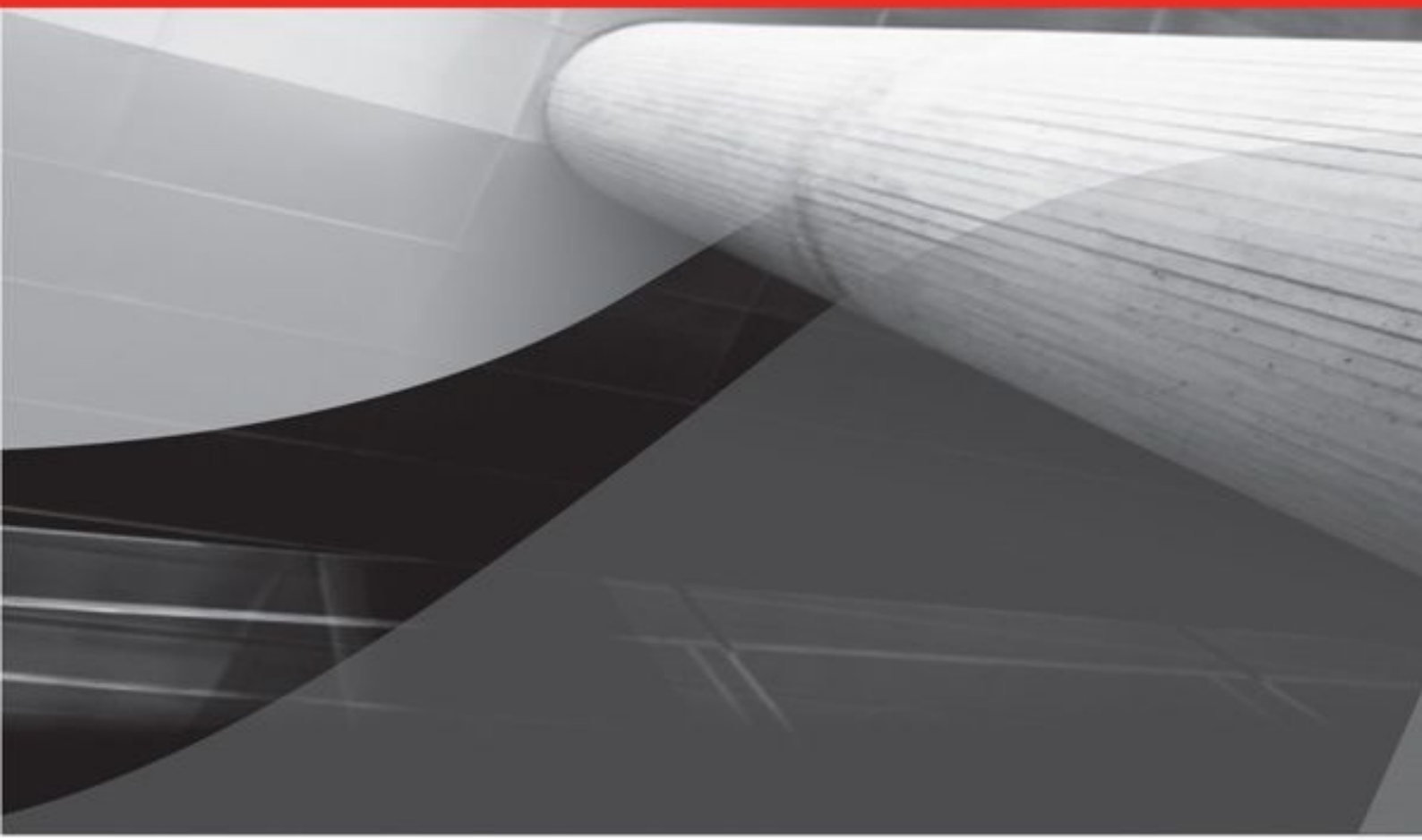
ORACLE



Effective MySQL: Optimizing SQL Statements

MySQL

Practical Knowledge for Performance Improvement

Ronald Bradford
Oracle ACE Director

ORIGINAL • AUTHENTIC
Oracle Press
ONLY FROM McGRAW-HILL

Oracle Press™

Effective MySQL

Optimizing SQL Statements

Ronald Bradford

New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan
Seoul Singapore Sydney Toronto

ISBN 978-0-07-178280-7
MHID 0-07-178280-X

The material in this eBook also appears in the print version of this title: ISBN 978-0-07-178279-1, MHID 0-07-178279-6.

Effective MySQL Optimizing SQL Statements

TERMS OF USE

About the Author

Ronald Bradford is an industry expert with more than 20 years of experience in the relational database field. With a professional background and more than a decade of working knowledge with Ingres and Oracle, Ronald has focused for the past 12 years in MySQL, the world's most popular open source database. With international recognition, including being named an Oracle ACE Director (2010) and MySQL Community Member of the Year (2009), Ronald combines his extensive consulting expertise with many public speaking events at conferences worldwide. Ronald is recognized as the all-time top individual MySQL blog contributor at Planet MySQL (2010) and is the published co-author of *Expert PHP and MySQL* (Wrox, 2010).

The Oracle Corporation acquisition has highlighted MySQL as a primary database offering and has enabled additional community evangelism opportunities. Ronald is the top invited MySQL speaker at Oracle User Groups worldwide, including North and South America, Europe, and the Asia Pacific region.

About the Technical Editors

Jay Pipes is a software engineer working for the Rackspace Cloud on various projects revolving around cloud infrastructure and databases. Previously, he was a software engineer at Sun Microsystems and the North American Community Relations Manager at MySQL. Co-author of *Pro MySQL* (Apress, 2005), Jay has also written articles for *Linux Magazine* and regularly assists developers in identifying how to make the most effective use of MySQL and other software.

He has given sessions on performance tuning at the MySQL Users Conference, RedHat Summit, NY PHP Conference, ZendCon, php-tek, OSCON, and Ohio LinuxFest, amongst others. He lives in Columbus, Ohio, with his wife, Julie, and his two dogs.

Hans Forbrich is an independent software consultant specializing in Oracle database performance and Linux environments. He received his Electrical Engineering degree from the University of Calgary many years ago and has been using, programming, fixing, administering, tuning, and architecting systems since the 1970s. Aside from being a Linux and Oracle consultant, Hans is also an Oracle University partner instructor. As an Oracle ACE Director, Hans regularly speaks at conferences in the Americas and abroad.

Darren Cassar is a senior MySQL Database Administrator at Lithium Technologies. He holds a Computer and Communications Engineering degree from the University of Malta and started his career doing systems administration in Malta, later moving on into database administration in Malta, London, New York, and San Francisco. Darren is the author of Securich, an open source security plugin for MySQL, a subject which he has presented at several conferences in both the United States and Europe.

CONTENTS

Acknoledgements

Introduction

1 The Five Minute DBA

Identifying Performance Problems

Finding a Slow SQL Statement

Confirming the Slow Query

Optimizing Your Query

What You Should Not Do

Confirm Your Optimization

The Correct Approach

An Alternative Solution

Conclusion

2 The Essential Analysis Commands

EXPLAIN

EXPLAIN PARTITIONS

EXPLAIN EXTENDED

SHOW CREATE TABLE

SHOW INDEXES

SHOW TABLE STATUS

SHOW STATUS

SHOW VARIABLES

INFORMATION_SCHEMA

Conclusion

3 Understanding MySQL Indexes

Example Tables

MySQL Index Usages

Data Integrity

Optimizing Data Access

Table Joins

Sorting Results
Aggregation
About Storage Engines
Index Terminology
MySQL Index Types
Index Data Structure Theory
MySQL Implementation
MySQL Partitioning
Conclusion
4 Creating MySQL Indexes
Example Tables
Existing Indexes
Single Column Indexes
Syntax
Restricting Rows with an Index
Joining Tables with an Index
Understanding Index Cardinality
Using Indexes for Pattern Matching
Selecting a Unique Row
Ordering Results
Multi Column Indexes
Determining Which Index to Use
Multi Column Index Syntax
Providing a Better Index
Many Column Indexes
Combining WHERE and ORDER BY
MySQL Optimizer Features
Query Hints
Complicated Queries
The Impact of Adding Indexes
DML Impact
DDL Impact

Disk Space Impact
MySQL Limitations
Cost Based Optimizer
QEP Pinning
Index Statistics
Function Based Indexes
Multiple Indexes per Table
Conclusion
5 Creating Better MySQL Indexes
Better Indexes
Covering Index
Storage Engine Implications
Partial Index
Conclusion
6 MySQL Configuration Options
Memory Related Variables
key_buffer_size
Named Key Buffers
innodb_buffer_pool_size
innodb_additional_mem_pool_size
query_cache_size
max_heap_table_size
tmp_table_size
join_buffer_size
sort_buffer_size
read_buffer_size
read_rnd_buffer_size
Instrumentation Related Variables
slow_query_log
slow_query_log_file
general_log
general_log_file

long_query_time
log_output
Profiling
Other Optimization Variables
optimizer_switch
default_storage_engine
max_allowed_packet
sql_mode
innodb_strict_mode
Other Variables
Conclusion
7 The SQL Lifecycle
Capture Statements
General Query Log
Slow Query Log
Binary Log
Processlist
Engine Status
MySQL Connectors
Application Code
INFORMATION_SCHEMA
PERFORMANCE_SCHEMA
SQL Statement Statistics Plugin
MySQL Proxy
TCP/IP
Identify Problematic Statements
Slow Query Log Analysis
TCP/IP Analysis
Confirm Statement Operation
Environment
Timing
Analyze Statements

Optimize Statements
Verify the Results
Conclusion
8 Hidden Performance Tips
Index Management Optimizations
Combining Your DDL
Removing Duplicate Indexes
Removing Unused Indexes
Monitoring Ineffective Indexes
Index Column Improvements
Data Types
Column Types
Other SQL Optimizations
Eliminating SQL Statements
Simplifying SQL Statements
Using MySQL Replication
Conclusion
9 Explaining the MySQL EXPLAIN
Syntax
Explain Columns
key
rows
possible_keys
key_len
table
select_type
partitions
Extra
id
ref
filtered
type

Interpreting EXPLAIN Output

Conclusion

For MySQL Culture, past, present, and future. To many in the MySQL and Oracle community: you are more than colleagues; you are great friends.

ACKNOWLEDGMENTS

Writing a book is not possible without the help, support, and sacrifice of many around you. My wife, Cindy, has been an essential support for all the time and effort it took to create this book.

My involvement with the Oracle ACE program has provided many new opportunities. The interaction with several top Oracle experts has enabled valuable new discussion with industry peers in related technologies. As an invited speaker to many different Oracle User Groups, I have had the pleasure of being able to share my vast knowledge and experience with new audiences. My goal is to share, promote, and educate the next generation in order to ensure the best ways of achieving success with the tools and technologies available. I especially hope that the Spanish translation of this book will help in the use, adoption, and appreciation of MySQL for many new friends in Latin America.

I would like to thank the team at McGraw-Hill, especially Paul and Stephanie. As with any new relationship, working with this title over time has enabled us to find an ideal balance of skills and talents to produce an excellent result.

Finally, to three friends who have been a barometer to what is detailed in this book. Jay, Darren, and Hans, you have helped to agree with my content, to correct, and even educate me at times.

To my longtime friend, Jay, your counsel is, as always, hugely valued. Your knowledge of MySQL internals and of practical user experiences ensures I got it right. Your community spirit has been an encouragement since my first MySQL Users Conference many years ago.

To Darren, your day-to-day role as a MySQL database administrator for a single employer has given me a different and important view to business and technical problems faced by users than the fast paced consulting experience.

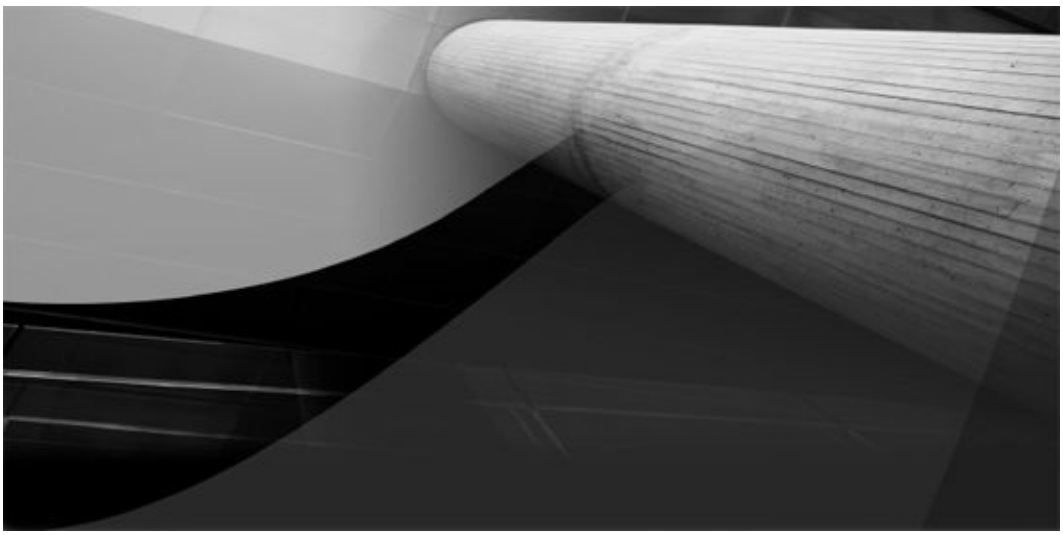
I especially wish to thank Hans. We met less than one year ago, and your background is not with MySQL; however, your insight from an experienced Oracle background has helped ensure that this title is not just for the MySQL reader, but for any skilled resource that can learn to use MySQL more effectively. As veterans of two successful OTN Latin America tours, your friendship and advice is greatly appreciated.

INTRODUCTION

The most common repetitive task for an operational DBA is the review and optimization of running SQL statements in a production environment. After the MySQL software is installed, configured, and operating correctly, monitoring the database for performance issues is a regular reoccurring task. Knowing how to capture problematic SQL statements correctly, review and tune appropriately are critical skills for a proficient resource. Although MySQL is an RDBMS, experienced Database Administrators from an Oracle or SQL Server background will need to learn how to apply the theory of SQL query analysis correctly in MySQL terms by reading and understanding the Query Execution Plan (QEP), knowing the limitations with the MySQL optimizer functionality, and understanding how different MySQL storage engines change how indexes are effectively used.

The optimization of SQL statements is not just the domain of the database administrator. This book will provide the education to help readers understand how MySQL indexes and storage engines operate, which is an important implementation consideration for an optimal database design by the data architect. Software developers will be able to capture and analyze all SQL statements written to ensure performance bottlenecks can be identified early in the development lifecycle and raised with applicable resources.

Optimizing SQL statements is a key component of improving performance and scalability.



1

The Five Minute DBA

Users are complaining that the application is slow. By reviewing your system and database performance, you have identified a slow running SQL query in the database. If you did not know how to tune an SQL statement in MySQL, what would you do? This book aims to address this need by discussing the ideal approach and best principles toward optimizing SQL statements. This chapter provides a few quick tips you can apply immediately.

In this chapter we will be using the following approach:

- Confirm your slow query
- Identify a missing index
- Apply a new index
- Verify your new index

Identifying Performance Problems

Users report that your application is too slow. After determining there is no physical system resource bottleneck, you turn your attention to the MySQL database.

Finding a Slow SQL Statement

Looking at the current running MySQL connections with the SHOW FULL PROCESSLIST command, you find the following details:

```
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
```

This information shows the SELECT statement in the Info column has been running for 3 seconds via the value in the Time column.

What do you do now?

Confirming the Slow Query

Your first step when identifying a potential slow query is to confirm that it is slow when repeated. Verify that it was not a unique instance that might have occurred because of other factors such as locking or a system bottleneck.

Run and Time Your SQL Statement

Re-running the SQL statement using the MySQL command line client or other client tool is an easy approach for verification:

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

This confirms the query took more than 3 seconds to execute. When the query takes more than 10 milliseconds, the output from the MySQL command line client is sufficient. Chapter 7 details alternative ways to determine the response time of a query.

> ***CAUTION*** *You should rerun only SELECT statements, because these do not modify any existing data. If your slow running query is an UPDATE or DELETE statement, you can simply rewrite this query as a SELECT statement for verification purposes. For example, if the SQL query was* `DELETE FROM inventory WHERE item_id = 16102176`*, you would have rewritten this query as the SELECT statement shown in this example to simulate the performance but not modify any information.*

Generate a Query Execution Plan (QEP)

When MySQL executes an SQL query, it first parses the SQL query for valid syntax, and then it constructs a QEP that determines how MySQL will retrieve information from its underlying storage engines. To show the QEP the MySQL query optimizer is expected to construct for an SQL statement, simply prefix the SELECT statement with the EXPLAIN keyword like so:

```
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
```

This vertical output is obtained using the \G statement terminator with the MySQL command line client. This is helpful for parsing output via automated operations and also for any printed form such as in this book. Using the semicolon (;) terminator provides a column orientated approach that is generally easier to read with multiple rows of output.

> ***NOTE*** *In most cases, an EXPLAIN does not run the actual SQL statement; however, there are some exceptions when part of a SELECT statement might be executed for the optimizer to determine how to construct the QEP. An example is the use of a derived table in the FROM clause, which you would identify with the word* DERIVED in the *select_type column. You can find more information about these limitations in the MySQL Reference Manual at* http://dev.mysql.com/doc/refman/5.5/en/from-clause-subqueries.html.

If you knew nothing about how to read a QEP, the first two columns you should scan are the indexes used and the number of rows affected. Any query that does not use an index signified by the key column in the preceding output can be considered a poorly tuned SQL query. The number of rows affected in evaluating this SQL statement, as signified by the rows column, contributes to an estimation of how much data is read and can directly correlate to the amount of time required to execute the query. The type column with a value of ALL is also an indicator of a potential problem; we will discuss this in more detail in Chapters 4 and 9.

> ***NOTE*** *Depending on the underlying storage engine, the number of affected rows will be either an estimate or an exact number of rows to be examined. Even when the number of affected rows is an estimate (such as when the InnoDB storage engine manages the table storage), the estimate is typically adequate for the optimizer to make an informed decision.*

In this EXPLAIN example no index value was found in the key column. Because this is a single table SELECT statement, this can be considered a full table scan to search for any rows that match the WHERE clause predicate. The rows value can then be considered an approximate value for the number of rows read in order to find the occurrences matching item_id=16102176.

Optimizing Your Query

Identifying a slow running SQL query is a necessary prerequisite for any type of optimization. Throughout this book, we will detail the tools and principles required to determine the various options for an ideal solution.

What You Should Not Do

If you lived in the wild west, where no rules applied, you might consider adding an index to this table based on the WHERE clause. Here is an example:

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

> ***CAUTION*** *Do not try this in a production environment without additional verification!*

There are many factors to choosing to add a new index and deploying it accordingly. This statement highlights just one potential impact on a production environment. This Data Definition Language (DDL) statement took about 55 seconds to complete. During that time, any additional queries that add or modify data for this table are blocked, because the ALTER statement is a blocking operation. Depending on the order in which other Data Manipulation Language (DML) statements are executed, SELECT statements are also blocked from completing during this time. For larger tables, an ALTER statement can takes hours, or even days, to complete! A second impact to consider is the performance overhead of DML statements when a table has multiple indexes.

Confirm Your Optimization

By re-running the SQL query, you can see an immediate improvement with the query now taking less than 10 milliseconds.

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

You can also confirm the effectiveness of the new index by looking at the revised QEP:

```
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:
```

The MySQL optimizer has now selected an index as indicated by the value in the key column, and the number of rows estimated to be examined during the execution of the SQL statement was 1, compared with the original value of 787,338.

The Correct Approach

Adding an index to a table offers benefits including performance optimization; however, there are always other implications for adding an index. Chapters 4 and 5 will discuss the pros and cons of adding indexes on table columns. Before you choose to add an index, you should always perform at least two checks: the first to verify the existing structure of the table, and the second to confirm the size of the table. You can obtain this information using the following SQL commands:

```
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
```

From these commands, you can determine that the current table structure includes a number of indexes, including an index that uses the item_id column. This index was not used, however, because the leftmost column of the index was not satisfied by this query. You also get an approximate size of the table by the Data_length and Rows information from the SHOW TABLE STATUS command. Chapters 4 and 5 will further discuss the importance of this information in determining the time impact of adding an index and the impact of having multiple indexes on the same column.

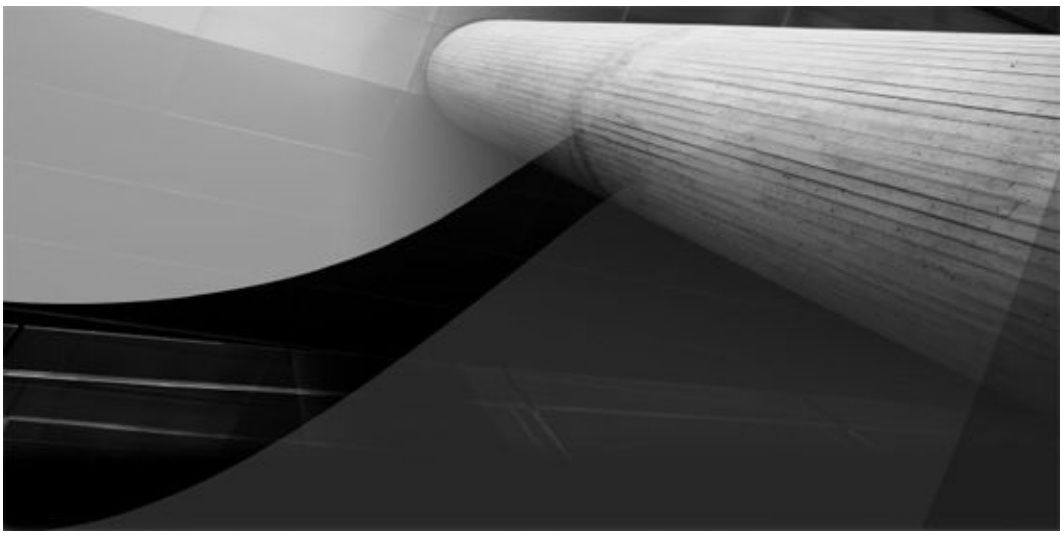
An Alternative Solution

By choosing to look at this SQL statement in isolation, the DBA or architect can elect to create an index, as described. The correct approach for optimizing SQL includes understanding and verifying the purpose for the SQL statement and related SQL statements for this table. By performing this analysis, you would highlight that the application code executing this SQL statement already maintains additional information to improve the query. The value for supp_id was known at the time this SQL statement was executed. By altering the SQL statement to include this column in the WHERE clause, the existing index would be used. No schema changes would be necessary to improve the SQL statement.

In this example, adding an index was not the ideal approach to addressing the observed slow query; without further analysis, the table would have the overhead of an additional unnecessary index.

Conclusion

Optimizing SQL statements is not about just adding an index. This chapter described several analysis tools used to help optimize a statement, including EXPLAIN and SHOW CREATE TABLE. We looked at some of the attributes that identify performance problems and outlined initial important information. We detailed some of the considerations that affect operations when adding an index, and we highlighted several business considerations in providing an optimal solution.



2

The Essential Analysis Commands

No single tool or command in MySQL will identify optimizations for SQL statements. The following SQL commands help identify areas of possible SQL optimization techniques, including creating indexes; however, these will not tell us which technique to apply. The art of SQL optimization requires a human to interpret the facts and deduce one or more optimal solutions. You have already seen a number of these commands in action in Chapter 1.

The following SQL commands will be outlined in this chapter:

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

EXPLAIN

The EXPLAIN command is the essential tool for determining the Query Execution Plan (QEP) of the SQL statement you intend to run. This command offers insight to the MySQL cost-based optimizer and provides details about access strategies the optimizer might have considered and what strategy the optimizer is expected to choose when running the SQL statement. It is important to realize that the generated QEP is not guaranteed and can change, depending on several factors. MySQL offers no way to pin a QEP for a given query, and the QEP is determined for every execution of an SQL statement. The use of stored procedures does not improve this evaluation. Although SQL statements are pre-parsed, the QEP is still determined for every invocation.

The following two QEP examples show how adding a WHERE condition can affect the query execution plan of the SELECT statement:

```
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
```

Throughout this book, we will review a number of different QEP examples and detail various techniques. Chapter 9 goes into great detail about the specifics of each column listed in these examples. The following simple guidelines can be used to identify primary issues quickly:

- No index used (NULL specified in the key column)
- A large number of rows processed (the rows column)
- A large number of indexes evaluated (the possible_keys column)

TIP *Ideally, the EXPLAIN command should be applied to all SQL statements in your system. You can easily prefix all SELECT statements with the EXPLAIN keyword. For UPDATE and DELETE statements, you need to rewrite the query as a SELECT statement to confirm index usage.*

Two optional keywords can be used with the EXPLAIN command: the PARTITIONS and EXTENDED keywords.

EXPLAIN PARTITIONS

The keyword PARTITIONS (since MySQL 5.1) for the EXPLAIN command provides additional information about the specific table partitions that are used to satisfy a query in the partitions column. Here is an example:

```
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
```

EXPLAIN EXTENDED

The EXTENDED keyword of the EXPLAIN command provides the additional column filtered. Here is an example:

```
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)
```

This syntax can also provide insight to the actual SQL statement that is executed. The MySQL optimizer can elect to simplify or rewrite your SQL command. This occurs, for example, when you are using a view or a search condition that is known always to be met (such as WHERE 1=1).

In isolated cases, the EXTENDED syntax can provide answers as to why an index is not ultimately used when it would appear obvious. The actual output that is important is not produced by the EXPLAIN EXTENDED command, but by the SHOW WARNINGS command executed immediately after. Here is an example:

```
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`)
```

In this example, both tables have a primary key of uid, yet the EXPLAIN plan does not show an expected join. Further analysis with EXPLAIN EXTENDED shows that the query is actually rewritten due to a character set conversion.

SHOW CREATE TABLE

The SHOW CREATE TABLE command provides the full details of the current column and index definitions for the necessary table(s) in a format that is easy to read and manipulate. This command provides the exact syntax to re-create the database table and can be easily altered to create optimizations for new or revised indexes, data types, nullability constraints, character sets, and storage engines on the given table.

```
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)
```

NOTE *The backtick (`) is generally not required when referencing database objects.*

It is necessary only when a schema object is a reserved word or uses a space. These are two practices architects and developers should avoid, so that the use of optional backticks are never required. Unfortunately, there is no way to remove this from generated syntax with this command.

The mysql dump command is a quick means of generating a definition of all tables in your schema or database instance. Here is an example:

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

You can also use several INFORMATION_SCHEMA tables to obtain components of table structure, including the following: TABLES, COLUMNS, TABLE_CONSTRAINTS, KEY_COLUMN_USAGE, REFERENTIAL_CONSTRAINTS (since MySQL 5.1), and PARTITIONS (since MySQL 5.1).

SHOW INDEXES

The SHOW INDEXES command provides additional index information, including the index type and the currently reported cardinality of MySQL indexes.

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

The full list of output columns from this command is shown here:

```
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 ***************************
…
```

You can also use the INFORMATION_SCHEMA.STATISTICS table to retrieve the same information.

The value of the Cardinality column is important to review. This value represents the approximate number of unique values per column within the index. In the preceding example, the primary key cardinality represents 2689 unique values, while the post_type column reports only 4 unique values. The purpose of cardinality is discussed in Chapter 4.

NOTE *Refer to the MySQL Reference Manual for additional information on SHOW INDEXES (http://dev.mysql.com/doc/refman/5.5/en/show-index.html).*

SHOW TABLE STATUS

The SHOW TABLE STATUS command provides information about the underlying size and structure of a table, including its storage engine type, version, data and index size, average row length, and number of rows.

```
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:
```

The accuracy of certain values depends on the storage engine used. For example, with MyISAM, MEMORY, ARCHIVE, and BLACKHOLE storage engines, the average row length and number of rows values are precise. For the InnoDB storage engine, these values are approximate and can be wildly inaccurate. The following shows the change in values between the MyISAM and InnoDB storage engines:

```
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
    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 |
+----------+
```

The INFORMATION_SCHEMA.TABLES table provides information about the underlying table and can provide information similar to the SHOW TABLE STATUS command. The benefit of using the INFORMATION_SCHEMA to retrieve information is the ability to use SQL SELECT syntax to restrict, sort, and format the output of information. Here is an example:

```
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)/1024/1024,2) 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
*************************** 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
```

CAUTION *SELECT queries against INFORMATION_SCHEMA tables can take a long time to execute. This example query could take more than 60 seconds for a schema with a large number of tables and large size tables.*

SHOW STATUS

The SHOW [GLOBAL|SESSION] STATUS command provides details of the current internal status of the MySQL server. This information is very valuable in a global sense to determine various indicators of load for a MySQL server. For any given session, valuable information can be determined from a large number of variables to provide the impact of a given SQL command—for example, the use of an internal temporary table, an index range scan, and the amount of rows read. When a scope is not specified for SHOW STATUS, the default is SESSION. Here is an example:

```
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)
```

You can also use the INFORMATION_SCHEMA.GLOBAL_STATUS and INFORMATION_SCHEMA.SESSION_STATUS tables (since MySQL 5.1) to retrieve the same information.

The interpretation of status variables can assist an experienced architect in fine tuning how MySQL internally executes an SQL statement. Following are example comparisons

of status values that can assist in evaluating the effectiveness of an index:

```
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 |
+-----------------------+--------+
```

This information shows that no key was used.

In the following query, the Handler_read_key value of 1 indicates an index was used, and the Handler_read_next value of 71 indicates that the index was used to read 71 additional rows:

```
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     |
+-----------------------+-------+
```

In the next query, you can see that the Handler_read_prev value of 71 shows the index was used to read 71 additional rows in reverse order:

```
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     |
+-----------------------+-------+
```

SHOW VARIABLES

The SHOW [GLOBAL|SESSION] VARIABLES command provides details of the current value of MySQL system variables. Certain variables affect how SQL statements can execute. For example, tmp_table_size limits the maximum memory size of an internally created temporary table. By knowing the current per session or global value of certain settings and by altering these dynamically (when applicable) using the SET command, you can alter the performance outcome of SQL statements. When a scope is not specified for SHOW VARIABLES, the default is SESSION. Here is an example:

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

You can also use the INFORMATION_SCHEMA.GLOBAL_VARIABLES and INFORMATION_SCHEMA.SESSION_VARIABLES tables (since MySQL 5.1) to retrieve the same information. This can be very helpful in identifying differences in the values of SESSION and GLOBAL scope of variables. Here is an example:

```
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       |
+---------+--------------------+----------------+
```

INFORMATION_SCHEMA

The INFORMATION_SCHEMA provides an ANSI SQL statement interface for many of the SHOW commands in this chapter. The number of INFORMATION_SCHEMA tables varies according to your MySQL version. MySQL 5.5 currently includes 37 default tables. The following tables refer to the commands listed in this chapter:

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

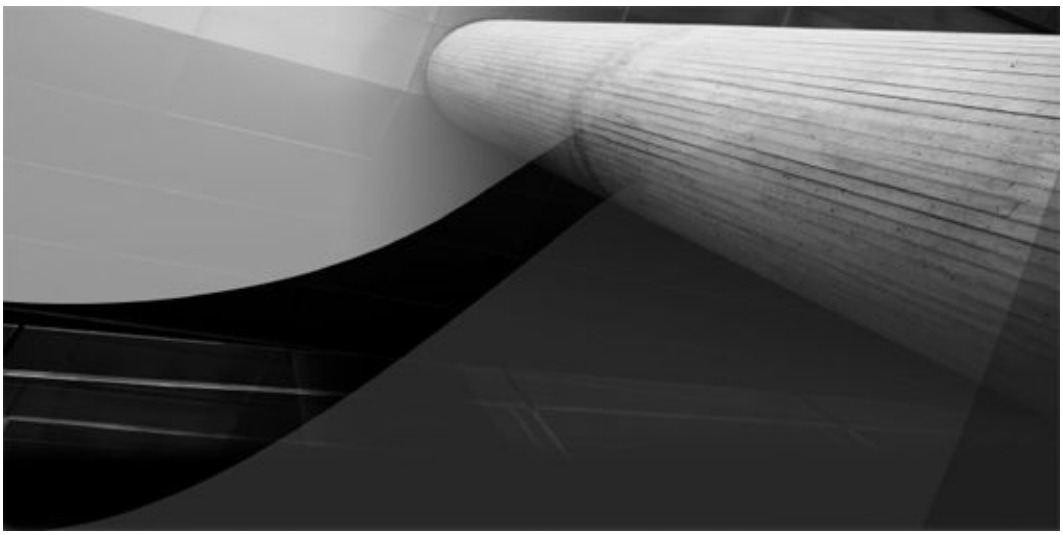
NOTE *For a full list of tables, refer to the MySQL Reference Manual at http://dev.mysql.com/doc/refman/5.5/en/information-schema.html.*

Conclusion

Familiarity with the output of these commands is an essential skill in a practical and reproducible approach to SQL optimization. Although these are the essential commands, several factors including time, data change, storage engines, and the version of MySQL can cause SQL statements to operate differently. The use of these commands should not be a "one off approach" when you are evaluating an SQL statement to optimize. The gathering and analysis of information should be done using an iterative approach.

TIP *Understanding the MySQL EXPLAIN statement is an art that can take many years of experience to perfect.*

All relevant SQL statements for this chapter can be downloaded at http://effectivemysql.com/book/optimizing-sql-statements.



3

Understanding MySQL Indexes

Creating the right indexes is one of the most significant techniques for SQL performance tuning. Before creating MySQL indexes, as discussed in the following chapter, you need to understand the MySQL architectural details about how indexes are organized on disk, how indexes and memory usage operate, and how the differences between storage engines affect index selection. This understanding is a necessary prerequisite to mastering optimal index creation in a MySQL production environment. You might find this chapter less interesting; however, the specific nature of indexes and a detailed knowledge of how MySQL operates with indexes will greatly enhance your DBA skills from other RDBMS products.

In this chapter we will be covering the following topics:

- The possible uses for MySQL indexes
- Understanding the theory of various index data structures
- Different index implementations between storage engines
- MySQL indexes with partitioning

Example Tables

In this chapter, several tables are created and referenced to demonstrate various MySQL index implementations. The following SQL statements are used to create a number of simple sample tables and populate them with a reasonable amount of data:

```
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;
```

The input file used in this example is the standard dictionary file found with the Community Enterprise OS (CentOS) 5.5 Linux distribution. The number of words in this file might differ depending on your operating system. You can obtain a copy of this file from http://EffectiveMySQL.com/downloads/words if you want to reproduce this exact example.

MySQL Index Usages

MySQL can use indexes to support a range of different functions. Indexes are not just for optimizing MySQL performance when reading data. These functions include the following:

- Maintaining data integrity
- Optimizing data access
- Improving table joins
- Sorting results
- Aggregating data

Data Integrity

MySQL uses both primary and unique keys to enforce a level of uniqueness of your storage data per table. The differences between primary and unique keys are as follows:

Primary key

- Only one primary key may exist per table.

- A primary key cannot contain a NULL value.
- A primary key provides a means of retrieving any specific row in the table
- If an AUTO INCREMENT column is defined, it must be part of the primary key.

Unique key

- More than one unique key per table is supported.
- A unique key can contain a NULL value where each NULL value is itself unique (that is, NULL != NULL).

The million_words table contains a primary key on the id column. This constraint ensures no duplicate values. Here is an example:

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

Likewise, the million_words table contains a unique key on the word column. This constraint ensures a duplicate word cannot be added.

```
mysql> INSERT INTO million_words(word) VALUES('oracle');
ERROR 1062 (23000): Duplicate entry 'oracle' for key 'word'
```

In Chapter 4 we will be providing additional examples of primary and unique key indexes.

In addition, some MySQL storage engines support foreign keys for data integrity. These are not actually an index; they are referred to as a constraint. However, a common prerequisite for certain implementations of foreign keys is that an index exists in both the source and parent tables to enable the management of foreign keys. Currently only the InnoDB storage engine of the default MySQL storage engines supports foreign key constraints and there is no requirement for a corresponding index; however, this is highly recommended for performance.

CAUTION *Although MyISAM does not support foreign key constraints, the CREATE TABLE (…) ENGINE=MyISAM syntax allows for the definition of foreign keys via the REFERENCES syntax.*

Optimizing Data Access

Indexes allow the optimizer to eliminate the need to examine all data from your table during query execution. By restricting the number of rows accessed, query speeds can be significantly improved. This is the most common use for an index.

For example, in our example table of one million words, if the word column is not indexed, each SELECT would need to scan all one million rows sequentially in the random order in which they were added to find zero or more matching rows every time. Even if the data were originally loaded in sequential order, SQL does not know this and must process every row to find a possible match.

For example, we will create a table without an index:

```
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)
```

When the table has an index on the word column, each SELECT would first scan the index that is ordered and is well optimized for searches to identify a reference to the zero or more rows that contain the matching information. When the index is defined as unique, the SELECT would know that the results contained at most one matching row. Here is another example, using our million_words table:

```
mysql> SELECT * FROM million_words WHERE word='oracle';
+--------+--------+
| id     | word   |
+--------+--------+
| 266877 | oracle |
+--------+--------+

1 row in set (0.00 sec)
```

The indexed column example retrieves a row in less than 10 milliseconds via this MySQL client output. When not indexed, the row(s) retrieved take 250 milliseconds.

> ***CAUTION*** *Adding an index is not an automatic improvement in performance for all types of SQL queries. Depending on the number of rows required, it might be more efficient to perform a full table scan of all data. This is a difference between random I/O operations of retrieving individual rows from index lookups and a sequential I/O operation to read all data.*

Throughout the remainder of the book, we will be providing more detailed examples of

how indexes are used for query restriction.

Table Joins

In addition to restricting data on a given table, the other primary purpose for an index is to join relational tables conveniently and efficiently. The use of an index on a join column provides the immediate performance benefit as described in the previous section when now matching a value in a different table. The mastering of creating correct indexes to perform efficient table joins is fundamental for SQL performance in all relational databases.

Sorting Results

MySQL indexes store data in a sorted form. This makes the use of the index very applicable when you would like the result of a SELECT statement in a given order. It is possible to sort data for any SELECT query via the ORDER BY operator. Without an index on the ordered-by column, MySQL will typically perform an internal filesort of the retrieved table rows. The use of a predefined index can have a significant performance improvement on a high concurrency system that is required to sort hundreds or thousands of individual queries per second, since the results are naturally ordered in the index. Simply having an index that matches the order you want for your results does not automatically mean that MySQL will choose to use this index. In Chapter 4, we discuss how MySQL uses indexes and specifically what limitations exist.

Aggregation

Indexes can be used as a means of calculating aggregated results more easily. For example, the sum of the total of all invoices for a given period might be more efficiently performed with an appropriate index on the date and invoice amount. Chapter 4 will show examples of the appropriate way to create indexes to support aggregation.

About Storage Engines

If you are unfamiliar with MySQL, or are familiar with other relational database systems, the concept of a storage engine can take some time to understand. In summary, although MySQL communicates and manages data via Structured Query Language (SQL), internally MySQL has different mechanisms to support the storage management and retrieval of the underlying data. The flexibility of MySQL storage engines is both a blessing and a curse. The saying “With great flexibility comes great responsibility” is applicable in this sense. We will not be detailing storage engines in this book, but it is critical that you understand some basic information about storage engine features and

capabilities, including the following:

- Transactional and non-transactional
- Persistent and non-persistent
- Table and row level locking
- Different index methods such as B-tree, B+tree, Hash, and R-tree
- Clustered indexes versus non-clustered indexes
- Primary versus secondary indexes
- Data compression
- Full text index capabilities

MySQL supports the capability of pluggable storage engines from other service providers, which includes both open source and commercial offerings. Being an open source product, MySQL offers variants that support additional different storage engines.

During this book we will be focused on discussing three primary storage engines that are included by default with MySQL:

- **MyISAM** A non-transactional storage engine that was the default for all MySQL versions prior to 5.5
- **InnoDB** The most popular transactional storage engine and the default engine starting with version 5.5
- **Memory** As the name suggests, a memory based, non-transactional, and non-persistent storage engine

NOTE *Starting with version 5.5, the default storage engine for tables has changed from the MyISAM storage engine to the InnoDB storage engine. This can have a significant effect when you are installing packaged software that relies on the default settings and was originally written for the MyISAM storage engine.*

Current versions of MySQL also include the built-in storage engines of ARCHIVE, MERGE, BLACKHOLE, and CSV. Some of the other popular storage engines provided by MySQL or third parties include Federated, ExtraDB, TokuDB, NDB, Maria, InfinDB,

Infobright, as well as many more.

> ***TIP*** *You can use the SHOW CREATE TABLE, SHOW TABLE STATUS, or INFORMATION_SCHEMA.TABLES to determine the storage engine of any given table. Chapter 2 provides detailed examples of these options.*

For more information about MySQL storage engines and a more detailed list, visit http://EffectiveMySQL.com/article/storage-engines/.

Index Terminology

Understanding all the different terminology relating to indexes can be confusing. The following table is a guide to these terms that are used for this text.

Index **Technique**	This is the theory behind how different data structures enable various approaches to accessing underlying information. These techniques include B-tree, B+tree, R-tree, and Hash. Each technique uses different concepts to achieve a particular goal or strength of the data structure.
Index **Implementation**	This is how MySQL and various storage engines implement the various data structure techniques. For example, the MyISAM storage engine implements B-tree differently from the InnoDB storage engine.
Index **Type**	This refers to the common usages of indexes. These include the primary key, unique key, secondary (also known as normal), fulltext, and spatial types. Each of these types can also support a single column, multi column (also known as compound), and/or partial column to define the given index type. Finally, one or more of these index types can also result in an index being referred to as a covering index.

These terms will be discussed in greater detail throughout this chapter and later chapters.

MySQL Index Types

MySQL supports the expected primary key, unique, and non unique secondary indexes found in all relational databases. In addition, MySQL can support full text and spatial index types in certain storage engines.

The built-in storage engines in MySQL have different implementations of various index techniques, including B-tree, B+tree, Hash, and R-tree. With additional pluggable storage engines other implementations of indexes are possible—for example the Fractal Tree index in TokuDB (see http://tokutek.com/technology/) or the column oriented storage approach in InfiniDB (see http://www.calpont.com/) and Infobright (see http://infobright.com/).

These different index implementations of the various data structure techniques are important as they can directly affect how SQL queries operate and perform. As a simple example, a Hash implementation provides optimal performance for direct lookups but is not efficient for range based, while a B-tree implementation is designed for range based queries. It is possible to determine the effectiveness and efficiency of a given data structure mathematically. Furthermore, the column oriented implementations with InfiniDB and Infobright engines actually eliminate the need for user defined indexing strategies.

Index Data Structure Theory

Before explaining how different index types are implemented in MySQL, it is helpful for you to understand the basic theory of the primary data structures used.

B-tree

The B-tree data structure supports data insertion, manipulation, and selection via the management of a series of connected nodes in a structure like the roots of a tree. There are two types of nodes: the index nodes that organize and direct you in an ordered sense to data that is stored in leaf nodes. The B-tree data structure is not to be confused with the Binary Tree data structure that is a simple implementation of a hierarchy of nodes.

Figure 3-1 provides a graphical representation of the B-tree data structure. For more information on the concept, mathematical proof, and a technical description of the B-tree data structure, see http://en.wikipedia.org/wiki/Btree.

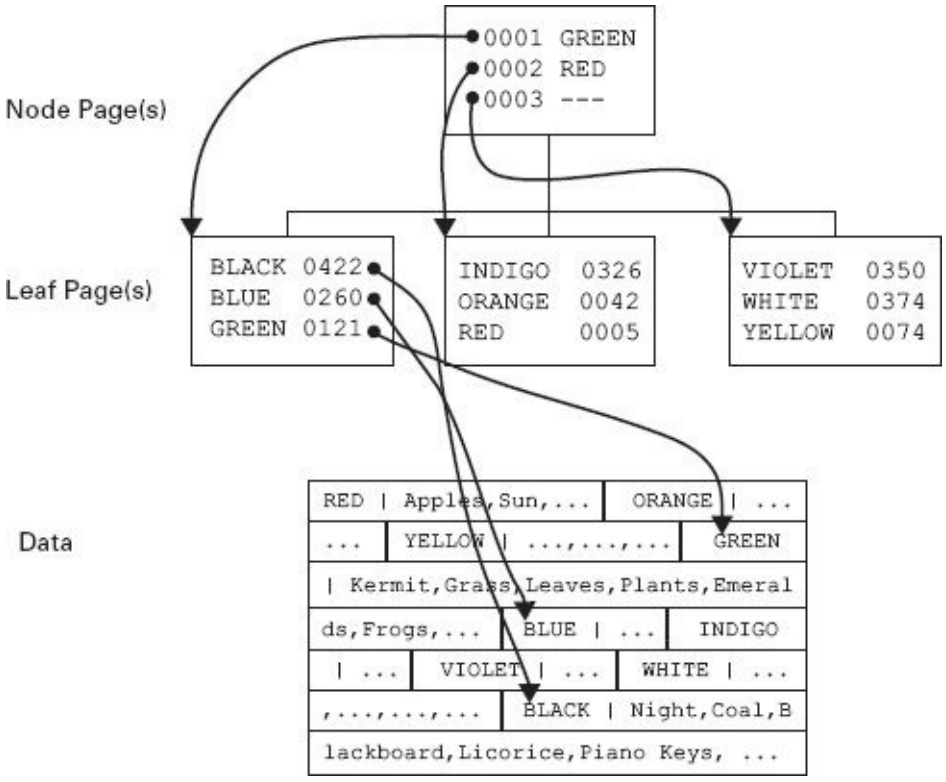


Figure 3-1 *B-tree data structure*

B+tree

The B+tree data structure is an enhanced version of the B-tree implementation. Although it supports all the features of the B-tree index, the significant difference is that the underlying data is ordered based on the indexed column in question. The B+tree data structure also utilizes additional references between leaf nodes for optimized scanning.

Figure 3-2 provides a graphical representation of the B+tree data structure. For more information on the concept and technical description of the B+tree data structure, see http://en.wikipedia.org/wiki/B%2B_tree.

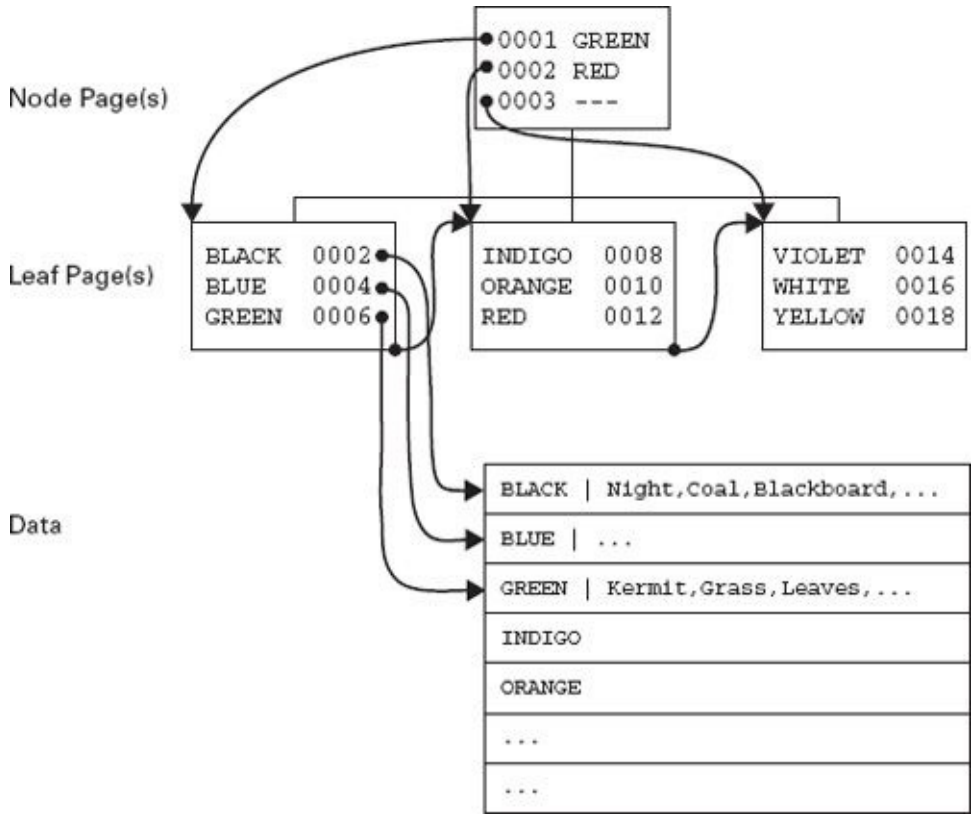


Figure 3-2 *B+tree data structure*

Hash

The Hash table data structure is a very simple concept, where an algorithm (or Hash function) is applied to the given value to return a unique pointer or position within the underlying storage of the data. The benefit of a Hash table is a consistent linear time to identify the position of any given row to retrieve, unlike the B-tree data structure, where multiple levels of nodes might need to be traversed to determine the same position.

Figure 3-3 provides a graphical representation of the Hash tree data structure. For more information on the concept and technical description of the Hash table data structure, see http://en.wikipedia.org/wiki/Hash_table.

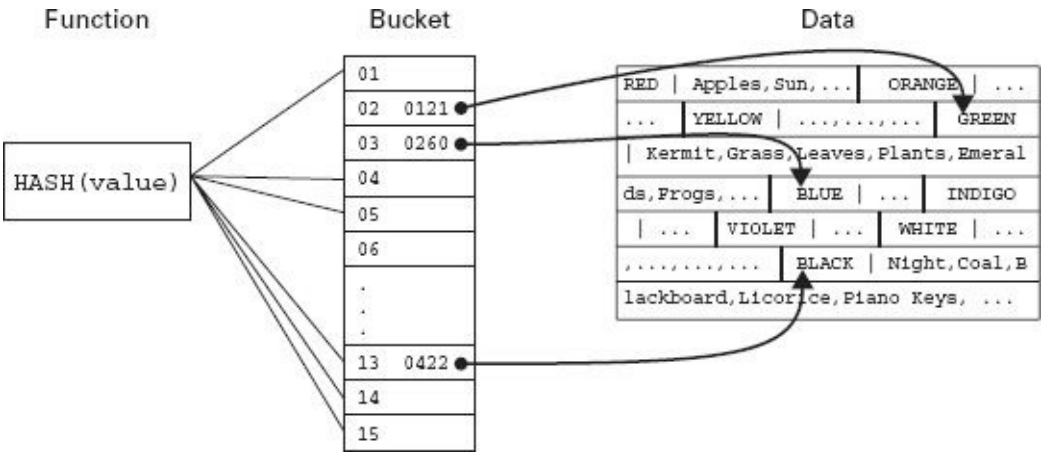


Figure 3-3 *Hash data structure*

R-tree

The R-tree data structure supports managing geometry based data types. Currently only MyISAM supports spatial indexes with an R-tree implementation. There are also several restrictions with spatial indexes, including only supporting a single NOT NULL column. Due to the infrequent use of spatial indexes, we will not be discussing them in any further detail during this book.

For more information on the concept and technical description of R-tree, see http://en.wikipedia.org/wiki/R-tree.

Fulltext

Completing the complement of basic data structure techniques found in MySQL is fulltext. This implementation is supported only by the MyISAM storage engine in the current version of MySQL. MySQL version 5.6 is expected to include fulltext functionality with the InnoDB storage engine. The use of this index is impractical in a large scale implementation where more applicable dedicated products exist for text indexing and searching. Due to the practical limitations of fulltext for large scale deployments we will not be discussing it in any further detail in this book.

For more information on the concept and technical description of fulltext indexes, see http://en.wikipedia.org/wiki/Full_text_search.

> ***TIP*** *The Sphinx search server can be integrated as a pluggable storage engine enabling support for a number of fulltext features and also integrating with data available in MySQL. See http://sphinxsearch.com/ for more information.*

For more introductory information regarding these index techniques, visit http://EffectiveMySQL.com/article/index-techniques/.

MySQL Implementation

With an appreciation of the basic concepts of B-tree, B+tree, and Hash data structures, we can discuss how MySQL implements different algorithms via storage engines to support these. Each implementation has relative advantages in SQL performance for certain types of queries. Each implementation also has a different impact on disk space and memory usage, which is an important consideration for supporting large databases.

MyISAM B-tree

The MyISAM storage engine uses a B-tree data structure for primary, unique, and secondary indexes. Within the MySQL instance data directory and database schema subdirectory you will find a corresponding .MYD and .MYI file for each MySQL table. Information for the defined table indexes is stored in the .MYI file. By default the block size for a .MYI file is 1024 bytes. This is configurable with the myisam-block-size system variable.

```
$ 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
```

The internal format of these file structures is available via the MySQL source code, which is freely available, and via the MySQL Internals Manual page found at http://forge.mysql.com/wiki/MySQL_Internals_MyISAM#The_.MYI_file.

In MyISAM, the B-tree structure of a secondary index stores the indexed value and a pointer to the primary key data. This is an important distinction between MyISAM and InnoDB. The impact of this changes how an index works differently between these two storage engines. Chapter 4 provides a specific example.

The MyISAM indexes are managed in memory in the common key buffer as defined by key_buffer_size or additional named key buffers. This is an important consideration when determining the size of the buffers based on the calculated and projected size of table indexes. These configuration options are described in Chapter 6.

You can actually see how MyISAM implements a B-tree data structure by simple analysis of the underlying file structure. The following creates a small sample table and corresponding index:

```
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);
```

You can look directly at the MyISAM data file to see the data is ordered as was specified in the INSERT statement. Here is an example:

```
$ od -c colors.MYD
0000000 001  \0 031 003   R   E   D 024   A   p   p   l   e   s   ,   S
0000020   u   n   ,   B   l   o   o   d   ,   .   .   . 003  \0 030  \0
0000040 006   O   R   A   N   G   E 020   O   r   a   n   g   e   s   ,
0000060   S   a   n   d   ,   .   .   . 003  \0  \v 005 006   Y   E   L
0000100   L   O   W 003   .   .   .  \0  \0  \0  \0  \0 003  \0   \  \0
0000120 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
0000240   r   a   f   f   i   c       L   i   g   h   t 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  \a   I   N   D   I   G   O 003   .   .   .   .
0000340  \0  \0  \0  \0 003  \0  \v 005 006   V   I   O   L   E   T 003
0000360   .   .   .  \0  \0  \0  \0  \0 003  \0  \n 006 005   W   H   I
0000400   T   E 003   .   .   .  \0  \0  \0  \0  \0  \0 003  \0   4  \0
0000420 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   ,   .   .
```

You can also look at the MyISAM index and see the ascending order of values. Here is an example:

```
$ 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
...
```

InnoDB B+tree Clustered Primary Key

The InnoDB storage engine uses a B+tree data structure for the primary key. This is also referred to as a clustered primary key. This structure organizes the data along with the primary key and also has additional next and previous page pointers on the leaf-level node pages, allowing for easier range scan operations.

> ***NOTE*** *In other database products including Oracle, this type of structure is referred to as an index organized table.*

On the filesystem, all InnoDB data and index information is managed within the common InnoDB tablespace by default. Unless otherwise specified with the innodb_data_file_path, this is a single file named ibdata1 that is found in the MySQL data

directory. Here is an example:

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

You can use the SHOW TABLE STATUS command and INFORMATION_SCHEMA.TABLE information to determine the size of individual tables. You can also define InnoDB to use a tablespace per table with the innodb_file_per_table option, which enables you to see the combined table data and index size per file. This per table tablespace is found in the appropriate schema subdirectory. Here is an example:

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

As InnoDB stores data with a clustered primary key, the amount of disk space for the underlying information can vary greatly due to the page fill factor. For sequential primary keys, InnoDB will use a fill factor of 15/16 of a 16K page size. For nonsequential primary keys, by default InnoDB will assign a 50 percent fill factor per page when inserting initial data. Knowing this fact is important for determining potential disk space requirements. For example, we can review the size of the existing table and create a new table based on a nonsequential primary key for comparison.

```
$ 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: Compact
table_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
```

In the original table using an AUTO_INCREMENT and sequential primary key, the underlying data size is 34MB. Using a nonsequential primary key, the same data set produces a data size of 66MB. Although this is not an exact means of measurement, this demonstrates the impact of the default fill factor.

The standard MySQL product currently provides no means of adjusting these fill factor calculations. In addition, the importance for understanding this implementation is that all InnoDB data and indexes are managed in memory with the InnoDB buffer pool defined by the innodb_buffer_pool_size configuration option. As your system grows, the impact of disk layout and memory utilization can have a direct impact on SQL performance.

The internal format of these file structures is available via the MySQL source code, which is freely available, and via the MySQL Internals Manual page found at http://forge.mysql.com/wiki/MySQL_Internals_InnoDB.

Looking under the hood and trying to verify InnoDB on the file system is much more complicated than the MyISAM example. If we used the color table example as previously described, we would not see data ordered in InnoDB. It would look like the following:

```
$ 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
```

Without going into full technical details, InnoDB stores data in 16K data pages. The B+tree for each page is ordered relative to all pages, and the underlying data within each page may not be ordered. To see ordered data in InnoDB, we need to increase the per row size to force less row per data page. Here is an example:

```
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  \0 230 354 352   Y   E   L   L   O   W  \0  \0  \0  \0
```

NOTE *I would not recommend you use this approach for any serious analysis. This is provided to give you a basic confirmation that data is stored in an ordered fashion in comparison to MyISAM.*

InnoDB B-tree Secondary Key

The secondary indexes in InnoDB use the B-tree data structure; however, they differ from the MyISAM implementation. In InnoDB, the secondary index stores the physical value of the primary key. In MyISAM, the secondary index stores a pointer to the data that contains the primary key value.

This is important for two reasons. First, the size of secondary indexes in InnoDB can be much larger when a large primary key is defined—for example when your primary key in InnoDB is 40 bytes in length. As the number of secondary indexes increase, the comparison size of the indexes can become significant. The second difference is that the secondary index now includes the primary key value and is not required as part of the index. This can be a significant performance improvement with table joins and covering indexes. This is demonstrated in Chapter 5.

Memory Hash Index

Of the included default MySQL engines, only the MEMORY engine supports a Hash data structure, which is the default for primary and secondary indexes. The strength of a Hash structure is simplicity of a direct key lookup. The following example shows a comparison with different queries:

```
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%'

In this example the simulated range pattern match was 1000 times slower for a Hash index than a direct value lookup.

It is also possible to specify a B-tree index implementation for the MEMORY storage engine. The following information shows the size of the table data and index space for later comparison with a MEMORY B-tree index:

```
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
```

MEMORY B-tree Index

For large MEMORY tables, an index range search can be very inefficient with a Hash index, as indicated in the previous example. Repeating the same SQL commands with a B-tree index shows the impact:

```
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%'

The downside is the increase in size of the index space:

```
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
```

As you can see, the B-tree index size is approximately twice the size as the Hash index.

> ***CAUTION*** *You might observe in this example that the B-tree index was actually faster for a direct key lookup than the default Hash index. Depending on the depth of the B-tree, it is possible that one or two operations could be faster than the hashing algorithm. This example emphasizes that testing index variations is important to identify optimal solutions for your specific use case.*

InnoDB Internal Hash Index

The InnoDB storage engine stores the primary key in a clustered B+tree index; however, internally InnoDB can determine that an in-memory Hash index is more effective for primary key lookups. This is managed by the InnoDB storage engine, and the only configuration setting is either to enable or disable via the innodb_adaptive_hash_index configuration option.

> ***NOTE*** *More information about the InnoDB adaptive Hash can be found at http://dev.mysql.com/doc/refman/5.5/en/innodb-adaptive-hash.html and http://dev.mysql.com/doc/innodb/1.1/en/innodb-performance-adaptive_hash_index.html.*

MySQL Partitioning

Starting with version 5.1, MySQL supports a means of table partitioning types via Range, List, Hash, Key, and (since 5.5) Columns. The implementation of partitioning has some effects on index usage and optimization.

A partitioned table does not support fulltext indexes, spatial indexes, or foreign keys. The primary and unique indexes in partitioned tables must include all columns used in the partitioning expression.

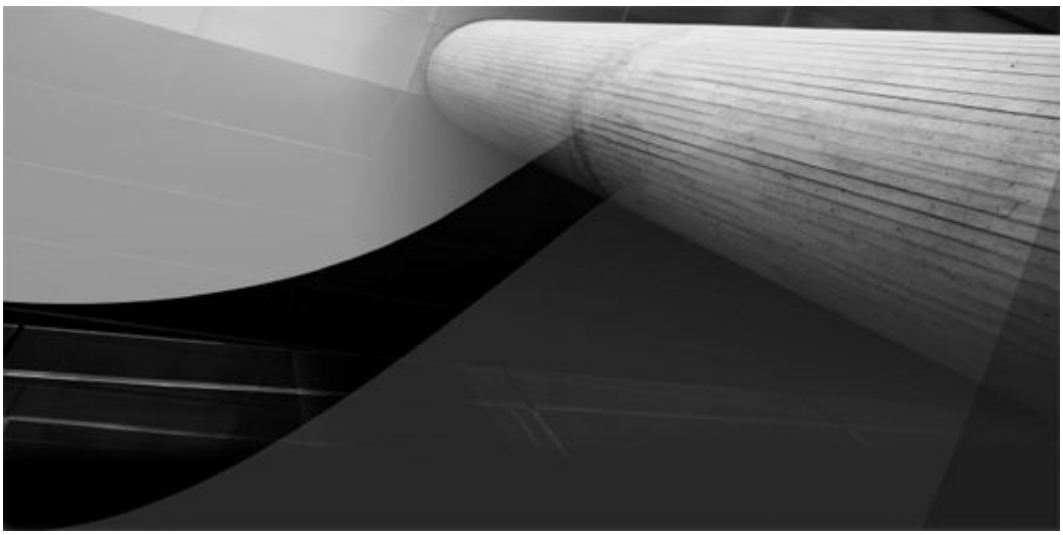
A benefit of partitioning is the enabling of partition pruning when executing SQL statements. As detailed in Chapter 2 with the EXPLAIN PARTITIONS command, MySQL can leverage partitioning to scan only indexes that are applicable rather than scanning an entire index.

> ***NOTE*** *For more information on the documented limitations of indexes in partitioning, see http://dev.mysql.com/doc/mysql-reslimits-excerpt/5.5/en/partitioning-limitations.html.*

Conclusion

In this chapter we discussed the various common data structures and the different MySQL storage engine implementations of these to support different index types. While this chapter provided no detailed examples of how to create MySQL indexes for optimizing SQL statements, it is very important that you understand how an index is used and is represented in MySQL in varying forms. The next two chapters will detail many examples and will reference this important supporting information.

You can download all SQL statements for this chapter at http://effectivemysql.com/book/optimizing-sql-statements.



4

Creating MySQL Indexes

In this chapter, we will discuss various types of indexes, how MySQL determines indexes, and the impact of adding indexes to your system's overall performance. We will also be describing how to create indexes for specific given queries. This does not imply that if your environment uses similar tables you should automatically create indexes; these index examples describe how to create and evaluate a particular situation.

Creating indexes is not the only way to optimize an SQL statement. Determining how to optimize the database schema or an application's use of data from an SQL statement can often have a far greater impact on overall system performance. Creating indexes can often provide a great performance gain with less overall impact and can be implemented more conveniently in certain environments.

In this chapter we will be covering the following topics:

- Basic index types including single and multi column indexes
- How MySQL uses indexes
- The performance impact of adding indexes
- Various MySQL index limitations

We will not be discussing in this chapter any examples for spatial or fulltext indexes that are available with the MyISAM storage engine.

Example Tables

In this chapter a number of example tables and sample data are used. Here are the primary tables:

```
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;
```

You can download a full schema and sample data for this chapter from http://EffectiveMySQL.com/downloads/music_example.tar.gz. The table data is obtained from the MusicBrainz open music encyclopedia at http://musicbrainz.org under a public data license. For more information see http://musicbrainz.org/doc/MusicBrainz_License.

> ***NOTE*** *These table structures are defined for the purpose of explaining the possible usage of MySQL indexes and are used to describe different examples. They do not represent how this data model should be defined in a production environment.*

Existing Indexes

In most situations you will be optimizing SQL statements on existing tables with existing indexes. For any given query you wish to review, Chapter 2 described the tools of the trade. We will progress through these tools during this chapter with different examples to describe how the output can help in optimizing SQL statements. The following is a commonly observed query to retrieve specific information about a given artist:

```
SELECT artist_id, type, founded
FROM artist
WHERE name = 'Coldplay';
```

The first action is always to view the Query Execution Plan (QEP) using the EXPLAIN keyword prefixed to the SELECT statement. This does not actually execute the SQL statement.

```
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
```

NOTE *While the EXPLAIN command does not execute an SQL statement, EXPLAIN can execute subqueries in the FROM clause while determining an execution plan.*

You should always verify the table structure, indexes, and storage engines when reviewing SQL statements with the following command:

```
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
```

This output confirms that an index for the name column exists and is used in the execution of this query.

Single Column Indexes

The most basic index is a single column index, which is an index for one specific column for a database table. There is no practical limitation to the number of indexes you can have on a table; however, there is a performance impact that we will discuss later in the section "The Impact of Adding Indexes." MySQL will generally select only one index per table. Starting with MySQL 5.0, there are a small number of exceptions when the optimizer may utilize more than one index. We will be discussing these in the section "MySQL Optimizer Features."

Syntax

To add an index to an existing table you use the following general syntax:

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

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

NOTE *Although the keywords KEY and INDEX may be interchanged for secondary indexes, only the keyword KEY is valid for the primary index.*

Restricting Rows with an Index

If there was a common requirement to retrieve artists by the year they were founded, you could create an index to avoid a full table scan. A full table scan can be determined with type=ALL and key=NULL. Here is an example:

```
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
```

We create an index on the founded column with the following syntax:

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

Re-executing the EXPLAIN SELECT from above will show us if this new index will now be used. The following output shows that the new index would indeed be used, and the number of rows estimated to be examined is dramatically lower than before, likely resulting in a much faster query time:

```
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
```

As previously mentioned, there is no practical limit to the number of indexes a table can have. There is also no restriction on creating duplicate indexes. If you accidentally created the same index a second time, you would see the following information:

```
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
```

NOTE *There is no requirement to name your index. MySQL will automatically identify your index based on the first column name and with additional optional extension for uniqueness. As you can see, we have now added a second index on the same column. Duplicate indexes are a performance overhead. In the following section on multi column indexes, we will discuss how to identify and remove duplicate indexes.*

Although there is no recommended standard for naming indexes, choosing to name all indexes for a table might cause an error rather than a duplicate index being created.

Joining Tables with an Index

A second benefit of an index is to improve performance when joining relational tables. For example the following SQL statement retrieves the albums for a given artist:

```
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
```

We can see in this SQL example that a full table scan is occurring with the album table. We can address this by adding an index for the join condition and repeating the EXPLAIN statement.

```
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:
```

For the album table, we now use the newly created artist_id index by the key value, and you also see a ref value that indicates this joins with the artist table.

Understanding Index Cardinality

When you have multiple different indexes that can be used for a query, MySQL tries to identify the most effective index for the query. It does so by analyzing statistics about the data distribution within each index. In our example, we are looking for all bands that were founded in 1980. Given these requirements, we create an index on the artist's type because that is what we will be searching on.

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

To demonstrate this correctly with all MySQL 5.*x* versions, we disable an optimizer setting for the purposes of this example:

```
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
```

In this example, MySQL has to make a choice between the possible indexes as listed in possible_keys. The optimizer chooses an index based on the estimated cost to do the least amount of work, not what a human considers the right order. We can use the index cardinality to confirm the likely reason for this decision. Here is an example:

```
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
…
```

This information shows that the founded column has a higher cardinality—that is, a higher number of unique values, and therefore there is a higher likelihood of finding the needed records in fewer reads from the index. The statistics information is only an estimate. We know from data analysis that there are only four unique values for type, yet statistics indicate otherwise.

A discussion of cardinality is not complete without discussing selectivity. Knowing the number of unique values in an index is not as useful as comparing that number to the total number of rows in the index. Selectivity is defined as the number of distinct values in relation to the number of records in the table. The ideal selectivity is a value of 1. This is a non null unique value for every value. Having an index with good selectivity means that fewer rows have the same value. Poor selectivity is when there are few distinct values—for example gender or a status. This determination can not only be used to determine when an index might not be effective, but also how to order columns in a multi column index when all columns are used for your queries.

The presented index cardinality provides a simple insight. The following two queries look for bands and combinations for the 1980s.

```
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
```

Although these queries appear similar, a different index path was chosen based on more detailed statistics of the distribution of information for the columns.

Using Indexes for Pattern Matching

You can utilize an index for a pattern match using a wildcard character. Here is an example:

```
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
```

MySQL will not use an index if your search term starts with a wildcard character. Here is an example:

```
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
```

TIP *If you always require a wildcard search for the start of a column, a common trick is to store the reverse of the value in the database. For example, suppose you wanted to find all email addresses with a ".com" domain. MySQL cannot use an index when searching with* `email LIKE '%.com'`*; however, searching* `reverse_email LIKE REVERSE('%.com')` *will use an index when defined on* `reverse_email`*.*

MySQL does not support function based indexes. Attempting to create an index with a column function will result in a syntax error. A common problem with developers from a different database background is to perform the following, expecting that an index on the name column will be used to satisfy the query:

```
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
```

MySQL will not use the name index because of the UPPER() function that is applied to this column.

TIP *MySQL by default uses a case insensitive character set for storing text information. There is no need to store data in a particular case format and use case specific comparisons in your SQL statements.*

Selecting a Unique Row

If we wanted to ensure that all artists had a unique name, we would create a unique index. A unique index serves two purposes:

- Provides data integrity to ensure that there is only one occurrence of any value for the column

- Informs the optimizer that there is at most one row for a given record; this is important because it reduces the need for an additional index scan

Using the SHOW STATUS command as described in Chapter 2, we can identify the internal impact of a normal index versus a unique index. Here is an example using the existing non unique index:

```
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';
```

This produces the following output:

```
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| Handler_read_next  | 0     |
+--------------------+-------+
+------+
| name |
+------+
| Enya |
+------+
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| Handler_read_next  | 1     |
+--------------------+-------+
```

Internally, MySQL had to read the next entry of the index to determine that the next

value in the name index was not also the value specified. Creating the index as unique, and running the same query, we see the following output:

```
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     |
+--------------------+-------+
```

In comparison, when using a unique index, MySQL knew that at most one row could be returned, and after finding a match no further scanning was required. It is always a good practice to define an index as a unique index when the data is indeed unique.

> ***TIP*** *It is possible to define a unique index with nullable columns. A NULL value is considered as unknown and NULL != NULL. This is an advantage of a three-state logic rather than using a default value or an empty string value.*

Ordering Results

An index can also be used to order the results. Without an index, MySQL will use an internal filesort to return rows in the order requested. Here is an example:

```
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
```

As you can see, by the Using Filesort information in the Extra attribute, MySQL internally used the sort_buffer to sort the results.

You can confirm this internally with the following:

```
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     |
+-------------------+-------+
```

By ordering the data based on the index, you eliminate sorting, as shown here:

```
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     |
+-------------------+-------+
```

In the following section we will discuss how to utilize indexes to restrict rows and return sorted results with multi column indexes.

Multi Column Indexes

It is possible for an index to have two or more columns. Multi column indexes are also known as compound or concatenated indexes.

Determining Which Index to Use

Let us look at a query that could use two different indexes on the table based on the WHERE clause restrictions. We first create these indexes.

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

It is more efficient to combine DML statements for a given table when possible. If you chose to run these ALTER statements as two individual statements, the following would occur:

```
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)
```

If this was a production size table, and an ALTER statement took 60 minutes or 6 hours, this is a significant saving.

> ***TIP*** *Creating an index is a time intensive operation and can block other operations. You can combine creating multiple indexes on a given table with a single ALTER statement.*

```
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
```

> ***NOTE*** *Depending on which version of MySQL you use, optimizer improvements can provide a different QEP in this example. You can generally guarantee the same*

output of the following examples in all MySQL 5.x versions with the following MySQL backward compatibility system variable setting:

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

However, if we run the same query for a different value for the album type, we now use a different index:

```
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
```

Why did MySQL make this decision? From the EXPLAIN rows column we draw a conclusion that the cost based optimizer determines a significantly less cost—that is, 58K rows to read compared with 154K rows to read.

```
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
…
```

If MySQL only used the index cardinality, then you would expect the QEP always to use the country_id column as this is generally more unique and would therefore retrieve less rows. Although the index cardinality is an indicator of uniqueness, MySQL also holds

additional statistics regarding the range and volume of unique values. We can confirm some of these numbers by looking at the actual table distribution.

```
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 |
+----------+
```

In the first query, the country_id index was selected. The actual results show 92K rows compared with 289K rows if the album_type_id was selected.

For the second query, the actual results show 92K rows compared with 111K rows, with the later actually being selected. If you compare the actual numbers with the QEP estimated rows values, you also find a reasonable discrepancy—for example the second query estimates 58K rows when there are actually 111K or almost two times more actual rows.

Multi Column Index Syntax

The syntax to create multi column indexes is identical except you specify your index across multiple columns:

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

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

Providing a Better Index

We could add a multi column index on both the country and album type columns so that the optimizer has more information. Here is an example:

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

When we rerun the SQL statement we get the following 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
```

The optimizer has now chosen to use the new index. You will also note the key_len=7. This represents a means of determining the effectiveness of the columns used by the index. We will be discussing this further in more examples.

It seemed logical to create the index in this order; however, providing your queries used both columns, you might elect to reverse the columns:

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

We look at the QEP again to find that this new index is used:

```
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
```

We can look at the index cardinality of the table to try to determine why this index was chosen:

```
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
…
```

If we looked solely at the index cardinality we would expect that the `m1` would be used because it provides a higher distribution of unique rows; however, this information does not provide sufficient details of the statistics used to determine the current executed plan.

> ***TIP*** *When you use a multi column index in an intersection table when both columns are always specified, switching the order of the columns may provide a better index.*

A multi column index serves a greater purpose than optimizing restricting rows. The leftmost columns of a multi column index can also be used effectively as if the index was on a single column. The leftmost columns can also be a great performance benefit when columns are used frequently in aggregation (that is, GROUP BY) and ordering (that is, ORDER BY).

Many Column Indexes

While indexes can contain multiple columns, there is a practical limit in the effectiveness of the index. Indexes are part of the relational model to improve performance. The index row width should be as short as practical in order to provide as many index records per index data page. The benefit is to traverse the index as quickly as possible by reading the least amount of data. You also want to keep your indexes efficient so you can maximize the use of your system memory. The EXPLAIN command provides the `key_len` and `ref` attributes to determine the column utilization of selected indexes. Here is an example:

```
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
```

As you can see the ref columns shows three constants, which match three columns of the index. The key_len of 6 can also confirm this: 1 byte for ENUM, 2 bytes for SMALLINT, 1 byte for nullability, 1 byte for ENUM, 1 byte for nullability.

If we did not restrict our query by country we would see the following:

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

This highlights that while the index was used, the additional column is not used for this query. If no other queries utilize this third column, this is an optimization to reduce the index row width.

Combining WHERE and ORDER BY

We have shown examples of how you can use an index to optimize the restriction of data rows and how you can use an index to optimize sort results. MySQL can utilize one multi column index to perform both operations.

```
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 ***************************
…
```

We can create an index that will satisfy both the WHERE clause and the ORDER BY clause:

```
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 ***************************
…
```

NOTE *The optimizer can choose to use an index for a WHERE restriction and an ORDER BY; however, the* key_len *will not reflect this.*

TIP *It can be difficult to create indexes that are used when ordering results; however, this can be a great benefit in certain applications when very frequent queries (such as hundreds per second) order the same information. The symptom of seeing* Sorting results *in a PROCESSLIST command highlights both a CPU impact and a strong candidate for considering a more optimal schema and SQL design.*

MySQL Optimizer Features

MySQL can use an index for a WHERE, ORDER BY, or GROUP BY column; however, generally MySQL will select only one index per table. Starting with MySQL 5.0, there are a small number of exceptions when the optimizer may utilize more than one index, but in earlier versions this could result in a slower query. The most commonly seen index merge

is a union between two indexes, generally found when you perform an OR operation on two high cardinality indexes. Here is an example:

```
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
```

NOTE *The optimizer_switch system variable that could enable or disable these additional options was first introduced in MySQL 5.1. For more information see http://dev.mysql.com/doc/refman/5.1/en/switchable-optimizations.html.*

A second type of index merge is the intersection of two less unique indexes, as shown here:

```
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
```

The third type of index merge is similar to a union between two indexes; however, one index must first be sorted:

```
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
```

You can find more information about these index merges at http://dev.mysql.com/doc/refman/5.5/en/index-merge-optimization.html.

As part of creating these examples, the author discovered a new case never seen before with any client queries. The following is a three index merge example:

```
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
```

TIP *You should always evaluate if a multi column index is more efficient than letting the optimizer merge indexes.*

The advantage of having multiple single column indexes or multiple multi column indexes can only be determined based on the types of queries and query volumes for your specific application. Several single column indexes of high cardinality columns combined with index merge capabilities may provide greater flexibility for very different query conditions. The performance considerations of writes may also affect the most optimal data access paths for retrieving data.

Query Hints

MySQL has a small number of query hints that can affect performance. There are hints that affect the total query and those that affect how individual table indexes are used.

Total Query Hints

All of the total query hints occur directly after the SELECT keyword. These options include SQL_CACHE, SQL_NO_CACHE, SQL_SMALL_RESULT, SQL_BIG_RESULT, SQL_BUFFER_RESULT, SQL_CALC_FOUND_ROWS, and HIGH_PRIORITY. None of these hints affect the use of any specific table index. At this time we will not be discussing any of these in more detail.

Only the STRAIGHT_JOIN query hint has an effect on how indexes are used for query execution. This hint is used to inform the optimizer to execute a query execution plan in the order the tables are specified in the query. Here is an example:

```
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
…
```

You can see in the first query the optimizer chose to join on the album table first. In the second query with the STRAIGHT_JOIN the optimizer was forced to join the names in the order the tables were specified. While the query uses an index for both tables, the second query has to process a much larger set of rows and is less efficient in this example.

Index Hints

With the exception of the STRAIGHT_JOIN query hint, all index hints are applied for each table in a join statement. You can elect to define a USE, IGNORE, or FORCE list of indexes per table. You can also elect to restrict the use of the index to the JOIN, the ORDER BY, or the GROUP BY portion of a query. After each table in your query you can specify the following syntax:

```
   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)
```

In this query, the optimizer had a choice of several indexes but chose the founded index.

In the next example, we instruct the optimizer to use a specific index:

```
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: ref
possible_keys: type
          key: type
      key_len: 1
          ref: const
         rows: 186720
        Extra: Using where
```

In this query we see the index specified was used.

We can also ask the optimizer to ignore an index:

```
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
```

You can provide multiple index names, and multiple index hints:

```
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
```

For more information see http://dev.mysql.com/doc/refman/5.5/en/index-hints.html.

The use of MySQL hints does not have an effect on changing the entire execution path, causing you then to specify multiple hints. The USE INDEX hint forces MySQL to choose from one of the specified indexes. The FORCE INDEX has the effect of influencing the cost based optimizer to prefer an index scan over a full table scan.

> ***CAUTION*** *Adding hints to SQL queries comes at a great risk. While this might help a query, over time the volume of data, for example, can change the query effectiveness. Changes in adding or revising indexes on tables will not affect a hard coded SQL statement that has specified a specific index. You should use hints only as a last resort.*

Complicated Queries

In the examples provided in this chapter, you have not seen a ten table join or more complicated queries. The rules described in this chapter and in Chapters 5 and 8 can simply be applied to more complex queries one table at a time. The same analysis tools are used; what is needed is to break down SQL statements, test and verify individual components to understand and verify the best possible optimized usage, and progressively increase your statement complexity to the required need.

The Impact of Adding Indexes

Although this chapter has shown many examples of adding indexes to optimize the performance of SQL statements, the addition of indexes comes at a significant cost.

DML Impact

Adding indexes to a table affects the performance of writes. This can be easily shown from the artist table used in this chapter. Looking at the current definition shows a large number of indexes:

```
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
```

By running a simple benchmark we can test the insert rate of the current album table with the original definition that included fewer indexes:

```
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;
```

Here are the timed results:

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

Inserting data into the table with additional indexes was four times slower. This is a simple bulk test and other factors can contribute to the slower speed; however, this provides a representative example that adding indexes to a table has a direct effect on write performance.

Duplicate Indexes

One of the easiest techniques for index optimization is to remove duplicate indexes. Although it is easy to spot an index that is identical, other common occurrences are an

index matching the primary key or indexes that are subsets of other indexes. Any index that is contained within the leftmost portion of another index is a duplicate index that will not be used. Here is an example:

```
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`)
…
```

```
  KEY `country_id` (`country_id`),
  KEY `m1` (`album_type_id`,`country_id`),
  KEY `m2` (`country_id`,`album_type_id`)
...
```

The country_id index is actually a duplicate due to the m2 index.

The Maatkit mk-duplicate-key-checker is an open source tool that can be used to identify duplicate indexes. The human verification of a desk check of your schema tables also works.

Index Usage

One of the deficiencies of MySQL instrumentation is the lack of determining the usage of an index. With the analysis of all your SQL statements, you could deduce what indexes are not used. It is important to determine what indexes are used and what indexes are not used. Indexes have a performance impact for writes and have a disk space impact that can affect your backup and recovery strategy. Indexes that are less effective can use valuable memory resources.

First released by Google in 2008, the SHOW INDEX_STATISTICS command enabled you to obtain this information in a more precise method. Various MySQL forks and variances now include this feature, but the official MySQL product does not.

For more information see http://code.google.com/p/google-mysql-tools/wiki/UserTableMonitoring.

Regardless of the tools used to determine whether an index is used or not, it is important that you analyze the effectiveness of columns defined in an index also to find portions of indexes that are ineffective.

DDL Impact

As the size of your tables grows, the impact of performance is generally affected. For example, the addition of indexes on the primary table took on average 20–30 seconds.

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

Traditionally the cost of any ALTER statement was a blocking statement as a new version of a table was created. It was possible to SELECT data, but any DML operation would then cause blocking of all statements due to standard escalation policies. When your table size is 1G or 100G, this blocking time will be significantly longer. In more recent versions there have been a number of advances, both with the MySQL product and with creative solutions.

There are some exceptions to the impact of adding indexes. With InnoDB, fast index creation features are available with the InnoDB plugin in MySQL 5.1 and by default in MySQL 5.5 or better. More information is at http://dev.mysql.com/doc/innodb/1.1/en/innodb-create-index.html.

Other storage engines also implement different ways of creating fast indexes that perform little to no locking. Tokutek is one such engine. Read more at http://tokutek.com/2011/03/hot-column-addition-and-deletion-part-i-performance/.

The impact of disk space is also an important consideration, especially if you are using the default common tablespace configuration for InnoDB. MySQL creates a copy of your table. If your table is 200GB in size, then you need at least 200GB more disk space to perform an ALTER TABLE. Using InnoDB, this additional disk space is added to the common tablespace during the operation. This is not reclaimed on the filesystem at the completion of the command. This additional space is reused internally when InnoDB requires additional disk space. Although you can switch to a per table tablespace, this has an impact on write intensive systems.

> ***TIP*** *There are also various techniques to minimize this blocking operation. You can elect to use a high availability master/fail-over master replication topology to support online alters. More recently Shlomi Noach introduced the oak-online-alter-table utility. See information at http://code.openark.org/blog/mysql/online-alter-table-now-available-in-openark-kit. Facebook also released its online schema change (OSC) tool that performs in a similar fashion. More information is at http://www.facebook.com/notes/mysql-at-facebook/online-schema-change-for-mysql/430801045932.*

Disk Space Impact

Using the INFORMATION_SCHEMA.TABLES query from Chapter 2 we can obtain the size of the album table used in this chapter.

Before adding indexes:

```
*************************** 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
```

After adding indexes:

```
*************************** 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
```

After adding indexes:

You can see a 7 times increase in the amount of index space used for this table. Depending on your backup and recovery procedures, this is a direct impact on the increased time for both processes. Adding indexes will have a direct impact in other areas. What is important is that you understand and consider the impact before adding indexes.

The use of InnoDB can also have a direct effect on the size of disk space with the choice of primary key and how that primary key is used. Secondary indexes always have the primary key appended to each secondary index record. Therefore, it is important to use as small a primary key data type as possible for InnoDB tables.

There is an exception when a greater disk footprint can be of longer term performance benefit. In cases of extreme table size (such as hundreds of gigabytes), an ordered primary key that is not a sequential key might produce more sequential disk activity if all queries use the primary key order. Although the fill factor causes a greater data size, the overall time of a highly concurrent system that retrieves large numbers of rows by the primary key order can result in more even disk performance and overall query performance. This is very rare example that highlights that detailed monitoring and suitable production volume testing is necessary to look at long term benefits in overall performance.

Page Fill Factor

Your choice of a natural primary key over a surrogate primary key has a direct influence of your default page fill factor. For a surrogate primary key, InnoDB will fill data pages when inserting new data to a 15/16th volume as the order is naturally increasing. When the primary key is a natural key, InnoDB tries to minimize reorganization of the data with page splitting when inserting new data. Generally InnoDB will fill data pages only to 50 percent initially. This results in a naturally larger disk footprint, and when the data volume exceeds the allocated memory to the InnoDB Buffer Pool, packing more data into 16K data pages might provide performance improvements. Chapter 3 provides an example of the disk size that occurs due to the sequential and natural fill factors.

Secondary Indexes

The internal implementation of the B-tree secondary index in InnoDB has a significant difference from a MyISAM B-tree secondary index. InnoDB uses the primary key value within a secondary index, not a pointer to the primary key. A copy of the applicable primary key is appended to each index record. When your database table has a primary key length of 40 bytes, and you have 15 indexes, the index size can be dramatically reduced by introducing a shorter primary key. This primary key value implementation has performance benefits combined with the use of internal InnoDB primary key hash.

MySQL Limitations

There are several limitations to how indexes are used and managed in MySQL in comparison to other RDBMS products.

Cost Based Optimizer

MySQL uses a cost based optimizer to prune the possible query tree to create the most optimal SQL execution path. MySQL has limited capabilities for using generated statistics to aid the optimizer as described. MySQL supports a small number of index hints to assist the optimizer in choosing a suitable path.

QEP Pinning

MySQL does not support the capability of pinning a specific QEP for a particular given query. There is no means of defining the QEP for a query where the data may change over time, affecting the possible QEP chosen. This results in the QEP being determined for every execution of every query.

Index Statistics

MySQL supports limited index statistics, and these vary depending on the storage engine. Using the MyISAM storage engine, the ANALYZE TABLE command generates statistics for the table. There is no way to specify a sampling amount. The InnoDB storage engine performs random sampling of data pages to generate statistics for a given table when the table is first opened, and then by a fuzzy method with a hard coded percentage change of the table rows.

The current development version of MySQL 5.6 includes the ability to save InnoDB statistics.

Function Based Indexes

MySQL does not currently support function based indexes. In addition, using functions on existing indexes results in sub-optimal performance. MySQL does support a partial column index, which is effectively a left substring. This is discussed in greater detail in Chapter 5.

You also cannot specify a reverse order of an index, as all data is effectively in ascending order. MySQL will traverse an existing index in reverse order if the DESC predicate is specified when ordering data.

Multiple Indexes per Table

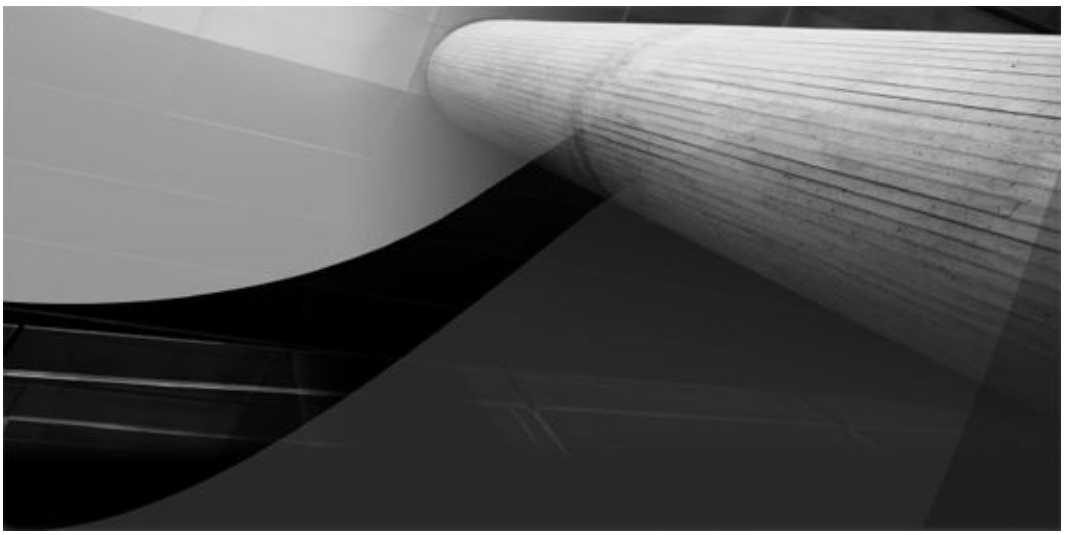
As described in this chapter, by default MySQL will use only one index per table. While there are five exceptions, it is a good practice to realize this limitation in the design of your tables, indexes, and SQL statements. Further improvements in the MySQL optimizer will help improve on this limitation in the future.

Conclusion

Although the principles of index creation are simple, deciding the right columns on which to create indexes, and determining when the impact on write performance outweighs the read improvements an index can produce, is not always easy. Regardless of the complexity of your SQL statement, you can always reduce and distill an SQL statement down to smaller working examples containing a subset of tables and then analyze these smaller components individually, building your SQL statement one join at a time.

In this chapter we have focused on how to create indexes in an existing schema and with existing SQL statements. We have not discussed the importance of structuring your SQL statements to utilize indexes the best possible way. Altering how you write an SQL statement with the columns you select, how you join tables, and what the order of the results are can all have an important effect on optimizing SQL statements.

> ***NOTE*** *In a large read scale out environment MySQL replication can provide a significant benefit to read performance on MySQL slaves by spreading the read load. It is possible to utilize the same database schema and have different indexing strategies on different servers that can be more optimized for one type of query over a different query. Although you may think you have to find the right balance of indexes for your database queries, you can also leverage your read and write scalability models for independent benefits.*



5

Creating Better MySQL Indexes

Creating appropriate indexes is an important optimization technique. In Chapter 4 we discussed the basics of creating MySQL indexes to improve performance for various types of queries. Creating multi column indexes can provide a significant improvement over a single column index. Two more indexing techniques can further improve query performance.

In this chapter we will be covering these additional indexing techniques:

- Creating a covering index
- Creating a partial column index

We will be using the table and data examples as provided in Chapter 4. It is recommended that the example tables and data are repopulated.

> ***CAUTION*** *As described in previous chapters, the MySQL storage engine in use can impact how you define an optimal index. Our example tables use the InnoDB transactional storage engine.*

Better Indexes

Improving the performance of queries with indexes by reducing query execution from seconds to milliseconds can result in a large boost in your application performance. Fine tuning your indexes for optimal usage can be just as important, especially for high throughput applications. The improvement of a few milliseconds for queries that execute 1000 times per second can also have a significant impact on performance. For example, saving 4 milliseconds (ms) on a 20 ms query that executes 1000 queries per second is just as important for optimizing SQL statements.

We will take the approach as described in the previous chapter for creating multi column indexes and build on this technique and create a better covering index.

Covering Index

If we wanted to select the names of all artists that were founded in 1969, we could run the

following query:

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

Our sample database is rather small, with this table having only approximately 500,000 rows; however, we can still demonstrate the impact of improving the indexes.

Without an index, this query takes 190 ms to run. Looking at the query execution plan we would observe a full table scan (not shown here). A recommended improvement would be to add an index. Here is an example:

```
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
```

With an index added on the founded column that is used in the WHERE condition, the query now takes 5.9 ms to execute. This single change has made this query execute 97 percent faster than the original query. However, it is still possible to create an index that performs this query faster:

```
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
```

With a multi column index, the query now takes 1.2 ms to run. This is 4 times faster, and the query execution time now an 80 percent reduction from our first improvement. We have managed overall to produce a 99 percent saving in total execution time.

Although this improvement is due to the multi column index, the true benefit is not because of the additional column restricting the number of rows. Using the analysis

techniques from Chapter 4, we can see that only 2 bytes of this multi column index were used. You might consider the additional columns in the multi column index to be ineffective, but notice that in the Extra column you see Using index.

When the Query Execution Plan (QEP) shows the information Using index in the Extra column, this does not indicate that an index is being used to access the underlying table data. This indicates that only the index is being used to satisfy the query requirements. This can have a significant improvement on larger or more frequent queries and is known as a covering *index*.

This is a covering index because all columns for the given table in the query are satisfied by the index for the specific table. For a covering index to work, all columns in the WHERE clause, ORDER BY and GROUP BY if present, and also all SELECT columns must be contained within the index for a given table.

You might be wondering why the artist_id is not in the index based on the preceding description. For those who elected to skip Chapter 3, "Understanding MySQL Indexes" and its description of the theory of indexes, you would have read about an important feature of InnoDB secondary indexes. In InnoDB, the value of the primary key is appended to each record in a secondary index, and therefore it is not necessary to specify the primary key in InnoDB secondary indexes. This important fact means that the artist_id column is implied in all secondary indexes for the InnoDB storage engine. Adding the primary key as the last element of an index for an InnoDB table is a common observation for tables that have been converted from the MyISAM storage engine.

> ***TIP*** *There are many reasons not to use SELECT * for SQL queries. This is one example where only selecting the columns truly required can enable better SQL optimization by creating appropriate indexes.*

Our new index, however, is good only for the particular query defined. If we wanted to restrict our query further by adding a particular type of artist, the following occurs:

```
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
```

You can see we are using the same index, but now we do not have the benefit of a

covering index. This query now takes 5.4 ms to execute. We could adjust the index based on this new column information with the following statement:

```
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
```

We see that the revised index is used. We confirm the type column is now part of the optimizer limitations with a key_len value of 3, and we see that founded_type_name is a covering index. The query is also now running at 1.3 ms.

> ***CAUTION*** *These indexes are created to describe the process of identifying covering indexes and might not be ideal in a production environment. Due to the limited dataset, a large character column was used in these examples. As your data volume grows, especially exceeding a memory/disk boundary, indexing large columns can have an impact on overall system performance. Covering indexes can be ideal for a large normalized schema with many small width primary and foreign keys used.*

Storage Engine Implications

Chapter 3 highlighted that for InnoDB secondary indexes, the actual value of the primary key was used in the index, not a pointer to the underlying data row. MyISAM uses a different implementation of a B-tree index, and the following example highlights this difference:

```
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
```

This query does not use the full extent of the index shown in the previous query execution plan. For reference, this MyISAM query benchmarks at approximately 3.3 ms, which is slower than the optimized InnoDB statement, but faster than the non-covering index example. To confirm this important underlying architectural difference, we can modify the index to satisfy the needs of the MyISAM engine:

```
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
```

> ***NOTE*** *In this book we will not be discussing the architectural considerations and relative merits of why you should choose one storage engine over another. The Effective MySQL web site at http://EffectiveMySQL.com offers valuable information for optimal data architecture. This does highlight that benchmarking is an important task in the optimizations of SQL statements.*

For ongoing tests, we will reset the storage engine of the table:

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

Partial Index

Although an index can be used to restrict the rows you need to select, if MySQL has to retrieve additional column data for a large number of rows, it can be more efficient to create a smaller index with a more compact row width. As shown with creating covering indexes, the benefit of using an index to perform a lot more work can significantly improve the performance of an SQL query. When your data size exceeds your physical memory resources, your choice of what to index should consider physical resources, not just an optimal query execution plan.

This INFORMATION_SCHEMA query reviews the size of table data and index space in the following examples:

```
$ 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
```

We will start by removing the existing index for albums:

```
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
```

The album table should have only a primary key defined. We can determine the size of the table and index space with the defined INFORMATION_SCHEMA.TABLES query:

```
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
```

The table has no index size, because the primary key of an InnoDB table is a clustered index and is combined with the underlying data. If this table were using the MyISAM storage engine, we would see the following information highlighting the size of data and indexes:

```
*************************** 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
```

We are now going to add an index on the name column for albums:

```
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
```

The new index on name uses approximately 30MB of disk space. Let us now create a more compact index for the name column:

```
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
   avg_row: 63
  total_mb: 57.14
   data_mb: 33.56
  index_mb: 23.58
```

A more compact version of the index now takes only 23MB, a 20 percent savings. Although this might seem small, the sample table contains only 500,000 rows. If the table contained 500 million rows, the saving would be 6GB.

The primary consideration here is to reduce the size of the index. A smaller index means less disk I/O, which in turn means faster access to the necessary rows, especially when the index and data volume on disk is far greater than configured system memory. The performance gain can outweigh the impact of a now non unique index that may have a smaller cardinality.

The application of using a partial index will depend on how the data is accessed. In the previous section for a covering index, you saw that choosing to record a shorter version of the name column would provide no benefit for the executed SQL statement. The greatest

benefit will occur if you restrict rows on the indexed column. Here is an example:

```
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
```

In this example there is no added benefit in recording the full name in the index. The provided partial column satisfies the WHERE condition. Choosing the correct length depends on the data distribution and access path. Determining how to calculate an appropriate length for the index is not an exact science. A comparison on the number of unique values for a given range of column lengths showing the diminishing rate of return for uniqueness for shorter lengths combined with the values used in actual queries would be necessary.

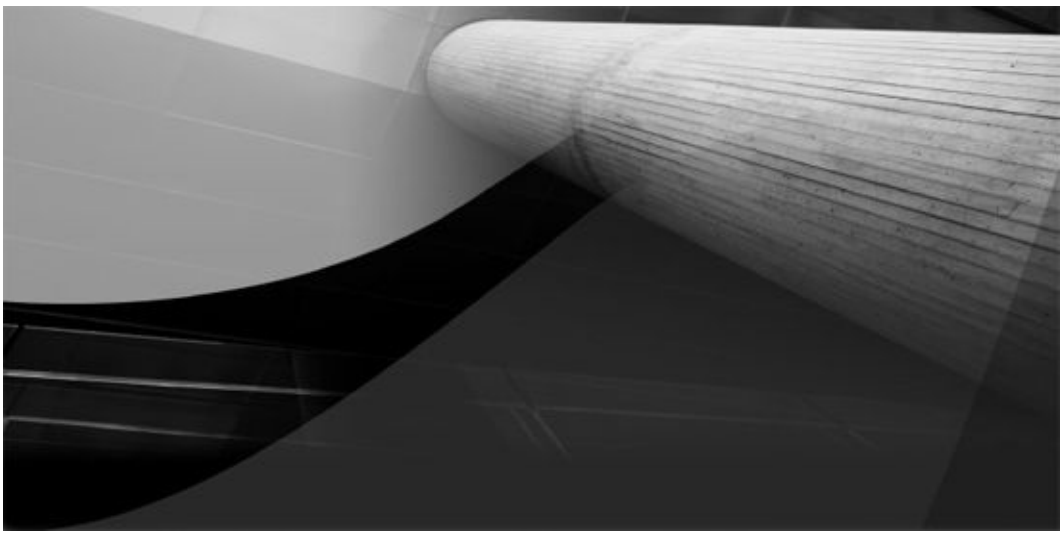
Conclusion

The correct column definitions combined with the order and placement of columns within an index can change the way an index is used effectively. An index can make a large improvement for a slow query. An index can also reduce very fast queries by a few milliseconds. In a high concurrency, a few milliseconds for 1 million queries can greatly improve performance and enable a greater capacity for scalability. Creating the most optimal indexes for SQL queries can be considered as much an art as a process.

Writing good SQL statements is also important for query optimization and to maximize current or future indexes. We will be discussing more about improving SQL queries in Chapter 8.

The Effective MySQL web site includes a presentation that provides detailed examples in how to identify and create better indexes. For more information see http://effectivemysql.com/presentation/improving-performance-with-better-indexes/.

You can download all SQL statements for this chapter at http://effectivemysql.com/book/optimizing-sql-statements.



6

MySQL Configuration Options

MySQL 5.5 supports more than 300 configurable system variables. Many system variables can have a significant effect on overall system performance when correctly tuned. System variables help to define how MySQL is configured, where important files and data reside, how to manage compatibility, interaction within a MySQL topology, and much more. In this chapter and for this book on optimizing SQL statements, we will focus on a small number of MySQL system variables that have a direct impact on the execution and optimization of individual SQL statements.

In this chapter we will discuss the following:

- Memory related system variables
- Logging and instrumentation system variables
- Miscellaneous query related system variables

Memory Related Variables

MySQL memory system variables can affect the global memory usage or the session memory usage of the multi-threaded single MySQL process. We will not discuss in this book the importance of system tuning for optimal memory usage in MySQL. It is important to note that MySQL has an unbounded process allocation and an unbounded memory storage engine allocation. Incorrect tuning can result in your system exhausting available memory resources.

These MySQL system variables can be defined in the MySQL configuration file at startup. Many MySQL variables are also dynamic—that is, they can be modified during runtime with the mysql client SET command. You should refer to the MySQL Reference Manual for your specific version for a full definition of those variables that are dynamic.

> ***NOTE*** *In MySQL, a reference to a session is relative to a given connection. By default, a connection is also a thread—that is, one connection has one and only one thread. As early as MySQL 5.0, custom commercial supported versions of MySQL supported connection thread decoupling where a pool of threads manage all MySQL connections. This has never been deployed into the mainstream version of MySQL; however, the system variables* thread_handling *and* thread_pool_size *expose information about this feature.*

NOTE *For Oracle readers, the global memory buffers can be considered the system global area (SGA) of the MySQL process. These buffers are preassigned when the mysqld process starts. The per session memory buffers and memory tables can be considered part of the process global area (PGA) of the running mysqld process.*

Unless otherwise specified, all of the following variables are available from MySQL 5.0. A number of these variables exist in the earliest versions of MySQL including 3.*x* and 4.*x*.

Global Memory Buffers

MySQL Variable Name	Description
key_buffer_size	This variable defines the size of the MyISAM index key buffer. This is often referred to as the key cache. It is possible to define multiple named key buffers of varying sizes.
innodb_buffer_pool_size	This variable defines the size of the InnoDB buffer pool that primarily holds InnoDB data and index pages.
innodb_additional_mem_pool_size	This variable defines the size of the InnoDB data dictionary and internal data structures buffer.
query_cache_size	This dynamic variable defines the size of the query cache that can be used to cache commonly executed SELECT statements.

Global/Session Memory Buffers

max_heap_table_size	This dynamic variable defines the maximum size of a MEMORY storage engine table.
tmp_table_size	This dynamic variable defines the maximum size of an internal memory based temporary table before the table is altered to write to disk. This variable is closely related to max_heap_table_size.

Session Buffers

join_buffer_size	This dynamic variable defines the memory buffer size for full table joins where a join condition cannot be met by indexes for a join between two tables.

sort_buffer_size	This dynamic variable defines the memory buffer size for sorting results when there is no index to satisfy the ordering of information.
read_buffer_size	This dynamic variable defines the memory buffer size for sequential data scans.
read_rnd_buffer_size	This dynamic variable defines the memory buffer size for ordered (that is, nonsequential) data scans.

CAUTION *A common problem is that administrators do not realize these four buffers are defined on a per-thread basis. The default values can vary from 128K/256K to 2M for the sort_buffer_size in MySQL 5.1 or better. When a buffer is defined as 10M or 100M, this has an adverse effect on query and system performance. Without evidence supporting specific performance improvements, your best approach is to reset these four variables to default values to maximize overall memory utilization.*

You can find more information regarding these variables and others in the MySQL Reference Manual at http://dev.mysql.com/doc/refman/5.5/en/memory-use.html.

key_buffer_size

The key_buffer_size is a global memory buffer that holds MyISAM index information only. Index data from a corresponding .MYI file is read from disk and stored in this buffer. Sizing the key_buffer_size variable can be as simple as calculating the total index size for all MyISAM tables and adjusting accordingly over time as your data grows.

When there is insufficient space in the key buffer to hold new index data, older pages are flushed appropriately via a least recently used (LRU) method.

NOTE *When running an all InnoDB schema you still need to define a key buffer as MySQL meta information is defined as MyISAM. A small configuration could be defined with a value of 8M.*

A poorly tuned MyISAM key buffer can manifest as different states when looking at individual MySQL connections. In the SHOW [FULL] PROCESSLIST State column, the value Repairing the keycache is a clear indicator that the current key buffer size is insufficient to perform a currently running SQL statement. This can result in excessive disk I/O activity.

For more advanced tuning, see also the key_cache_age_threshold, key_cache_block_size, key_cache_division_limit, and preload_buffer_size variables.

Named Key Buffers

In addition to the default MyISAM key buffer, MySQL supports additional named key buffers. This enables the administrator to define dedicated memory pools for MyISAM indexes that you can assign and pin particular tables to a named key buffer. For example, here is how to create a named 64M cache:

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

You can also define this within the MySQL configuration file my.cnf.

> ***CAUTION*** *In the MySQL configuration file, you can define a variable size as bytes or with K,M,G common format. When defining the same variable dynamically with the SET command, only a numeric bytes value is valid. The syntax 1024 * 1024 * 64 is used to translate easily from bytes to 1K, 1M, and 64M and is not a requirement.*

To assign table indexes to a specific named cache, you use the CACHE INDEX statement:

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

Information about which tables to cache to a named cache is not persisted. During a MySQL restart, this reference is lost. To achieve this functionality for a system restart, use the init_file variable to run specific SQL statements when the server starts.

Additional performance can be achieved when you pre-populate key buffers with all indexes of a given table when the key cache can hold the entire indexes or index without leaf node pages. Here is an example:

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

You can use this syntax to load indexes into any key cache, including the default key cache.

innodb_buffer_pool_size

The innodb_buffer_pool_size is a global memory buffer for all InnoDB data and indexes. For an InnoDB only schema, this is the most important buffer to assign correctly. Incorrectly assigning this buffer can result in excessive disk I/O and very slow query performance.

For smaller systems, you can use the total size of all InnoDB tables plus a small overhead to calculate a suitable size for this buffer. This is important if you have a small finite amount of RAM. For example, if you assigned a 1GB InnoDB buffer on an Amazon Web Services small instance that supported 1.7G of RAM when the total database size was 200MB, you would be assigning unnecessary memory.

There is no best estimate for sizing this variable, because many other factors contribute to the total amount of memory MySQL uses. A popular myth is to size this at 80 percent of RAM, but this can be an inappropriate calculation for many reasons.

Monitoring the usage of the InnoDB buffer pool is possible with the SHOW GLOBAL STATUS or SHOW ENGINE INNODB STATUS command. Here is an example:

```
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]
…
```

There are more than 50 InnoDB related system variables. All these variables are prefixed with innodb_. Several InnoDB variables that can affect overall system performance and should be considered when defining the innodb_key_buffer_size variable include innodb_buffer_pool_instances (since 5.5), innodb_max_dirty_pages_pct, innodb_thread_concurrency, innodb_spin_wait_delay, innodb_purge_threads (since 5.5), innodb_checksums, innodb_file_per_table, and innodb_log_file_size.

innodb_additional_mem_pool_size

The innodb_additional_mem_pool_size variable defines the memory pool for InnoDB specific data dictionary information. There is little instrumentation to determine an optimal value for this variable. A common value for this variable is 10M. The default value in MySQL 5.5 is 8M. In earlier versions, the default value was 1M. MySQL will report a warning message in the error log if this buffer is not a sufficient size.

query_cache_size

The query_cache_size is a global memory buffer for holding frequently cached queries. When enabled with the query_cache_type variable, SELECT queries are automatically cached. You can further control what statements are cached using the SQL_CACHE and SQL_NO_CACHE query hints. Enabling the query cache can be a large performance improvement in a high read environment; however, it can also slow down overall performance in a read/write environment.

The overall enabling and disabling of the query cache is done with the query_cache_type variable. When enabled, a query_cache_size of 0 is possible, which indicates that no queries are to be cached; however, the MySQL instance may still support caching at some time by dynamically changing the query_cache_size value. A small performance overhead can be gained by physically disabling the query cache if this is not required.

To enable query cache dynamically, use the following statements:

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

To disable the query cache dynamically, use this:

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

You can monitor the effectiveness of the query cache with the SHOW GLOBAL STATUS command or the INFORMATION_SCHEMA.GLOBAL_STATUS table. Here is an example:

```
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      |
+-------------------------+------------+
```

You can determine the query cache effectiveness with the following equation:

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

It is important to consider the Qcache_queries_in_cache status variable when determining read/write volume. This should be combined with the Com_select status variable to determine the total reads satisfied for your MySQL instance.

The query cache can be further tuned with a number of different system variables that affect the allocation of individual queries. These include the query_cache_limit, query_case_min_res_unit, query_cache_alloc_block_size, query_cache_wlock_invalidate, and query_prealloc_size.

max_heap_table_size

This defines the maximum size of a MySQL MEMORY storage engine table. When the maximum size is exceeded for an individual table, the application will receive the following message:

```
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
```

This variable has a default global value and can also be specified on a per-thread basis as shown in the above example. There is no limit to the total size of all MEMORY tables. This variable applies only to individual tables.

The total size of a MEMORY storage engine table can be determined by the SHOW TABLE STATUS command and the INFORMATION_SCHEMA.TABLES table.

For additional reference, a MEMORY storage engine table used a fixed width value for any character columns. In the preceding example, the VARCHAR(1000) column is effectively a CHAR(1000) column when defined in a MEMORY table. Given this constraint, a MEMORY table cannot contain any TEXT (aka CLOB) or BLOB fields. This is also important for the tmp_table_size variable.

tmp_table_size

The minimum of max_heap_table_size and tmp_table_size defines the maximum size for an internal temporary table used in query execution that can be stored in memory. You can recognize if a temporary table is being used during query execution if you see Using temporary appear in the Extra column of an EXPLAIN SELECT output. One SQL query can

use many temporary tables.

MySQL uses the MEMORY storage engine for these internal temporary tables. When the internal temporary table exceeds the minimum of tmp_table_size and max_heap_table_size, a MyISAM disk based table is created in the tmpdir location. Due to the limitation of the MEMORY storage engine, any TEXT or BLOB columns defined for the temporary table will result in a MyISAM disk based temporary table. The definition of large variable character data types can also cause a disk based temporary table as these are converted to fixed width fields.

There is no easy way to determine the total size of an internal temporary table. The MySQL status variables can confirm the creation of a temporary table or disk based temporary tables with the created_tmp_tables and created_tmp_disk_tables variables, respectively. Here is an example:

```
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     |
+-------------------------+-------+
```

NOTE *The act of measuring MySQL status variables affects the returned results. For example, executing a SHOW STATUS command generates a temporary table. The values also change between MySQL versions and global and session scope. You should always test the impact of measurement.*

Your understanding of how MySQL internal temporary tables operate can be very important to system performance. For example, too many concurrent internal temporary tables that exceed this size written to disk can manifest as the following error:

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

This is the result of the disk location defined by the tmpdir variable being full. There is no way to repair a temporary disk table as referenced in this error message. In MySQL 5.5, you can use the PERFORMANCE_SCHEMA to assist in the true calculation of the size of disk based temporary tables.

The tmp_table_size variable has a default global value and can also be specified on a per-thread basis, which may be of benefit for queries that generate very large temporary tables.

You can find some additional information about how MySQL uses internal temporary tables in the MySQL Reference Manual at http://dev.mysql.com/doc/refman/5.5/en/internal-temporary-tables.html.

join_buffer_size

The join_buffer_size is a per-thread memory buffer that is used when a query must join two sets of table data and no index is used. This buffer is for each non-index-bound join per thread. This can be confirmed in a query execution plan where you see Using join buffer in the Extra column. It is recommended you leave this buffer at the default size. Increasing the size will not provide a faster full join. Setting this variable to a large size can have a memory impact on a high connection environment.

sort_buffer_size

This variable is a per-thread buffer that is used when sorting the result set. This can be confirmed in a query execution plan when you see Using filesort in the Extra column. It is not recommended you increase this buffer as it is fully assigned per request and can slow down queries when a default value is too large.

read_buffer_size

This buffer is used when an SQL query performs a sequential scan of table data. Increasing this buffer is applicable only for large sequential table data scans.

read_rnd_buffer_size

This buffer is used to hold data that is read as the result of a sorting operation. This differs from the read_buffer_size that reads data sequential based on how this is stored on disk. Increasing this buffer is applicable only when performing large ORDER BY statements.

Instrumentation Related Variables

slow_query_log (since 5.1	This dynamic boolean variable determines whether slow running queries are logged.
slow_query_log_file (since 5.1) log_slow_queries (for 5.0 and earlier;	This dynamic variable specifies the file name for the

deprecated in 5.1)	slow query log when output is logged to a file.
long_query_time	This dynamic variable specifies the maximum execution time of queries before they are determined to be slow.
general_log (since 5.1)	This dynamic boolean variable determines whether all database queries are logged.
general_log_file (since 5.1) log (for 5.0 and earlier; deprecated in 5.1)	This dynamic variable specifies the file name for the general log when the output is logged to a file.
log_output (since 5.1)	This dynamic variable defines the destination type for slow log and general log output.
profiling	This dynamic variable defines statement profiling per thread.

slow_query_log

This boolean variable enables the slow query log, which will report any queries that exceed the value of long_query_time. It is highly recommended this option is always defined and the output is analyzed frequently for long running queries that can be optimized. This is a global variable that can be defined dynamically.

> ***TIP*** *The slow and general query logs are dynamic with the slow_query_log and general_log system variables, respectively. In MySQL 5.0 and earlier, these logs were not dynamic and would require a MySQL instance restart to enable and disable, respectively.*

slow_query_log_file

This variable defines the file name for all queries that are logged when the slow query log is enabled. This is a global variable that can be defined dynamically.

general_log

This variable enables the general query log that reports every SQL statement that is executed. This variable can be enabled or disabled on a per server instance value only. This is a global variable that can be defined dynamically.

general_log_file

This variable defines the file name for all SQL queries logged when the general log is enabled. This is a global variable that can be defined dynamically.

long_query_time

This variable specifies a limit for queries exceeding this time to be logged to the slow query log when enabled. Starting with MySQL 5.1.21, this variable accepts a value in milliseconds. When specifying a log output of TABLE, this variable supports values only in units of seconds. This is a global variable that can be defined dynamically.

log_output

This variable defines the output location for the slow query and general query logs. The valid options are FILE, TABLE, or NONE. When defined as FILE the output file will be as defined by slow_query_log_file and general_log_file system variables, respectively. If defined as TABLE, the output will be logged to the mysql.slow_log and mysql.general_log tables, respectively. These two tables are internally defined as a CSV storage engine, which does not support any indexes. When querying these tables frequently, it is recommended that you create a copy of the table and optimize your queries accordingly. This is a global variable that can be defined dynamically.

Profiling

MySQL exposes detailed internal information for statement execution with profiling. This boolean session variable enables detailed statement profiling and can be ideal for determining accurate timing for SQL execution to microsecond precision. Also, it can identify timing for key internal components and steps in the query execution process.

You can control the number of SQL statements in the per-thread profile buffer with the profiling_history_size variable.

Other Optimization Variables

optimizer_switch (since 5.1)	This variable determines which advanced index merge capabilities are enabled in the MySQL optimizer.
default_storage_engine	This variable defines the storage engine for tables when no engine is specified.
max_allowed_packet	This variable defines the maximum size of the result set.
sql_mode	This variable defines the different server SQL modes that are supported.

innodb_strict_mode (since 5.1)	This variable defines a level of server SQL mode specifically for the InnoDB plugin.

optimizer_switch

This option defines a number of advanced switches for the MySQL query optimizer that can be disabled (enabled by default) for the three different index merges conditions and for engine pushdown conditions.

See also the max_seeks_for_key, optimizer_prune_level, optimizer_search_depth, and engine_condition_pushdown system variables.

default_storage_engine

This variable specifies the storage engine for the CREATE TABLE command when no ENGINE value is specified. In MySQL 5.5, the default storage engine changed from MyISAM to InnoDB. Any historical applications can experience issues if the expectation is to use MyISAM and InnoDB is now the default.

max_allowed_packet

You can define the maximum size of an SQL query result with the max_allowed_packet variable. Increasing this value will allow much larger result sets to be returned. By limiting the size, you have a possible indicator of large running SQL queries that might be inefficient.

sql_mode

MySQL supports a number of different server SQL modes. These modes vary greatly and can provide a more ANSI compliant SQL syntax, provide default expected data validation, and support different management for null and zero dates. These settings can be important as they can change the impact of how an SQL statement operates.

innodb_strict_mode

This variable provides sql_mode capabilities for the InnoDB plugin, which was optional in MySQL 5.1 and is now the default InnoDB implementation in MySQL 5.5. These options extend the native sql_mode to include management of table objects and row checking.

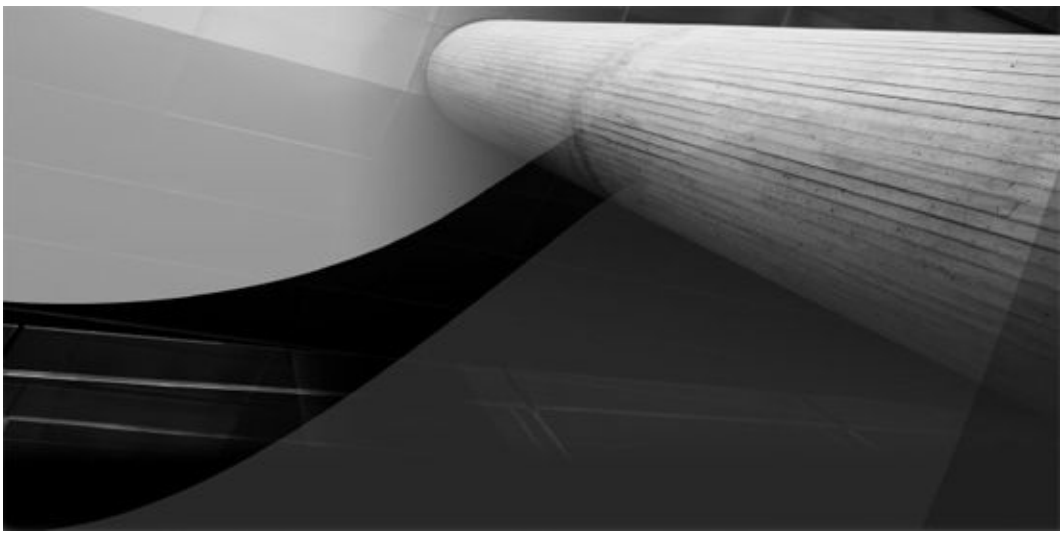
Other Variables

MySQL system variables that are not discussed in this chapter that you might consider for more in-depth analysis include the following:

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	

Conclusion

This chapter gives a detailed account of the most common and applicable MySQL system variables that relate to the primary purpose of this book for general SQL optimization. There are many additional variables, especially those that can affect DML performance and optimizations for the popular InnoDB storage engine. Please check the MySQL Reference Manual for a detailed list.



7

The SQL Lifecycle

The lifecycle of optimizing SQL statements involves six distinct steps. These include the essential prerequisite steps of how you capture SQL statements, how you identify problem SQL statements, and how you confirm your SQL statement before you begin analysis.

In this chapter we discuss the following six steps of the full SQL optimization lifecycle:

- Capture SQL statements
- Identify and categorize problematic SQL statements
- Confirm current operation of SQL statements
- Analyze your SQL statement and supporting information
- Optimize your SQL statement
- Verify the results of SQL optimization

Capture Statements

Before you can review and optimize SQL statements, you need to collect possible SQL statements to review. You can do so using a sampling process that collects a portion of all executed statements, or by a detailed collection process that collects all statements executed during a period of time. The following shows the various popular capture techniques in MySQL:

- General query log
- Slow query log
- Binary log
- Processlist
- Engine status

- MySQL connectors
- Application code
- INFORMATION_SCHEMA
- PERFORMANCE_SCHEMA
- SQL Statement Statistics plugin
- MySQL proxy
- TCP/IP

Note that this is not an exhaustive list of all possible SQL capture techniques.

General Query Log

The MySQL general query log allows you to capture all SQL statements that are executed on the database instance. As of version 5.1, this can be configured to write to a file or to a database table. There is no granularity other than on or off. The general query log is enabled with the following MySQL configuration settings:

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

> ***NOTE*** *These configuration options are valid for MySQL 5.1 or later. For MySQL 5.0 or earlier, refer to the MySQL Reference Manual at http://dev.mysql.com/doc/refman/5.0/en/query-log.html.*

It is also possible to dynamically enable and disable the general log via SQL and to log output to a table, a file, or both. Here is an example:

```
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
…
```

NOTE *The general query log is ideal for providing a sequential order of all SQL statements; however, it does not provide execution timing. The enabling of the general query log for development and low volume testing environments is an ideal way to review SQL statements, but the general query log should never be enabled in a production environment.*

Slow Query Log

The MySQL slow query log allows you to capture those SQL statements that are executed for a given database instance that exceed a given amount of real time. The slow query log is enabled with the following MySQL configuration settings:

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

NOTE *These configuration options are valid for MySQL 5.1 or later. For MySQL 5.0 or earlier, refer to the MySQL Reference Manual at http://dev.mysql.com/doc/refman/5.0/en/slow-query-log.html.*

Starting with MySQL 5.1, is it possible to specify a long_query_time in microseconds; however, this option is supported only for file output. You can easily test the validity of the slow query log with the SLEEP() function, which also supports microseconds:

```
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);
```

As you can see in the output, the first SELECT SLEEP(0.1) statement is not recorded in the slow query log, as this is under the long_query_time threshold.

Binary Log

The MySQL binary log contains all non SELECT statements including all DML and DDL commands. This can be used to provide historical analysis of the volume of statements with a table level granularity. The capture of UPDATE and DELETE statements can shed light on possible index optimizations. The binary log is enabled with the following MySQL configuration settings:

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

MySQL provides the mysqlbinlog command to create a text version of the information within the MySQL binary log. The generated output includes a lot of additional verbose information that is not relevant when looking at DML statements only. Here is an example:

```
$ 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
…
```

NOTE *Starting with MySQL 5.1, you can specify the ROW or MIXED binary log format. The analysis of binary log SQL statements is possible only when the default STATEMENT format is used.*

Processlist

This sampling process can find long running or common queries that are currently executing. In addition, this command can give an insight into the internal step of the executing SQL statement. This is a quick means of confirming long running DML statements that are blocking other SQL statements. These are determined by a State value of Locked.

```
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 ...
----+----------+-----+---------+------+--------------------+---------------------
```

This output has been modified for display purposes.

You can also obtain processlist information from the command line mysqladmin tool with the following syntax:

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

You can also obtain this information from the INFORMATION_SCHEMA (since 5.1) using the following SQL statement:

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

Engine Status

The storage engine specific SHOW ENGINE [engine] STATUS command can provide some additional SQL information. The only default included MySQL storage engine that uses this command is InnoDB. This is not an ideal source of SQL statements, as it primarily displays other system metrics; however, this is included for completeness as this is the only means of displaying SQL statements that have occurred due to certain types of errors.

The SHOW ENGINE INNODB STATUS command can provide details of SQL statements including those running and those that have caused foreign key validation failure or a deadlock. Here is an example:

```
----
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' ….
….
```

You can also glean SQL statements from the LATEST DETECTED DEADLOCK and the FOREIGN KEY ERROR sections of the SHOW ENGINE INNODB STATUS output.

MySQL Connectors

Various MySQL connectors provide easy access to capture SQL statements. These include the following:

Connector/J

Refer to the following Connector/J datasource configuration properties:

- logSlowQueries
- slowQueryThresholdMillis
- useNanosForElapsedTime
- slowQueryThresholdNanos
- autoSlowLog

For more information regarding these properties, see http://dev.mysql.com/doc/refman/5.5/en/connector-j-reference-configuration-properties.html.

PHP mysqlnd

This new native driver for MySQL that is available since PHP 5.3 enables the creation of plugins that can manage and extend access to SQL communication protocol. For example, this enables you to capture and log SQL statements that are executed via PHP with the following steps.

1. Download PHP 5.3.*x* or better
2. Download PECL/mysqlnd_qc
3. Configure PHP with —enable-mysqlnd-qc
4. Add the following to your PHP output: var_dump(mysqlnd_qc_get_query_trace_log());

Application Code

Using abstract classes in your application code for all database access is an ideal way to capture detailed SQL information, including the actual SQL statement, the query

execution time, and the order in which SQL statements executed. A well defined application infrastructure that can be dynamically enabled and disabled can provide detailed instrumentation for capturing SQL statements.

Using a simple legacy PHP application, adding application code can be as simple as the following steps.

1. Replace the occurrences of mysql_query with db_query.

2. Create an abstract database query function for db_query that executes the replaced mysql_query.

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

With this basic abstract function you can now add timing and output display:

```
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;
}
```

For the purposes of this example, we simply output the SQL statements within an HTML comment. Ideally, you would determine by some parameter or variable whether to display the SQL output, and you would collate and format your SQL statements in a more manageable way. Application level SQL logging can assist in identifying SQL injection by comparing what SQL was passed to MySQL and what was executed in MySQL.

INFORMATION_SCHEMA

The current INFORMATION_SCHEMA.INNODB_TRX table provides the same information available in the SHOW ENGINE INNODB STATUS transactions list. The INFORMATION_SCHEMA enables additional plugins to create tables that may also expose SQL statements. This can be utilized by additional storage engines.

For more information, see http://dev.mysql.com/doc/refman/5.5/en/innodb-i_s-tables.html.

PERFORMANCE_SCHEMA

The PERFORMANCE_SCHEMA does not currently provide SQL syntax information on individual SQL statements that is relevant for capturing SQL statements. However, this information can be very valuable in the detailed analysis of problematic SQL statements in high performance systems. While we do not discuss the PERFORMANCE_SCHEMA in this book, this is a source of low level monitoring information for very specific SQL analysis.

For more information, see http://dev.mysql.com/doc/refman/5.5/en/performance-schema.html.

SQL Statement Statistics Plugin

The open source sqlstats project is a MySQL plugin that utilizes the audit plugin interface to capture all SQL statements for MySQL 5.5 and above. More information is available at http://sourceforge.net/projects/sqlstats/.

More information on the audit plugin interface available in MySQL can be found at http://dev.mysql.com/doc/refman/5.5/en/audit-plugins.html.

MySQL Proxy

The MySQL Proxy, or similar proxy derivatives, act as a man in the middle when communicating with a MySQL instance over the network. This command can intercept TCP/IP packets between the client and server and process these based on many flexible and programmable needs. The default installation of MySQL Proxy includes several examples that include capturing, logging, and analyzing SQL statements.

You can find more information including software download, installation instructions, and a discussion list at https://launchpad.net/mysql-proxy.

TCP/IP

When you are connecting to MySQL across a network, it is possible to capture the physical network packets that communicate with the MySQL server. The MySQL protocol is a simple and open source communication stream that can be interpreted manually in some detail and also with various analysis tools.

There are many different ways of capturing TCPI/IP traffic. The following simple command will produce text output of the MySQL protocol that includes SQL statements and other commands:

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

Identify Problematic Statements

The identification of problematic SQL statements from those captured in the first step will help in prioritization. Looking at the slowest running SQL query is not the only technique needed for optimizing SQL statements. Optimizations of high frequency statements can also have a significant effect on improving system performance. The benefit of a 10 millisecond gain on a query that is executed hundreds or thousands of times per second is a far better gain than a 1 second gain on a query that executes only a few times per second.

> ***NOTE*** *Frequently executed and very fast SQL statements are often not considered statements necessary to tune. Speeding up these queries can have a far greater benefit on system resources that enables your database system to process more SQL statements.*

The identification process can be streamlined by adding a C style comment to an SQL statement. Here is an example:

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

This technique can be useful for identifying queries that naturally take a long time to run and might require a lower priority for identification and classification for optimization. For example, with an application that combines online transaction processing (OLTP), batch, reporting, and caching queries, the identification of an important, but low priority, long running SQL statement can indicate that other SQL statements should be reviewed first for optimization.

The aggregation of captured SQL statements will generally identify the most likely candidates for SQL optimization. Identifying the most frequent and the longest running queries will typically yield high value candidates.

With a standard installation, only the mysqldumpslow command provides any level of aggregation identification. Here is an example:

```
$ 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
```

For more information, refer to http://dev.mysql.com/doc/refman/5.5/en/mysqldumpslow.html.

You can use the mysqlbinlog command with a simple one line Linux syntax to provide aggregate information for the binary log. Here is an example:

```
$ 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
…
```

NOTE *For ease of readability, this command is displayed on multiple lines.*

Refer to http://ronaldbradford.com/blog/mysql-dml-stats-per-table-2009-09-09/ for additional information.

The Maatkit mk-query-digest tool is a general purpose open source tool that has the ability to identify an SQL statement from various sources, including the slow query log, the general query log, the binary log, and the TCP/IP stack. The benefit of this tool is that it provides valuable aggregate information on the frequency of SQL statements, as well as min/avg/max execution time. The output is conveniently formatted to provide cut/paste syntax of several of the essential analysis commands described earlier.

Slow Query Log Analysis

For queries identified and analyzed, the following output is provided:

```
$ 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 500.73k  500.96k 500.90k 485.50k       0 485.50k
# 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
```

As you can see, this output contains a wealth of information. You can determine the frequency of statements for a given time range, the source users, and hosts. You can get min/max/avg/median statistics of the execution time, rows processed, and result set size. You can also simply cut/paste provided information to determine the Query Execution Plan (QEP) and other valuable analysis information.

A number of other open source tools and code snippets also perform slow query log analysis. One example is http://code.google.com/p/mysql-slow-query-log-parser/.

TCP/IP Analysis

One of the least intrusive and near real time tools for capturing SQL statements is via TCP/IP. The mk-query-digest output is identical for other sources; however, the summary information about the sample size can be particularly useful for troubleshooting. The output provides three unique sections of information that can be used for different levels of query identification, categorization, and initial analysis.

The first section provides a top level summary of the overall time, query frequency, and summary breakdown of important details:

```
# 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
```

We can see from this sampling period that 2710 queries were analyzed, producing 197 unique MySQL requests.

The second section provides a detailed list of the top offending unique queries identified ordered by total execution time:

```
# 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>
```

In this example, two occurrences of a single DELETE statement contributed to 10 percent of the execution time of more than 2000 queries, identifying an obvious query to review in greater detail.

Part of reviewing and optimizing statements is determining SQL queries that should not be occurring. In the following example, the simple act of looking at executed SQL statements showed a high occurrence of the ROLLBACK statement. While the result of the SQL statement has no data impact when using MyISAM, the time taken to execute the statement affects overall performance.

```
# 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
```

Confirm Statement Operation

The confirmation of an SQL statement can be as easy as running the SQL statement and confirming the response time. Factors that have to be considered when confirming accurate timings for SQL include the current system load, query concurrency, the network overhead, the MySQL query cache, or the in-memory access to the necessary table index and data. It is important that you attempt to reproduce the working conditions when confirming an SQL statement. A statement in isolation must by definition respond differently than the same statement in a multi-tasking system.

Environment

Your goal should be to confirm your identified SQL statement in a reproducible means. This situation will enable you to reproduce information following optimizations.

When confirming SQL statements, you should add the SQL_NO_CACHE hint to SELECT statements to ensure the MySQL query cache is not used. Another technique might be to attempt to load all table data into the appropriate MySQL memory buffer prior to confirmation to confirm a more consistent response time. System resources, including memory, CPU, and disk, can have a significant effect on SQL query execution and should be monitored in conjunction with confirming your SQL statements to ensure consistency in results.

The testing of SQL statements using a different network path can also contribute to misleading information. When possible, you should verify SQL statements both with the network overhead and without the network overhead.

It might not be possible to confirm an SQL statement similar to a production system. What is important is to create a reproducible confirmation so that any optimization can be easily verified.

Timing

The mysql client by default shows a query execution granularity only to 10 ms. Many queries on modern hardware run under this historical minimum threshold. There is a simple patch to enable microsecond precision with the stock mysql client. See http://EffectiveMySQL.com/article/microsecond-mysql-client for more information.

The SHOW PROFILES command, when enabled with your session, can provide microsecond granularity for SQL statements. The following varying random SQL

statements show you a set of sample results that run with different durations to demonstrate microsecond precision:

```
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)                  |
+----------+------------+----------------------------------+
```

This technique was used in the analysis of SQL statements that were described in Chapter 1. The timing results that were used to confirm details were as follows:

```
|    15 | 3.18892330 | SELECT * FROM inventory WHERE item_id=16102176 |
|    15 | 0.00072900 | SELECT * FROM inventory WHERE item_id=16102176 |
```

A well defined infrastructure that is proactive in SQL performance will gather not only the SQL statements, the QEP, and query execution, but additional query details including the number of rows retrieved, the result set size, the underlying representative data size of tables, and the MySQL configuration used for the query analysis. This information gathered for SQL statements over time can help to identify changes in query performance and help in the verification process for testing different MySQL configurations, schema structures, and MySQL versions. MySQL Proxy is an excellent option to be able to combine all of the above information and produce in any customized format.

The distribution of data and the growth of data over time can have a significant effect on the performance of SQL queries.

Analyze Statements

The list of SQL commands that are used in the analysis of your identified and confirmed SQL statements are detailed in Chapter 2.

Optimize Statements

Chapters 4 and 5 detailed how to optimize SQL statements with indexes. Other techniques in the optimization of SQL statements may also provide a far better performance improvement. This includes simplifying SQL statements by removing joins or columns and improving the structure of tables by streamlining data types and constraints such as nullability. Chapter 8 goes into more detail regarding these various techniques.

Verify the Results

The most important step in any optimization is the verification process. Without detailed confirmation of your situation, it is impossible to realize the true benefit of any optimization. The verification process should also be confirmed in varying conditions, not just repeating the steps in the confirmation process. Verification with a high concurrency and a mixed workload will identify the true benefits in a production environment.

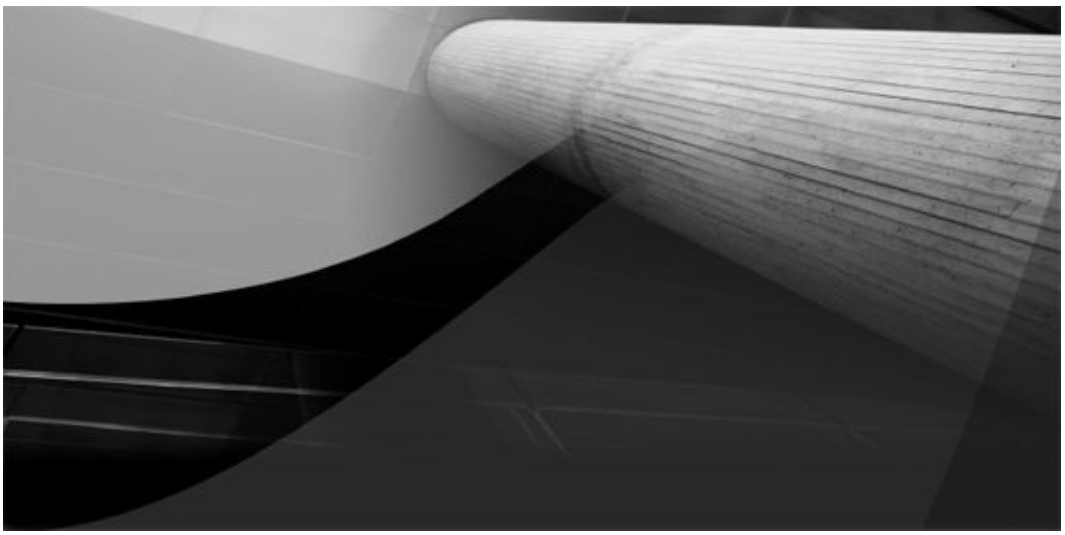
Does the query perform as expected in your controlled environment? Without a detailed integrated monitoring environment, you cannot ensure the total benefit of a single SQL optimization. Has the optimization of one query affected the performance of other system functionality? Does the optimization degrade over time as the volume of data changes? It is important not to test an optimization and deploy without carefully monitoring to determine the true improvement. This is true for any optimization including configuration changes.

Conclusion

SQL optimization is a continual improvement process. These steps should be executed repetitively to ensure that you optimize the right SQL statements at the appropriate time. A well defined infrastructure will capture, graph, and monitor key timing components from individual query times across your various MySQL environments. This should also be combined with detailed instrumentation of full page load times for important key pages.

The QEP can alter over time as the distribution and growth of data changes. New application functionality, schema design changes, an increase in query concurrency, or physical hardware bottlenecks are all variables in the identification and optimization of SQL statements.

All relevant SQL statement examples for this chapter can be downloaded at http://EffectiveMySQL.com/book/optimizing-sql-statements.



8

Hidden Performance Tips

In previous chapters, we have shown that adding indexes can dramatically improve SQL performance. There are additional ways to improve index efficiency further. Various techniques can be used to simplify or reduce SQL statements to create an immediate performance and throughput improvement.

In this chapter we will be discussing several performance tips:

- Removing duplicate indexes
- Identifying unused and ineffective indexes
- Improving indexes
- Eliminating SQL statements
- Simplifying SQL statements
- Caching options

Index Management Optimizations

The management of indexes—how they are created and maintained—can impact the performance of SQL statements.

Combining Your DDL

An important management requirement when adding indexes to MySQL is the blocking nature of a DDL statement. Historically, the impact of an ALTER statement required that a new copy of the table be created. This could be a significant operation for time and disk volume when altering large tables. With the InnoDB plugin, first available in MySQL 5.1, and with other third party storage engines, various ALTER statements are now very fast, as they do not perform a full table copy. You should refer to the system documentation for the specific storage engine and MySQL version to confirm the full impact of your ALTER statement.

Combining multiple ALTER statements into one SQL statement is an easy optimization improvement. For example, if you needed to add a new index, modify an index, and add a new column, you could perform the following individual SQL commands:

```
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;
```

You can optimize this SQL by combining all statements for a single table into one SQL statement:

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

This optimization can result in significant performance improvement of administration tasks.

Removing Duplicate Indexes

A duplicate index has two significant impacts: All DML statements will be slower as additional work is performed to maintain the data and index consistency. Additionally, the disk footprint of the database is now larger and can lead to increased backup and recovery time.

Several simple conditions can cause duplicate indexes. MySQL does not require a primary key column also to be indexed. Here is an example:

```
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)
);
```

In this DDL the defined index on `id` is a duplicate and should be removed.

A duplicate index also exists when the leftmost portion of a given index is contained within another index. Here is an example:

```
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)
);
```

The name1 index is redundant as the index columns are contained within the leftmost portion of the name2 index.

The Maatkit mk-duplicate-index-checker command is one open source tool that can quickly review your database schema for duplicate indexes. Refer to http://maatkit.org/mk-duplicate-key-checker for more information.

Removing Unused Indexes

In addition to duplicate indexes that are unused, other defined indexes might be unused. These indexes have the same performance impact as duplicate indexes. The official MySQL product provides no means to identify what indexes are unused; however, several MySQL variants do provide this feature.

The Google MySQL patch (http://code.google.com/p/google-mysql-tools/wiki/Mysql5Patches) first introduced the SHOW INDEX_STATISTICS functionality. This feature is part of a number of new commands that measure per user monitoring.

For the official MySQL product, to determine unused indexes, you first need to collect all SQL statements executed. By using these SQL statements to capture and aggregate the Query Execution Plan (QEP) for all SQL statements, per table analysis will provide information about unused indexes. The process of SQL capture and QEP generation is a good practice for all applications.

Monitoring Ineffective Indexes

When defining multi column indexes, it is important that you determine the true effectiveness of all columns specified. This instrumentation is also not part of the official MySQL product.

Analysis of the key_len column for all SQL statements on a given table can identify any indexes that might contain unused columns.

Chapter 9 provides detailed examples for understanding and analyzing the key_len column.

Index Column Improvements

In addition to creating new indexes to improve performance, you can improve database performance with additional schema optimizations. These optimizations include using specific data types and/or column types. The benefit is a smaller disk footprint producing less disk I/O and results in more index data being packed in available system memory.

Data Types

Several data types can be replaced or modified with little or no impact to an existing schema.

BIGINT vs. INT

When a primary key is defined as a BIGINT AUTO_INCREMENT data type, there is generally no requirement why this datatype is required. An INT UNSIGNED AUTO_INCREMENT datatype is capable of supporting a maximum value of 4.3 billion. If the table holds more than 4.3 billion rows, other architecture considerations are generally necessary before this requirement.

The impact of modifying a BIGINT data type to an INT data type is a 50 percent reduction in per row size of the primary key from 8 bytes to 4 bytes. The impact is also not just for a primary key; all foreign keys that are defined as BIGINT can now be modified to INT. This savings can significantly reduce the space required for indexes in a heavily normalized database. For a more detailed explanation, see http://ronaldbradford.com/blog/bigint-v-int-is-there-a-big-deal-2008-07-18/.

> ***TIP*** *The change of BIGINT to INT UNSIGNED for an AUTO_INCREMENT column can be one of the best immediate schema improvements for a database server with limited memory and a heavily normalized data model.*

DATETIME vs. TIMESTAMP

Is the requirement for recording a date/time value an epoch value—that is, the number of seconds since 1/1/1970? If the value being stored is only an epoch value, then a TIMESTAMP column supports all necessary values. A DATETIME column supports all possible date/time values. A DATETIME data type is 8 bytes, and a TIMESTAMP data type is 4 bytes.

The disadvantage of using a TIMESTAMP is the implied DEFAULT 0, which is incompatible with any related SQL_MODE settings that disable zero date settings. The TIMESTAMP data type also does not support a NULL value. This column type is ideal for creating the date and time of a created or an update value for a row when a value always exists.

ENUM

MySQL provides the ENUM data type, which is ideal for static code values. For example, when recording the values for gender, you could define a column as either of the following.

```
gender1     VARCHAR(6) NOT NULL
```

```
gender2     ENUM (‘Male’,’Female’) NOT NULL
```

There are three benefits of using the ENUM data type:

- An ENUM column provides additional data integrity with an implied check constraint.
- An ENUM column uses only 1 byte for up to 255 distinct values.
- The values of an ENUM column are better for readability. For example, if you have a status field, the use of an ENUM is compact in data size and provides a meaningful description of the column value.

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

Historically, the impact of using an ENUM was the database dependence of changing the range of values when a new value was required. This was an ALTER DDL statement, which is a blocking statement. Starting with MySQL 5.1, adding a value to an ENUM column is very fast and unrelated to the size of the table.

NULL vs. NOT NULL

Unless you are sure that a column can contain an unknown value (a NULL), it is best to define this column as NOT NULL. Frameworks, for example Ruby on Rails, are notorious for not defining mandatory columns as NOT NULL. When the column is defined within an index, there is an improved size reduction and simplification of index processing, as no additional NULL conditions are required. In addition, having NOT NULL places an extra integrity constraint on the data in the column, ensuring all rows have a value for the column.

Implied Conversions

When you are choosing an indexed data type for table joins, it is important that the data type is identified. Implied type conversion can be an unnecessary overhead. For numeric integer columns, ensure that SIGNED and UNSIGNED is consistent. For a variable data type, the added complexity includes the character set and collation. When you are defining indexed columns for table join conditions, ensure that these match. A common problem is an implied conversion between LATIN1 and UTF8 character sets.

Column Types

Several types of data are commonly defined with inefficient column types. Changing the column’s data type can result in more efficient storage, especially when these columns are included within indexes. Following are a few common examples.

IP Address

An IPv4 address can be defined as an INT UNSIGNED data type requiring 4 bytes. A common occurrence is a column definition of VARCHAR(15), which averages 12 bytes in size. This one improvement can save two-thirds of the original column data size. The INET_ATON() and INET_NTOA() functions manage the conversion between an IP string and a numeric value. Here is an example:

```
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 |
+--------------+------------+--------------+
```

This technique is applicable only to IPv4. With IPv6 becoming commonly acceptable, it is important that you also store these 128-bit integer values efficiently with a BINARY(16) data type and not use a VARCHAR data type for the human-readable format.

MD5

Storing a MD5 value of any value as a CHAR(32) column is a normal practice. If your application is using VARCHAR(32) you have additional unnecessary overhead of the length of the string for every value. This hexadecimal value can be stored more efficiently using the UNHEX() and HEX() functions and storing the data in a BINARY(16) data type. Doing this conversion reduces the per-row space consumed from 32 bytes to 16 bytes.

The following example shows the size of the MD5 and MD5 compressed column:

```
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 |
+----------------------------------+---------+-----------+
```

This principle can be applied to any hexadecimal value—for example, if an index is defined for a hash of all columns.

Other SQL Optimizations

Adding indexes can provide significant performance benefits. However, the most effective SQL optimization for a relational database is to eliminate the need to execute the SQL statement completely. For a highly tuned application, the greatest amount of time for the total execution of the statement is the network overhead. Removing SQL statements can reduce the application processing time. Additional steps necessary for each SQL statement

include parsing, permission security checks, and generation of the query execution plan. These are all overheads that add unnecessary load to the database server when statements are unnecessary. You can use the profiling functionality to get detailed timing of steps within the execution of a query. Here is an example:

```
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 |
+----------------------+----------+------------------+---------------+----------+
```

The Status values listed can be misleading, as these expose internal comments not originally intended for public view. Reviewing the actual MySQL source code file and line as specified is recommended to understand the full description.

Eliminating SQL Statements

Several simple techniques can be used to eliminate SQL statements:

- Remove duplicate SQL statements.
- Remove repeating SQL statements.
- Remove unnecessary SQL statements.
- Cache SQL statement results.

Removing Duplicate SQL Statements

Capture of all SQL statements for a given function or process will highlight any duplicate SQL statements that are executed to complete a specific request. The best practice is to enable the general query log in development environments. Analysis of all SQL statements should be the responsibility of the developer to ensure that only necessary SQL statements are executed. Adding instrumentation to your application to report the number of SQL statements and provide debugging for dynamic viewing of all SQL statements easily enables more information to identify duplicate statements. The use of application frameworks can be a primary cause of unnecessary duplicate SQL statements.

Removing Repeating SQL Statements

Many applications suffer from Row At a Time (RAT) processing. Also known as the N+1 problem, the cause is an outer loop that generates an SQL statement per row. This can result in hundreds to thousands of repeating SQL statements. In many situations, applying a single SQL statement to achieve Chunk At a Time (CAT) processing can eliminate repeating SQL statements. Using the capability of set processing, which is a strength of SQL, can greatly improve performance. Here is an example:

```
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;
```

This sequential row processing can be rewritten as follows:

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

The profiling functionality can demonstrate the full overhead of this type of SQL execution:

```
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 |
+------------------------+------------+
```

The use of application frameworks can be a primary cause of unnecessary repeating SQL statements.

This example of repeating SQL statements is part of a more common N+1 scenario, where the SQL developer does not understand how to use table joins correctly. This practice involves a loop on an outer set of records, and then querying a subsequent table for all records in the loop. The result is the combined SELECT statement can be eliminated completely. Here is an example:

```
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;
```

This loop of SQL statements can be replaced with the following single SQL statement:

```
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’;
```

Removing Unnecessary SQL Statements

Applications that are modified and enhanced over time, such as the following examples, can introduce unnecessary SQL statements:

- Information selected that is no longer required
- Information selected that is used only for certain paths of a given function
- Information that can be selected with a preceding SQL statement

Functionality that has grown over time can easily suffer from this. Capturing and reviewing all SQL statements might not indicate occurrences of unnecessary SQL. Understanding the application needs and what is presented to the end user or used in processing might be necessary to identify improvements. The classic SELECT * is an example for which there is no understanding of what data is actually required without full analysis of the code executing the SQL statement.

Caching SQL Results

When the rate of change of common data is relatively low, caching SQL results can provide a significant performance boost to your application and enable additional scalability of your database server.

MySQL Caching

The MySQL query cache can provide a boost in performance for a high read environment and can be implemented without any additional application overhead. The following is an example using the profiling functionality to show the execution time and the individual complexity of a regular SQL statement and a subsequent cached query:

```
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 |
+----------+------------+-------------------------------------+
```

As you can see, the second query is 10 times faster. You can use the profiling functionality to provide a detailed analysis of individual steps:

```
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)
```

As you can see, the second query performs significantly less work.

The impact of the MySQL query cache can degrade the performance for environments with heavy write to read ratios. This is due to the coarse nature of the query cache, where a change of any data to a given table will result in an invalidation of all SQL statements that are cached that use the table. In an environment with a high number of reads and writes, this can result in significant invalidations of SELECT statements that do not gain the benefit of ever being used.

Application Caching

Adding caching at the application level can dramatically reduce unnecessary SQL statement execution. There is no single recommended technique for application caching. Strategies include caching the results of a given SQL statement in memory or in local files, or it can involve lazy object instantiation in the application object relational mapping (ORM) layer. The implementation should reflect the needs of your application. For example, the following pseudo code would replace a simple SELECT with:

```
// Existing SELECT
SELECT FROM TABLE
RETURN VALUE
```

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

What is necessary is that any changes to the table are also reflected in the cache. For example, the following pseudo code is required:

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

Alternatively, you could enable a time-to-live (TTL) value for cached values if your caching system allows periodic cache invalidation.

The popular Memcached product (refer to http://memcached.org/) is widely used for many applications. This caching infrastructure can be easily integrated with many popular programming languages.

There are different ways to implement a MySQL/Memcached data synchronization for invalidation. One implementation is with an integration of the MySQL User Defined Functions (UDFs) for Memcached combined with database triggers to manage invalidations. Another option is to process the MySQL binary log events for SQL statements and then appropriately manage the synchronization with the implementing cache infrastructure.

Simplifying SQL Statements

When it is not possible to eliminate an SQL statement, it might be possible to simplify the statement. Consider the following questions:

- Are all columns required?
- Can a table join be removed?
- Is a join or WHERE restriction necessary for additional SQL statements in a given function?

Column Improvement

An important requirement of simplification is to capture all SQL statements in order for a given executed function. Using a sampling process will not identify all possible improvements. Here is an example of a query simplification:

```
mysql> SELECT fid, val, val
    -> FROM table1
```

```
    -> WHERE fid = X;
```

This query returned 350,000 rows of data that was cached by the application server during system startup. For this query, two-thirds of the result set that was passed over the network was redundant. The fid column was passed as a restriction to the SQL statement, and therefore the value was identical for all rows. The val column was also unnecessarily duplicated. The optimal SQL statement is shown here:

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

Join Improvement

The following is an example of a table join simplification that requires knowledge of all SQL statements for a given function. During the process, the following SQL statement is executed to retrieve a number of rows:

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

At a later time in the execution of this function, the following statement was executed using the id value from the previous SQL statement:

```
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;
```

This second SQL statement could be simplified to this:

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

As the first query performs the necessary restriction for the id column (that is, col1 = X and col2 = Y), the join and restriction clauses for table1 are redundant. Removing this join condition simplifies the SQL statement and removes a potential problem if only one statement is altered in the future.

Rewriting Subqueries

The MySQL database supports subqueries. The performance of subqueries may be significantly slower in certain circumstances than using a normal table join. Here is an example:

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

This SQL statement can simply be rewritten as follows:

```
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
```

The change might appear to be subtle; however, this approach for more complex queries can result in improved query performance.

Understanding the Impact of Views

Developers should know the true class of the table that is used for SQL statements. If the object is actually a view, the impact of SQL optimizations can be masked by the complexity of join conditions for the view definition. A common problem with data warehouse (DW) and business intelligence (BI) tools is the creation of views on top of views. For any view definition, additional information may be retrieved that is not ultimately required for an SQL statement. In MySQL, complex queries involving views can be easily improved by using the necessary underlying tables.

Using MySQL Replication

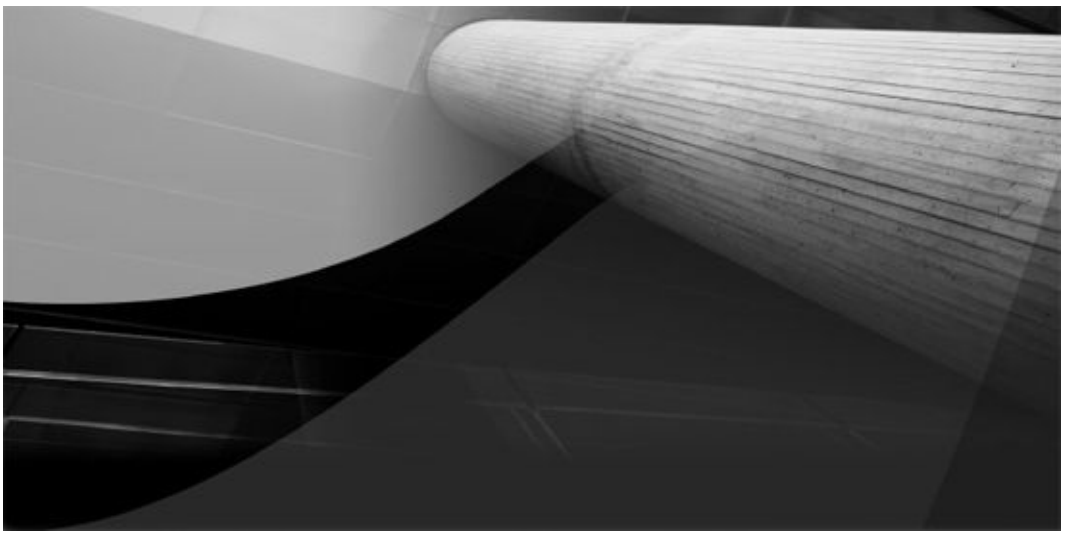
You can use MySQL slaves to increase read scalability across many servers. This principle is common for applications that have a much higher read capacity requirement.

While executing SELECT statements on a MySQL slave or slaves can spread read load, MySQL replication supports an additional feature where you can customize the schema design on slaves. Providing an SQL statement executes successfully on a MySQL slave, MySQL replication will accept these modifications. This could include using a different storage engine for some tables or using different indexes on certain tables. More complex environments may support different classes of MySQL slaves with differing index requirements on a range of servers. It is important that for any index changes, primary and unique keys that are used to enforce data integrity are not removed.

Conclusion

There are many different approaches to improving application and database performance. In this chapter, we have discussed a number of architectural and design approaches that can be applied to optimizing SQL statements. There are many more overall techniques for ongoing optimizations. The skills of a software developer can be improved with the understanding and appreciation of skills known by a data architect and database administrator. These skills separate experienced SQL developers from those with less SQL

experience.



9

Explaining the MySQL EXPLAIN

The MySQL EXPLAIN command is used to show the Query Execution Plan (QEP) for your SQL statement. The output from the command provides insight into how the MySQL optimizer will execute the SQL statement. The MySQL EXPLAIN command does not provide any tuning recommendations, but it does provide valuable information to help you make tuning decisions.

In this final chapter, we will be detailing the full syntax and options for the EXPLAIN command.

Syntax

The QEP is produced with the EXPLAIN command. The syntax has two options:

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

or

```
mysql> EXPLAIN table
```

The MySQL EXPLAIN syntax operates on a SELECT statement or a specific table. The table option is a synonym for the DESC table command. UPDATE and DELETE commands are not immune to performance improvements and should be rewritten as SELECT statements when not working directly with the table primary key to ensure optimal index usage. Here is an example:

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

This UPDATE statement can be rewritten as the following SELECT statement:

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

The MySQL optimizer is a cost based optimizer. It does not provide any pinning of a QEP. This means the QEP is calculated for every SQL execution. SQL statements found in MySQL stored procedures are also calculated for every execution. Stored procedures

cache only the parsed query tree.

Explain Columns

The MySQL EXPLAIN command produces the following information for each table within the SQL statement:

```
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
```

This QEP shows no index used (known as full table scan) and a large number of rows processed to satisfy the query. An optimized QEP for the same SELECT statement is shown here:

```
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:
```

In this QEP, we see that an index is used and that it is estimated that only one row will be retrieved. In this chapter we will detail how to read and interpret this information. The full list of columns for each row in the QEP is shown here:

- id
- select_type
- table
- partitions (this column is available only with EXPLAIN PARTITIONS syntax)

- possible_keys
- key
- key_len
- ref
- rows
- filtered (this column is available only with the EXPLAINED EXTENDED syntax)
- Extra

These columns are displayed for each table for the QEP of your SELECT statement. A table may relate to a physical schema table or an internal temporary table (for example, from a subquery or union) that is generated at the time of SQL execution.

You can also refer to the MySQL Reference Manual for more information at http://dev.mysql.com/doc/refman/5.5/en/explain-output.html.

The following column descriptions are ordered based on importance of analysis and understanding for quickly and efficiently understanding your QEP.

key

The key column identifies the index(es) the optimizer has selected to use. Generally each individual table within an SQL query will use only one index. There are a small number of index merge exceptions where two or more indexes for a given table are possible.

Here are example occurrences of the key column in a QEP:

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

The SHOW CREATE TABLE <table> command is the easiest method of providing the details of the table and index columns.

Related columns include possible_keys, rows, and key_len.

rows

The rows column provides an estimate of the number of rows the MySQL optimizer expects to analyze with all existing rows in the cumulative result set. This is a difficult calculation to describe easily for a QEP. The total number of all read operations for a query is based on the sequential accumulation of the rows value for each row of the combined proceeding rows. This is a nested row algorithm.

For example, a QEP joins two tables. The first row as identified by id=1 has a rows value of 1. This equates to 1 read operation for the first table. The second row as identified by id=2 has a rows value of 5. This equates to 5 read operations that are matched with the current accumulation of 1. The total number of reads when referencing both tables is 6. In another QEP, the first rows value is 5 and the second rows value is 1. This equates to 5 read operations for the first table, and 1 read operation per current accumulation of 5. The total number of reads when referencing both tables is 10 (5+5).

The best estimate is 1 row, which is generally the case when the search-for rows in the table can be found using a primary or unique key.

In the following QEP, the outer nested loop as indicated by id=1 has an estimate of 1 physical row. The second loop processes 10 rows.

```
*************************** 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:
```

A review of the actual row operations can be found using the SHOW STATUS command. This gives the best confirmation of the physical row operations. Here is an example:

```
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     |
+------------------------+-------+
```

In the next QEP, the outer nested loop as indicated by id=1 has an estimate of 160 rows. The second loop has an estimate of 1 row.

```
*************************** 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
```

A review of the actual row operations can be found using the SHOW STATUS command, which shows a large increase of physical read operations. Here is an example:

```
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   |
+------------------------+-------+
```

Related QEP columns include key.

possible_keys

The possible_keys column identifies the index(es) the optimizer chose to consider for the query.

A QEP that shows a large number of possible indexes, for example three or more, is an indicator of too many indexes to choose from and a probable sign that there are ineffective single column indexes. The SHOW INDEXES command detailed in Chapter 2 can be used to verify indexes in place and applicable cardinality.

The speed of determining the QEP for a query can affect query performance. Identifying a large number of possible indexes can indicate that indexes are not being utilized.

Related QEP columns include key.

key_len

The key_len column defines the length of the key that is used for the join condition of your SQL statement. This column value is important to identify the effectiveness of an index and confirm how many columns of a multi column index are used.

Example values include the following:

```
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
```

As shown in these examples, nullability, variable length columns, and character sets all affect the internal memory size of table indexes.

The value of the key_len column references only index columns used in the join or where restriction of the query. Further columns of the index can be utilized with the ORDER BY or GROUP BY clauses.

The following table from the popular open source blogging software WordPress demonstrates how best to use SQL statements with the defined table indexes:

```
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
```

The index for this table includes the columns post_type, post_status, post_date, and ID. The following is an example SQL query used to demonstrate index column usage:

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

The QEP of this query reports a key_len of 62. This equates to the post_type column only (that is, (20 × 3) + 2). Although the query uses the post_type and post_date columns in the WHERE clause, only the post_type portion of the index is used. The reason why more of the index is not utilized is because MySQL will use only the leftmost portions of the defined index. To utilize the index better, you would modify the query to leverage the index columns. Here is an example:

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

By adding a restriction with the post_status column to the SELECT query, the QEP shows a key_len of 132. This equates to the post_type, post_status, and post_date columns of the index (that is 62 + 62 + 8, (20 × 3) + 2, (20 × 3) + 2, 8).

In addition, the definition of the primary key column ID for this index is a legacy of using the MyISAM storage index. When using the InnoDB storage engine, the inclusion of the primary key column in a secondary index is redundant and would be represented by the key_len usage.

Related QEP columns include Extra with a value of Using index.

table

The table column is the unique indicator for an individual row of the EXPLAIN output. This value could be the table name, table alias, or an indicator of a generated temporary table for the query such as a derived table, subquery, or a union.

Here are example occurrences of the table column in a QEP:

```
table: item
table: <derivedN>
table: <unionN,M>
```

The values of N and M in the table value reference a different table row matching the id column value.

Related QEP columns include select_type.

select_type

The select_type column provides a type to represent how the table column reference is used. The most common values include SIMPLE, PRIMARY, DERIVED, and UNION. Other possible values are UNION RESULT, DEPENDENT SUBQUERY, DEPENDENT UNION, UNCACHEABLE UNION, and UNCACHEABLE QUERY.

SIMPLE

This is the common type for simple queries that do not include subqueries or other complicated syntax.

PRIMARY

This is the primary table (that is, the outermost table) for a more complex query. This is primarily found when combined with DERIVED and UNION select types.

DERIVED

When a table does not represent a physical table this is referenced as DERIVED. The following SQL statement for example will provide an example DERIVED select_type in a QEP:

```
mysql> EXPLAIN SELECT MAX(id)
    -> FROM (SELECT id FROM users WHERE first = 'west') c;
```

DEPENDENT SUBQUERY

This select_type value is defined when a subquery is used. The following SQL example produces this value:

```
mysql> EXPLAIN SELECT p.*
    -> FROM parent p
    -> WHERE p.id NOT IN (SELECT c.parent_id FROM child c);
```

UNION

This is one SQL element of the UNION statement.

UNION RESULT

This is the result of a number of tables that are defined within a UNION statement. This is also observed with a table value of <unionN,M>, which indicates which matching id rows are part of the union.

The following SQL example produces a UNION and 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;
```

partitions

The partitions column identifies what partitions are used for the given table. This column is available only when the EXPLAIN PARTITIONS syntax is used.

Extra

This column provides a bucket of additional information about various paths of the MySQL optimizer. The Extra column can contain multiple values. There are many different values and these increase as newer versions of MySQL are released. The following is a list of common values. A more complete list can be found at http://dev.mysql.com/doc/refman/5.5/en/explain-output.html.

Using where

This represents that the query uses the where clause for processing—for example, with a full table scan. When an index is also referenced, the restriction of rows was done via processing the read buffer after retrieving the necessary data.

Using temporary

This represents that an internal temporary (memory based) table was used. It is possible for a query to use multiple temporary tables. There are many different reasons why MySQL may create a temporary table during query execution. Two common causes are using a DISTINCT on columns from different tables, or using a different ORDER BY and GROUP BY columns.

For more information see http://forge.mysql.com/wiki/Overview_of_query_execution_and_use_of_temp_tables.

A temporary table can also be forced to use the disk based MyISAM storage engine. There are two primary causes for this:

- The size of internal table exceeds min(tmp_table_size, max_heap_table_size) system variables
- Use of TEXT/BLOB column

Using filesort

This is the result of an ORDER BY predicate. This can be a CPU intensive process.

A performance improvement can be to try and achieve ordering of results with an applicable selected index. See Chapter 4 for more information.

Using index

This highlights that only the index was used to satisfy the requirements for the table, and no direct table access was needed. See Chapter 5 for detailed examples that demonstrate

this value.

Using join buffer

This highlights that no index was used in the retrieval of a join condition and the join buffer was required to store the intermediate results.

Depending on the query it might be possible to add an index to improve performance.

Impossible where

This highlights that the where clause results in no possible rows. Here is an example:

mysql> EXPLAIN SELECT * FROM user WHERE 1=2;

Select tables optimized away

This indicates that the optimizer was able to return only one row of results from aggregated functions using just the index. Here is an example:

```
mysql> EXPLAIN SELECT COUNT(*)
    -> FROM (SELECT id FROM users WHERE first = ‘west’) c
```

Distinct

This indicates that MySQL will stop searching for additional rows after the first matching row.

Index merges

When MySQL has determined to use more than one index for a given table, one of the following formats is presented, detailing the type of merge and the indexes used.

- Using sort_union(…)
- Using union(…)
- Using intersect(…)

id

The id column is a sequential reference of each table that is represented in the QEP.

ref

The ref column can be used to identify the column or constant that is used for index

comparison.

filtered

The filtered column gives a percentage that when used with the rows column gives the estimated rows that will be joined with earlier tables in the QEP. An earlier table is defined as a lower id column value.

This column is provided only with the EXPLAIN EXTENDED syntax.

type

The type column refers to the join type that was used for this specific table in the QEP. Following are the most common join types:

- const This table has at most one matching row.
- system This is a special case of const where the table has only one row.
- eq_ref One row is read for each of the previously identified tables.
- ref All rows with a matching value of the index are used.
- range All index rows that match a supplied range of values are used.
- ALL A full table scan is required.

Other type values include fulltext, ref_or_null, index_merge, unique_subquery, index_subquery, and index.

For more information, see http://dev.mysql.com/doc/refman/5.5/en/explain-output.html.

Interpreting EXPLAIN Output

Understanding your application, including the technology and implementation possibilities, is just as important as trying to tune an SQL statement. The following is an example of a business need that retrieves orphaned parent records in a parent/child relationship. This SQL query can be written in three different ways. While the output produces the same results, the QEP shows three different paths.

```
mysql> EXPLAIN SELECT p.*
    -> FROM parent p
```

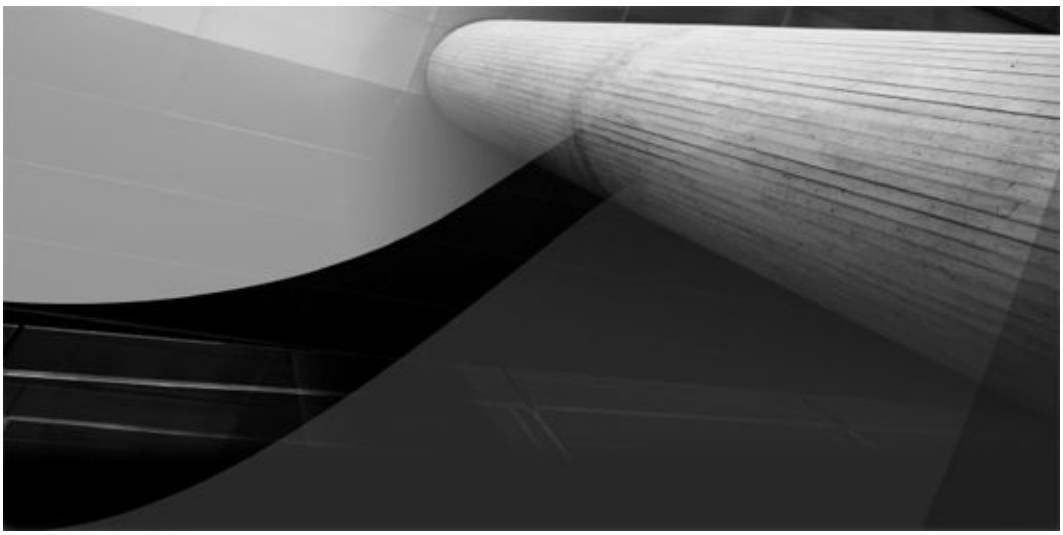
```
    -> WHERE p.id NOT IN (SELECT c.parent_id FROM child c)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: p
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 160
        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
Explaining the MySQL EXPLAIN 155
      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)
```

Which is best? Will data growth over time cause a different QEP to perform better? The purpose of this book is to present the tools and practices to optimize your SQL statements.

Conclusion

Reading and understanding the MySQL Query Execution Plan (QEP) with the EXPLAIN command is one of the most essential commands used to identify information for the optimization of SQL statements. Combining this with additional information from various sources as detailed in this book ensures you have the necessary data to make informed decisions.



Index

A

aggregation
identifying problematic SQL statements, 115–117
of left columns in multi column index, 63
supporting with indexes, 28
ALTER statements
combining statements into one SQL, 124–125
impact of adding indexes, 73–74
performance overhead of, 5
ALTER TABLE statement, 74
analysis commands
EXPLAIN, 10–13
INFORMATION_SCHEMA, 20–21
overview of, 9–10
SHOW CREATE TABLE, 13
SHOW INDEXES, 14–15
SHOW STATUS, 17–19
SHOW TABLE STATUS, 15–17
SHOW VARIABLES, 19–20
analysis, of SQL statements, 121
ANALYZE table command, 77
ANSI SQL statement, 20
applications
adding caching to, 136
capturing SQL statements with code, 112–113
ARCHIVE storage engine, 15
AUTO INCREMENT column
InnoDB B+tree clustered primary key, 39
primary keys, 25

B

B-tree data structure
MEMORY tables, 41–42
MyISAM storage engine using, 35–37
overview of, 31–32
secondary indexes in InnoDB using, 40, 76
benchmarking, 84
BIGINT vs. INT datatypes, 127
binary log, 109–110
BLACKHOLE storage engine, 15
B+tree data structure
indexes, 32–33
InnoDB clustered primary key, 37–40
bytes, defining variable size, 93

C

C style SQL comments, 115
CACHE INDEX statement, 93
caching
application, 136
assigning table indexes to specific named caches, 93
defined, 134
global memory buffer for frequently cached queries, 95
Memcached product,
application caching, 136
MySQL, 134–136
capture statements
application code, 112–113
binary log, 109–110
engine status, 110–111
general query log, 107–108
INFORMATION_SCHEMA, 113
MySQL connectors, 111–112

MySQL Proxy, 114
overview of, 106–107
PERFORMANCE_SCHEMA, 112
processlist, 110
slow query log, 108–109
SQL statement statistics plugin, 114
TCP/IP, 114–115
Cardinality column, SHOW INDEXES command, 14
cardinality, index
multi column indexes, 62–63
understanding, 51–53
CAT (Chunk At a Time) processing, 132–133
Chunk At a Time (CAT) processing, 132–133
columns
aggregating in multi column index, 63
improving, 137
indexes with multiple. *see* indexes, multi column
indexes with single. *see* indexes, single column
inefficient, 129–130
QEP. *see* QEP (Query Execution Plan) columns
configuration options
logging and instrumentation related variables, 100–102
memory related variables. *see* memory related variables
miscellaneous query related variables, 102–103
overview of, 89
confirmation operation, SQL statement, 119–121
connection threads, 90
Connector/J, 112
connectors, capturing SQL statements, 111–112
constraints, foreign key, 26
cost based optimizer
MySQL limitations, 76–77

MySQL optimizer as, 142
CREATE TABLE command
MyISAM syntax, 26
specifying storage engine for, 102
created_tmp_disk_tables variable, 98
created_tmp_tables variable, 98

D

data
choosing optimal architecture for, 84
maintaining integrity with indexes, 25–26
optimizing access with indexes, 26–27
data structure theory, index types, 31–35
data types, index column improvements, 127–129
DATETIME vs. TIMESTAMP datatypes, 127
DDL statements
adding indexes to tables, 73–74
capturing with binary log, 109
index optimization, 124–125
removing duplicate indexes, 125–126
default_storage_engine variable, 102
DELETE statement
capturing with binary log, 109
rewriting as SELECT, 142
DEPENDENT SUBQUERY value, select_type column, 150
DERIVED value, select_type column, 149
DESC command, 77
disk space impact, of indexes, 74–75
distinct keyword, EXPLAIN command, 152
DML statements
capturing long running or common queries, 110
capturing with binary log, 109
impact of adding indexes to table, 71–72

performance overhead of indexes, 5
removing duplicate indexes, 125–126
duplicate indexes, removing, 72–73, 125–126
duplicate SQL statements, removing, 131
dynamic variables, 90

E

engine status, capturing statements with, 110–111
ENUM datatype, 128
environment, confirming SQL statements, 119–120
EXPLAIN command
for column utilization of multi column indexes, 63–64
columns, overview, 142–144
distinct, 152
experience required to understand, 21
EXTENDED keyword of, 12–13
Extra column, 150
filtered column, 152
id column, 152
impossible WHERE, 151–152
index merges, 152
interpreting output, 153–155
joining tables, 51
key column, 144
key_len column, 147–148
optimizing existing indexes, 47
optimizing SQL statements, 3–4
overview of, 10–11, 141
partitions column, 150
PARTITIONS keyword of, 11
possible_keys column, 147
rows column, 144–146
select tables optimized away, 152

select_type column, 149–150
syntax, 142
table column, 148–149
type column, 153
using filesort, 151
using index, 151
using join buffer, 151
using temporary tables, 151
using where, 150–151
EXPLAIN EXTENDED command, 12–13, 152
EXPLAIN PARTITIONS command
overview of, 11
partitions column, 150
scanning only applicable indexes, 43
EXTENDED keyword, EXPLAIN command, 12–13, 152
Extra column
creating covering index, 82
EXPLAIN, 150

F

filtered column, EXPLAIN EXTENDED command, 11, 152
FORCE INDEX hint, 69–71
FOREIGN KEY ERROR section, SHOW ENGINE INNODB STATUS, 111
foreign keys
for data integrity, 26
partitioned tables not supporting, 43
Fractal Tree index, TokuDB, 31
full table scans, avoiding, 48–49
fulltext indexes
defined, 30
overview of, 34
partitioned tables not supporting, 43
function based indexes, 77

G

\G statement terminator, 3

general query log, 107–108

general_log variable, 100–101

general_log_file variable, 100–101

global memory buffers

global/session memory buffers, 91

innodb_additional_mem_pool_size, 95

innodb_buffer_pool_size, 93–95

key_buffer_size, 92

overview of, 91

global/session memory buffers, 91

Google MySQL Patch, removing unused indexes, 126

GROUP BY operator, covering indexes, 82

H

Hash table data structure

MEMORY engine supporting, 40–41

overview of, 33–34

HEX() function, MD5 performance, 129–130

hints

index, 69–71

query, 67–69

I

id column, EXPLAIN command, 152

identification process, SQL optimization lifecycle

overview of, 115–117

slow query log analysis, 117–118

TCP/IP analysis, 118–119

IGNORE INDEX hint, 69–71

implied type conversions, overhead, 128–129

impossible WHERE, EXPLAIN command, 151–152

index hints, 69–71

index implementations
defined, 30
InnoDB B-tree secondary key, 40
InnoDB B+tree clustered primary key, 37–40
InnoDB internal Hash index, 42
Memory B-tree index, 41–42
Memory Hash index, 40–41
MyISAM B-tree, 35–37
overview of, 34
theory of data structure, 31–35
index merges
EXPLAIN, 152
overview of, 66–67
index techniques, 30
index types
data structure theory for, 31–34
defined, 30
overview of, 30–31
indexes
aggregation and, 28
assigning to specific named cache, 93
B-tree data structure, 31–32
B+tree data structure, 32–33
column improvements for, 126–130
data access optimization with, 25–27
data integrity with, 25–26
example tables, 24–25
fulltext, 34–35
generating query execution plan, 4
Hash table data structure, 33–34
managing optimization of, 124–125
optimizing SQL statements, 4–7

partitioning, 43
R-tree data structure, 34
removing duplicate, 125–126
SHOW INDEXES output for, 14
showing size with SHOW TABLE STATUS, 15
for sorting results, 28
storage engines and, 28–31
for table joins, 28
terminology, 30
types of, 30–31
understanding, 23–24
indexes, creating
covering indexes, 80–83
example tables, 46
MySQL limitations, 76–77
optimizing existing indexes, 47–48
overview of, 45–46, 79–80
partial indexes, 85–88
performance impact of adding indexes, 71–76
storage engine implications, 84–85
indexes, multi column
combining WHERE and ORDER BY, 64–65
complicated queries, 71
creating covering index, 80–83
determining which index to use, 58–61
improving effectiveness of, 63–64
index hints, 69–71
monitoring for unused columns, 126
optimizer features, 66–67
providing better index, 61–63
query hints, 67–69
syntax, 61

indexes, single column
joining tables, 50–51
ordering results, 57–58
overview of, 48
pattern matching, 54–55
restricting rows, 48–49
selecting unique row, 55–57
syntax, 48
understanding cardinality, 51–53
INET_ATON() function, IPv4 performance, 129
INET_NTOA() function, IPv4 performance, 129
InfiniDB index implementation, 31
Infobright index implementation, 31
INFORMATION_SCHEMA command
capturing SQL statements, 110, 113
creating partial index, 85
obtaining components of table structure, 13
overview of, 20–21
INFORMATION_SCHEMA.GLOBAL_STATUS, 17, 96
INFORMATION_SCHEMA.GLOBAL_VARIABLES, 19–20
INFORMATION_SCHEMA.SESSION_STATUS, 17
INFORMATION_SCHEMA.SESSION_VARIABLES, 19–20
INFORMATION_SCHEMA.STATISTICS, 14
INFORMATION_SCHEMA.TABLES table
determining storage engine, 30
disk space impact of adding indexes, 74–75
retrieving information, 16
showing size of MEMORY engine table, 97
init_file variable, 93
InnoDB storage engine
B-tree secondary key, 40
B+tree clustered primary key, 37–40

covering indexes and, 84
defined, 30
fast index creation, 74
foreign key constraints for, 26
impacting disk space, 75
innodb_additional_mem_pool_size, 95
innodb_buffer_pool_size for, 93–95
innodb_strict_mode for, 103
internal Hash index, 42
limited index statistics of, 77
MyISAM vs., 35
page fill impact of adding indexes, 76
innodb_adaptive_hash_index, 42
innodb_additional_mem_pool_size variable, 95
innodb_buffer_pool_size option, InnoDB, 39
innodb_buffer_pool_size variable, 93–95
innodb_file_per_table option, InnoDB, 37
innodb_strict_mode variable, 103
instrumentation related system variables, 100–102
INT datatype
vs. BIGINT, 127
UNSIGNED, IPv4, 129
internal Hash index, InnoDB, 42
internal temporary tables, 97–98
IPv4 addresses, performance tips, 129

J

join_buffer_size variable, 99
joining tables
applying index hints, 69–71
with indexes, 48–49

K

key column, EXPLAIN command, 4, 11, 144

key_buffer_size variable, 92
key_len column, EXPLAIN command
defined, 64–65
monitoring ineffective indexes, 126
overview of, 147–148
K,M,G format (variable size), 93

L

LATEST DETECTED DEADLOCK, SHOW ENGINE INNODB STATUS, 111
LATIN1, and implied conversions, 129
least recently used (LRU) method, key_buffer_size, 92
lifecycle. *See* SQL optimization lifecycle
logging system variables, 100–102
log_output variable, 100–101
long_query_time variable, 100–101, 108
LRU (least recently used) method, key_buffer_size, 92

M

Maatkit commands
mk-duplicate-key-checker, 73, 125–126
mk-query-digest, 116–117
max_allowed_packet variable, 103
max_heap_table_size variable, 97
MD5 value, and performance, 129–130
Memcached product, application caching, 136
memory related variables
configuration options, 90
global memory buffers, 91
innodb_additional_mem_pool_size, 95
innodb_buffer_pool_size, 93–95
join_buffer_size, 99
key_buffer_size, 92
max_heap_table_size, 97
named key buffers, 93

query_cache_size, 95–96
read_buffer_size, 99
read_rnd_buffer_size, 99
session buffers, 91–92
sort_buffer_size, 99
tmp_table_size, 97–99
MEMORY storage engine
B-tree index, 41–42
defined, 29
defining maximum size of table, 97
Hash index, 40–41
for internal temporary tables, 97–98
SHOW TABLE STATUS, 15
merges, index
EXPLAIN, 152
overview of, 66–67
MIXED binary log format, 110
mk-duplicate-key-checker command, 73, 125–126
mk-query-digest command, 116–117
multi column indexes. *See* indexes, multi column
.MYD file, MyISAM, 35
.MYI file, MyISAM, 35
MyISAM storage engine
B-tree data structure, 35–37
covering indexes and, 84
default key_buffer_size variable for, 92
defined, 29
foreign key constraints and, 26
InnoDB B-tree secondary key vs., 40
limited index statistics of, 77
partial indexes and, 85
SHOW TABLE STATUS, 15–16

spatial indexes with R-tree, 34
MySQL caching, 134–136
MySQL connectors, 111–112
MySQL Proxy, 114
mysqlbinlog command, 109, 116
mysqldump command, 13
mysqldumpslow command, 116

N

N+1 problem, eliminating SQL statements, 132–133
named key buffers, memory related variables, 93
naming indexes, 50
non unique secondary indexes, 30–31
NULL datatype
defining unique index with, 57
NOT NULL vs., 128
primary keys and, 25
restricting rows with index, 48–50
unique keys and, 26

O

oak-online-alter-table utility, 74
object relational mapping (ORM) layer, application caching, 136
online schema change (OSC) tool, 74
optimization, of SQL statements, 121
alternative solution, 7
confirming slow query, 2–4
confirming your, 5–6
correct approach to, 6–7
finding slow query, 2
what you should not do, 5
optimizer
as cost based, 142
features, 66–67

optimizer_switch system variable, 66, 102
ORDER BY operator
creating covering index, 82
multi column indexes, 64–65
sorting data for SELECT query, 28
Using filesort, 151
ordering
of leftmost columns in multi column index, 63
non-support for reverse, 77
results with indexes, 57–58
results with WHERE and ORDER BY in multi column indexes, 64–65
ORM (object relational mapping) layer, application caching, 136
OSC (online schema change) tool, 74
output
interpreting EXPLAIN, 153–155
log_output variable, 100–101
SHOW INDEXES, 14

P

page fill factor, InnoDB, 76
partial column indexes, 77
partitioning, table, 43
partitions column, EXPLAIN PARTITIONS, 11, 150
PARTITIONS keyword, EXPLAIN command
identifying partitions used with, 150
overview of, 11
scanning only applicable indexes, 43
pattern matching, with indexes, 54–55
per-thread basis
defining global/session memory buffers on, 92
join_buffer_size, 99
max_heap_table_size, 97
sort_buffer_size, 99

tmp_table_size, 99
performance
adding indexes for, 26–27
alternative optimization solution, 7
confirming optimization, 5–6
finding and confirming slow query, 2–3
generating QEP, 3–4
impact of adding indexes, 71–76
what not to do, 5
performance tips
application caching, 136
caching SQL results, 134
combining your DDL, 124–125
eliminating SQL statements, 130–131
index column improvements, 126–130
monitoring ineffective indexes, 126
MySQL caching, 134–136
removing duplicate indexes, 125–126
removing duplicate SQL statements, 131
removing repeating SQL statements, 132–133
removing unnecessary SQL statements, 134
removing unused indexes, 126
rewriting subqueries, 138
simplifying SQL statements, 137–138
understanding impact of views, 139
using MySQL replication, 139
PERFORMANCE_SCHEMA, 112
PHP mysqlnd driver, 112
pinning limitations, QEP, 77
pluggable storage engines, 29
possible_keys column, EXPLAIN, 11, 147
primary keys

data integrity of, 25–26
disk space impact of adding indexes, 75
index types, 30–31
InnoDB B+tree clustered, 37–40
MyISAM vs. InnoDB, 35–36
PRIMARY value, select_type column, 149
PROCESSLIST command
capturing SQL statements, 110
sorting results in, 65
profiling, 101–102
profiling_history_size variable, 102

Q

Qcache_queries_in_cache status variable, 96
QEP (Query Execution Plan)
confirming your optimization, 5–6
creating covering index, 82
determining index usage, 126
determining with EXPLAIN, 10–11, 142
generating, 3–4
identifying problematic SQL statements, 117
optimizing existing indexes, 47–48
pinning limitations, 77
QEP (Query Execution Plan) columns
Extra, 150
key, 144
key_len, 147–148
overview of, 143–144
partitions, 150
possible_keys, 147
rows, 144–146
select_type, 149–150
table, 148–149

queries, complicated, 71
query hints, 67–69
query_cache_size variable, 95–96
query_cache_type variable, 95

R

R-tree data structure, 34
RAT (Row At a Time) processing, eliminating SQL statements, 132
read scalabilty, MySQL replication, 139
read_buffer_size variable, 99
read_rnd_buffer_size variable, 99
ref attribute, EXPLAIN command, 64
removing duplicate SQL statements, 131
removing repeating SQL statements, 132
removing unnecessary SQL statements, 133
Repairing the keycache, key_buffer_size, 92
replication, MySQL, 139
Row At a Time (RAT) processing, eliminating SQL statements, 132
ROW binary log format, 110
rows
generating query execution plan, 4
restricting with index, 48–50
restricting with WHERE and ORDER BY, 64–65
selecting unique, 55–57
SHOW TABLE STATUS and, 15
rows column, generating QEP, 4, 11, 144–146
running SQL statement, 2–3

S

secondary indexes
disk space impact of, 75–76
InnoDB B-tree secondary key, 40
MyISAM vs. InnoDB, 35–36
SELECT statements

creating covering indexes, 82
EXPLAIN, 142, 152
generating QEP, 3–4
against INFORMATION_SCHEMA tables, 16–17
sorting results with ORDER BY, 28
total query hints after, 67
selectivity, and cardinality, 52–53
select_type column, EXPLAIN command, 149–150
semicolon (;) terminator, 3
session buffers, 91–92
session memory usage, profiling, 101–102
SET command
defining variable size, 93
modifying dynamic variables at runtime, 90
SHOW VARIABLES with, 19
SHOW commands, INFORMATION_SCHEMA, 20
SHOW CREATE TABLE command
detailing table and index columns, 144
determining storage engine for table, 30
DML impact of adding indexes to table, 71–72
optimizing SQL statements, 6
overview of, 13
SHOW ENGINE INNODB STATUS command, 94–95, 110–111
SHOW FULL PROCESSLIST command, 2
SHOW [GLOBAL | SESSION] STATUS command, 17–19
SHOW [GLOBAL | SESSION] VARIABLES command, 19–20
SHOW GLOBAL STATUS command, 94–96
SHOW INDEXES command, 14–15, 147
SHOW INDEX_STATISTICS command Google Patch, 73, 126
SHOW PROFILES command, 120–121
SHOW STATUS command, 55–56, 145–146
SHOW TABLE STATUS command

determining storage engine for table, 30
InnoDB B+tree clustered primary key, 37
optimizing SQL statements, 6–7
overview of, 15–17
showing size of MEMORY engine table, 97
SHOW WARNINGS command, 12
SIGNED datatype, 128–129
SIMPLE value, select_type column, 149
simplifying SQL statements, 137–138
single column indexes. *See* indexes, single column
slaves, MySQL replication, 139
SLEEP() function, testing slow query log, 108–109
slow query log
capturing with, 108–109
identifying problematic statements, 117–118
slow_query_log variable, 100
slow_query_log_file variable, 100
sort_buffer_size variable, 99
sorting results, with indexes, 28
spatial indexes
MyISAM supporting, 34
overview of, 30–31
partitioned tables not supporting, 43
Sphinx search server, 35
SQL optimization lifecycle
analyzing statements, 121
capturing statements. *see* capture statements
confirming statement operation, 119–121
identifying problematic statements, 115–119
optimizing statements, 121
verifying results, 121–122
SQL statement statistics (sqlstats) plugin, 114

sql_mode variable, 103
SQL_NO_CACHE hint, 119–120
sqlstats (SQL statement statistics) plugin, 114
status, 17–19
SHOW ENGINE INNODB STATUS, 110–111
SHOW [GLOBAL | SESSION] STATUS, 17–19
SHOW GLOBAL STATUS, 94–96
SHOW STATUS, 17–19, 145–146
SHOW TABLE STATUS. *see* SHOW TABLE STATUS command
storage engines
choosing, 84
fast index creation, 74
generating limited index statistics, 77
implications of covering indexes, 84
index types and, 31
SHOW TABLE STATUS, 15
understanding, 28–30
STRAIGHT_JOIN query hint, 68–69
subqueries, rewriting, 138
system variables
configuration options, overview, 89
logging and instrumentation related, 100–102
memory related. *see* memory related variables
miscellaneous query related, 102–103

T

table column, EXPLAIN command, 148–149
table joins
improving with indexes, 28
simplifying SQL statement to improve, 137–138
TCP/IP
capturing SQL statements with, 114–115
identifying problematic SQL statements, 118–119

temporary tables, EXPLAIN command, 151
terminology, index, 30
thread_handling variable, 90
thread_pool_size variable, 90
TIMESTAMP vs. DATETIME datatypes, 127
timing
confirm statement operations, 120–121
SQL statement, 2–3
tmp_table_size variable, 97–99
Tokutek storage engine, 74
total query hints, 67–68
TTL (time-to-live), application caching, 136
type column
EXPLAIN, 153
generating QEP, 4

U

UDFs (User Defined Functions), application caching, 136
UNHEX() function, MD5 performance, 129–130
UNION RESULT value, select_type column, 150
UNION value, select_type column, 150
unique indexes, 55–57, 66
unique keys, 25–26, 30
UNSIGNED datatype, 128–129
unused indexes, removing, 126
UPDATE statement, 109, 142
usage, determining index, 73
USE INDEX hint, 69–71
User Defined Functions (UDFs), application caching, 136
Using filesort
EXPLAIN, 151
ordering results using index, 57
sort_buffer_size, 99

Using index, 82, 151
Using join buffer, 99, 151
Using where, EXPLAIN, 150–151
UTF8 character set, implied conversions, 129

V

VARCHAR data type, 129–130
variables. *See also* system variables
SHOW VARIABLES, 19–20
verification process, optimization, 121–122
version, SHOW TABLE STATUS, 15
views, impact of, 139

W

WHERE clause
affecting QEP of SELECT, 10–11
creating covering index, 82
EXPLAIN, 150–151
multi column indexes, 64–65
wildcards, pattern matching with, 54–55