MySQL Query & Index Tuning

Keith Murphy

MySQL DBA - iContact, Inc
Editor of MySQL Magazine @ http://www.paragon-cs.com/mag/

kmurphy@icontact.com

blog: http://blog.paragon-cs.com

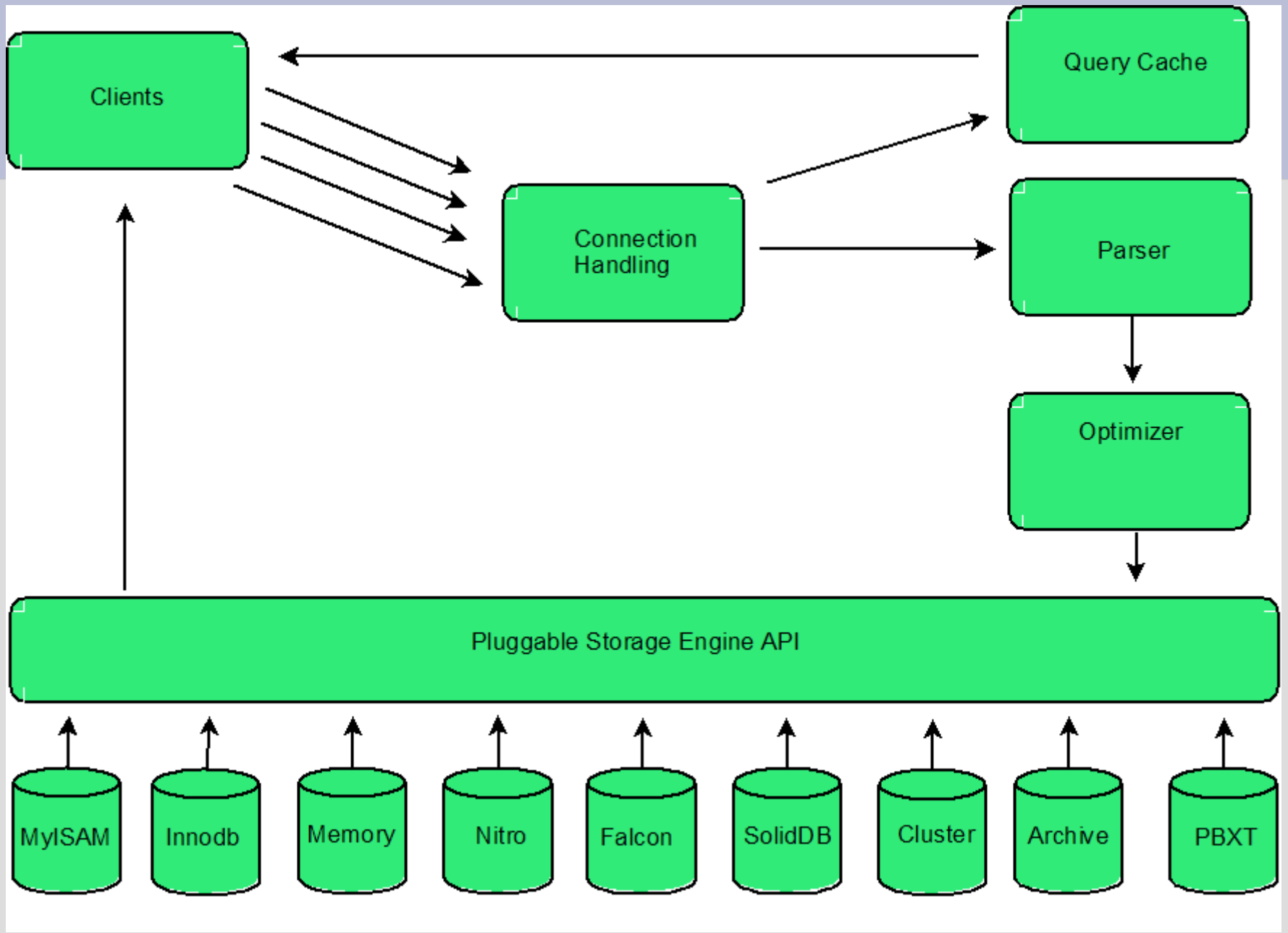
Rules

- Questions OK after any slide

The views and opinions expressed here are mine and may not reflect the views and opinions of iContact, Inc.

What We'll Cover

- MySQL Server Overview
- Slow Query Logging
- The EXPLAIN statement
- Things to Avoid in Queries
- Indexing Strategies
- The MySQL Optimizer
- Schema Guidelines
- Query Cache
- Benchmarking


Clients
Query Cache
Connection Handling
Parser
Optimizer
Pluggable Storage Engine API
MyISAM
Innodb
Memory
Nitro
Falcon
SolidDB
Cluster
Archive
PBXT

Slow Query Log

- logs queries that take more than a specified amount of time (defaults to ten seconds)
- resulting log can be “cat-ed/tail-ed” or you can use the mysqldumpslow command:
  - mysqldumpslow -s t
  - mysqldumpslow -s -c

sample output from log file:

```
# Time: 070911  2:06:37
# User@Host: username @ hostname [ip_address]
# Query_time: 20  Lock_time: 0  Rows_sent: 3979  Rows_examined: 3979
use database;
SELECT field_1 FROM table WHERE field_2 = '360973';
```

EXPLAIN command

- query tuners friend!!
- shows the execution path chosen by the MySQL optimizer for a specific SELECT statement

```
explain SELECT field_1 FROM table_1 WHERE field_2 = '360973'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: table_1
         type: ref
possible_keys: field_2
          key: field_2
      key_len: 4
          ref: const
         rows: 7174
        Extra: Using where
```

EXPLAIN Output Syntax

- id
- select_type
- table
- type
- possible_keys
- keys
- key_len
- ref
- rows
- extras

Explain Output

- **id** the sequential number of the table(s)
- **select_type** the type of SELECT
- **table** the name of the table or alias
- **type** the type of join for the query
- **possible_keys** which indexes MySQL could use
- **keys** which indexes MySQL will use
- **key_len** the length of keys used
- **ref** any columns used with the key to retrieve results
- **rows** estimated number of rows returned
- **extra** any additional information

Select Types – Part I

SIMPLE

basic SELECT without UNION or sub-queries

Example:

SELECT * FROM customers WHERE last_name = “smith”

PRIMARY

Outermost or highest level SELECT

DERIVED

If a query involves a derived table (including a view) it is assigned a DERIVED select type.

UNION

Second or later SELECT statement in a UNION

Example:

SELECT cm.customer_id FROM CUSTOMER_MASTER cm
WHERE cm.date_joined_program > “2002-01-15” and cm.home_airport_code = 'SEA'
UNION
SELECT cm1.csutomer_id FROM CUSTOMER_MASTER cm1 WHERE cm1.sex = 'M';

DEPENDENT UNION

Second or later SELECT statement in a UNION - dependent on outer query

Select Types – Part II

UNION RESULT

When the optimizer needs to create a temporary intermediate table to hold the results of a UNION

SUBQUERY

When a query includes a subquery the first SELECT in the subquery is identified as SUBQUERY

DEPENDENT SUBQUERY

If a subquery relies on information from an outer query the first SELECT in the subquery is labeled DEPENDENT SUBQUERY.

Example:

```
SELECT cm.last_name, cm.first_name
FROM customer_master cm
WHERE cm.customer_id IN
(
      SELECT ca.customer_id
      FROM customer_address ca
      WHERE ca.country = “Canada”
);
```

UNCACHEABLE SUBQUERY

A sub-query result which can’t be cached and is evaluated for each row of the outer query

Join Types – Part I

const – The query will have type of **const** (constant value) if the optimizer is able to fully use a unique index or primary key to satisfy your search. Because there is only one row, values from the column in this row can be regarded as constants by the rest of the optimizer. **const** tables are very fast because they are read only once. A **system** join is just a **const** join type reading a system table.

Example:

```
SELECT * FROM tbl_name WHERE primary_key=1;
SELECT * FROM tbl_name WHERE primary_key_part1=1 AND primary_key_part2=2;
```

eq_ref - one row is read from this table for each combination of rows from the previous tables. Other than the system and const types, this is the best possible join type. It is used when all parts of an index are used by the join and the index is a PRIMARY KEY or UNIQUE index. eq_ref can be used for indexed columns that are compared using the = operator. The comparison value can be a constant or an expression that uses columns from tables that are read before this table. In the following examples, MySQL can use an eq_ref join to process ref_table:

```
SELECT * FROM ref_table,other_table WHERE ref_table.key_column=other_table.COLUMN;

SELECT * FROM ref_table,other_table
WHERE ref_table.key_column_part1=other_table.COLUMN AND ref_table.key_column_part2=1;
```

Join Types – Part II

ref / **ref_or_null** – these types of joins use a nonunique index to find anywhere from one to many rows from a table.

Example:

SELECT * FROM customer_address ca where ca.postal_code=”27713”;

If the join type is **ref_or_null** than it means that query is also looking for null values from an index column.

index_merge – in this type of join multiple indexes are used to locate the data. With the index_merge the key column in the output row contains a list of indexes used.

unique_subquery - unique_subquery is just an index lookup function that replaces the subquery completely for better efficiency. This type replaces ref for some IN subqueries of the following form:

value IN (SELECT primary_key FROM single_table WHERE some_expr)

Join Types – Part III

index_subquery – this kind of query takes advantage of an index to speed results on a subquery:

Example:

SELECT cm.last_name, cm.first_name FROM customer_master cm WHERE cm.customer_id IN (SELECT ca.customer_id FROM customer_address ca WHERE ca.postal_code = “TVC15-3CPU”);

range - only rows that are in a given range are retrieved, using an index to select the rows. The key column in the output row indicates which index is used. The key_len contains the longest key part that was used. The ref column is NULL for this type. range can be used when a key column is compared to a constant using any of the =, <>, >, >=, <, <=, IS NULL, <=>, BETWEEN, or IN operators:

SELECT * FROM tbl_name WHERE key_column BETWEEN 10 AND 20;

index - this join type is the same as ALL, except that the index tree is scanned (instead of the entire table). This usually is faster than ALL because the index file usually is smaller than the data file. MySQL can use this join type when the query uses only columns that are part of a single index.

ALL – the optimizer is forced to perform a full-table scan and read all rows in a table to find your results.

Example:

SELECT last_name from customer_table where last_name LIKE “%john%”;

What to Avoid in Queries

- correlated subquery
- Version specific “gotchas”:
  - Prior to MySQL 5.0 queries only use one index
  - Prior to MySQL 5.0 in the case of OR conditions in a WHERE clause – MySQL uses a full-table scan

Indexing Strategies

- you should minimally use one index per table
- however, don't index every column!!
  - indexes are more costly to update
- always try and use indexes with high selectivity

Selectivity

Selectivity of an index - the ratio of the number of distinct values in the indexed column(s) to the number of records. The ideal selectivity is one. Such a selectivity can be reached only by unique indexes on NOT NULL columns (UNIQUE or PRIMARY KEY columns).

Example:

Query A select count (*) from users;
Query B select count(distinct username) from users;
B/A = selectivity

A higher selectivity (closer to one) is more desirable. If selectivity is too low the query optimizer won't use it.

Full Table Scans – Part I

Full table scans are where the database will read the entire table without an index.

- almost always not the desired behavior

How do we determine if MySQL is going to perform a full table scan?

- EXPLAIN

Full Table Scans – Part II

Reasons why full table scans are performed

- no WHERE clause
- no index on any field in WHERE clause
- poor selectivity on an indexed field
- too many records meet WHERE conditions
- MySQL version less than 5.0 and using OR in a WHERE clause
- using SELECT * FROM

Covering Indexes – Part I

A **covering index** is an indexing strategy where the index encompasses every field returned by a SELECT statement.

Why is this a good thing? Because it is faster to read everything from the index than use the index to look up information in rows.

Example:

You have an index (a,b) on table_1

SELECT b from table_1 where a=5

Covering Indexes – Part II

Where do covering indexes benefit?

- When you have large tables
- When you have long rows (BLOBS for example)
- When extra columns do not increase key length significantly
- When you have a large join with a number of secondary table lookups
- When a lot of rows match the same key value
- MyISAM tables benefit more than Innodb because MyISAM does not cache rows

Duplicate Indexes – Part I

When tables have multiple indexes defined on the same columns it has **duplicate indexes**.

Example:

```
PRIMARY KEY (id)
UNIQUE KEY id(id)
 KEY id2(id)
```

Duplicate Indexes – Part II

Notes:

Duplicate indexes apply to indexes of the same type. It may make sense to have indexes of different types created on the same column.

Example: BTREE index and a FULLTEXT index

Order of columns is important -
index (a,b) is not a duplicate of index (b,a)

Redundant Indexes – Part I

Redundant indexes are prefixes of other indexes

Example:

KEY (A)
KEY (A,B)
KEY (A(10))

Redundant indexes are almost never helpful.

Queries that take advantage of redundant indexes will also be able to make use of the longer indexes.

Redundant Indexes – Part II

When are redundant indexes useful?

If an index is just too long

Example: if A is an int and B is a varchar(255) which holds a great deal of long data than using KEY(A) might be significantly faster than using KEY (A,B)

There are no tools currently bundled with the MySQL distribution to check schemas for redundant and duplicate indexes.

There are available tools to do this though including:

- MySQL Toolkit (http://mysqltoolkit.sourceforge.net)
- A Java one also: (http://jroller.com/dschneller/entry/mysql_indices)

Complete Example

```
create database mysql_query optimization;

use mysql_query_optimization;

create table students (
student_id integer not null auto_increment primary key,
 name char(40));

create table tests (
test_id integer not null auto_increment primary key,
 total integer, name char(40));

create table grades (
test_id integer,
 student_id integer,
 score decimal(5,2));
```

Inserting Data

```
echo "insert into tests values (1,100, 'test 1');" > insert_tests.sql

echo "insert into tests values (2,100, 'test 2');" >> insert_tests.sql

echo "insert into tests values (3,300, 'test 3');" >> insert_tests.sql

echo "-- insert students" > insert_students.sql

for i in `seq 1 50000`
do
echo "insert into students values ($i, '$i diff');" >> insert_students.sql
done

mysql -u root -p mysql_query_optimization < insert_tests.sql

mysql -u root -p mysql_query_optimization < insert_students.sql
```

Inserting Grading Data

```
insert into grades select 1, student_id, rand()*100 from students
    order by rand() limit 40000 ;
insert into grades select 2, student_id, rand()*100 from students
    order by rand() limit 40000 ;
insert into grades select 3, student_id, rand()*300 from students
    order by rand() limit 40000 ;
```

Testing w/o Indexes – Part I

```
-- what was the average score on test 1?

mysql> select avg(score) from grades where test_id=1;
+--------------+
| avg(score)  |
+--------------+
 | 49.887198 |
+--------------+
1 row in set (0.06 sec)

-- what students didn't take test 1?

select count(*) from students s left join grades g on (s.student_id =
    g.student_id and test_id=2) where g.student_id is null ;

*** canceled query after 4200 seconds
```

Testing w/o Indexes – Part I EXPLAINed

```
mysql> explain select count(*) from students s left join grades g on (s.student_id =
    g.student_id and test_id=2) where g.student_id is null\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 50000
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 120000
        Extra: Using where
```

Testing w/o Indexes – Part II

-- how many students took zero tests?

```
select count(*) from students s left join grades g on (s.student_id =
    g.student_id) where g.student_id is null;
```

***canceled query after 5700 seconds

Testing w/o Indexes – Part II EXPLAINed

```
mysql> explain select count(*) from students s left join grades g on (s.student_id =
    g.student_id) where g.student_id is null\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 50000
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 120000
        Extra: Using where
```

Changes!!!

```
-mysql> alter table grades add index(test_id);
Query OK, 120000 rows affected (0.52 sec)
Records: 120000  Duplicates: 0  Warnings: 0

mysql> alter table grades add index(student_id);
Query OK, 120000 rows affected (0.88 sec)
Records: 120000  Duplicates: 0  Warnings: 0
```

Testing with Indexes – Part I

```
-- what students didn't take test 1?

mysql> select count(*) from students s left join grades g on
    (s.student_id = g.student_id and test_id=2) where g.student_id is
    null ;
+----------+
| count(*) |
+----------+
|    10000 |
+----------+
1 row in set (1.40 sec)
```

Testing with Indexes – Part I EXPLAINed

```
mysql> explain select count(*) from students s left join grades g on (s.student_id = g.student_id and
    test_id=2) where g.student_id is null\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 50000
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ref
possible_keys: test_id,student_id
          key: student_id
      key_len: 5
          ref: mysql_query_optimization.s.student_id
         rows: 2
        Extra: Using where
```

Testing with Indexes – Part II

```
-- how many students took zero tests?

mysql>  select count(*) from students s left join grades g on
    (s.student_id = g.student_id) where g.student_id is null;
+----------+
| count(*) |
+----------+
|      443  |
+----------+
1 row in set (1.19 sec)
```

Testing with Indexes – Part II EXPLAINed

```
mysql> explain select count(*) from students s left join grades g on (s.student_id =
    g.student_id) where g.student_id is null\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: s
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 50000
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ref
possible_keys: student_id
          key: student_id
      key_len: 5
          ref: mysql_query_optimization.s.student_id
         rows: 2
        Extra: Using where; Using index
```

MySQL Optimizer – Part I

There are two types of optimizers:

- Cost-based
- Rule-based

Cost-based optimizers estimates the execution times of performing a query various ways. It *tries* to choose the lowest execution time.

Rule-based optimizers build the best query based on a set of rules. Rule-based optimizers are quicker than cost-based optimizers but cost-based optimizers usually offer better performance. There are too many exceptions to the rules that the calculation overhead of cost-based optimizations is worthwhile.

MySQL Optimizer – Part II

Both cost-based and rule-based optimizers will return correct results.

The cost-based optimizer uses index selectivity to determine the best execution for joins.

Wrong Choices

The query optimizer will make wrong choices from time to time

- EXPLAIN is your friend!!

Schema Guidelines

Always use the smallest data type possible? Do you REALLY need that BIGINT?

- the smaller your data type the more index and data records can fit into a single block of memory
- normalize first and denormalize later

Query Cache

To use the QC effectively you must understand your applications read/write ratio.

- QC design is a compromise between CPU usage and read performance
- Bigger QC does not automatically mean better performance – even for heavy read applications
- Any modification of any table referenced in a SELECT will invalidate any QC entry that uses that table

Benchmarking – Part I

When performing benchmarks:

- change one item at a time
  - configuration (my.cnf) variable
  - addition of an index
  - modification to a schema table
  - change to a SQL script
- re-run the benchmark after making a change
- make comparisons apple–apple not apple-orange

Benchmarking – Part II

Isolate the benchmarking environment

- Disable non-essential services
  - Network traffic analyzers
  - Non essential daemons
  - MySQL query cache
- Use a testing machine if at all possible

Save the benchmark results

- Save all benchmark files
  - Result files of test runs
  - Configuration files at time of runs
  - OS/hardware configuration changes

Benchmarking – Part III

Why a benchmarking framework?

- automates the most tedious part of testing
- standardizes benchmark results
- can customize to your needs
- framework can be analyzed by outside parties
  - determine if framework provides consistent results
  - if framework is biased

Benchmarking Tools

Sysbench
Supersmack
AB (ApacheBench)

Generic Frameworks:

Junit/Ant (Java)
MyBench (Perl)
thewench (PHP)
QPP (Query Processing Programs)

Resources

Books:

SQL Tuning by Dan Tow
MySQL Database Design and Tuning by Robert Schneider

Websites:

http://www.mysqlperformanceblog.com
http://www.planetmysql.org
http://jpipes.com -- Target Practice Query Tuning Seminar

Mailing Lists:

http://lists.mysql.com (general list)