ORACLE

Mc Graw Hill Education

Effective MySQL: Optimizing SQL Statements

MySQL

Effective MySQL 之SQL语句最优化

性能改进的实用知识

[美] Ronald Bradford 著
李雪锋 译

清华大学出版社

Effective MySQL 之 SQL 语句最优化

[美] Ronald Bradford 著

李雪锋 译

清华大学出版社

北 京

Ronald Bradford
Effective MySQL: Optimizing SQL Statements
EISBN：978-0-07-178279-1

图书在版编目(CIP)数据

Effective MySQL 之 SQL 语句最优化/(美) 布拉德福(Bradford，R.) 著；李雪锋 译. —北京：清华大学出版社，2013.1
书名原文：Effective MySQL：Optimizing SQL Statements
ISBN 978-7-302-30429-6
Ⅰ. ①E… Ⅱ. ①布… ②李… Ⅲ. ①关系数据库—数据库管理系统 Ⅳ. ①TP311.138

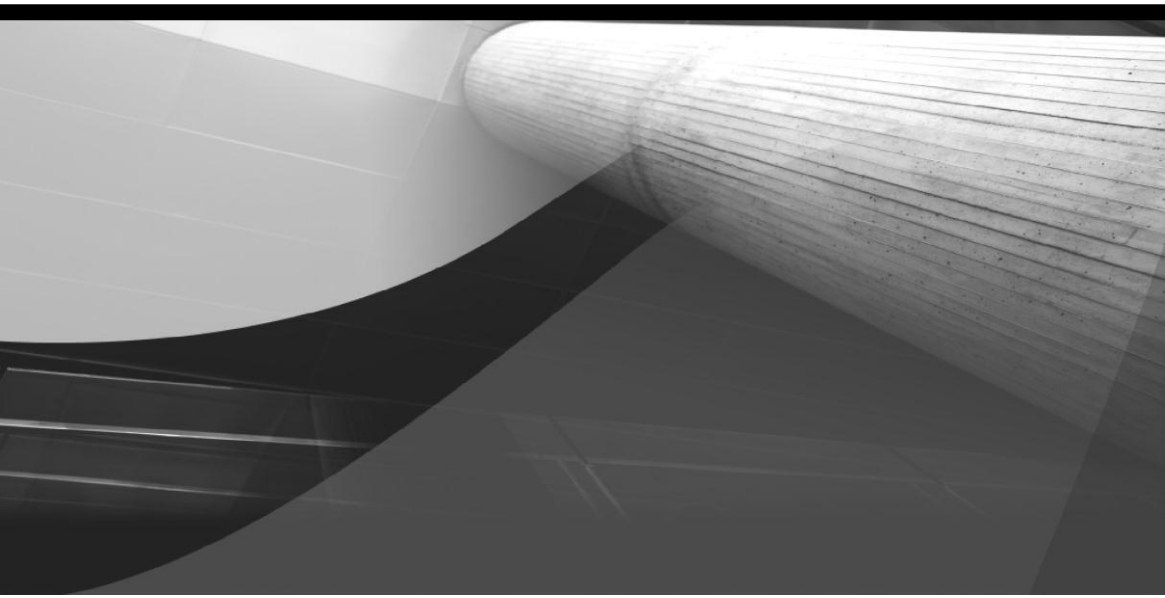
中国版本图书馆 CIP 数据核字(2012)第 244771 号

责任编辑：王　军　于　平
封面设计：牛艳敏
责任校对：邱晓玉
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

装 订 者：
经　　销：全国新华书店
开　　本：148mm×210mm　　印　　张：7　　字　　数：172 千字
版　　次：2013 年 1 月第 1 版　　印　　次：2013 年 1 月第 1 次印刷
印　　数：1～3500
定　　价：29.00 元

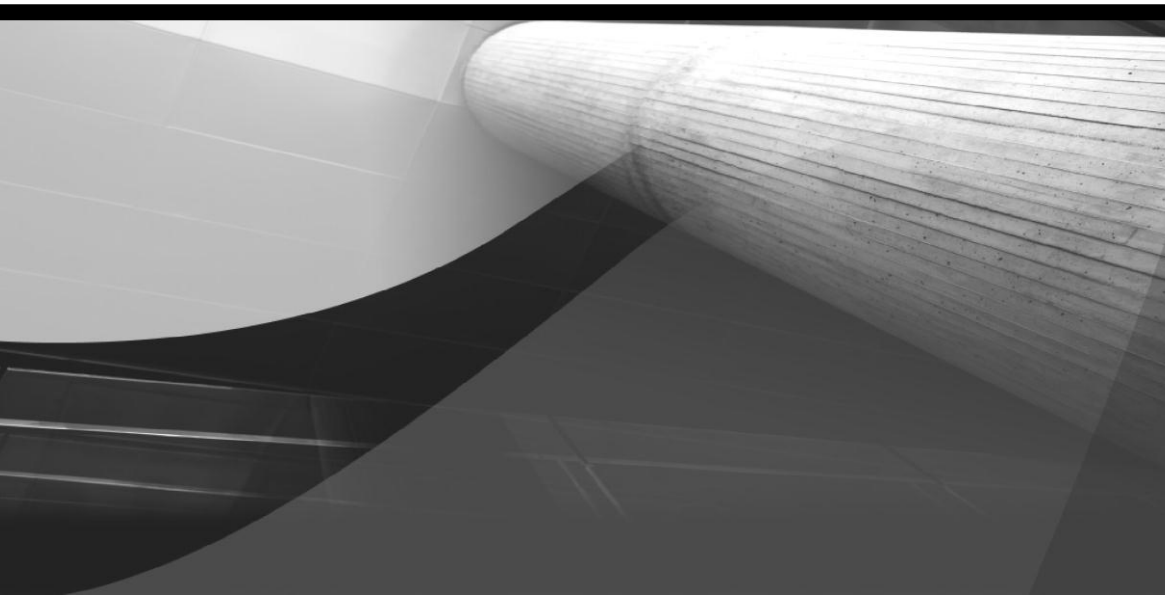
产品编号：



作 者 简 介

Ronald Bradford 是一位在关系型数据库领域拥有 20 多年丰富经验的专家。他拥有深厚的专业背景以及 10 年以上 Ingres 和 Oracle 系统的工作知识，在过去 12 年中他致力于 MySQL——世界上最流行的开源数据库的发展。他曾在 2009 年被提名为 MySQL 社区成员和 2010 年的 Oracle ACE Director，其咨询领域的专家背景以及多次在国际会议上的发言也为他赢得了广泛的国际知名度。他还是 Planet MySQL(2010)最受欢迎的个人 MySQL 技术博客作者，并且是清华大学出版社引进并出版的《PHP+MySQL 专家编程》一书的作者之一。

MySQL 在被 Oracle 公司收购之后成为主要的数据库解决方案，并获得了更多社区推广的机会。Ronald 是世界范围的 Oracle 用户组中最受欢迎的 MySQL 的受邀发言人，该用户组的范围遍及北美、南美、欧洲以及亚太地区。



技术编辑简介

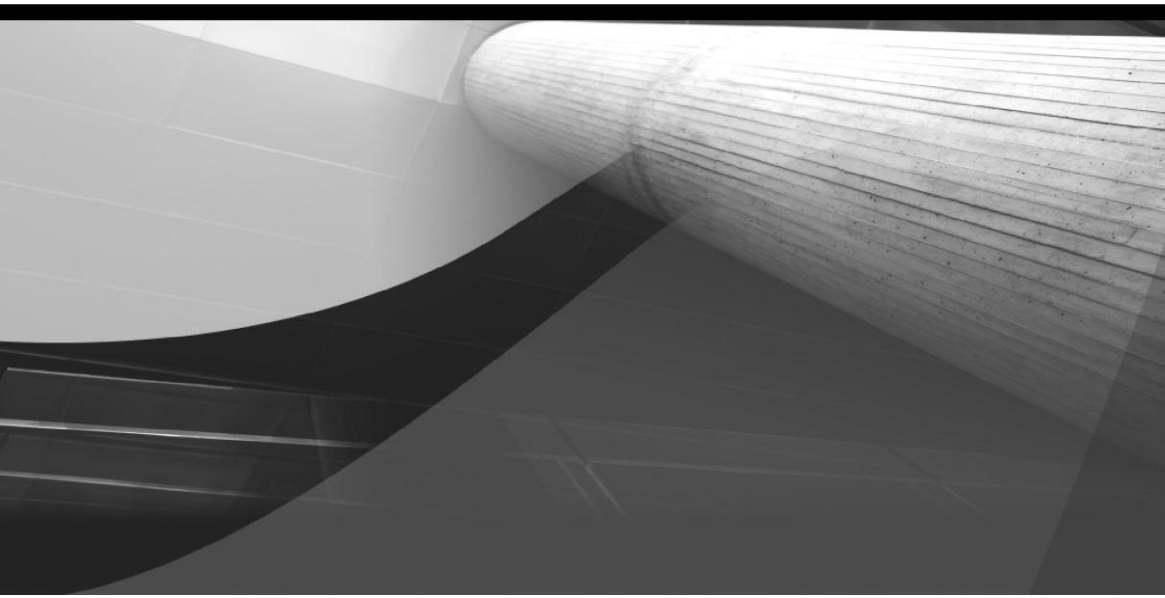
Jay Pipes 是 Rackspace Cloud 公司的软件工程师，他曾参与过多个和云计算基础设施以及数据库相关的项目。在此之前，他曾是 Sun Microsystems 公司的软件工程师和 MySQL 北美社区的社区关系经理。作为 *Pro MySQL*(Apress,2005)一书的作者之一，他也曾为 *Linux Magazine* 撰写文章，并经常帮助开发人员发现最有效地使用 MySQL 和其他软件的方法。

他曾在 MySQL Users Conference、RedHat Summit、NY PHP Conference、ZendCon、php-tek、OSCON 和 Ohio LinuxFest 等会议上做过有关性能优化的讲座。目前他和他的妻子 Julie 以及两只

宠物狗住在俄亥俄州的哥伦布。

Hans Forbrich 是一位独立的软件咨询专家，他擅长的领域是 Oracle 数据库性能以及 Linux 环境。他早年从卡尔加里大学获得电子工程学的学位，并从 20 世纪 70 年代就开始使用数据库和 Linux 系统，他为系统编写代码并提交补丁，还参与了很多系统管理、性能调优以及系统架构方面的工作。除了从事 Oracle 和 Linux 的咨询工作之外，他还是 Oracle 大学的客座讲师。作为 Oracle ACE Director，Hans 也经常在美国和其他一些国家的会议上发言。

Darren Cassar 是 Lithium Technologies 公司的高级数据库管理员。他拥有马耳他大学的计算机和通信工程的学位，并以系统管理员作为最初的职业生涯，之后他在马耳他、伦敦、纽约以及旧金山等多地从事数据库管理方面的工作。Darren 是 Securich(一个开源的 MySQL 的安全插件)的作者，他曾在美国和欧洲的多次会议上展示这个项目。



译　者　序

MySQL 数据库由于性能高、成本低、可靠性好等优点，已经成为最流行的开源关系型数据库产品，广泛地被使用在互联网上的中小型网站中。并且随着 MySQL 的不断成熟，以及一些企业特性的加入，它也逐渐被用于更大规模的网站和应用系统中，如维基百科、Google 和 facebook 等。目前 Internet 上流行的网站构架方式 LAMP 和 LNMP，其中的“M”都是 MySQL，这些都是免费或开放源码软件(FLOSS)，使用这种方式可以以极低的成本构建网站系统。因此 MySQL 越来越受到企业和个人开发者的喜爱。

MySQL 是一个非常容易上手的数据库产品，很多开发者经过短时间的学习就可以使用它。但是有经验的数据库应用程序开发者写出来的 SQL 查询语句和新手写出的 SQL 语句在执行性能方面有巨大的差距，最根本的原因就在于有经验的开发者善于运用 MySQL 的索引以及各种性能分析调优工具优化自己的查询。我本人就是从事 DB2 数据库管理应用开发和性能监控工具的开发工作，深刻地认识到数据库查询性能优化在数据库应用程序开发中是至关重要的。然而 MySQL 数据库在管理和性能调优方面还没有公认的主流工具可供开发者使用。因此 MySQL 的用户只能依靠自己在数据库性能优化领域的经验，使用 MySQL 提供的原生工具对自己的查询进行优化调整。

《Effective MySQL 之 SQL 语句最优化》是一本不可多得的参考指南。该书虽然非常轻薄，却是从最实用的角度帮助开发者。该书首先介绍了最为常用的性能调优技巧；然后逐步介绍 MySQL 的查询执行计划和索引机制，介绍如何在数据库应用程序的开发和生产环境中分析、定位并且解决性能问题；然后还介绍了如何从管理配置和参数调整的角度优化应用程序。诚然，SQL 语句性能优化、数据库管理配置调整都是需要长期经验积累，不是仅仅学习一些技巧就可以做到的。本书通过介绍最实用的技巧来引导初学者在实践开发中开始自己的性能优化之旅，为开发者以后成为优秀的数据库应用程序开发者铺平了道路。除了入门技巧之外，书中还会给出很多性能优化方面的忠告建议，这些都是作者汇集很多数据库大师级人物的智慧而得出的经验之谈，仔细研究这些建议会让读者受益匪浅。

这次翻译技术类书籍，最大的体会就是，翻译和自己阅读外文技术书籍差别非常大。虽然我阅读外文技术类书籍没有什么障碍，但开始翻译工作之后才体会到，想要准确地表达原作者的意思，绝不是自己读懂那么简单。况且本书凝聚了很多数据库大师的独到见解，即便是专门从事数据库管理工具开发的我，也没有

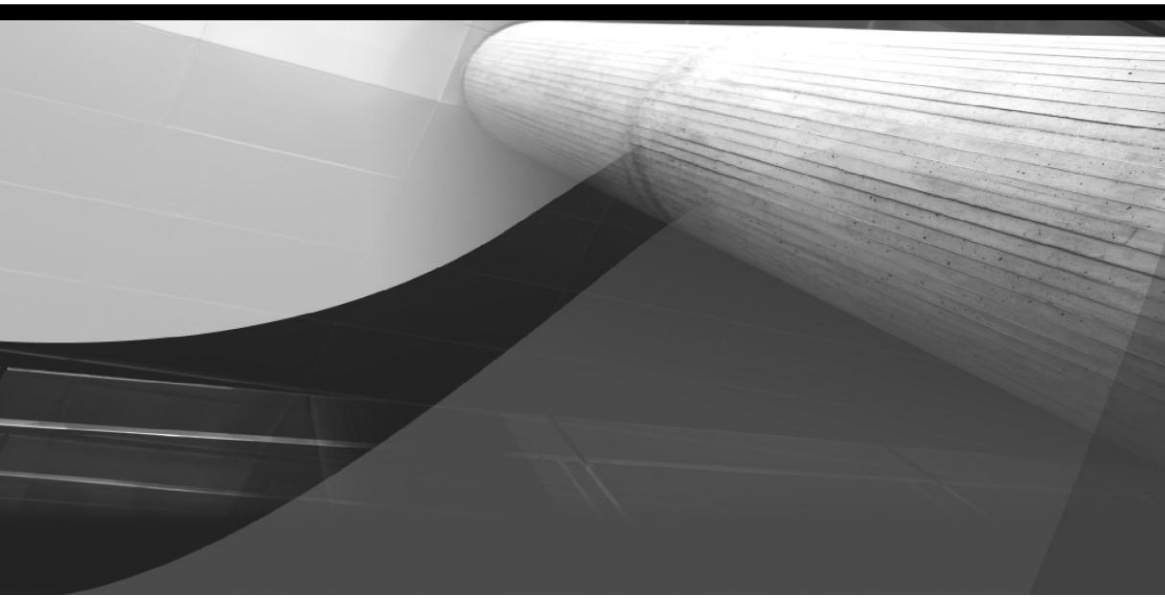
完全准确地理解并表达所有内容的自信。为了尽可能保证翻译内容的准确完整性，我在翻译过程中，曾多次就不确定的部分征求我们产品开发团队同事的意见。在此我想对我们团队的全体同事表达我最真挚的谢意。

我能够成为这本书的译者，要特别感谢清华大学出版社的李阳编辑。作为初次接触技术类书籍翻译工作的新人，李阳编辑给我提供了很多学习参考资料，并为我的译稿提出宝贵的修改意见。

我还要感谢从事数据库建模工具开发的同事陶佰明，他在我初稿完成后帮助我审稿，并提出很多修改意见。

由于译者水平有限，翻译工作中可能会有不准确的内容，如果读者在阅读过程中发现了失误和遗漏之处，希望能够多多包涵，并欢迎批评指正。敬请广大读者提供反馈意见，读者可以将意见发到 wkservice@163.com，我会仔细查阅读者发来的每一封邮件，以求进一步提高今后译著的质量。

译者



致　　谢

为了 MySQL 文化的过去、现在和未来！

献给 MySQL 和 Oracle 社区的所有人：你们不仅仅是我的同事，更是我最好的朋友！

如果没有来自身边所有人的帮助、支持以及贡献，我是不可能写出这本书的。我最希望感谢的人是我的妻子 Cindy，她为支持这本书的创作付出的时间和努力是不可或缺的。

我所参与的 Oracle ACE 项目为我提供了很多新的机会。和几位顶尖 Oracle 专家的合作也促使我和工业界的同僚开展了在相关技术领域崭新且很有意义的讨论。作为很多 Oracle 用户组的受邀

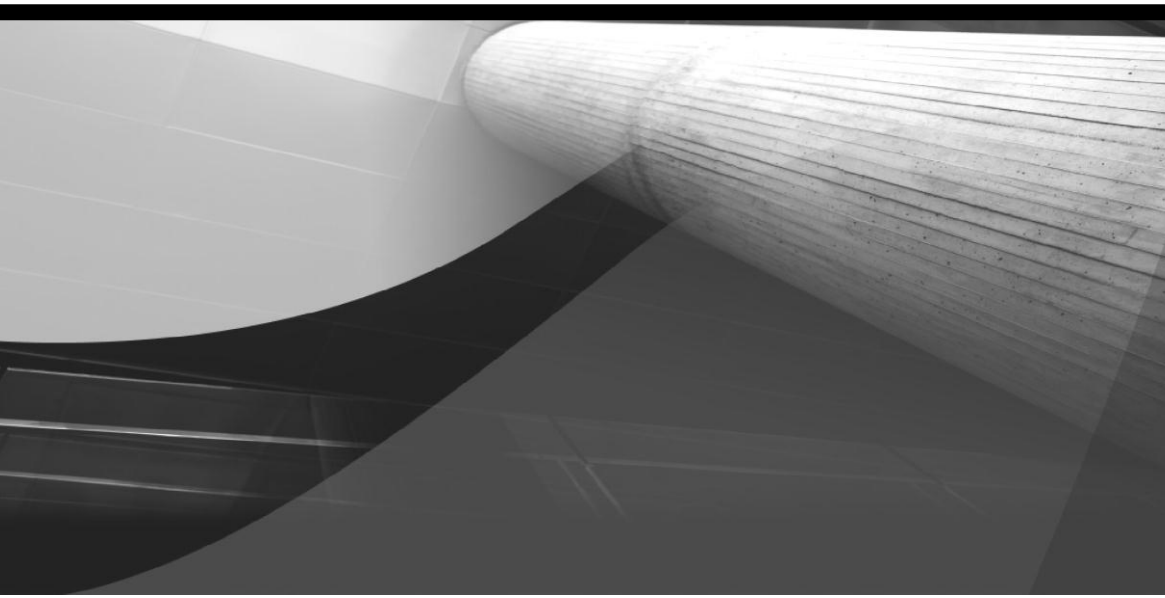
发言人，我也很荣幸能和更多听众分享我在这方面的知识和经验。我的目标是和新一代的开发人员共享信息、教导并促进他们，以确保他们能够使用各种可行的工具和技术，用最佳的方式达成他们的目标。我特殊希望这本书的西班牙语版能够帮助更多拉丁美洲的朋友使用、学习和理解 MySQL。

我要感谢在 McGraw-Hill 的小组成员，尤其是 Paul 和 Stephanie。随着我们的关系越来越密切，在 MySQL 这个主题中一起共事的这段时间让我们在学到的技巧和天赋之间找到了一个理想的平衡点并创造出杰出的成果。

我的老朋友 Jay，你的建议一直是金玉良言。你对 MySQL 内部原理的掌握以及实际用户的使用经验保证了本书内容的正确性。你的团队精神从很多年前我第一次参加 MySQL 用户会议开始就一直鼓舞我。

Darren，你作为 MySQL 数据库管理员的日常工作经验给我提供了非常重要的用户面临的业务和技术问题的观点，这些和快节奏的咨询经验带来的观点是截然不同的。

我尤其想要感谢 Hans。我们认识不到一年，你的知识背景并不是 MySQL，然而你深厚的 Oracle 经验背景带来的洞察力帮助我让本书不仅能够面向 MySQL 读者，同时也是为了有其他技术背景然而还想学习如何更高效地使用 MySQL 的工作人员。我非常感谢你作为两次成功的 Oracle 技术网络(OTN)拉丁美洲旅行的先驱者对我的友谊和建议。



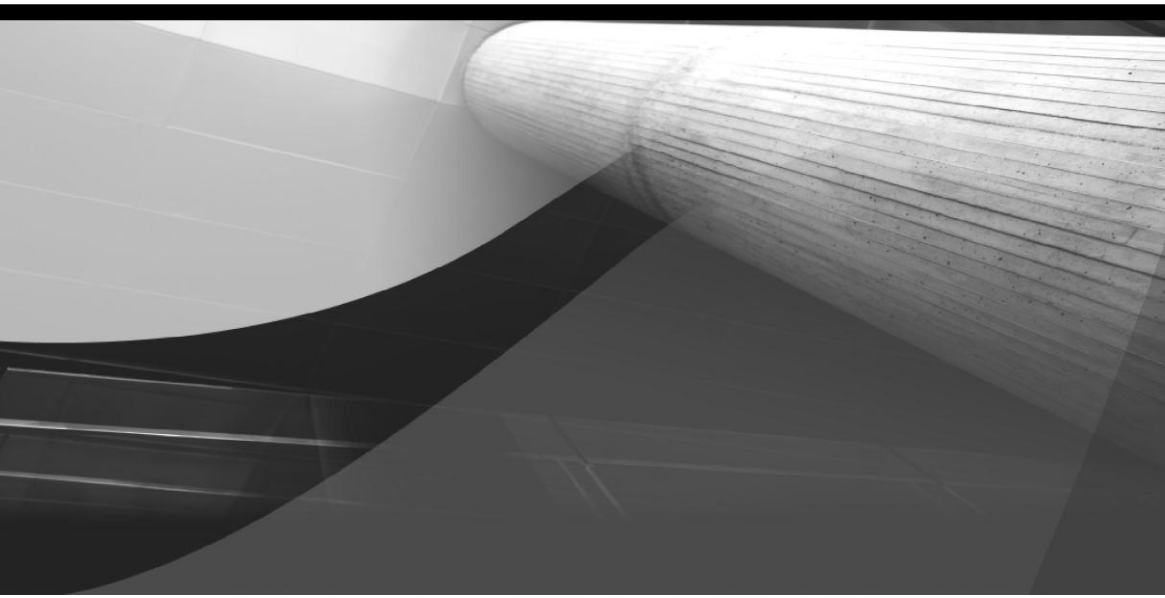
前　言

作为一名日常 DBA，最常重复的任务就是在生产环境中检查和优化运行的 SQL 语句。在 MySQL 软件安装、配置以及正常运行之后，监控数据库的性能问题就成为一项经常重复的工作。了解如何正确地截取有问题的 SQL 语句以及检查并做适当的调整，这已经成为一个专业 DBA 的必备技能。尽管 MySQL 是一个关系型数据库管理系统(RDBMS)，有 Oracle 或者 SQL Server 背景的有经验的数据库管理员还是需要学习如何在 MySQL 术语中正确地应用 SQL 查询分析理论，而这需要阅读并理解查询执行计划(QEP)，

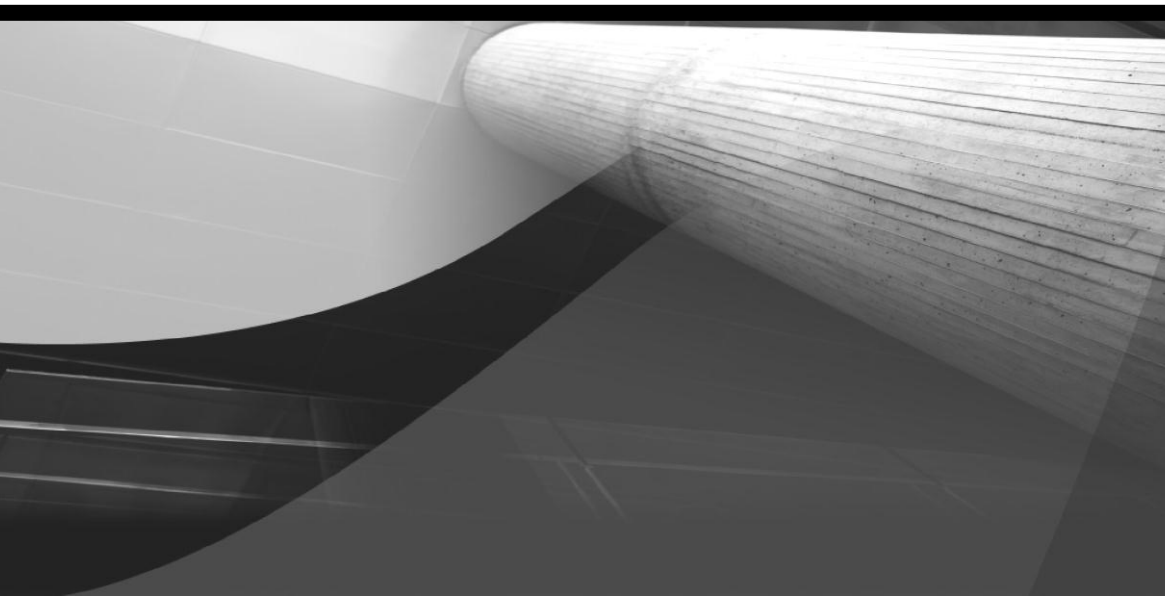
了解 MySQL 优化器功能的限制和不足，还要理解不同的 MySQL 存储引擎是如何改变索引的高效使用方式的。

SQL 语句的优化不仅仅是数据库管理员的责任。本书将帮助读者理解 MySQL 索引和存储引擎是如何运行的，这对一个由数据架构师设计的优化过的数据库来说是更重要的实现考虑因素。软件开发人员将能够截取和分析所有 SQL 语句，以此来确保性能瓶颈能够在开发早期被发现然后由合适的人去处理。

优化 SQL 语句是改进性能和扩展性的一个关键部分。



目　　录



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/Music-Brainz_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

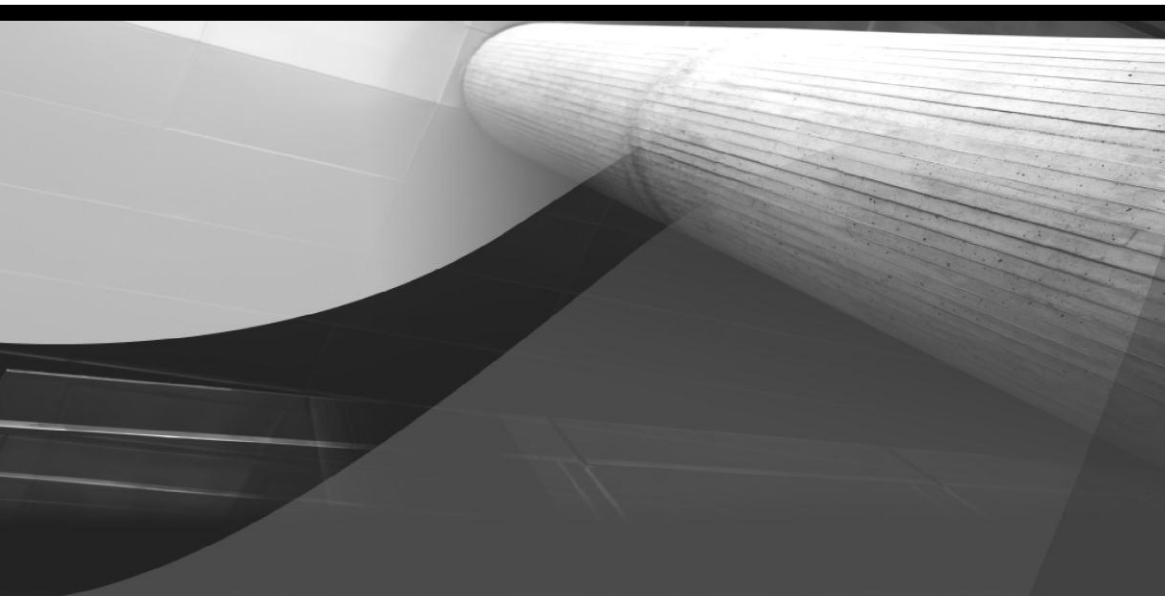
尽管创建索引的原则很简单，但是要选择在合适的列上创建索引，并且要判断在当前情况下添加索引带来的读数据性能的改进是否超过了对写数据性能的影响则不是那么容易。不管你的 SQL 语句有多么复杂，你都可以将一个 SQL 语句简化成为包含用到的表的子集的更小的工作单元，然后分别分析这些小型组件，最后再一点一点重新构造成为你的 SQL 语句。

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_
  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。

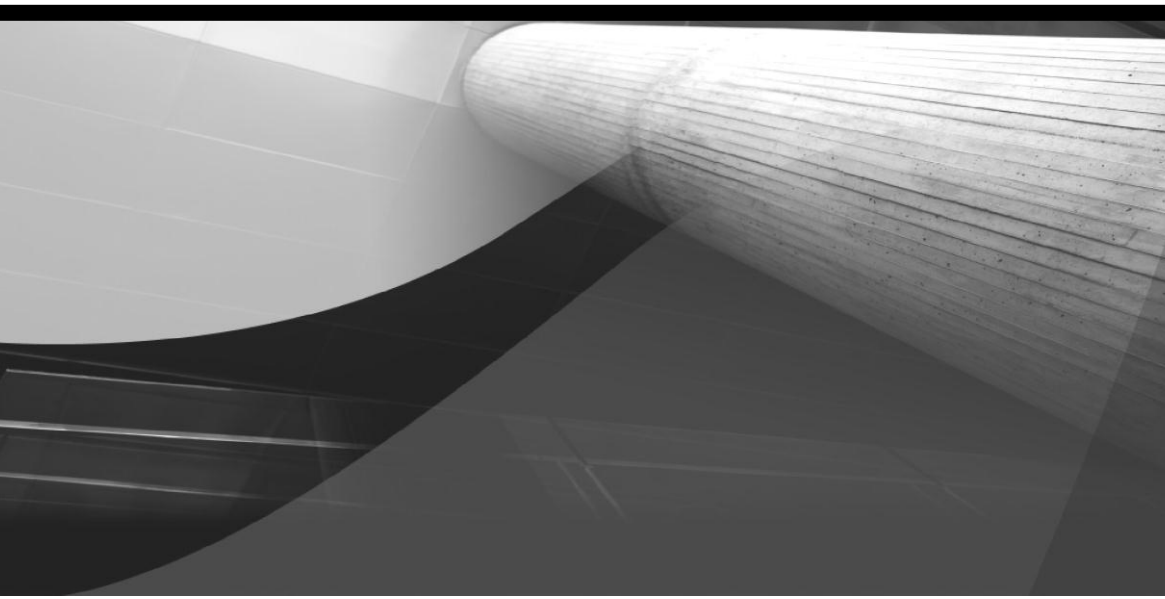
5.2 本章小结

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 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。