slow_query_log_file=/path/to/file
long_query_time=0.2
log_output=FILE
```

注意

这些配置选项在 MySQL 5.1 及以后的版本是合法的。对于 5.0 或更早的版本请参考 MySQL 参考手册 http://dev.mysql.com/doc/refman/5.0/en/slow-query-log.html。

从 MySQL 5.1 版本开始，可以以毫秒为单位来指定 long_query_time 变量，但只在文件形式的输出时才支持该选项。可以通过 SLEEP()函数来方便地测试慢查询日志的有效性，这个函数也支持毫秒单位：

```
mysql> SELECT SLEEP(0.1);
mysql> SELECT SLEEP(0.2);
mysql> SELECT SLEEP(0.3);

$ tail -f /var/log/mysql/slow.log
# Time: 110420 22:53:58
# User@Host: dba[dba] @ localhost []
# Query_time: 0.201908 Lock_time: 0.000000 Rows_sent:
1 Rows_examined: 0
SET timestamp=1303354438;
SELECT SLEEP(0.2);
# Time: 110420 22:54:01
# User@Host: dba[dba] @ localhost []
# Query_time: 0.302270 Lock_time: 0.000000 Rows_sent:
1 Rows_examined: 0
SET timestamp=1303354441;
SELECT SLEEP(0.3);
```

从结果中可以看到，第一个 SELECT SLEEP(0.1)语句没有被记录在慢查询日志中，因为它的执行时间小于 long_query_time 的阈值。

7.1.3 二进制日志

MySQL 的二进制日志涵盖了所有非 SELECT 语句，其中包括 DML 和 DDL 语句。这个日志功能可以用来提供表级别粒度的语句容量的历史分析。UPDATE 和 DELETE 语句的截取内容能够揭示出潜在的可优化索引。二进制日志功能可以通过下面的 MySQL 配置命令来启用：

```
[mysqld]
log-bin=/path/to/file
```

MySQL 提供了 mysqlbinlog 命令来创建 MySQL 二进制日志中信息的纯文本版本。生成的输出结果包含很多额外的文本信息，如果只关注 DML 语句的话，这些信息可以忽略。请看下面的示例：

```
$ mysqlbinlog path/to/file.000001
...
# at 178
#110420 22:47:17 server id 1 end_log_pos 336
Query thread_id=5 exec_time=0
      error_code=0
use blog/*!*/;
SET TIMESTAMP=1303354037/*!*/;
UPDATE `wp_options` SET `option_value` = '124348'
WHERE `option_name` =
'akismet_spam_count'
/*!*/;
# at 336
#110420 22:47:17 server id 1 end_log_pos 408 Query
thread_id=5 exec_time=0
       error_code=0
SET TIMESTAMP=1303354037/*!*/;
COMMIT
/*!*/;
```

```
# at 408
#110420 22:48:02 server id 1 end_log_pos 480 Query
thread_id=12 exec_time=0
       error_code=0
SET TIMESTAMP=1303354082/*!*/;
BEGIN
/*!*/;# at 480
...
```

注意

从 MySQL 5.1 开始，你可以指定 ROW 或者 MIXED 的二进制日志格式。但只有当使用默认的 STATEMENT 格式时，对二进制日志 SQL 语句的分析才可能进行。

7.1.4　进程列表

这个样例进程可以找出当前运行的执行时间很长的或者常用的查询。另外，这个命令还可以提供一些执行 SQL 语句的内部步骤信息。它还是一个找出执行时间很长从而阻塞了其他 SQL 语句的 DML 语句的简单方法。以上可以通过 State 的值为 Locked 来判断。

```
mysql> SHOW FULL PROCESSLIST;
---+----------+---+-------+-----------------------+--------------
...| Host     | db | Command | Time | State              | Info
   | db:44376 | db1 | Query |    3 | Sending data       | SELECT...
   | db:44703 | db1 | Query |    0 | statistics         | SELECT...FOR UPDATE
   | db:44812 | db1 | Query |    0 | init               | UPDATE...
   | db:44866 | db1 | Sleep |    2 |                    | NULL
   | db:44866 | db1 | Sleep |    2 |                    | NULL
   | db:56751 | db1 | Query |    0 | Updating           | UPDATE...
   | db:44868 | db1 | Query |    0 | query end          | SELECT * FROM...
   | db:56792 | db1 | Query |    0 | logging slow query | INSERT INTO...
---+----------+---+-------+----+-------------------+--------------
```

为了便于在本书中显示，以上输出结果格式做了一些修改。

你也可以通过命令行工具 mysqladmin 来获得进程列表的信息，语法如下：

```
$ mysqladmin -uroot -p [-v] processlist
```

使用如下的SQL语句同样可以从INFORMATION_SCHEMA(从5.1版本开始)中获得这些信息：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST;
```

7.1.5 引擎状态

存储引擎特有的 SHOW ENGINE[引擎] STATUS 命令可以提供一些额外的 SQL 信息。默认包含的 MySQL 存储引擎中，唯一使用这个命令的是 InnoDB。这不是一个理想的 SQL 语句来源，因为它主要展现其他系统的特性；但出于完整性考虑还是要有这个功能，因为这是展示出由于某类错误而执行的 SQL 语句的唯一途径。

SHOW ENGINE INNODB STATUS 命令可以提供 SQL 语句的详细信息，包括那些运行的 SQL 语句以及那些导致外键验证失败或者造成死锁的 SQL 语句。请看下面的示例：

```
------------
TRANSACTIONS
------------
Trx id counter 0 480206585
Purge done for trx's n:o < 0 480205682 undo n:o < 0 0
History list length 5
LIST OF TRANSACTIONS FOR EACH SESSION:
---TRANSACTION 0 0, not started, process no 27230, OS
thread id 1182685536
MySQL thread id 427, query id 123563 10.1.1.1 dba
```

```
SHOW ENGINE INNODB STATUS
---TRANSACTION 0 0, not started, process no 27230, OS
thread id 1169639776
MySQL thread id 44, query id 123445 10.1.1.1 dba
---TRANSACTION 0 480206305, ACTIVE 67 sec, process no
27230, OS thread id 1192270176
mysql tables in use 1, locked 1
1 lock struct(s), heap size 368, 0 row lock(s), undo
log entries 1
MySQL thread id 370, query id 123171 10.1.1.2 user
query end
INSERT INTO `table` ....
---TRANSACTION 0 480206290, ACTIVE 70 sec, process no
27230, OS thread id 1178159456
mysql tables in use 1, locked 1
1 lock struct(s), heap size 368, 0 row lock(s), undo
log entries 1
MySQL thread id 372, query id 123150 10.1.1.2 user
query end
INSERT INTO `table` ....
---TRANSACTION 0 480206289, ACTIVE 70 sec, process no
27230, OS thread id 1178425696
mysql tables in use 1, locked 1
1 lock struct(s), heap size 368, 0 row lock(s), undo
log entries 1
MySQL thread id 373, query id 123149 10.1.1.2 user
query end
INSERT INTO `table` ....
....
```

也可以从 SHOW ENGINE INNODB STATUS 的输出结果的 LATEST DETECTED DEADLOCK 和 FOREIGN KEY ERROR 部分获取 SQL 语句。

7.1.6　MySQL 连接器

很多种类的 MySQL 连接器提供了截取 SQL 语句的简便方

法。这些连接器包括：

1. Connector/J

可以参考以下 Connector/J 数据源配置属性：

- logSlowQueries
- slowQueryThresholdMillis
- useNanosForElapsedTime
- slowQueryThresholdNanos
- autoSlowLog

想了解更多有关这些属性的信息请参考 http://dev.mysql.com/doc/refman/5.5/en/connector-j-reference-configuration-properties.html。

2. PHP mysqlnd

这个从PHP 5.3版本开始新加入的原生MySQL驱动器用于创建管理和扩展访问 SQL 通信协议的插件。例如，这个驱动器可以让你截取通过 PHP 运行的 SQL 语句并将其写入日志中，只需要按照下面的步骤：

(1) 下载 PHP 5.3.x 或更新的版本。

(2) 下载 PECL/mysqlnd_qc。

(3) 通过--enable-mysqlnd-qc 来配置 PHP。

(4) 把下面的变量添加到你的 PHP 输出结果中：var_dump (mysqlnd_qc_get_query_trace_log())。

7.1.7 应用程序代码

在你的应用程序代码中使用抽象类来实现所有数据库的访问是一种理想的截取详细 SQL 信息的方式，可以截取到的信息包括实际的 SQL 语句、查询的执行时间以及 SQL 语句的执行顺序。

一个定义好的能够动态开启或者关闭的应用程序基础框架可以提供截取 SQL 语句的详细说明。

我们使用一个简单的 PHP 旧有应用程序为例，添加应用程序代码可以通过下面的简单步骤做到：

(1) 用 db_query 替换所有的 mysql_query。

(2) 为 db_query 创建一个抽象的数据库查询函数，其中 db_query 替换了 mysql_query。

```
function db_query($sql) {
    $rs = mysql_query($sql);
    return $rs;
}
```

有了这个基本的抽象函数，你就可以添加即时和输出显示的代码了：

```
function db_query($sql) {
  $start_time = microtime(true);
  $rs = mysql_query($sql);
  debug(format_time(microtime(true) - $start_time),
  $sql);
  return $rs;
}
function format_time($time) {
  return number_format($time, 8, '.', '');
}
function debug($exec_time, $str) {
  // Add some flag to enable/disable
  echo '<!-- '.$str.' ('.$exec_time.') -->';
  return 0;
}
```

在这个示例中，我们简单地把 SQL 语句输出到 HTML 注释中。理想情况下，可以通过一些参数或者变量来决定是否显示 SQL

输出，并且可以用一种更便于管理的方式核对和格式化 SQL 语句。应用程序级别的 SQL 日志有助于识别 SQL 注入，这是通过把传递给 MySQL 的 SQL 语句和真正在 MySQL 中执行的 SQL 语句对比实现的。

7.1.8 INFORMATION_SCHEMA

目前的 INFORMATION_SCHEMA.INNODB_TRX 表提供了一些同样可用于 SHOW ENGINE INNODB STATUS 事务列表中的信息。INFORMATION_SCHEMA 允许安装其他插件用于创建有可能会暴露 SQL 语句的表，这一点可以被其他存储引擎利用。

想了解更多信息请访问 http://dev.mysql.com/doc/refman/5.5/en/innodb-i_s-tables.html。

7.1.9 PERFORMANCE_SCHEMA

PERFORMANCE_SCHEMA 目前并不提供与截取 SQL 语句相关的单个 SQL 语句的语法信息。但是这些信息在高性能要求的系统中对有问题的 SQL 语句进行深入分析时非常重要。我们也不会在本书中讨论 PERFORMANCE_SCHEMA，它是在进行非常专业的 SQL 分析时用到的底层监控信息的来源。

想了解更多信息，请访问：http://dev.mysql.com/doc/refman/5.5/en/performance-schema.html。

7.1.10 SQL 语句统计插件

开源项目 sqlstats 是一个利用审计插件接口来截取 MySQL 5.5 或更高版本中所有 SQL 语句的 MySQL 插件。更多信息可以参考 http://sourceforge.net/projects/sqlstats/。

更多关于 MySQL 的审查插件接口的信息可以从这里了解到：

http://dev.mysql.com/doc/refman/5.5/en/audit-plugins.html。

7.1.11　MySQL Proxy

MySQL Proxy 或者其他代理的派生产品，在通过网络与 MySQL 实例的通信中扮演了一个中间人的角色。这个命令可以拦截客户端和服务器之间的 TCP/IP 数据包，并根据很多灵活的且可以用程序定制的需求来处理这些数据包。MySQL Proxy 默认安装中附带了一些样例，包括 SQL 截取、SQL 日志记录以及 SQL 语句分析。

可以在下面的地址找到更多包括软件下载、安装等的信息：https://launchpad.net/mysql-proxy。

7.1.12　TCP/IP

当你通过网络连接到 MySQL 时，可能会截取到和 MySQL 服务器通信的物理层网络数据包。MySQL 协议是一个简单的开源通信流，你可以人工解读部分细节，也可以使用各种分析工具来解析这些流。

有很多不同的截取 TCP/IP 通信量的方式。下面的简单命令会产生一个 MySQL 协议的文本输出结果，其中，它包含 SQL 语句和其他一些命令：

```
$ sudo tcpdump -l -i eth0 -w - src or dst port 3306
-c 1000 | strings
```

7.2　识别有问题的语句

从上一步截取下来的 SQL 语句中找到有问题的语句，也是有

优先级顺序的。关注运行最慢的 SQL 查询并不是 SQL 语句优化的唯一技术需求。优化执行频率高的语句同样能够显著改进系统性能。一条每秒执行成百上千次的查询获得 10 毫秒的速度提升带来的益处远远大于一条每秒仅仅执行几次的查询得到 1 秒的速度提升的好处。

注意

频繁执行且速度很快的 SQL 语句通常被认为不是必须要调整的。但提升这些查询的速度能够为系统资源带来非常大的益处，这些资源使得你的数据库系统能够处理更多的 SQL 语句。

识别的过程可以用流水线的方式进行，通过在 SQL 语句中添加 C 语言风格的注释来实现。请看下面的示例：

```
SELECT /* 5m cache */ ...
SELECT /* AddPost */ ...
SELECT /* CustomerReport */ ...
UPDATE /* EOM processing */ ...
```

该技术在识别那些原本就需要很长的运行时间并且被定位和分类的优先级要求都比较低的查询时很有用。例如，在一个集合了在线事务处理(OLTP)、批处理、报表以及缓存查询功能于一身的应用程序中，找到那些重要但低优先级的且运行时间长的 SQL 语句能够为我们指出其他语句应该被优先审查和优化。

被截取的 SQL 语句的集合将会大致找到最有可能需要优化的 SQL 语句候选。找到最高执行频率和运行时间最长的查询，这些查询通常是典型的最需要优化的候选者。

在标准安装下，只有 mysqldumpslow 命令提供了任意级别的集合查找。请看下面的示例：

```
$ mysqldumpslow /path/to/slow.log
Count: 1 Time=4.61s (4s) Lock=0.00s (0s) Rows=0.0 (0),
rb[rb]@localhost
  SELECT option_name, option_value FROM wp_options
WHERE autoload = 'S'
Count: 1 Time=3.83s (3s) Lock=0.00s (0s) Rows=0.0 (0),
rb[rb]@localhost
  UPDATE `wp_postmeta` SET `meta_value` = 'S' WHERE
`meta_key` = 'S' AND `post_id` = N
Count: 1 Time=3.26s (3s) Lock=0.00s (0s) Rows=154.0
(154), rb[rb]@localhost
  SELECT /*!N SQL_NO_CACHE */ * FROM `wp_options`
Count: 1 Time=2.14s (2s) Lock=0.00s (0s) Rows=0.0 (0),
rb[rb]@localhost
 SELECT option_value FROM wp_options WHERE
 option_name = 'S' LIMIT N
```

想了解更多信息请参考：http://dev.mysql.com/doc/refman/5.5/en/mysqldumpslow.html。

可以通过一行简单的 Linux 语句使用 mysqlbinlog 命令为二进制日志生成聚合信息，请看下面的示例：

```
$ mysqlbinlog /path/to/mysql-bin.000999 | \
   grep -i -e "^update" -e "^insert" -e "^delete" -e
"^replace" -e "^alter"| \
   cut -c1-100 | tr '[A-Z]' '[a-z]' | \
   sed -e "s/\t/ /g;s/\`//g;s/(.*$//;s/ set .*$//;s/
   as .*$//" | \
   sed -e "s/ where .*$//" | \
   sort | uniq -c | sort -nr

  33389 update e_acc
  17680 insert into r_b
  17680 insert into e_rec
  14332 insert into rcv_c
  13543 update e_rec
```

```
  10805 update loc
   3339 insert into r_att
   2781 insert into o_att
...
```

注意

为了提高可读性，这个命令被分成多行来显示。

想了解更多信息请参考：http://ronaldbradford.com/blog/mysql-dml-stats-per-table-2009-09-09/。

Maatkit 的 mk-query-digest 工具是一个通用的开源工具，它能够从各种来源中定位 SQL 语句，包括慢查询日志、全面查询日志、二进制日志以及 TCP/IP 栈。这个工具的优点是它能够提供有价值的高频率 SQL 语句的集合信息，以及最小、最大以及平均的执行时间。它的输出结果是经过格式化的，能够很方便地提供前面介绍过的一些必要分析命令的删除、粘贴语法。

7.2.1 慢查询日志分析

对于被找到和经过分析的查询会有如下的输出结果：

```
$ mk-query-digest --type slowlog /var/log/mysql/slow.log
...
# Query 1: 0.00 QPS, 2.74x concurrency, ID 0x036A4959F0EBB3BD at byte -1 _
#             pct   total     min     max     avg     95%  stddev median
# Count         1     395
# Exec time    28 478528s     11s   4733s   1211s   4168s   1431s    511s
# Lock time    20  31777s       0   2769s     80s    685s    291s       0
# Rows sent     0   3.86k      10      10      10      10       0      10
# Rows exam     3 193.22M 500.73k500.96k500.90k485.50k       0485.50k
# Users                1 app
# Hosts                5 app5.xxxxx... (128), app1.xxxx... (85)... 3 more
# Databases            1 db
# Time range 2009-05-18 10:23:18 to 2009-05-20 10:50:14
```

```
# bytes           0 184.67k  476  479  478.73 463.90  0  463.90
# Query_time distribution
# 1us
...
# 1s
# 10s+ ##################################################
# Tables
#      SHOW TABLE STATUS FROM `db` LIKE 'gg'\G
#      SHOW CREATE TABLE `db`.`gg`\G
#      SHOW TABLE STATUS FROM `db ` LIKE 'gr'\G
#      SHOW CREATE TABLE `dnb`.`gr`\G
#      SHOW TABLE STATUS FROM `db ` LIKE 'tt'\G
#      SHOW CREATE TABLE `db `.`tt`\G
# EXPLAIN
SELECT tt.*,
          (SELECT COUNT(*) FROM gg WHERE gg.tt_user_id
          = tt.id) AS g_c,
          (SELECT COUNT(*) FROM WHERE gr.tt_user_id =
          tt.id ) AS r_c
           FROM tt AS tt
           WHERE 1 AND tt.user_type = 'CUSTOMER'
           GROUP BY tt.id HAVING g_c > 1 OR r_c > 1
           ORDER BY tt.id desc, g_c DESC , r_c DESC
           LIMIT 10 OFFSET 0\G
```

可以看出这个结果包含很多信息。你可以确定给定时间范围内每条语句的执行频率、来源用户以及主机。你能够得到执行时间、处理过的行以及结果集大小的最小、最大、平均以及中间值的统计信息。还可以方便地对这些信息进行删除/粘贴来获得 QEP 以及其他有价值的分析信息。

很多其他开源工具以及代码片段也可以执行慢查询日志分析。例如这个开源项目：http://code.google.com/p/mysql-slow-query-log-parser/。

7.2.2 TCP/IP 分析

各种截取 SQL 语句的方法中侵入性最小也最接近实时工具的方式是通过 TCP/IP。mk-query-digest 工具的输出结果和其他来源是一样的；但关于样本容量的汇总信息在排错时将会非常有用。这个命令的输出结果提供了 3 个不同部分的信息，可以用在不同级别的查询定位、分类和初始化分析中。

第一部分提供了总体时间、查询频率以及硬件故障细节的顶级汇总信息：

```
# 4.5s user time, 40ms system time, 20.69M rss, 108.81M
vsz
# Current date: Fri Mar 18 09:50:13 2011
# Hostname: xxx.example.com
# Files: 110318.0949.tcp
# Overall: 2.71k total, 197 unique, 149.83 QPS, 3.65x
concurrency ________
# Time range: 2011-03-18 09:49:50.178180 to
09:50:08.258622
# Attribute      total min max  avg 95% stddev median
# ==========  ======= ======= ======= ======= =======
# Exec time       66s       0     5s    24ms    34ms
                                     216ms   119us
# Rows affected   241       0      2    0.09    0.99
                                       0.33       0
# Query size  612.79k      14  7.17k  231.63   1.53k
                                     586.40   26.08
# Warning count   115       0      3    0.04       0
                                       0.28       0
# Boolean:
# No index use 3% yes, 96% no
```

从这个示例时间段中我们可以看到分析了 2 710 个查询，产生了 197 个不同的 MySQL 请求。

第二部分提供了最差的不同查询的详细列表，这个列表是按照总执行时间排序的：

```
# Profile
# Rank Query ID           Response time Calls R/Call Apdx V/M Item
# ==== ================== ============= ===== ====== ==== =====
#    1 0xE9850B5E2CB37759 6.7643 10.2%     2 3.3821 0.25 0.43
DELETE pip
#    2 0x1EDF0DD7357332ED 4.5564  6.9%     1 4.5564 0.00 0.00
SELECT oi iu u uu
#    3 0x763E3EEAFA1CFEAE 4.4828  6.8%     2 2.2414 0.50 0.01
SELECT pip
#    4 0xB1154FBE5C91CB61 4.4195  6.7%    47 0.0940 1.00 0.20
INSERT lp
#    5 0x5D379C422F763082 4.3450  6.6%     1 4.3450 0.00 0.00
SELECT i ip oip u
#    6 0xD83B315B5BA461E9 3.6669  5.5%     1 3.6669 0.50 0.00
SELECT i uip p
...
#   47 0xDBB6833B67C98FD3 0.1097  0.2%    28 0.0039 1.00 0.09
SELECT s
#   48 0xAEBB8C3A072C2ED5 0.1059  0.2%    14 0.0076 1.00 0.00
SELECT ci cp c
#   49 0x1EF255F0C1F71360 0.1057  0.2%     1 0.1057 1.00 0.00
SELECT am ac
#   50 0xC9F21CA3FDD63ABF 0.1028  0.2%     1 0.1028 1.00 0.00
SELECT pp
# MISC 0xMISC 2.1004 3.2% 936 0.0022 NS 0.0 <147 ITEMS>
```

在这个示例中，一条 DELETE 语句的两次调用占用了的时间是超过 2 000 条查询的总执行时间的 10%，很显然这是找到了一条需要进一步审查更多细节的查询。

审查和优化语句的其中一步是找到那些不应该被调用的 SQL 查询。在下面的示例中，通过查看执行过的 SQL 语句我们发现 ROOLBACK 语句被调用了很多次。而当使用 MyISAM 引擎时，

SQL 语句的结果并不会影响到数据，执行语句耗费的时间会对总体性能产生影响。

```
# Profile
# Rank Query ID           Response time    Calls R/Call  Item
# ==== ================== ================ ===== ======= =======
#    1 0x4ED092EFA577DAB7    0.0106 24.8%     1  0.0106  SELECT p
#    2 0xC9ECBBF2C88C2336    0.0102 23.8%    52  0.0002  SELECT rc
#    3 0x19C8068B5C1997CD    0.0092 21.6%   138  0.0001  ROLLBACK
#    4 0x448E4AEB7E02AF72    0.0091 21.3%    52  0.0002  SELECT rt
#    5 0x56438040F4B2B894     0.0015 3.6%     2  0.0008  SELECT hc
```

7.3 确认语句执行

确认一条 SQL 语句和运行 SQL 语句以及确认响应时间一样简单。在确认 SQL 语句的精确返回时间时应该考虑到的因素包括：当前系统负载、查询并发程度、网络开销、MySQL 查询缓存以及能否在内存内部访问必要的表索引和数据。在确认 SQL 语句时，一定要尽量重现这些工作环境。一条语句在隔绝环境中的运行肯定会和同样的语句在多任务环境中运行的返回不同的结果。

7.3.1 环境

你的最终目标是使用一种可以重现的方式来确认你发现的 SQL 语句。这种情况将允许你为后面的优化重现这些信息。

在确认 SQL 语句时，你应该在 SELECT 语句上添加 SQL_NO_CACHE 提示来保证没有使用 MySQL 查询缓存。另一个技巧是在确认 SQL 语句之前尽量把所有表数据都加载到合适的 MySQL 内存缓冲区中，这样可以保证更加一致的结果返回时间。包括内存、CPU 以及磁盘在内的系统资源将会对 SQL 查询

执行产生重要影响，因此这些系统资源应该在确认你的 SQL 语句时被监控以此来保证结果的一致性。

使用不同的网络路径来测试 SQL 语句也会产生误导性的信息。所以如果可能的话，你应该在有网络开销和没有网络开销两种环境中验证你的 SQL 语句。

在类似生产系统的环境中也许不可能确认 SQL 语句。重要的是创建一个可重现的确认信息，这样的话任何优化都可以很容易地得到验证。

7.3.2　时间统计

mysql 客户端默认以 10 毫秒为单位显示查询时间。在现代强大的硬件条件下，很多查询的执行时间在 10 毫秒这一古老的限制之下。有一个补丁可以让你在原来的客户端中得到微秒级别的精度。可以从这里了解更多信息：http://EffectiveMySQL.com/article/microsecond-mysql-client。

启用会话时，SHOW PROFILES 命令后可以提供 SQL 语句的微秒级别粒度。下面一系列随机的查询语句展示了一些以不同的持续时间运行的样例结果，以此来说明微秒级精度。

```
mysql> SET PROFILING=1;
mysql> SELECT NOW();
mysql> SELECT BENCHMARK(1+1,100000);
mysql> SELECT BENCHMARK('1'+'1',100000);
mysql> SELECT SLEEP(1);
mysql> SELECT SLEEP(2);
mysql> SHOW PROFILES;
+----------+------------+----------------------------------+
| Query_ID | Duration   | Query                            |
```

```
+----------+------------+-----------------------------------+
|        1 | 0.01768300 | SELECT NOW()                      |
|        2 | 0.00014600 | SELECT BENCHMARK(1+1,100000)      |
|        3 | 0.02124700 | SELECT BENCHMARK('1'+'1',100000)  |
|        4 | 1.00130800 | SELECT SLEEP(1)                   |
|        5 | 2.00152800 | SELECT SLEEP(2)                   |
+----------+------------+-----------------------------------+
```

这个技巧在第 1 章介绍的 SQL 语句分析过程中用到过。用于确认细节信息的时间统计结果如下所示：

```
|  15 | 3.18892300 | SELECT * FROM inventory WHERE
item_id = 16102176 |
|  16 | 0.00072900 | SELECT * FROM inventory WHERE
item_id = 16102176 |
```

一个定义良好且在 SQL 性能方面很积极的基础架构能够主动收集很多信息，不仅仅包括 SQL 语句、QEP、查询执行时间，还包含很多其他的查询细节，例如获取到的行数、结果集大小、底层有代表性的表数据大小以及查询分析中用到的 MySQL 配置信息。这些长期收集的 SQL 语句信息能够帮助我们找到查询性能的变化，还有助于验证不同的 MySQL 配置、模式结构以及 MySQL 版本。如果想要把上述信息合并并以自定义的格式输出，那么 MySQL Proxy 是一个不错的选择。

数据的分布以及数据量随时间的增长而增加也会对 SQL 查询的性能产生很大的影响。

7.4 语句分析

在你找到并确认那些 SQL 语句之后，就可以参考第 2 章介绍的命令和工具来分析 SQL 语句了。

7.5　语句优化

第 4 章和第 5 章详细介绍了使用索引来优化 SQL 语句的过程。其他优化 SQL 语句的技术也可能提供更好的性能改进，包括通过去除连接操作或减少列的数目来简化 SQL 语句，还有通过精简数据类型和约束条件(比如是否可以为空)来改进表的结构。第 8 章会详细介绍各种相关的技术。

7.6　结果验证

任何优化过程的最重要一步都是最后的验证过程。如果没有详细的情况的确认，你就不可能体会到任何优化的真正益处。验证过程应该通过不同的条件来确认，而不是仅仅重复确认过程的那些步骤。在生产环境中，一个高并发并且混合的工作负载来验证会带来真正的益处。

查询在受控环境中的运行情况和你预期的一样吗？没有一个详细的集成监控环境，你就无法确保单个 SQL 优化所带来的总体益处。对一个查询的优化是否会对其他系统功能模块的性能产生影响？优化效果经过一定时间之后会随着数据容量的变化而退化吗？如果没有仔细检视一个优化过程并确定它带来真实改进，一定不能测试部署这个优化。这一点对任何类型的优化都是适用的，包括改变配置这种优化方法。

7.7 本章小结

SQL 优化是一个持续改进的过程。本章介绍的这些步骤应该反复地执行来确保合适的 SQL 语句在恰当的时间被优化。一个定义良好的基础架构能够从各种 MySQL 环境中的个别查询时间中截取、记录并监控其中对作用时间要求很高的关键组件。

QEP 也会在一段时间之后随着数据的分布和增长的变化而发生改变。新的应用程序功能模块、模式设计的变更、查询并发性的增加或者物理硬件瓶颈都会成为定位和优化 SQL 语句时的未知因素。

本章所有相关的 SQL 语句示例都可以从下面的地址下载：http://EffectiveMySQL.com/book/optimizing-sql-statements。

第 8 章

性能优化之隐藏秘籍

在前面的章节中，我们介绍了添加索引能大幅度提升 SQL 的性能。除此之外，还有其他进一步改进索引效率的方法。有多种技术可以用于简化或者减少 SQL 语句，从而对性能和吞吐量提升

带来立竿见影的效果。

本章中将讨论下面几种性能优化的技巧：

- 去除重复的索引
- 找到没有被使用的或者无效的索引
- 改进索引
- 减少 SQL 语句
- 简化 SQL 语句
- 缓存选项

8.1 索引管理优化

索引的管理过程，即索引创建和维护的方式，能够影响到 SQL 语句的性能。

8.1.1 整合 DDL 语句

在将索引添加到 MySQL 表的过程中，一个需要注意的管理问题就是 DDL 语句是阻塞性。在以前，由于 ALTER 语句的阻塞性影响，执行 ALTER 语句时需要为表创建一个新的副本。在改变大型表时，这个操作将消耗大量的时间和磁盘存储空间。有了在 MySQL 5.1 中首次出现的 InnoDB 插件以及其他第三方存储引擎，各种 ALTER 语句现在运行非常迅速，因为它们不用再执行全表复制操作了。你可以参考特定存储引擎和 MySQL 版本的系统文档来了解 ALTER 语句的全面影响。

把多条 ALTER 语句整合成一条 SQL 语句是一种简单的优化改进。例如，如果需要添加一个新的索引、修改一个索引以及添加新的一列，可以分别运行下面这些 SQL 命令：

```
ALTER TABLE test ADD INDEX (username);
ALTER TABLE test DROP INDEX name, ADD INDEX name
(last_name,first_name);
ALTER TABLE test ADD COLUMN last_visit DATE NULL;
```

可以通过把这些在同一张表上运行的语句整合成为一条 SQL 语句的方式来优化这个 SQL：

```
ALTER TABLE test
ADD INDEX (username),
DROP INDEX name,
ADD INDEX name (last_name,first_name),
ADD COLUMN last_visit DATE NULL;
```

该优化能够大幅度提升管理任务的性能。

8.1.2　去除重复索引

重复的索引有两个主要的影响：第一，所有 DML 语句都会运行得更慢，因为需要做更多工作来保持数据和索引的一致性；第二，数据库的磁盘占用量将会更大，这将导致备份和恢复需要的时间增加。

一些简单的情况会导致索引重复。MySQL 并不要求主码所在的列也被索引。请看下面的示例：

```
CREATE TABLE test(
   id INT UNSIGNED NOT NULL,
   first_name VARCHAR(30) NOT NULL,
   last_name VARCHAR(30) NOT NULL,
   joined     DATE NOT NULL,
   PRIMARY KEY(id),
   INDEX (id)
);
```

在这个 DDL 中，在 id 列上定义的索引是一个重复索引，应该被

移除。

当一个给定索引的最左边部分被包含在其他索引中时也会产生重复索引，请看下面的示例：

```
CREATE TABLE test(
  id INT UNSIGNED NOT NULL,
  first_name VARCHAR(30) NOT NULL,
  last_name VARCHAR(30) NOT NULL,
  joined DATE NOT NULL,
  PRIMARY KEY(id),
  INDEX name1 (last_name),
  INDEX name2 (last_name, first_name)
);
```

name1 这个索引是多余的，因为此索引所在的列已经被包含在索引 name2 的最左边部分里面了。

Maatkit 的 mk-duplicate-index-checker 命令是一个开源工具，可以快速地检查你的数据库模式中是否包含重复索引。可以参考这里了解更多信息：http://maatkit.org/mk-duplicate-key-checker。

8.1.3 删除不用的索引

除了重复索引没有被使用到之外，还有其他索引可能也没有被用到。这些索引和重复索引一样会影响性能。官方 MySQL 产品没有提供识别不用的索引的方法；然而，一些其他版本的 MySQL 提供了这个特性。

Google 的 MySQL 补丁(http://code.google.com/p/google-mysql-tools/wiki/Mysql5Patches)首次引入了 SHOW INDEX_STATISTICS 功能。这个特性是诸多用来衡量每个用户监控的新命令的一部分。

对于官方 MySQL 产品，想要找到没有使用的索引，首先需要收集所有执行过的 SQL 语句。使用这些 SQL 语句为所有 SQL

语句捕捉和聚合 QEP，然后对每个表进行分析可以得到没有使用的索引的信息。这个截取 SQL 语句和生成 QEP 的过程对所有应用程序都是适用的最佳实践。

8.1.4　监控无效的索引

当定义多列索引时，一定要注意确定所指定的每一列是否真的有效。官方 MySQL 产品同样没有提供任何相关的说明。

可以通过分析指定表上的所有 SQL 语句的 key_len 列来找到那些可能包含没有使用到的列的索引。

第 9 章会给出详细的示例说明如何理解和分析 key_len 列。

8.2　索引列的改进

除了创建新的索引来改进查询性能之外，还可以通过其他模式优化的方法来改进数据库性能，包括使用特定的数据类型以及/或者列类型。这样做的好处是可以使用更小的磁盘空间，从而减少了 I/O 的开销，并且使得更多的索引数据可以被打包装载进入可用的系统内存中。

8.2.1　数据类型

有几种数据类型是可以被替换或修改的，而且这样做几乎不会对已有的模式造成任何影响。

1. BIGINT 和 INT

当一个主码被定义为 BIGINT AUTO_INCREMENT 数据类型时，一般来说不需要搞清楚这个数据类型存在的原因。一个 INT

UNSIGNED AUTO_INCREMENT 数据类型能够支持的最大值是 43 亿。如果一个表包含的行数大于 43 亿，那么除了使用 BIGINT 数据类型之外，还有其他架构因素需要考虑。

把主码列的数据类型从 BIGINT 改成 INT 的影响是此列中每行的容量减小 50%，即从 8 字节减少到 4 字节。这个改动不仅仅针对主码，所有定义为 BIGINT 的外键都可以改成 INT。这样做节省下来的空间能够显著地减少在高度规范化的数据库中索引所需要的存储空间。更多详细解释请参考：http://ronaldbradford.com/blog/bigint-v-int-is-there-a-big-deal-2008-07-18/。

技巧

在 AUTO_INCREMENT 列上把数据类型从 BIGINT 改成 INT UNSIGNED 是最有效的模式改进方法之一，尤其是对那些只有有限的内存同时具有高度规范化的数据模型的数据库服务器。

2. DATETIME 和 TIMESTAMP

记录一个日期或者时间的值时可以用一个纪元值(即从 1970 年 1 月 1 日起到现在的秒数)吗？如果所存储的值只是一个纪元值，那么一个 TIMESTAMP 列支持所有必需的值。一个 DATETIME 列支持所有可能的日期/时间的值。一个 DATETIME 数据类型占用 8 字节，而一个 TIMESTAMP 类型占用 4 字节。

使用 TIMESTAMP 类型的缺点是这种类型隐含的默认值为 0，这和任何与 SQL_MODE 相关的禁用 0 日期设置相违背。TIMESTAMP 数据类型也不支持 NULL 值。这个列类型适合存储那些总是有值存在的时间或日期，例如一行数据的创建时间或者修改时间。

3. ENUM

MySQL 提供了 ENUM 数据类型，这个类型适合存储静态的代码值。例如，当要记录性别的值时，可以选择以下方法中的任意一种来定义：

```
gender1 VARCHAR(6) NOT NULL
gender2 ENUM ('Male','Female') NOT NULL
```

使用 ENUM 数据类型有 3 个优点：

- 一个 ENUM 列利用隐含的检查限制提供了额外的数据完整性支持。
- 一个 ENUM 列仅使用 1 字节来存储 255 个不同的值。
- ENUM 列的值更具有可读性。例如，如果你有一个状态字段，使用 ENUM 类型会使数据容量更加紧凑，并且为每一列的值提供有意义的说明。

```
status1 CHAR(1) NOT NULL DEFAULT 'N', // 'N','A','X'
status2 ENUM ('New','Active','Archived') NOT NULL
DEFAULT 'New'
```

在以前，使用 ENUM 类型会带来一些影响，即当需要加入新的值时，数据库就需要改变 ENUM 值的范围。而这种改变范围是一个 ALTER DDL 语句，它将会阻塞其他语句。从 MySQL 5.1 开始，可以很快地在 ENUM 列中添加新值并且不会影响到表的大小。

4. NULL 和 NOT NULL

除非你确定一个列能包含一个未知的值(一个 NULL 值)，否则最好把这一列定义为 NOT NULL。有些框架(如 Ruby on Rails)因为没有把强制的列定义为 NOT NULL 而饱受诟病。当列被定义

在一个索引中时，那么此列可以占用更小的空间，并简化索引的处理过程，因为不需要其他 NULL 条件。而且，NOT NULL 会在列的数据上添加额外的完整性约束检查，以保证所有行在此列上都有数据。

5. 隐含的变换

当你为表连接选择一个索引数据类型时，一定要确保这个数据类型是相同的。隐含的类型转换将带来不必要的开销。对于整数类型的列，要确保 SIGNED 和 UNSIGNED 类型是统一的。对于可变的数据类型，会出现字符集和“逻辑与”运算的复杂情况。当为表连接条件定义索引列时，要确保这些都匹配。这里经常出现在 LATIN1 和 UTF8 字符集之间的隐含转换的问题。

8.2.2 列的类型

有几种数据类型被发现广泛地用在一些并不高效的列类型中。改变这些列的数据类型能够带来更高效的存储，尤其是当这些列属于某个索引时。下面就是一些常见的示例。

1. IP 地址

一个 IPv4 地址可以被定义成 INT UNSIGNED 数据类型，只占用 4 字节。通常情况下将列定义为 VARCHAR(15)，这样平均要占用 12 字节。这样一个改进就可以节约 2/3 的列数据占用的空间。INET_ATON()和 INET_NTOA()函数可以管理 IP 地址在字符串和数字值之间的转换。请看下面的示例：

```
mysql> SET @ip='123.100.0.16';
mysql> SELECT @ip, INET_ATON(@ip) AS str_to_i,
    -> INET_NTOA(INET_ATON(@ip)) as i_to_str;
```

```
+-----------------+-------------+---------------+
| @ip             | str_to_i    | i_to_str      |
+-----------------+-------------+---------------+
| 123.100.0.16    | 2070151184  | 123.100.0.16  |
+-----------------+-------------+---------------+
```

这个技术只适用于 IPv4。随着 IPv6 的广泛采用，一定要将这些 128 比特的整数值以 BINARY(16)的类型存储，而不是用 VARCHAR 数据类型那种人们可以阅读的方式。

2. MD5

用 CHAR(32)来存储 MD5 值是一个常见的技巧。如果你的应用程序使用 VARCHAR(32)，则对每个值的字符串长度都需要花费额外的不必要的开销。这个十六进制的值可以使用 UNHEX()和 HEX()函数来存储在 BINARY(16)数据类型中且更为高效。用这种转换可以让每行占用的存储空间从 32 字节减少到 16 字节。

下面这个示例展示了原始 MD5 以及压缩过的 MD5 占用的空间对比：

```
mysql> SET @str='somevalue';
mysql> SELECT MD5(@str),
    -> LENGTH(MD5(@str)) AS len_md5,
    -> LENGTH(UNHEX(MD5(@str))) as len_unhex;
+-----------------+-----------------+-------------+
| MD5(@str)       | len_md5         | len_unhex   |
+-----------------+-----------------+-------------+
| d5d984e0a00665878320727318ac378c |      32 |  16 |
+-----------------+-----------------+-------------+
```

这个原则也适用于其他 16 进制的值，例如为所有列的散列值而定义的索引时。

8.3 其他 SQL 优化

添加索引能够带来显著的性能提升。然而，对关系型数据库来说最有效的 SQL 优化方法是完全删除不需要执行的 SQL 语句。对于一个高度优化的应用程序来说，占总执行时间最大比重的是网络开销。去除 SQL 语句能够减少应用程序的处理时间。对于 SQL 语句来说其他必要的步骤还包括解析语句、安全许可检查以及生成查询执行计划。如果其中有不必要的语句，那么这些都会为数据库服务器不必要的负担。可以使用分析功能在查询执行过程中获得这些步骤的详细时间统计。请看下面的例子：

```
mysql> show profile source for query 7;
+----------------------+--------+------------------+--------------+------+
|Status                |Duration|Source_function   |Source_file   |Src_line|
+----------------------+--------+------------------+--------------+------+
|starting              |0.000202|NULL              |NULL          | NULL |
|checking permissions |0.000011|unknown function  |sql_parse.cc  | 5160 |
|Opening tables        |0.000261|unknown function  |sql_base.cc   | 4472 |
|System lock           |0.000012|unknown function  |lock.cc       |  258 |
|Table lock            |0.000015|unknown function  |lock.cc       |  269 |
|init                  |0.000115|unknown function  |sql_select.cc | 2373 |
|optimizing            |0.000013|unknown function  |sql_select.cc |  771 |
|statistics            |0.000034|unknown function  |sql_select.cc |  953 |
|preparing             |0.000016|unknown function  |sql_select.cc |  963 |
|Creating tmp table    |0.000041|unknown function  |sql_select.cc | 1417 |
|executing             |0.000006|unknown function  |sql_select.cc | 1647 |
|Copying to tmp table |0.000038|unknown function  |sql_select.cc | 1794 |
|optimizing            |0.000009|unknown function  |sql_select.cc |  771 |
|statistics            |0.000047|unknown function  |sql_select.cc |  953 |
|preparing             |4.344404| unknown function |sql_select.cc |  963 |
|Sending data          |2.234372|unknown function  |sql_select.cc | 2204 |
|end                   |0.000010|unknown function  |sql_select.cc | 2419 |
```

```
|removing tmp table   |0.003175|unknown function  |sql_select.cc |10596 |
|end                  |0.000023|unknown function  |sql_select.cc | 10621|
|query end            |0.000005|unknown function  |sql_parse.cc  |  4922|
|freeing items        |0.009538|unknown function  |sql_parse.cc  |  5949|
|logging slow query   |0.000009|unknown function  |sql_parse.cc  |  1623|
|cleaning up          |0.000007|unknown function  |sql_parse.cc  |  1591|
+---------------------+--------+------------------+--------------+------+
```

Status 列的值可能会有误导性，因为这些值揭示内部信息，原本并不是用来公开访问的。强烈推荐你检查实际的 MySQL 源代码文件以及指定的行，这样有助于理解所有的描述信息。

8.3.1　减少 SQL 语句

有一些简单的技巧可以用来减少 SQL 语句：

- 删除内容重复的 SQL 语句。
- 删除重复执行的 SQL 语句。
- 删除不必要的 SQL 语句。
- 缓存 SQL 语句的返回结果。

1. 删除内容重复的 SQL 语句

指定函数或者进程的所有 SQL 语句的结果表明那些为了完成一个特定请求而重复执行的 SQL 语句。最好的做法是在开发环境中启用全面查询日志。开发人员有责任分析所有 SQL 语句以确保只执行必要的 SQL 语句。在应用程序中添加工具来报告 SQL 语句的总数量，同时提供动态查看所有 SQL 语句的调试工具，这些都会让你有更多信息来识别重复的语句。使用应用程序框架可能是造成不必要的重复语句的主要原因。

2. 删除重复执行的 SQL 语句

很多应用程序遇到过每次一行(Row At a Time，RAT)的处理方式，这个问题也被叫做 N+1 问题，导致这个问题的原因是外层循环为每一行都生成一个 SQL 语句。这会导致成百上千的重复执行的 SQL 语句。通常情况下，用一条 SQL 语句来实现每次一块(Chunk At a Time，CAT)的处理方式能够减少重复 SQL 语句。使用 SQL 的强项——集合处理能力可以有效地改进性能。请看下面的示例：

```
SELECT name FROM firms WHERE id=727;
SELECT name FROM firms WHERE id=758;
SELECT name FROM firms WHERE id=857;
SELECT name FROM firms WHERE id=740;
SELECT name FROM firms WHERE id=849;
SELECT name FROM firms WHERE id=839;
SELECT name FROM firms WHERE id=847;
SELECT name FROM firms WHERE id=867;
SELECT name FROM firms WHERE id=829;
SELECT name FROM firms WHERE id=812;
SELECT name FROM firms WHERE id=868;
SELECT name FROM firms WHERE id=723;
```

这个连续的行处理过程可以被重写成如下所示的形式：

```
SELECT id, name
FROM firms
WHERE id IN (723, 727, 740, 758, 812, 829, 839, 847,
849, 857, 867, 868);
```

使用分析功能可以看出这类 SQL 执行的总开销：

```
SET PROFILING=1;
SELECT . . .
SHOW PROFILES;
```

```
+----------+-----------+-------------------------------------
| Query_ID | Duration  | Query
+----------+-----------+-------------------------------------
|        1 | 0.00030400 | SELECT name FROM firms WHERE id=727
|        2 | 0.00014400 | SELECT name FROM firms WHERE id=758
|        3 | 0.00014300 | SELECT name FROM firms WHERE id=857
|        4 | 0.00014000 | SELECT name FROM firms WHERE id=740
|        5 | 0.00012300 | SELECT name FROM firms WHERE id=849
|        6 | 0.00012200 | SELECT name FROM firms WHERE id=839
|        7 | 0.00011600 | SELECT name FROM firms WHERE id=847
|        8 | 0.00014300 | SELECT name FROM firms WHERE id=867
|        9 | 0.00013900 | SELECT name FROM firms WHERE id=829
|       10 | 0.00014000 | SELECT name FROM firms WHERE id=812
|       11 | 0.00012800 | SELECT name FROM firms WHERE id=868
|       12 | 0.00011700 | SELECT name FROM firms WHERE id=723
|       13 | 0.00034200 | SELECT id, name FROM firms WHERE id IN (723 ...
+----------+-----------+-------------------------------------
SELECT 'Sum Individual Queries' AS txt,SUM(DURATION) AS total_time
FROM
INFORMATION_SCHEMA.PROFILING WHERE QUERY_ID BETWEEN 1 AND 12 UNION
SELECT 'Combined Query',SUM(DURATION) FROM
INFORMATION_SCHEMA.PROFILING WHERE
QUERY_ID = 13;
+-------------------------+-------------+
| txt                     | total_time  |
+-------------------------+-------------+
| Sum Individual Queries | 0.001311    |
| Combined Query         | 0.000342    |
+-------------------------+-------------+
```

使用应用程序框架可能是造成不必要的重复 SQL 语句的主要原因。

这个重复 SQL 语句的示例是一个更加常见的 N+1 场景的一部分，在这个场景中 SQL 开发人员不了解如何正确地使用表连接。这种做法涉及一个在外层记录集合上的循环，然后再次对循环中每条记录查询下一个表。这样做的结果就是合并的 SELECT

语句完全被消除。请看下面的示例：

```
SELECT  a.id, a.firm_id, a.title
FROM    article a
WHERE   a.type=2
AND   a.created > '2011-06-01';

// For  loop for all records SELECT id, name
        FROM firm
        WHERE id = :firm_id;
```

这个 SQL 语句的循环可以用下面的单条 SQL 语句代替：

```
SELECT a.id, a.firm_id, f.name, a.title
FROM article a
INNER JOIN firm f ON a.firm_id = f.id
WHERE a.type=2
AND a.created > '2011-06-01';
```

3. 删除不必要的 SQL 语句

随着时间的推移，应用程序会不断修改和增加功能，这可能产生不必要的 SQL 语句，例如：

- 不再需要的选择信息。
- 仅仅在给定函数的某些路径上用到的选择信息。
- 可以从之前的 SQL 语句中选择的信息。

应用程序的功能模块在随时间推移的过程中很容易遇到这些问题。截取并审查所有的 SQL 语句可能无法找出不必要的 SQL 语句。深入理解应用程序的需求以及到底应该把什么内容呈现给终端用户的过程对于找到可以改进的点来说是必不可少的。没有仔细分析那些执行 SQL 语句的代码，就无法了解哪些数据是实际需要的数据，比如传统的 SELECT*语句。

4. 缓存 SQL 结果

当普通数据的变化率相对较低时，缓存 SQL 结果能够为你的应用程序带来性能提升和对数据库服务器的可扩展性。

5. MySQL 缓存

MySQL 查询缓存能够为读操作频繁的环境带来性能提升，且在不需要其他应用程序开销的情况下就可以实现。下面就是一个使用分析功能来展示常规查询和随后被缓存的查询的执行时间和复杂性：

```
SET GLOBAL query_cache_size=1024*1024*16;
SET GLOBAL query_cache_type=1;
SET PROFILING=1;
SELECT name FROM firms WHERE id=727;
SELECT name FROM firms WHERE id=727;
SHOW PROFILES;
+----------+-------------+-----------------------------------+
| Query_ID | Duration    | Query                             |
+----------+-------------+-----------------------------------+
|        1 | 0.00030200  | SELECT name FROM firms WHERE id=727 |
|        2 | 0.00003300  | SELECT name FROM firms WHERE id=727 |
+----------+-------------+-----------------------------------+
```

可以看出，第二个查询快了 10 倍。可以使用分析功能来提供每一步的详细分析数据：

```
mysql> SHOW PROFILE FOR QUERY 1;
+------------------------------+--------------+
| Status                       | Duration     |
+------------------------------+--------------+
| starting                     | 0.000014     |
| checking query cache for query | 0.000033     |
| Opening tables               | 0.000031     |
```

```
| System lock                          | 0.000005    |
| Table lock                           | 0.000019    |
| init                                 | 0.000020    |
| optimizing                           | 0.000009    |
| statistics                           | 0.000043    |
| preparing                            | 0.000019    |
| executing                            | 0.000059    |
| Sending data                         | 0.000012    |
| end                                  | 0.000005    |
| query end                            | 0.000003    |
| freeing items                        | 0.000019    |
| storing result in query cache        | 0.000006    |
| logging slow query                   | 0.000002    |
| cleaning up                          | 0.000003    |
+--------------------------------------+-------------+
17 rows in set (0.00 sec)

mysql> SHOW PROFILE FOR QUERY 2;
+--------------------------------------+-------------+
| Status                               | Duration    |
+--------------------------------------+-------------+
| starting                             | 0.000008    |
| checking query cache for query       | 0.000005    |
| checking privileges on cached        | 0.000004    |
| sending cached result to clien       | 0.000010    |
| logging slow query                   | 0.000003    |
| cleaning up                          | 0.000003    |
+--------------------------------------+-------------+
6 rows in set (0.00 sec)
```

可以看出第二个查询减少了很多执行工作。

MySQL 查询缓存可能会让那些写操作多于读操作的系统性能退化。这是由查询缓存的粗略性导致的，对给定表的任意数据的改变都会导致所有使用那个表的缓存的 SQL 语句无效。读操作和写操作都很多的环境会导致没有从之前的使用中受益的 SELECT 语句失效。

6. 应用程序缓存

在应用程序级别添加缓存能够大幅度减少不必要的 SQL 语句的执行。对于应用程序缓存没有单独的推荐的技术。具体策略包括把特定 SQL 语句缓存在内存或本地文件中，或者可以在应用程序对象关系隐射(ORM)层引入延迟对象实例化机制。应用程序缓存的实现要能够反映应用程序的需求。例如，下面的伪代码可以代替一个简单的 SELECT 语句：

```
// Existing SELECT
SELECT FROM TABLE
RETURN VALUE
// Cached SELECT
GET FROM CACHE
IF EMPTY
   SELECT FROM TABLE
   PUT INTO CACHE
END
RTURN VALUE
```

一定不要忘记对表做的任何改动都要在缓存中有所反应。例如，下面的伪代码是必须的：

```
INSERT/UPDATE/DELETE FROM TABLE
REMOVE FROM CACHE
```

如果缓存系统允许周期缓存失效或者也可以启用缓存值的有效时间(TTL)值。

主流的 Memcached 产品(请参考 http://memcached.org/)被广泛用于在很多应用程序中。这个缓存基础架构能够很容易地和很多流行的编程语言整合在一起。

有很多种方式实现 MySQL/Memcached 数据的同步失效。其中一种实现方式就是用集成的 MySQL 的 Memcached 专用的用户

定义函数(UDF)结合数据库触发器来管理失效。还可以处理MySQL二进制日志事件，然后用缓存基础架构的同步实现来正确管理同步。

8.3.2 简化 SQL 语句

进一步减少 SQL 语句已无可能时，还可以简化 SQL 语句。简化语句时需要考虑到以下问题：

- 查询中所有的列都是必须的吗？
- 表的连接操作能被省去吗？
- 在给定的函数中，连接或 WHERE 条件限制对其他 SQL 语句是必要的吗？

1. 改进列

简化过程的一个重要要求就是按顺序截取指定执行函数中的所有 SQL 语句。使用采样过程不能找到所有可能的改进点。下面就是一个查询简化的示例：

```
mysql> SELECT fid, val, val
    -> FROM table1
    -> WHERE fid = X;
```

这个查询返回了 350 000 行数据，这些数据在系统重启时已经被应用程序服务器缓存了。对于这个查询，通过网络传输的结果集中 2/3 的数据都是冗余的。fid 列被传递过去作为这个 SQL 语句的条件限制，因此这个值对所有的行都是同样的。val 列也是不必要的重复。理想的 SQL 语句如下：

```
mysql> SELECT val
    -> FROM table1
    -> WHERE fid = X;
```

2. 改进连接操作

下面这个表连接简化的示例需要用到给定函数的所有 SQL 语句的信息。在这个过程中，下面的 SQL 语句用来获取行数：

```
mysql> SELECT /* Query 1 */ id FROM table1
    -> WHERE col1 = X
    -> AND col2 = Y;
```

之后在执行这个函数时，执行下面的语句会使用来自前一个 SQL 语句的 id 值：

```
mysql> SELECT /* Query 2 */ table2.val1, table2.val2,
table2.val3
    -> FROM table2 INNER JOIN table1 USING (id)
    -> WHERE table2.id = 9
    -> AND table1.col1 = X
    -> AND table1.col2 = Y
    -> AND table2.col1 = Z;
```

第二个 SQL 语句可以被简化成下面的语句：

```
mysql> SELECT /* Query 2 */ val1, val2, val3
    -> FROM table2
    -> WHERE table2.id = 9
    -> AND table2.col1 = Z
```

因为第一个查询在 id 列上添加了必要的条件限制(即 col1=X 且 col2=Y)，所以 table1 上的连接和条件语句都是多余的。去掉这个连接条件简化了 SQL 语句，同时消除了潜在的问题，只要以后某个语句被改变。

3. 重写子查询

MySQL 数据库支持子查询。在某些环境下，子查询的性能比

直接使用一般的表连接要慢很多。请看下面的示例：

```
SELECT id, label
FROM   code_opts
WHERE  code_id = (SELECT id FROM codes WHERE
typ='CATEGORIES')
ORDER BY seq
```

这条 SQL 语句能够简单地重写成如下的语句：

```
SELECT o.id, o.label
FROM  code_opts o INNER JOIN codes c ON o.code_id = c.id
WHERE c.typ='CATEGORIES'
ORDER BY o.seq
```

这些改变看起来很不起眼，但它能够改进复杂查询的性能。

4. 理解视图(views)所带来的影响

开发人员应该知道 SQL 语句中所使用的表的真正类别。如果这个对象实际上是一个视图，那么 SQL 优化的影响将会被视图定义的连接条件的复杂性掩盖。使用数据仓库(DW)和商务智能(BI)工具的一个常见问题就是在视图之上创建视图。对任何的视图定义，你将会获取到额外的信息，这些信息最终都是不会被 SQL 语句用到的。在 MySQL 中，涉及视图的复杂查询都可以通过使用必要的基础表来简单优化。

8.3.3 使用 MySQL 的复制功能

在多台服务器中可以使用 MySQL 的附属扩展节点(slaves)来增强读扩展性。这个原理在对读操作要求很高的应用程序中是很常见的。

尽管在 MySQL 附属扩展节点上执行 SELECT 语句能够分担

读操作负载，MySQL 的复制功能支持更多特性，可以使用这些特性来自定义附属节点上的模式设计。假设一条 SQL 语句在 MySQL 附属节点上执行成功，MySQL 复制功能将会接受这个改动。这包括对同样的表使用不同的存储引擎，以及在同样的表上使用不同的索引。MySQL 复制功能还支持更复杂的情况，例如支持不同类别的 MySQL 附属节点在不同的服务器上使用不同的索引。要特别注意的是，任何索引改变都不会删除主键唯一键这些用来保持数据完整性的索引。

8.4　本章小结

可以用很多种不同的方法来提高应用程序和数据库的性能。本章介绍了一些从架构和设计角度来优化 SQL 语句的方法。还有很多技术可以用来进一步优化。软件开发人员可以通过向数据架构师和数据库管理员学习并理解他们所掌握的技能来改进自己的技术实力。这些技能会让一个有经验的 SQL 开发人员从那些没有经验的开发人员中脱颖而出。

第 9 章

MySQL EXPLAIN 命令详解

MySQL 的 EXPLAIN 命令用于 SQL 语句的查询执行计划(QEP)。这条命令的输出结果能够让我们了解 MySQL 优化器是如何执行 SQL 语句的。这条命令并没有提供任何调整建议，但它能够提供

重要的信息帮助你做出调优决策。

在最后的这一章中，我们将详细介绍 EXPLAIN 命令的全部语法和选项。

9.1 语法

QEP 是通过 EXPLAIN 命令生成的，它的语法包含两项：

```
mysql> EXPLAIN [EXTENDED | PARTITIONS ]
    -> SELECT ...
```

或者

```
mysql> EXPLAIN table
```

MySQL 的 EXPLAIN 语法可以运行在 SELECT 语句或者特定表上。如果作用在表上，那么此命令等同于 DESC 表命令。UPDATE 和 DELETE 命令也需要进行性能改进，当这些命令不是直接在表的主码上运行时，为了确保最优化的索引使用率，需要把它们改写成 SELECT 语句(以便对它们执行 EXPLAIN 命令)。请看下面的示例：

```
UPDATE  table1
SET   col1 = X, col2 = Y
WHERE  id1 = 9
AND   dt >= '2010-01-01';
```

这个UPDATE语句可以被重写成为下面这样的SELECT语句：

```
SELECT  col1, col2
FROM    table1
WHERE   id1 = 9
AND   dt >= '2010-01-01';
```

MySQL 优化器是基于开销来工作的，它并不提供任何的 QEP 的位置。这意味着 QEP 是在每条 SQL 语句执行的时候动态地计算出来的。在 MySQL 存储过程中的 SQL 语句也是在每次执行时计算 QEP 的。存储过程缓存仅仅解析查询树。

9.2 各列详解

MySQL EXPLAIN 命令能够为 SQL 语句中的每个表生成以下信息：

```
mysql> EXPLAIN SELECT * FROM inventory
    -> WHERE item_id = 16102176\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: inventory
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 787338
        Extra: Using where
```

这个 QEP 显示没有使用任何索引(也就是全表扫描)并且处理了大量的行来满足查询。对同样一条 SELECT 语句，一个优化过的 QEP 如下所示：

```
mysql> EXPLAIN SELECT * FROM inventory
    -> WHERE item_id = 16102176\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
```

```
           table: inventory
            type: ref
   possible_keys: item_id
             key: item_id
         key_len: 4
             ref: const
            rows: 1
           Extra:
```

在这个 QEP 中，我们看到使用了一个索引，且估计只有一行数据将被获取。本章中将会详细介绍如何读取并解释这些信息。QEP 中每个行的所有列表如下所示：

- id
- select_type
- table
- partitions(这一列只有在 EXPLAIN PARTITIONS 语法中才会出现)
- possible_keys
- key
- key_len
- ref
- rows
- filtered(这一列只有在 EXPLAINED EXTENDED 语法中才会出现)
- Extra

这些列展示了 SELECT 语句对每一个表的 QEP。一个表可能和一个物理模式表或者在 SQL 执行时生成的内部临时表(例如从子查询或者合并操作会产生内部临时表)相关联。

可以参考 MySQL Reference Manual 获得更多信息：http://dev.mysql.com/doc/refman/5.5/en/explain-output.html。

下面我们将按照分析以及快速有效地理解 QEP 的重要程度的顺序来介绍这些列。

9.2.1　key

key 列指出优化器选择使用的索引。一般来说 SQL 查询中的每个表都仅使用一个索引。也存在索引合并的少数例外情况，如给定表上用到了两个或者更多索引。

下面是 QEP 中 key 列的示例：

```
key: item_id
key: NULL
key: first, last
```

SHOW CREATE TABLE <table>命令是最简单的查看表和索引列细节的方式。

和 key 列相关的列还包括 possible_keys、rows 以及 key_len。

9.2.2　rows

rows 列提供了试图分析所有存在于累计结果集中的行数目的 MySQL 优化器估计值。QEP 很容易描述这个很困难的统计量。查询中总的读操作数量是基于合并之前行的每一行的 rows 值的连续积累而得出的。这是一种嵌套行算法。

以连接两个表的 QEP 为例。通过 id=1 这个条件找到的第一行的 rows 值为 1，这等于对第一个表做了一次读操作。第二行是通过 id=2 找到的，rows 的值为 5。这等于有 5 次读操作符合当前 1 的积累量。参考两个表，读操作的总数目是 6。在另一个 QEP 中，第一 rows 的值是 5，第二 rows 的值是 1。这等于第一个表有

5 次读操作，对 5 个积累量中每个都有一个读操作。因此两个表总的读操作的次数是 10(5+5)次。

最好的估计值是 1，一般来说这种情况发生在当寻找的行在表中可以通过主键或者唯一键找到的时候。

在下面的 QEP 中，外面的嵌套循环可以通过 id=1 来找到，其估计的物理行数是 1。第二个循环处理了 10 行。

```
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: p
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: const
         rows: 1
        Extra:
*********************** 2. row ***********************
           id: 1
  select_type: SIMPLE
        table: c
         type: ref
possible_keys: parent_id
          key: parent_id
      key_len: 4
          ref: const
         rows: 10
        Extra:
```

可以使用 SHOW STATUS 命令来查看实际的行操作。这个命令可以提供最佳的确认物理行操作的方式。请看下面的示例：

```
mysql> SHOW SESSION STATUS LIKE 'Handler_read%';
+---------------------------------------+----------+
```

```
| Variable_name                          | Value    |
+----------------------------------------+----------+
| Handler_read_first                     | 0        |
| Handler_read_key                       | 5        |
| Handler_read_last                      | 0        |
| Handler_read_next                      | 10       |
| Handler_read_prev                      | 0        |
| Handler_read_rnd                       | 0        |
| Handler_read_rnd_next                  | 0        |
+----------------------------------------+----------+
```

在下一个 QEP 中，通过 id=1 找到的外层嵌套循环估计有 160 行。第二个循环估计有 1 行。

```
*********************** 1. row ***********************
           id: 1
  select_type: SIMPLE
        table: p
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 160
        Extra:
*********************** 2. row ***********************
           id: 1
  select_type: SIMPLE
        table: c
         type: ref
possible_keys: PRIMARY,parent_id
          key: parent_id
      key_len: 4
          ref: test.p.parent_id
         rows: 1
        Extra: Using where
```

通过 SHOW STATUS 命令可以查看实际的行操作，该命令表明物理读操作数量大幅增加。请看下面的示例：

```
mysql> SHOW SESSION STATUS LIKE 'Handler_read%';
+-----------------------------------+----------+
| Variable_name                     | Value    |
+-----------------------------------+----------+
| Handler_read_first                | 1        |
| Handler_read_key                  | 164      |
| Handler_read_last                 | 0        |
| Handler_read_next                 | 107      |
| Handler_read_prev                 | 0        |
| Handler_read_rnd                  | 0        |
| Handler_read_rnd_next             | 161      |
+-----------------------------------+----------+
```

相关的 QEP 列还包括 key 列。

9.2.3 possible_keys

possible_keys 列指出优化器为查询选定的索引。

一个会列出大量可能的索引(例如多于 3 个)的 QEP 意味着备选索引数量太多了，同时也可能提示存在一个无效的单列索引。可以用第 2 章详细介绍过的 SHOW INDEXES 命令来检查索引是否有效且是否具有合适的基数。

为查询确定 QEP 的速度也会影响到查询的性能。如果发现有大量的可能的索引，则意味着这些索引没有被使用到。

相关的 QEP 列还包括 key 列。

9.2.4 key_len

key_len 列定义了用于 SQL 语句的连接条件的键的长度。此列值对于确认索引的有效性以及多列索引中用到的列的数目很

重要。

此列的一些示例值如下所示：

```
key_len: 4   // INT NOT NULL
key_len: 5   // INT NULL
key_len: 30  // CHAR(30) NOT NULL
key_len: 32  // VARCHAR(30) NOT NULL
key_len: 92  // VARCHAR(30) NULL CHARSET=utf8
```

从这些示例中可以看出，是否可以为空、可变长度的列以及字符集都会影响到表索引的内部内存大小。

key_len 列的值只和用在连接和 WHERE 条件中的索引的列有关。索引中的其他列会在 ORDER BY 或者 GROUP BY 语句中被用到。

下面这个来自于著名的开源博客软件 WordPress 的表展示了如何以最佳方式使用带有定义好的表索引的 SQL 语句：

```
CREATE TABLE `wp_posts` (
  `ID` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
...
  `post_date` datetime NOT NULL DEFAULT '0000-00-00
  00:00:00',
  `post_status` varchar(20) NOT NULL DEFAULT
  'publish',
  `post_type` varchar(20) NOT NULL DEFAULT 'post',
...
  PRIMARY KEY (`ID`),
  KEY `type_status_date`
  (`post_type`,`post_status`,`post_date`,`ID`),
) DEFAULT CHARSET=utf8
```

这个表的索引包括 post_type、post_status、post_date 以及 ID 列。下面是一个演示索引列用法的 SQL 查询：

```
mysql> EXPLAIN SELECT ID, post_title
```

```
    -> FROM wp_posts
    -> WHERE post_type='post'
    -> AND post_date > '2010-06-01';
```

这个查询的 QEP 返回的 key_len 是 62。这说明只有 post_type 列上的索引用到了(因为(20×3)+2=62)。尽管查询在 WHERE 语句中使用了 post_type 和 post_date 列，但只有 post_type 部分被用到了。其他索引没有被使用的原因是 MySQL 只能使用定义索引的最左边部分。为了更好地利用这个索引，可以修改这个查询来调整索引的列。请看下面的示例：

```
mysql> EXPLAIN SELECT ID, post_title
    -> FROM wp_posts
    -> WHERE post_type='post'
    -> AND post_status='publish'
    -> AND post_date > '2010-06-01';
```

在 SELECT 查询的添加一个 post_status 列的限制条件后，QEP 显示 key_len 的值为 132，这意味着 post_type、post_status、post_date 三列(62+62+8，(20×3)+2，(20×3)+2，8)都被用到了。

此外，这个索引的主码列 ID 的定义是使用 MyISAM 存储索引的遗留痕迹。当使用 InnoDB 存储引擎时，在非主码索引中包含主码列是多余的，这可以从 key_len 的用法看出来。

相关的 QEP 列还包括带有 Using index 值的 Extra 列。

9.2.5 table

table 列是 EXPLAIN 命令输出结果中的一个单独行的唯一标识符。这个值可能是表名、表的别名或者一个为查询产生临时表的标识符，如派生表、子查询或集合。

下面是 QEP 中 table 列的一些示例：

```
table: item
table: <derivedN>
table: <unionN,M>
```

表中 N 和 M 的值参考了另一个符合 id 列值的 table 行。

相关的 QEP 列还有 select_type。

9.2.6　select_type

select_type 列提供了各种表示 table 列引用的使用方式的类型。最常见的值包括 SIMPLE、PRIMARY、DERIVED 和 UNION。其他可能的值还有 UNION RESULT、DEPENDENT SUBQUERY、DEPENDENT UNION、UNCACHEABLE UNION 以及 UNCACHEABLE QUERY。

1. SIMPLE

对于不包含子查询和其他复杂语法的简单查询，这是一个常见的类型。

2. PRIMARY

这是为更复杂的查询而创建的首要表(也就是最外层的表)。这个类型通常可以在 DERIVED 和 UNION 类型混合使用时见到。

3. DERIVED

当一个表不是一个物理表时，那么就被叫做 DERIVED。下面的 SQL 语句给出了一个 QEP 中 DERIVED select-type 类型的示例：

```
mysql> EXPLAIN SELECT MAX(id)
    -> FROM (SELECT id FROM users WHERE first = 'west') c;
```

4. DEPENDENT SUBQUERY

这个 select-type 值是为使用子查询而定义的。下面的 SQL 语句提供了这个值：

```
mysql> EXPLAIN SELECT p.*
    -> FROM parent p
    -> WHERE p.id NOT IN (SELECT c.parent_id FROM child
    c);
```

5. UNION

这是 UNION 语句其中的一个 SQL 元素。

6. UNION RESULT

这是一系列定义在 UNION 语句中的表的返回结果。当 select_type 为这个值时，经常可以看到 table 的值是<unionN,M>，这说明匹配的 id 行是这个集合的一部分。

下面的SQL产生了一个UNION和UNION RESULT select-type：

```
mysql> EXPLAIN SELECT p.* FROM parent p WHERE p.val
LIKE 'a%'
    -> UNION
    -> SELECT p.* FROM parent p WHERE p.id > 5;
```

9.2.7 partitions

partitions 列代表给定表所使用的分区。这一列只会在 EXPLAIN PARTITIONS 语句中出现。

9.2.8 Extra

Extra 列提供了有关不同种类的 MySQL 优化器路径的一系列额外信息。Extra 列可以包含多个值，可以有很多不同的取值，并且这些值还在随着 MySQL 新版本的发布而进一步增加。下面给出常用值的列表。你可以从下面的地址找到更全面的值的列表：http://dev.mysql.com/doc/refman/5.5/en/explain-output.html。

1. Using where

这个值表示查询使用了 where 语句来处理结果——例如执行全表扫描。如果也用到了索引，那么行的限制条件是通过获取必要的数据之后处理读缓冲区来实现的。

2. Using temporary

这个值表示使用了内部临时(基于内存的)表。一个查询可能用到多个临时表。有很多原因都会导致 MySQL 在执行查询期间创建临时表。两个常见的原因是在来自不同表的列上使用了 DISTINCT，或者使用了不同的 ORDER BY 和 GROUP BY 列。

想了解更多内容可以访问 http://forge.mysql.com/wiki/Overview_of_query_execution_and_use_of_temp_tables。

可以强制指定一个临时表使用基于磁盘的 MyISAM 存储引擎。这样做的原因主要有两个：

- 内部临时表占用的空间超过 min(tmp_table_size，max_heap_table_size)系统变量的限制
- 使用了 TEXT/BLOB 列

3. Using filesort

这是 ORDER BY 语句的结果。这可能是一个 CPU 密集型的过程。

可以通过选择合适的索引来改进性能，用索引来为查询结果排序。详细过程请参考第 4 章。

4. Using index

这个值重点强调了只需要使用索引就可以满足查询表的要求，不需要直接访问表数据。请参考第 5 章的详细示例来理解这个值。

5. Using join buffer

这个值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。

如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进性能。

6. Impossible where

这个值强调了 where 语句会导致没有符合条件的行。请看下面的示例：

```
mysql> EXPLAIN SELECT * FROM user WHERE 1=2;
```

7. Select tables optimized away

这个值意味着仅通过使用索引，优化器可能仅从聚合函数结果中返回一行。请看下面的示例：

```
mysql> EXPLAIN SELECT COUNT(*)
    -> FROM (SELECT id FROM users WHERE first = 'west') c
```

8. Distinct

这个值意味着 MySQL 在找到第一个匹配的行之后就会停止搜索其他行。

9. Index merges

当 MySQL 决定要在一个给定的表上使用超过一个索引的时候，就会出现以下格式中的一个，详细说明使用的索引以及合并的类型。

- Using sort_union(...)
- Using union(...)
- Using intersect(...)

9.2.9　id

id 列是在 QEP 中展示的表的连续引用。

9.2.10　ref

ref 列可以被用来标识那些用来进行索引比较的列或者常量。

9.2.11　filtered

filtered 列给出了一个百分比的值，这个百分比值和 rows 列的值一起使用，可以估计出那些将要和 QEP 中的前一个表进行连接的行的数目。前一个表就是指 id 列的值比当前表的 id 小的表。

这一列只有在 EXPLAIN EXTENDED 语句中才会出现。

9.2.12 type

type 列代表 QEP 中指定的表使用的连接方式。下面是最常用的几种连接方式：

- const 当这个表最多只有一行匹配的行时出现
- system 这是 const 的特例，当表只有一个 row 时会出现
- eq_ref 这个值表示有一行是为了每个之前确定的表而读取的
- ref 这个值表示所有具有匹配的索引值的行都被用到
- range 这个值表示所有符合一个给定范围值的索引行都被用到
- ALL 这个值表示需要一次全表扫描

其他类型的值还有 fulltext、ref_or_null、index_merge、unique_subquery、index_subquery 以及 index。

想了解更多信息可以访问 http://dev.mysql.com/doc/refman/5.5/en/explain-output.html。

9.3 解释 EXPLAIN 输出结果

理解你的应用程序(包括技术和实现可能性)和优化 SQL 语句同等重要。下面给出一个从父子关系中获取孤立的父辈记录的商业需求的例子。这个查询可以用三种不同的方式构造。尽管会产生相同的结果，但 QEP 会显示三种不同的路径。

```
mysql> EXPLAIN SELECT p.*
    -> FROM parent p
   -> WHERE p.id NOT IN (SELECT c.parent_id FROM child
   c)\G
********************** 1. row **********************
```

```
           id: 1
  select_type: PRIMARY
        table: p
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 160
        Extra: Using where
********************** 2. row **********************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: c
         type: index_subquery
possible_keys: parent_id
          key: parent_id
      key_len: 4
          ref: func
         rows: 1
        Extra: Using index
2 rows in set (0.00 sec)

mysql> EXPLAIN SELECT p.*
    -> FROM parent p
    -> LEFT JOIN child c ON p.id = c.parent_id
    -> WHERE c.child_id IS NULL\G
********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: p
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 160
        Extra:
```

```
*********************** 2. row ***********************
           id: 1
  select_type: SIMPLE
        table: c
         type: ref
possible_keys: parent_id
          key: parent_id
      key_len: 4
          ref: test.p.id
         rows: 1
        Extra: Using where; Using index; Not exists
2 rows in set (0.00 sec)

mysql> EXPLAIN SELECT p.*
    -> FROM parent p
    -> WHERE NOT EXISTS
    -> SELECT parent_id FROM child c WHERE c.parent_id
    = p.id)\G
*********************** 1. row ***********************
           id: 1
  select_type: PRIMARY
        table: p
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 160
        Extra: Using where
*********************** 2. row ***********************
           id: 2
  select_type: DEPENDENT SUBQUERY

        table: c
         type: ref
possible_keys: parent_id
          key: parent_id
      key_len: 4
```

```
          ref: test.p.id
         rows: 1
        Extra: Using index
2 rows in set (0.00 sec)
```

哪种方式是最好的？随着数据的增长另一种 QEP 会更高效吗？这本书的目的就是介绍那些用来优化你的 SQL 语句的工具和方法。

9.4　本章小结

使用 EXPLAIN 命令阅读并理解 MySQL 查询执行计划(QEP)是获取优化 SQL 语句的信息的不可或缺的途径之一。把这个命令和本书中介绍的其他来源的信息相结合就能够确保你掌握足够的数据来做出合理的选择。