

Technical White Paper  
**MySQL Optimization**

---

OriosTech

orios

Author: Shahid Sheikh  
Amitava Mukherjee  
Date : 14/09/2004

---

**Table of Contents**

---

Optimization Overview.....3

Query Speed .....8

Locking Issues.....22

Optimizing Database Structure.....24

Optimizing the Server.....30

Disk issues.....38

How to optimize Application.....41

# MySQL Optimization

- [Optimize Overview](#): Optimization Overview
- [Query Speed](#): Optimizing **SELECT**s and Other Queries
- [Locking Issues](#): Locking Issues
- [Optimizing Database Structure](#): Optimizing Database Structure
- [Optimizing the Server](#): Optimizing the MySQL Server
- [Disk issues](#): Disk Issues

Optimization is a complicated task because it ultimately requires understanding of the whole system. While it may be possible to do some local optimizations with small knowledge of your system or application, the more optimal you want your system to become the more you will have to know about it.

This chapter will try to explain and give some examples of different ways to optimize MySQL. Remember, however, that there are always some (increasingly harder) additional ways to make the system even faster.

## 1. Optimization Overview

The most important part for getting a system fast is of course the basic design. You also need to know what kinds of things your system will be doing, and what your bottlenecks are.

The most common bottlenecks are: Disk seeks. It takes time for the disk to find a piece of data. With modern disks in 1999, the mean time for this is usually lower than 10ms, so we can in theory do about 1000 seeks a second. This time improves slowly with new disks and is very hard to optimize for a single table. The way to optimize this is to spread the data on more than one disk.

- Disk reading/writing. When the disk is at the correct position we need to read the data. With modern disks in 1999, one disk delivers something like 10-20Mb/s. This is easier to optimize than seeks because you can read in parallel from multiple disks.
- CPU cycles. When we have the data in main memory (or if it already were there) we need to process it to get to our result. Having small tables compared to the memory is the most common limiting factor. But then, with small tables speed is usually not the problem.
- Memory bandwidth. When the CPU needs more data than can fit in the CPU cache the main memory bandwidth becomes a bottleneck. This is an uncommon bottleneck for most systems, but one should be aware of it.
- [Design Limitations](#): MySQL Design Limitations/Tradeoffs
- [Portability](#): Portability
- [Internal use](#): What Have We Used MySQL For?
- [MySQL Benchmarks](#): The MySQL Benchmark Suite

- [Custom Benchmarks](#): Using Your Own Benchmarks

## 1.1 MySQL Design Limitations/Tradeoffs

Because MySQL uses extremely fast table locking (multiple readers / single writers) the biggest remaining problem is a mix of a steady stream of inserts and slow selects on the same table.

We believe that for a huge number of systems the extremely fast performance in other cases make this choice a win. This case is usually also possible to solve by having multiple copies of the table, but it takes more effort and hardware.

We are also working on some extensions to solve this problem for some common application niches.

## 1.2 Portability

Because all SQL servers implement different parts of SQL, it takes work to write portable SQL applications. For very simple Selects/Inserts it is very easy, but the more you need the harder it gets. If you want an application that is fast with many databases it becomes even harder!

To make a complex application portable you need to choose a number of SQL servers that it should work with.

You can use the MySQL crash-me program/web-page <http://www.mysql.com/information/crash-me.php> to find functions, types, and limits you can use with a selection of database servers. Crash-me now tests far from everything possible, but it is still comprehensive with about 450 things tested.

For example, you shouldn't have column names longer than 18 characters if you want to be able to use Informix or DB2.

Both the MySQL benchmarks and crash-me programs are very database-independent. By taking a look at how we have handled this, you can get a feeling for what you have to do to write your application database-independent. The benchmarks themselves can be found in the `'sql-bench'` directory in the MySQL source distribution. They are written in Perl with DBI database interface (which solves the access part of the problem).

For the results from this benchmark:

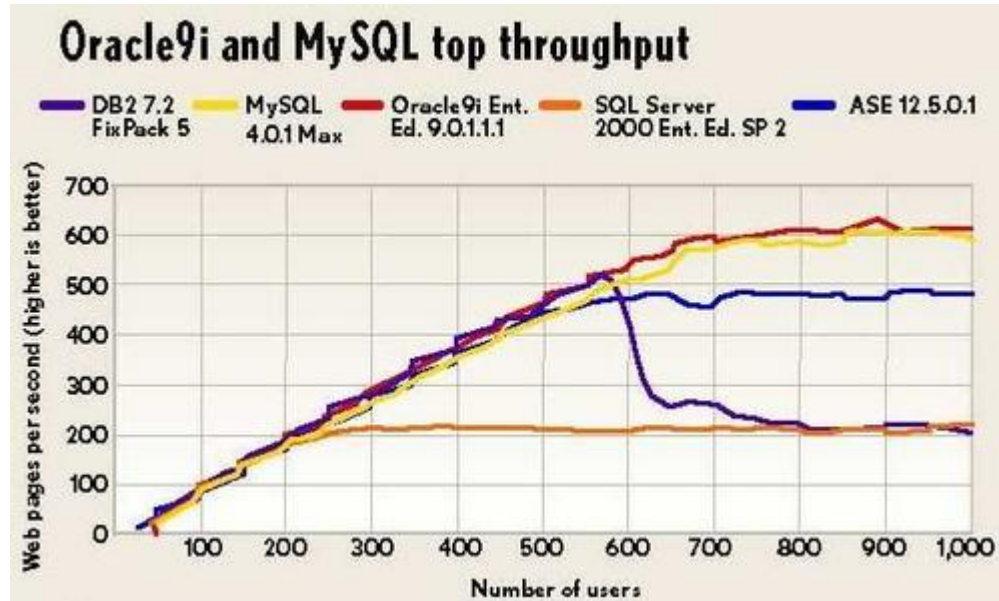
In a February 2002 database benchmark test performed by Ziff Davis Media Inc., the company behind PC Magazine, eWeek and other well-known publications, the MySQL database server stands out as a winner. The MySQL server is presented as having the overall best performance and scalability along with Oracle9i. Also, the MySQL server excelled in stability, ease of tuning and connectivity.

Some quotes from the eWeek article:

"Of the five databases we tested, only Oracle9i and MySQL were able to run our Nile application as originally written for 8 hours without problems."

"The Oracle and MySQL drivers had the best combination of a complete JDBC feature set and stability."

"SQL Server and MySQL were the easiest to tune, and Oracle9i was the most difficult because it has so many separate memory caches that can be adjusted."



The databases tested were: IBM DB2 7.2, Microsoft SQL Server 2000, MySQL-Max 4.0.1, Oracle 9i 9.0.1.1.1 and Sybase ASE 12.5.0.1

As you can see in these results, all databases have some weak points. That is, they have different design compromises that lead to different behavior.

If you strive for database independence, you need to get a good feeling for each SQL server's bottlenecks. MySQL is VERY fast in retrieving and updating things, but will have a problem in mixing slow readers/writers on the same table. Oracle, on the other hand, has a big problem when you try to access rows that you have recently updated (until they are flushed to disk). Transaction databases in general are not very good at generating summary tables from log tables, as in this case row locking is almost useless.

To get your application *really* database-independent, you need to define an easy extendable interface through which you manipulate your data. As C++ is available on most systems, it makes sense to use a C++ classes interface to the databases.

If you use some specific feature for some database (like the **REPLACE** command in MySQL), you should code a method for the other SQL servers to implement the same feature (but slower). With MySQL you can use the `/*! */` syntax to add MySQL-specific keywords to a query. The code inside `/**/` will be treated as a comment (ignored) by most other SQL servers.

If REAL high performance is more important than exactness, as in some Web applications, a possibility is to create an application layer that caches all results to give you even higher performance. By letting old results 'expire' after a while, you can keep the cache reasonably fresh. This is quite nice in case of extremely high load, in which case you can dynamically increase the cache and set the expire timeout higher until things get back to normal.

In this case the table creation information should contain information of the initial size of the cache and how often the table should normally be refreshed.

### 1.3 The MySQL Benchmark Suite

This should contain a technical description of the MySQL benchmark suite (and [crash-me](#)), but that description is not written yet. Currently, you can get a good idea of the benchmark by looking at the code and results in the ``sql-bench'` directory in any MySQL source distributions.

This benchmark suite is meant to be a benchmark that will tell any user what things a given SQL implementation performs well or poorly at.

Note that this benchmark is single threaded, so it measures the minimum time for the operations. We plan to in the future add a lot of multi-threaded tests to the benchmark suite.

For example, (run on the same NT 4.0 machine):

Reading 2000000 rows by index		
Seconds		Seconds
mysql	367	249
mysql_odbc	464	
db2_odbc	1206	
informix_odbc	121126	
ms-sql_odbc	1634	
oracle_odbc	20800	
solid_odbc	877	
sybase_odbc	17614	
Inserting (350768) rows		
Seconds		Seconds
mysql	381	206
mysql_odbc	619	
db2_odbc	3460	
informix_odbc	2692	
ms-sql_odbc	4012	
oracle_odbc	11291	
solid_odbc	1801	
sybase_odbc	4802	

In the above test MySQL was run with a 8M index cache.

We have gather some more benchmark results at <http://www.mysql.com/information/benchmarks.html>.

Note that Oracle is not included because they asked to be removed. All Oracle benchmarks have to be passed by Oracle! We believe that makes Oracle benchmarks **VERY** biased because the above benchmarks are supposed to show what a standard installation can do for a single client.

To run the benchmark suite, you have to download a MySQL source distribution, install the perl DBI driver, the perl DBD driver for the database you want to test and then do:

```
cd sql-bench
perl run-all-tests --server=#
```

where # is one of supported servers. You can get a list of all options and supported servers by doing `run-all-tests --help`.

`crash-me` tries to determine what features a database supports and what its capabilities and limitations are by actually running queries. For example, it determines:

- What column types are supported
- How many indexes are supported
- What functions are supported
- How big a query can be
- How big a `VARCHAR` column can be

We can find the result from `crash-me` on a lot of different databases at <http://www.mysql.com/information/crash-me.php>.

## 1.4 Using Your Own Benchmarks

Benchmarking is fundamentally a “what if” game. To make benchmarking as realistic and hassle-free as possible, here are several ways to consider:

*Change one thing at a time:* In science this is called as isolating the variable. No matter how well we think we understand the effects our changes will have, don’t make more than one change between test runs. Otherwise we’ll never know which one was responsible for the doubling of performance. We might be surprised to find that an adjustment we made once before to improve performance actually makes it worse in our current tests.

*Test Iteratively:* Try not to make dramatic changes. When adjusting MySQL’s buffers and caches, we’ll often be trying to find the smallest value that comfortably handles our load. Rather than increasing a value by 500%, start with a 50% or 100% increase and continue using that percentage increase on subsequent tests. We’ll find optimal values faster this way. Similarly, If while working from larger values to smaller, the time-tested “divide and conquer” technique is our best bet. Cut the current value in half, retest, and repeat the process until we’ve zeroed in close to the correct value.

*Always repeat tests:* By running each test several times (not fewer than four) and throwing out the first result, minimizes the chance of an outside influence getting in the way. Also consider restarting MySQL and even rebooting server between test runs to factor out caching artifacts.

*Use real data:* Try to use a realistic amount of data. If we planned to have 45 million rows in a table but test with only 45 thousand, we’ll find that performance drops quite a bit after the table is filled up and it has nothing to do with limits in MySQL.

Don’t use too many clients: We can find the optimal number of clients by using a simple iterative testing method. Start with number such as 20, and run the benchmark. Double the number, and run it again. Continue doubling it until the performance does not increase, meaning that the total queries per second stays same or decreases. Another option is to use data from logs to find out roughly how many concurrent users you handle during peak times.

## 2 Optimizing SELECTs and Other Queries

First, one thing that affects all queries: The more complex permission system setup you have, the more overhead you get.

If you do not have any **GRANT** statements done, MySQL will optimize the permission checking somewhat. So if you have a very high volume it may be worth the time to avoid grants. Otherwise more permission check results in a larger overhead.

If your problem is with some explicit MySQL function, you can always time this in the MySQL client:

```
mysql> select benchmark(1000000,1+1);
+-----+
| benchmark(1000000,1+1) |
+-----+
|           0 |
+-----+
1 row in set (0.32 sec)
```

The above shows that MySQL can execute 1,000,000 + expressions in 0.32 seconds on a PentiumII 400MHz.

All MySQL functions should be very optimized, but there may be some exceptions, and the **benchmark(loop\_count,expression)** is a great tool to find out if this is a problem with your query.

- [EXPLAIN](#): **EXPLAIN** Syntax (Get Information About a **SELECT**)
- [Estimating performance](#): Estimating query performance
- [SELECT speed](#): Speed of **SELECT** queries
- [Where optimizations](#): How MySQL optimizes **WHERE** clauses
- [DISTINCT optimization](#): How MySQL Optimizes **DISTINCT**
- [LEFT JOIN optimization](#): How MySQL optimizes **LEFT JOIN**
- [LIMIT optimization](#): How MySQL optimizes **LIMIT**
- [Insert speed](#): Speed of **INSERT** queries
- [Update speed](#): Speed of **UPDATE** queries
- [Delete speed](#): Speed of **DELETE** queries
- [Tips](#): Other Optimization Tips

## **2.1 EXPLAIN Syntax (Get Information About a SELECT)**

**EXPLAIN** tbl\_name or **EXPLAIN SELECT** select\_options

**EXPLAIN** tbl\_name is a synonym for **DESCRIBE** tbl\_name or **SHOW COLUMNS FROM** tbl\_name.

When you precede a **SELECT** statement with the keyword **EXPLAIN**, MySQL explains how it would process the **SELECT**, providing information about how tables are joined and in which order.

With the help of **EXPLAIN**, you can see when you must add indexes to tables to get a faster **SELECT** that uses indexes to find the records. You can also see if the optimizer joins the tables

in an optimal order. To force the optimizer to use a specific join order for a **SELECT** statement, add a **STRAIGHT\_JOIN** clause.

For non-simple joins, **EXPLAIN** returns a row of information for each table used in the **SELECT** statement. The tables are listed in the order they would be read. MySQL resolves all joins using a single-sweep multi-join method. This means that MySQL reads a row from the first table, then finds a matching row in the second table, then in the third table and so on. When all tables are processed, it outputs the selected columns and backtracks through the table list until a table is found for which there are more matching rows. The next row is read from this table and the process continues with the next table.

Output from **EXPLAIN** includes the following columns:

**table**

The table to which the row of output refers.

**type**

The join type. Information about the various types is given below.

**possible\_keys**

The **possible\_keys** column indicates which indexes MySQL could use to find the rows in this table. Note that this column is totally independent of the order of the tables. That means that some of the keys in **possible\_keys** may not be usable in practice with the generated table order. If this column is empty, there are no relevant indexes. In this case, you may be able to improve the performance of your query by examining the **WHERE** clause to see if it refers to some column or columns that would be suitable for indexing. If so, create an appropriate index and check the query with **EXPLAIN** again. To see what indexes a table has, use **SHOW INDEX FROM tbl\_name**.

**key**

The **key** column indicates the key that MySQL actually decided to use. The key is **NULL** if no index was chosen. If MySQL chooses the wrong index, you can probably force MySQL to use another index by using **mysamchk --analyze**, or by using **USE INDEX/IGNORE INDEX**.

**key\_len**

The **key\_len** column indicates the length of the key that MySQL decided to use. The length is **NULL** if the **key** is **NULL**. Note that this tells us how many parts of a multi-part key MySQL will actually use.

**ref**

The **ref** column shows which columns or constants are used with the **key** to select rows from the table.

**rows**

The **rows** column indicates the number of rows MySQL believes it must examine to execute the query.

**Extra**

This column contains additional information of how MySQL will resolve the query. Here is an explanation of the different text strings that can be found in this column:

**Distinct**

MySQL will not continue searching for more rows for the current row combination after it has found the first matching row.

**Not exists**

MySQL was able to do a **LEFT JOIN** optimization on the query and will not examine more rows in this table for the previous row combination after it finds one row that matches the **LEFT JOIN** criteria. Here is an example for this:

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.id=t2.id WHERE t2.id IS NULL;
```

Assume that **t2.id** is defined with **NOT NULL**. In this case MySQL will scan **t1** and look up the rows in **t2** through **t1.id**. If MySQL finds a matching row in **t2**, it knows that **t2.id** can never be **NULL**, and will not scan through the rest of the rows in **t2** that has the same **id**.

In other words, for each row in `t1`, MySQL only needs to do a single lookup in `t2`, independent of how many matching rows there are in `t2`.

**range checked for each record (index map: #)**

MySQL didn't find a real good index to use. It will, instead, for each row combination in the preceding tables, do a check on which index to use (if any), and use this index to retrieve the rows from the table. This isn't very fast but is faster than having to do a join without an index.

**Using filesort**

MySQL will need to do an extra pass to find out how to retrieve the rows in sorted order. The sort is done by going through all rows according to the **join type** and storing the sort key + pointer to the row for all rows that match the **WHERE**. Then the keys are sorted. Finally the rows are retrieved in sorted order.

**Using index**

The column information is retrieved from the table using only information in the index tree without having to do an additional seek to read the actual row. This can be done when all the used columns for the table are part of the same index.

**Using temporary**

To resolve the query MySQL will need to create a temporary table to hold the result. This typically happens if you do an **ORDER BY** on a different column set than you did a **GROUP BY** on.

**Where used**

A **WHERE** clause will be used to restrict which rows will be matched against the next table or sent to the client. If you don't have this information and the table is of type **ALL** or **index**, you may have something wrong in your query (if you don't intend to fetch/examine all rows from the table).

If you want to get your queries as fast as possible, you should look out for **Using filesort** and **Using temporary**.

The different join types are listed below, ordered from best to worst type:

**system**

The table has only one row (= system table). This is a special case of the **const** join type.

**const**

The table has at most one matching row, which will be read at the start of the query. 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 as they are read only once!

**eq\_ref**

One row will be read from this table for each combination of rows from the previous tables. This is the best possible join type, other than the **const** types. It is used when all parts of an index are used by the join and the index is **UNIQUE** or a **PRIMARY KEY**.

**ref**

All rows with matching index values will be read from this table for each combination of rows from the previous tables. **ref** is used if the join uses only a leftmost prefix of the key, or if the key is not **UNIQUE** or a **PRIMARY KEY** (in other words, if the join cannot select a single row based on the key value). If the key that is used matches only a few rows, this join type is good.

**range**

Only rows that are in a given range will be retrieved, using an index to select the rows. The **key** column indicates which index is used. The **key\_len** contains the longest key part that was used. The **ref** column will be NULL for this type.

**index**

This is the same as **ALL**, except that only the index tree is scanned. This is usually faster than **ALL**, as the index file is usually smaller than the data file.

**ALL**

A full table scan will be done for each combination of rows from the previous tables. This is normally not good if the table is the first table not marked **const**, and usually **very** bad in all other cases. You normally can avoid **ALL** by adding more indexes, so that the row can be retrieved based on constant values or column values from earlier tables.

You can get a good indication of how good a join is by multiplying all values in the **rows** column of the **EXPLAIN** output. This should tell you roughly how many rows MySQL must examine to execute the query. This number is also used when you restrict queries with the **max\_join\_size** variable.

The following example shows how a **JOIN** can be optimized progressively using the information provided by **EXPLAIN**.

Suppose you have the **SELECT** statement shown below, that you examine using **EXPLAIN**:

```
EXPLAIN SELECT tt.TicketNumber, tt.TimeIn,
             tt.ProjectReference, tt.EstimatedShipDate,
             tt.ActualShipDate, tt.ClientID,
             tt.ServiceCodes, tt.RepetitiveID,
             tt.CurrentProcess, tt.CurrentDPPerson,
             tt.RecordVolume, tt.DPPrinted, et.COUNTRY,
             et_1.COUNTRY, do.CUSTNAME
FROM tt, et, et AS et_1, do
WHERE tt.SubmitTime IS NULL
      AND tt.ActualPC = et.EMPLOYID
      AND tt.AssignedPC = et_1.EMPLOYID
      AND tt.ClientID = do.CUSTNMBR;
```

For this example, assume that:

- The columns being compared have been declared as follows:

Table	Column	Column type
tt	ActualPC	CHAR(10)
tt	AssignedPC	CHAR(10)
tt	ClientID	CHAR(10)
et	EMPLOYID	CHAR(15)
do	CUSTNMBR	CHAR(15)

- The tables have the indexes shown below:

Table	Index
tt	ActualPC
tt	AssignedPC
tt	ClientID
et	EMPLOYID (primary key)
do	CUSTNMBR (primary key)

- The **tt.ActualPC** values aren't evenly distributed.

Initially, before any optimizations have been performed, the **EXPLAIN** statement produces the following information:

```
table type possible_keys          key key_len ref rows Extra
et ALL PRIMARY                   NULL NULL NULL 74
do ALL PRIMARY                   NULL NULL NULL 2135
et_1 ALL PRIMARY                 NULL NULL NULL 74
```

```
tt ALL AssignedPC,ClientID,ActualIPC NULL NULL NULL 3872
   range checked for each record (key map: 35)
```

Because `type` is `ALL` for each table, this output indicates that MySQL is doing a full join for all tables! This will take quite a long time, as the product of the number of rows in each table must be examined! For the case at hand, this is  $74 * 2135 * 74 * 3872 = 45,268,558,720$  rows. If the tables were bigger, you can only imagine how long it would take.

One problem here is that MySQL can't (yet) use indexes on columns efficiently if they are declared differently. In this context, `VARCHAR` and `CHAR` are the same unless they are declared as different lengths. Because `tt.ActualIPC` is declared as `CHAR(10)` and `et.EMPLOYID` is declared as `CHAR(15)`, there is a length mismatch.

To fix this disparity between column lengths, use `ALTER TABLE` to lengthen `ActualIPC` from 10 characters to 15 characters:

```
mysql> ALTER TABLE tt MODIFY ActualIPC VARCHAR(15);
```

Now `tt.ActualIPC` and `et.EMPLOYID` are both `VARCHAR(15)`. Executing the `EXPLAIN` statement again produces this result:

```
table type possible_keys key key_len ref rows Extra
tt ALL AssignedPC,ClientID,ActualIPC NULL NULL NULL 3872 where used
do ALL PRIMARY NULL NULL NULL 2135
   range checked for each record (key map: 1)
et_1 ALL PRIMARY NULL NULL NULL 74
   range checked for each record (key map: 1)
et eq_ref PRIMARY PRIMARY 15 tt.ActualPC 1
```

This is not perfect, but is much better (the product of the `rows` values is now less by a factor of 74). This version is executed in a couple of seconds.

A second alteration can be made to eliminate the column length mismatches for the `tt.AssignedPC = et_1.EMPLOYID` and `tt.ClientID = do.CUSTNMBR` comparisons:

```
mysql> ALTER TABLE tt MODIFY AssignedPC VARCHAR(15),
        MODIFY ClientID VARCHAR(15);
```

Now `EXPLAIN` produces the output shown below:

```
table type possible_keys key key_len ref rows Extra
et ALL PRIMARY NULL NULL NULL 74
tt ref AssignedPC,ClientID,ActualIPC ActualPC 15 et.EMPLOYID 52 where used
et_1 eq_ref PRIMARY PRIMARY 15 tt.AssignedPC 1
do eq_ref PRIMARY PRIMARY 15 tt.ClientID 1
```

This is almost as good as it can get.

The remaining problem is that, by default, MySQL assumes that values in the `tt.ActualIPC` column are evenly distributed, and that isn't the case for the `tt` table. Fortunately, it is easy to tell MySQL about this:

```
shell> myisamchk --analyze PATH_TO_MYSQL_DATABASE/tt
shell> mysqladmin refresh
```

Now the join is perfect, and `EXPLAIN` produces this result:

```
table type possible_keys key key_len ref rows Extra
tt ALL AssignedPC,ClientID,ActualIPC NULL NULL NULL 3872 where used
et eq_ref PRIMARY PRIMARY 15 tt.ActualPC 1
et_1 eq_ref PRIMARY PRIMARY 15 tt.AssignedPC 1
do eq_ref PRIMARY PRIMARY 15 tt.ClientID 1
```

Note that the `rows` column in the output from `EXPLAIN` is an educated guess from the MySQL join optimizer. To optimize a query, you should check if the numbers are even close to the truth. If not, you may get better performance by using `STRAIGHT_JOIN` in your `SELECT` statement and trying to list the tables in a different order in the `FROM` clause.

## 2.2 Estimating Query Performance

In most cases you can estimate the performance by counting disk seeks. For small tables, you can usually find the row in 1 disk seek (as the index is probably cached). For bigger tables, you can estimate that (using B++ tree indexes) you will need:  $\log(\text{row\_count}) / \log(\text{index\_block\_length} / 3 * 2 / (\text{index\_length} + \text{data\_pointer\_length})) + 1$  seeks to find a row.

In MySQL an index block is usually 1024 bytes and the data pointer is usually 4 bytes. A 500,000 row table with an index length of 3 (medium integer) gives you:  $\log(500,000) / \log(1024/3*2/(3+4)) + 1 = 4$  seeks.

As the above index would require about  $500,000 * 7 * 3/2 = 5.2\text{M}$ , (assuming that the index buffers are filled to 2/3, which is typical) you will probably have much of the index in memory and you will probably only need 1-2 calls to read data from the OS to find the row.

For writes, however, you will need 4 seek requests (as above) to find where to place the new index and normally 2 seeks to update the index and write the row.

Note that the above doesn't mean that your application will slowly degenerate by  $N \log N$ ! As long as everything is cached by the OS or SQL server things will only go marginally slower while the table gets bigger. After the data gets too big to be cached, things will start to go much slower until your applications is only bound by disk-seeks (which increase by  $N \log N$ ). To avoid this, increase the index cache as the data grows.

## 2.3 Speed of SELECT Queries

In general, when you want to make a slow `SELECT ... WHERE` faster, the first thing to check is whether or not you can add an index. All references between different tables should usually be done with indexes. You can use the `EXPLAIN` command to determine which indexes are used for a `SELECT`..

Some general tips:

- To help MySQL optimize queries better, run `myisamchk --analyze` on a table after it has been loaded with relevant data. This updates a value for each index part that indicates the average number of rows that have the same value. (For unique indexes, this is always 1, of course.). MySQL will use this to decide which index to choose when you connect two tables with 'a non-constant expression'. You can check the result from the `analyze` run by doing `SHOW INDEX FROM table_name` and examining the `Cardinality` column.
- To sort an index and data according to an index, use `myisamchk --sort-index --sort-records=1` (if you want to sort on index 1). If you have a unique index from which you want to read all records in order according to that index, this is a good way to make that faster. Note, however, that this sorting isn't written optimally and will take a long time for a large table!

## 2.4 How MySQL Optimizes WHERE Clauses

The **WHERE** optimizations are put in the **SELECT** part here because they are mostly used with **SELECT**, but the same optimizations apply for **WHERE** in **DELETE** and **UPDATE** statements.

Also note that this section is incomplete. MySQL does many optimizations, and we have not had time to document them all.

Some of the optimizations performed by MySQL are listed below:

- Removal of unnecessary parentheses:
  - `((a AND b) AND c OR (((a AND b) AND (c AND d))))`
  - `-> (a AND b AND c) OR (a AND b AND c AND d)`
- Constant folding:
  - `(a<b AND b=c) AND a=5`
  - `-> b>5 AND b=c AND a=5`
- Constant condition removal (needed because of constant folding):
  - `(B>=5 AND B=5) OR (B=6 AND 5=5) OR (B=7 AND 5=6)`
  - `-> B=5 OR B=6`
- Constant expressions used by indexes are evaluated only once.
- **COUNT(\*)** on a single table without a **WHERE** is retrieved directly from the table information. This is also done for any **NOT NULL** expression when used with only one table.
- Early detection of invalid constant expressions. MySQL quickly detects that some **SELECT** statements are impossible and returns no rows.
- **HAVING** is merged with **WHERE** if you don't use **GROUP BY** or group functions (**COUNT()**, **MIN()**...).
- For each sub-join, a simpler **WHERE** is constructed to get a fast **WHERE** evaluation for each sub-join and also to skip records as soon as possible.
- All constant tables are read first, before any other tables in the query. A constant table is:
  - An empty table or a table with 1 row.
  - A table that is used with a **WHERE** clause on a **UNIQUE** index, or a **PRIMARY KEY**, where all index parts are used with constant expressions and the index parts are defined as **NOT NULL**.

All the following tables are used as constant tables:

```
mysql> SELECT * FROM t WHERE primary_key=1;
mysql> SELECT * FROM t1,t2
WHERE t1.primary_key=1 AND t2.primary_key=t1.id;
```

- The best join combination to join the tables is found by trying all possibilities. If all columns in **ORDER BY** and in **GROUP BY** come from the same table, then this table is preferred first when joining.

- If there is an **ORDER BY** clause and a different **GROUP BY** clause, or if the **ORDER BY** or **GROUP BY** contains columns from tables other than the first table in the join queue, a temporary table is created.
- If you use **SQL\_SMALL\_RESULT**, MySQL will use an in-memory temporary table.
- Each table index is queried, and the best index that spans fewer than 30% of the rows is used. If no such index can be found, a quick table scan is used.
- In some cases, MySQL can read rows from the index without even consulting the data file. If all columns used from the index are numeric, then only the index tree is used to resolve the query.
- Before each record is output, those that do not match the **HAVING** clause are skipped.

Some examples of queries that are very fast:

```
mysql> SELECT COUNT(*) FROM tbl_name;
mysql> SELECT MIN(key_part1),MAX(key_part1) FROM tbl_name;
mysql> SELECT MAX(key_part2) FROM tbl_name
      WHERE key_part_1=constant;
mysql> SELECT ... FROM tbl_name
      ORDER BY key_part1,key_part2,... LIMIT 10;
mysql> SELECT ... FROM tbl_name
      ORDER BY key_part1 DESC,key_part2 DESC,... LIMIT 10;
```

The following queries are resolved using only the index tree (assuming the indexed columns are numeric):

```
mysql> SELECT key_part1,key_part2 FROM tbl_name WHERE key_part1=val;
mysql> SELECT COUNT(*) FROM tbl_name
      WHERE key_part1=val1 AND key_part2=val2;
mysql> SELECT key_part2 FROM tbl_name GROUP BY key_part1;
```

The following queries use indexing to retrieve the rows in sorted order without a separate sorting pass:

```
mysql> SELECT ... FROM tbl_name ORDER BY key_part1,key_part2,... ;
mysql> SELECT ... FROM tbl_name ORDER BY key_part1 DESC,key_part2 DESC,... ;
```

## **2.5 How MySQL Optimizes DISTINCT**

**DISTINCT** is converted to a **GROUP BY** on all columns, **DISTINCT** combined with **ORDER BY** will in many cases also need a temporary table.

When combining **LIMIT #** with **DISTINCT**, MySQL will stop as soon as it finds **#** unique rows.

If you don't use columns from all used tables, MySQL will stop the scanning of the not used tables as soon as it has found the first match.

```
SELECT DISTINCT t1.a FROM t1,t2 where t1.a=t2.a;
```

In the case, assuming t1 is used before t2 (check with **EXPLAIN**), then MySQL will stop reading from t2 (for that particular row in t1) when the first row in t2 is found.

## 2.6 How MySQL Optimizes LEFT JOIN and RIGHT JOIN

A LEFT JOIN B in MySQL is implemented as follows:

- The table B is set to be dependent on table A and all tables that A is dependent on.
- The table A is set to be dependent on all tables (except B) that are used in the LEFT JOIN condition.
- All LEFT JOIN conditions are moved to the WHERE clause.
- All standard join optimizations are done, with the exception that a table is always read after all tables it is dependent on. If there is a circular dependence then MySQL will issue an error.
- All standard WHERE optimizations are done.
- If there is a row in A that matches the WHERE clause, but there wasn't any row in B that matched the LEFT JOIN condition, then an extra B row is generated with all columns set to NULL.
- If you use LEFT JOIN to find rows that don't exist in some table and you have the following test: column\_name IS NULL in the WHERE part, where column\_name is a column that is declared as NOT NULL, then MySQL will stop searching after more rows (for a particular key combination) after it has found one row that matches the LEFT JOIN condition.

RIGHT JOIN is implemented analogously as LEFT JOIN.

The table read order forced by LEFT JOIN and STRAIGHT JOIN will help the join optimizer (which calculates in which order tables should be joined) to do its work much more quickly, as there are fewer table permutations to check.

Note that the above means that if you do a query of type:

```
SELECT * FROM a,b LEFT JOIN c ON (c.key=a.key) LEFT JOIN d (d.key=a.key) WHERE b.key=d.key
```

MySQL will do a full scan on b as the LEFT JOIN will force it to be read before d.

The fix in this case is to change the query to:

```
SELECT * FROM b,a LEFT JOIN c ON (c.key=a.key) LEFT JOIN d (d.key=a.key) WHERE b.key=d.key
```

## 2.7 How MySQL Optimizes LIMIT

In some cases MySQL will handle the query differently when you are using LIMIT # and not using HAVING:

- If you are selecting only a few rows with LIMIT, MySQL will use indexes in some cases when it normally would prefer to do a full table scan.
- If you use LIMIT # with ORDER BY, MySQL will end the sorting as soon as it has found the first # lines instead of sorting the whole table.

- When combining **LIMIT #** with **DISTINCT**, MySQL will stop as soon as it finds # unique rows.
- In some cases a **GROUP BY** can be resolved by reading the key in order (or do a sort on the key) and then calculate summaries until the key value changes. In this case **LIMIT #** will not calculate any unnecessary **GROUP BY**'s.
- As soon as MySQL has sent the first # rows to the client, it will abort the query.
- **LIMIT 0** will always quickly return an empty set. This is useful to check the query and to get the column types of the result columns.
- The size of temporary tables uses the **LIMIT #** to calculate how much space is needed to resolve the query.

## 2.8 Speed of INSERT Queries

The time to insert a record consists approximately of:

- Connect: (3)
- Sending query to server: (2)
- Parsing query: (2)
- Inserting record: (1 x size of record)
- Inserting indexes: (1 x number of indexes)
- Close: (1)

where the numbers are somewhat proportional to the overall time. This does not take into consideration the initial overhead to open tables (which is done once for each concurrently running query).

The size of the table slows down the insertion of indexes by  $N \log N$  (B-trees).

Some ways to speed up inserts:

- If you are inserting many rows from the same client at the same time, use multiple value lists **INSERT** statements. This is much faster (many times in some cases) than using separate **INSERT** statements.
- If you are inserting a lot of rows from different clients, you can get higher speed by using the **INSERT DELAYED** statement.
- Note that with **MyISAM** you can insert rows at the same time **SELECT**s are running if there are no deleted rows in the tables.
- When loading a table from a text file, use **LOAD DATA INFILE**. This is usually 20 times faster than using a lot of **INSERT** statements.
- It is possible with some extra work to make **LOAD DATA INFILE** run even faster when the table has many indexes. Use the following procedure:

1. Optionally create the table with **CREATE TABLE**. For example, using `mysql` or `Perl-DBI`.
2. Execute a **FLUSH TABLES** statement or the shell command `mysqladmin flush-tables`.
3. Use `myisamchk --keys-used=0 -rq /path/to/db/tbl_name`. This will remove all usage of all indexes from the table.
4. Insert data into the table with **LOAD DATA INFILE**. This will not update any indexes and will therefore be very fast.
5. If you are going to only read the table in the future, run `myisampack` on it to make it smaller.
6. Re-create the indexes with `myisamchk -r -q /path/to/db/tbl_name`. This will create the index tree in memory before writing it to disk, which is much faster because it avoids lots of disk seeks. The resulting index tree is also perfectly balanced.
7. Execute a **FLUSH TABLES** statement or the shell command `mysqladmin flush-tables`.

This procedure will be built into **LOAD DATA INFILE** in some future version of MySQL.

- You can speed up insertions by locking your tables:
- `mysql> LOCK TABLES a WRITE;`
- `mysql> INSERT INTO a VALUES (1,23),(2,34),(4,33);`
- `mysql> INSERT INTO a VALUES (8,26),(6,29);`
- `mysql> UNLOCK TABLES;`

The main speed difference is that the index buffer is flushed to disk only once, after all **INSERT** statements have completed. Normally there would be as many index buffer flushes as there are different **INSERT** statements. Locking is not needed if you can insert all rows with a single statement. Locking will also lower the total time of multi-connection tests, but the maximum wait time for some threads will go up (because they wait for locks). For example:

```
thread 1 does 1000 inserts
thread 2, 3, and 4 does 1 insert
thread 5 does 1000 inserts
```

If you don't use locking, 2, 3, and 4 will finish before 1 and 5, but the total time should be about 40% faster. As **INSERT**, **UPDATE**, and **DELETE** operations are very fast in MySQL, you will obtain better overall performance by adding locks around everything that does more than about 5 inserts or updates in a row. If you do very many inserts in a row, you could do a **LOCK TABLES** followed by an **UNLOCK TABLES** once in a while (about every 1000 rows) to allow other threads access to the table. This would still result in a nice performance gain. Of course, **LOAD DATA INFILE** is much faster for loading data.

To get some more speed for both **LOAD DATA INFILE** and **INSERT**, enlarge the key buffer.

## 2.9 Speed of UPDATE Queries

Update queries are optimized as a **SELECT** query with the additional overhead of a write. The speed of the write is dependent on the size of the data that is being updated and the number of indexes that are updated. Indexes that are not changed will not be updated.

Also, another way to get fast updates is to delay updates and then do many updates in a row later. Doing many updates in a row is much quicker than doing one at a time if you lock the table.

Note that, with dynamic record format, updating a record to a longer total length may split the record. So if you do this often, it is very important to **OPTIMIZE TABLE** sometimes.

## **2.10 Speed of DELETE Queries**

If you want to delete all rows in the table, you should use **TRUNCATE TABLE table\_name..**

The time to delete a record is exactly proportional to the number of indexes. To delete records more quickly, you can increase the size of the index cache.

## **2.11 Other Optimization Tips**

Unsorted tips for faster systems:

- Use persistent connections to the database to avoid the connection overhead. If you can't use persistent connections and you are doing a lot of new connections to the database, you may want to change the value of the **thread\_cache\_size** variable.
- Always check that all your queries really use the indexes you have created in the tables. In MySQL you can do this with the **EXPLAIN** command.
- Try to avoid complex **SELECT** queries on tables that are updated a lot. This is to avoid problems with table locking.
- The new **MyISAM** tables can insert rows in a table without deleted rows at the same time another table is reading from it. If this is important for you, you should consider methods where you don't have to delete rows or run **OPTIMIZE TABLE** after you have deleted a lot of rows.
- Use **ALTER TABLE ... ORDER BY expr1,expr2...** if you mostly retrieve rows in **expr1,expr2..** order. By using this option after big changes to the table, you may be able to get higher performance.
- In some cases it may make sense to introduce a column that is 'hashed' based on information from other columns. If this column is short and reasonably unique it may be much faster than a big index on many columns. In MySQL it's very easy to use this extra column: **SELECT \* FROM table\_name WHERE hash=MD5(concat(col1,col2)) AND col\_1='constant' AND col\_2='constant'**
- For tables that change a lot you should try to avoid all **VARCHAR** or **BLOB** columns. You will get dynamic row length as soon as you are using a single **VARCHAR** or **BLOB** column.
- It's not normally useful to split a table into different tables just because the rows gets 'big'. To access a row, the biggest performance hit is the disk seek to find the first byte of the row. After finding the data most new disks can read the whole row fast enough for most applications. The only cases where it really matters to split up a table is if it's a dynamic row size table (see above) that you can change to a fixed row size, or if you very often need to scan the table and don't need most of the columns.
- If you very often need to calculate things based on information from a lot of rows (like counts of things), it's probably much better to introduce a new table and update the

- counter in real time. An update of type `UPDATE table set count=count+1 where index_column=constant` is very fast! This is really important when you use databases like MySQL that only have table locking (multiple readers / single writers). This will also give better performance with most databases, as the row locking manager in this case will have less to do.
- If you need to collect statistics from big log tables, use summary tables instead of scanning the whole table. Maintaining the summaries should be much faster than trying to do statistics 'live'. It's much faster to regenerate new summary tables from the logs when things change (depending on business decisions) than to have to change the running application!
  - If possible, one should classify reports as 'live' or 'statistical', where data needed for statistical reports are only generated based on summary tables that are generated from the actual data.
  - Take advantage of the fact that columns have default values. Insert values explicitly only when the value to be inserted differs from the default. This reduces the parsing that MySQL need to do and improves the insert speed.
  - In some cases it's convenient to pack and store data into a blob. In this case you have to add some extra code in your application to pack/unpack things in the blob, but this may save a lot of accesses at some stage. This is practical when you have data that doesn't conform to a static table structure.
  - Normally you should try to keep all data non-redundant (what is called 3rd normal form in database theory), but you should not be afraid of duplicating things or creating summary tables if you need these to gain more speed.
  - Stored procedures or UDF (user-defined functions) may be a good way to get more performance. In this case you should, however, always have a way to do this some other (slower) way if you use some database that doesn't support this.
  - You can always gain something by caching queries/answers in your application and trying to do many inserts/updates at the same time. If your database supports lock tables (like MySQL and Oracle), this should help to ensure that the index cache is only flushed once after all updates.
  - Use `INSERT /*! DELAYED */` when you do not need to know when your data is written. This speeds things up because many records can be written with a single disk write.
  - Use `INSERT /*! LOW_PRIORITY */` when you want your selects to be more important.
  - Use `SELECT /*! HIGH_PRIORITY */` to get selects that jump the queue. That is, the select is done even if there is somebody waiting to do a write.
  - Use the multi-line `INSERT` statement to store many rows with one SQL command (many SQL servers supports this).
  - Use `LOAD DATA INFILE` to load bigger amounts of data. This is faster than normal inserts and will be even faster when `myisamchk` is integrated in `mysqld`.
  - Use `AUTO_INCREMENT` columns to make unique values.

- Use **OPTIMIZE TABLE** once in a while to avoid fragmentation when using dynamic table format.
- Use **HEAP** tables to get more speed when possible.
- When using a normal Web server setup, images should be stored as files. That is, store only a file reference in the database. The main reason for this is that a normal Web server is much better at caching files than database contents. So it's much easier to get a fast system if you are using files.
- Use in memory tables for non-critical data that are accessed often (like information about the last shown banner for users that don't have cookies).
- Columns with identical information in different tables should be declared identical and have identical names. Before Version 3.23 you got slow joins otherwise. Try to keep the names simple (use **name** instead of **customer\_name** in the customer table). To make your names portable to other SQL servers you should keep them shorter than 18 characters.
- If you need REALLY high speed, you should take a look at the low-level interfaces for data storage that the different SQL servers support! For example, by accessing the MySQL **MyISAM** directly, you could get a speed increase of 2-5 times compared to using the SQL interface. To be able to do this the data must be on the same server as the application, and usually it should only be accessed by one process (because external file locking is really slow). One could eliminate the above problems by introducing low-level **MyISAM** commands in the MySQL server (this could be one easy way to get more performance if needed). By carefully designing the database interface, it should be quite easy to support this types of optimization.
- In many cases it's faster to access data from a database (using a live connection) than accessing a text file, just because the database is likely to be more compact than the text file (if you are using numerical data), and this will involve fewer disk accesses. You will also save code because you don't have to parse your text files to find line and column boundaries.
- You can also use replication to speed things up.
- Declaring a table with **DELAY\_KEY\_WRITE=1** will make the updating of indexes faster, as these are not logged to disk until the file is closed. The downside is that you should run **myisamchk** on these tables before you start **mysqld** to ensure that they are okay if something killed **mysqld** in the middle. As the key information can always be generated from the data, you should not lose anything by using **DELAY\_KEY\_WRITE**.

## 3 Locking Issues

- [Internal locking](#): How MySQL Locks Tables
- [Table locking](#): Table Locking Issues

### 3.1 How MySQL Locks Tables

You can find a discussion about different locking methods in the appendix.

All locking in MySQL is deadlock-free. This is managed by always requesting all needed locks at once at the beginning of a query and always locking the tables in the same order.

The locking method MySQL uses for **WRITE** locks works as follows:

- If there are no locks on the table, put a write lock on it.
- Otherwise, put the lock request in the write lock queue.

The locking method MySQL uses for **READ** locks works as follows:

- If there are no write locks on the table, put a read lock on it.
- Otherwise, put the lock request in the read lock queue.

When a lock is released, the lock is made available to the threads in the write lock queue, then to the threads in the read lock queue.

This means that if you have many updates on a table, **SELECT** statements will wait until there are no more updates.

To work around this for the case where you want to do many **INSERT** and **SELECT** operations on a table, you can insert rows in a temporary table and update the real table with the records from the temporary table once in a while.

This can be done with the following code:

```
mysql> LOCK TABLES real_table WRITE, insert_table WRITE;
mysql> insert into real_table select * from insert_table;
mysql> TRUNCATE TABLE insert_table;
mysql> UNLOCK TABLES;
```

You can use the **LOW\_PRIORITY** options with **INSERT**, **UPDATE** or **DELETE** or **HIGH\_PRIORITY** with **SELECT** if you want to prioritize retrieval in some specific cases. You can also start `mysqld` with `--low-priority-updates` to get the same behaviour.

Using **SQL\_BUFFER\_RESULT** can also help making table locks shorter.

You could also change the locking code in ``mysys/thr_lock.c'` to use a single queue. In this case, write locks and read locks would have the same priority, which might help some applications.

## 3.2 Table Locking Issues

The table locking code in MySQL is deadlock free.

MySQL uses table locking (instead of row locking or column locking) on all table types, except **BDB** tables, to achieve a very high lock speed. For large tables, table locking is MUCH better than row locking for most applications, but there are, of course, some pitfalls.

For **BDB** and **InnoDB** tables, MySQL only uses table locking if you explicitly lock the table with **LOCK TABLES** or execute a command that will modify every row in the table, like **ALTER TABLE**. For these table types we recommend you to not use **LOCK TABLES** at all.

In MySQL Version 3.23.7 and above, you can insert rows into **MyISAM** tables at the same time other threads are reading from the table. Note that currently this only works if there are no holes

after deleted rows in the table at the time the insert is made. When all holes has been filled with new data, concurrent inserts will automatically be enabled again.

Table locking enables many threads to read from a table at the same time, but if a thread wants to write to a table, it must first get exclusive access. During the update, all other threads that want to access this particular table will wait until the update is ready.

As updates on tables normally are considered to be more important than **SELECT**, all statements that update a table have higher priority than statements that retrieve information from a table. This should ensure that updates are not 'starved' because one issues a lot of heavy queries against a specific table. (You can change this by using **LOW\_PRIORITY** with the statement that does the update or **HIGH\_PRIORITY** with the **SELECT** statement.)

Starting from MySQL Version 3.23.7 one can use the **max\_write\_lock\_count** variable to force MySQL to temporary give all **SELECT** statements, that wait for a table, a higher priority after a specific number of inserts on a table.

Table locking is, however, not very good under the following senario:

- A client issues a **SELECT** that takes a long time to run.
- Another client then issues an **UPDATE** on a used table. This client will wait until the **SELECT** is finished.
- Another client issues another **SELECT** statement on the same table. As **UPDATE** has higher priority than **SELECT**, this **SELECT** will wait for the **UPDATE** to finish. It will also wait for the first **SELECT** to finish!
- A thread is waiting for something like **full disk**, in which case all threads that wants to access the problem table will also be put in a waiting state until more disk space is made available.

Some possible solutions to this problem are:

- Try to get the **SELECT** statements to run faster. You may have to create some summary tables to do this.
- Start **mysqld** with **--low-priority-updates**. This will give all statements that update (modify) a table lower priority than a **SELECT** statement. In this case the last **SELECT** statement in the previous scenario would execute before the **INSERT** statement.
- You can give a specific **INSERT**, **UPDATE**, or **DELETE** statement lower priority with the **LOW\_PRIORITY** attribute.
- Start **mysqld** with a low value for **max\_write\_lock\_count** to give **READ** locks after a certain number of **WRITE** locks.
- You can specify that all updates from a specific thread should be done with low priority by using the SQL command: **SET SQL\_LOW\_PRIORITY\_UPDATES=1**.
- You can specify that a specific **SELECT** is very important with the **HIGH\_PRIORITY** attribute.
- If you have problems with **INSERT** combined with **SELECT**, switch to use the new **MyISAM** tables as these support concurrent **SELECT**s and **INSERT**s.

- If you mainly mix **INSERT** and **SELECT** statements, the **DELAYED** attribute to **INSERT** will probably solve your problems.
- If you have problems with **SELECT** and **DELETE**, the **LIMIT** option to **DELETE** may help.

## **4 Optimizing Database Structure**

- [Design](#): Design Choices
- [Data size](#): Get Your Data as Small as Possible
- [MySQL indexes](#): How MySQL Uses Indexes
- [Indexes](#): Column Indexes
- [Multiple-column indexes](#): Multiple-Column Indexes
- [Table cache](#): How MySQL Opens and Closes Tables
- [Creating many tables](#): Drawbacks to Creating Large Numbers of Tables in the Same Database
- [Open tables](#): Why So Many Open tables?

### **4.1 Design Choices**

MySQL keeps row data and index data in separate files. Many (almost all) other databases mix row and index data in the same file. We believe that the MySQL choice is better for a very wide range of modern systems.

Another way to store the row data is to keep the information for each column in a separate area (examples are SDBM and Focus). This will cause a performance hit for every query that accesses more than one column. Because this degenerates so quickly when more than one column is accessed, we believe that this model is not good for general purpose databases.

The more common case is that the index and data are stored together (like in Oracle/Sybase et al). In this case you will find the row information at the leaf page of the index. The good thing with this layout is that it, in many cases, depending on how well the index is cached, saves a disk read. The bad things with this layout are:

- Table scanning is much slower because you have to read through the indexes to get at the data.
- You can't use only the index table to retrieve data for a query.
- You lose a lot of space, as you must duplicate indexes from the nodes (as you can't store the row in the nodes).
- Deletes will degenerate the table over time (as indexes in nodes are usually not updated on delete).
- It's harder to cache ONLY the index data.

### **4.2 Get Your Data as Small as Possible**

One of the most basic optimization is to get your data (and indexes) to take as little space on the disk (and in memory) as possible. This can give huge improvements because disk reads are faster and normally less main memory will be used. Indexing also takes less resources if done on smaller columns.

MySQL supports a lot of different table types and row formats. Choosing the right table format may give you a big performance gain.

You can get better performance on a table and minimize storage space using the techniques listed below:

- Use the most efficient (smallest) types possible. MySQL has many specialized types that save disk space and memory.
- Use the smaller integer types if possible to get smaller tables. For example, `MEDIUMINT` is often better than `INT`.
- Declare columns to be `NOT NULL` if possible. It makes everything faster and you save one bit per column. Note that if you really need `NULL` in your application you should definitely use it. Just avoid having it on all columns by default.
- If you don't have any variable-length columns (`VARCHAR`, `TEXT`, or `BLOB` columns), a fixed-size record format is used. This is faster but unfortunately may waste some space.
- The primary index of a table should be as short as possible. This makes identification of one row easy and efficient.
- For each table, you have to decide which storage/index method to use.
- Only create the indexes that you really need. Indexes are good for retrieval but bad when you need to store things fast. If you mostly access a table by searching on a combination of columns, make an index on them. The first index part should be the most used column. If you are ALWAYS using many columns, you should use the column with more duplicates first to get better compression of the index.
- If it's very likely that a column has a unique prefix on the first number of characters, it's better to only index this prefix. MySQL supports an index on a part of a character column. Shorter indexes are faster not only because they take less disk space but also because they will give you more hits in the index cache and thus fewer disk seeks.
- In some circumstances it can be beneficial to split into two a table that is scanned very often. This is especially true if it is a dynamic format table and it is possible to use a smaller static format table that can be used to find the relevant rows when scanning the table.

### 4.3 How MySQL Uses Indexes

Indexes are used to find rows with a specific value of one column fast. Without an index MySQL has to start with the first record and then read through the whole table until it finds the relevant rows. The bigger the table, the more this costs. If the table has an index for the columns in question, MySQL can quickly get a position to seek to in the middle of the data file without having to look at all the data. If a table has 1000 rows, this is at least 100 times faster than reading sequentially. Note that if you need to access almost all 1000 rows it is faster to read sequentially because we then avoid disk seeks.

All MySQL indexes (**PRIMARY**, **UNIQUE**, and **INDEX**) are stored in B-trees. Strings are automatically prefix- and end-space compressed. See section [CREATE INDEX Syntax](#).

Indexes are used to:

- Quickly find the rows that match a **WHERE** clause.
- Retrieve rows from other tables when performing joins.
- Find the **MAX()** or **MIN()** value for a specific indexed column. This is optimized by a preprocessor that checks if you are using **WHERE** `key_part_# = constant` on all key parts < N. In this case MySQL will do a single key lookup and replace the **MIN()** expression with a constant. If all expressions are replaced with constants, the query will return at once:
  - **SELECT MIN(key\_part2),MAX(key\_part2) FROM table\_name where key\_part1=10**
- Sort or group a table if the sorting or grouping is done on a leftmost prefix of a usable key (for example, **ORDER BY** `key_part_1,key_part_2`). The key is read in reverse order if all key parts are followed by **DESC**. The index can also be used even if the **ORDER BY** doesn't match the index exactly, as long as all the unused index parts and all the extra **ORDER BY** columns are constants in the **WHERE** clause. The following queries will use the index to resolve the **ORDER BY** part:
  - **SELECT \* FROM foo ORDER BY key\_part1,key\_part2,key\_part3;**
  - **SELECT \* FROM foo WHERE column=constant ORDER BY column, key\_part1;**
  - **SELECT \* FROM foo WHERE key\_part1=const GROUP BY key\_part2;**
- In some cases a query can be optimized to retrieve values without consulting the data file. If all used columns for some table are numeric and form a leftmost prefix for some key, the values may be retrieved from the index tree for greater speed:
  - **SELECT key\_part3 FROM table\_name WHERE key\_part1=1**

Suppose you issue the following **SELECT** statement:

```
mysql> SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2;
```

If a multiple-column index exists on **col1** and **col2**, the appropriate rows can be fetched directly. If separate single-column indexes exist on **col1** and **col2**, the optimizer tries to find the most restrictive index by deciding which index will find fewer rows and using that index to fetch the rows.

If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to find rows. For example, if you have a three-column index on **(col1,col2,col3)**, you have indexed search capabilities on **(col1)**, **(col1,col2)**, and **(col1,col2,col3)**.

MySQL can't use a partial index if the columns don't form a leftmost prefix of the index. Suppose you have the **SELECT** statements shown below:

```
mysql> SELECT * FROM tbl_name WHERE col1=val1;
mysql> SELECT * FROM tbl_name WHERE col2=val2;
mysql> SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3;
```

If an index exists on **(col1,col2,col3)**, only the first query shown above uses the index. The second and third queries do involve indexed columns, but **(col2)** and **(col2,col3)** are not leftmost prefixes of **(col1,col2,col3)**.

MySQL also uses indexes for **LIKE** comparisons if the argument to **LIKE** is a constant string that doesn't start with a wild-card character. For example, the following **SELECT** statements use indexes:

```
mysql> select * from tbl_name where key_col LIKE "Patrick%";
mysql> select * from tbl_name where key_col LIKE "Pat%_ck%";
```

In the first statement, only rows with "Patrick" <= key\_col < "Patricl" are considered. In the second statement, only rows with "Pat" <= key\_col < "Pau" are considered.

The following **SELECT** statements will not use indexes:

```
mysql> select * from tbl_name where key_col LIKE "%Patrick%";
mysql> select * from tbl_name where key_col LIKE other_col;
```

In the first statement, the **LIKE** value begins with a wild-card character. In the second statement, the **LIKE** value is not a constant.

Searching using **column\_name IS NULL** will use indexes if column\_name is an index.

MySQL normally uses the index that finds the least number of rows. An index is used for columns that you compare with the following operators: =, >, >=, <, <=, **BETWEEN**, and a **LIKE** with a non-wild-card prefix like 'something%'.

Any index that doesn't span all **AND** levels in the **WHERE** clause is not used to optimize the query. In other words: To be able to use an index, a prefix of the index must be used in every **AND** group.

The following **WHERE** clauses use indexes:

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3
... WHERE index=1 OR A=10 AND index=2    /* index = 1 OR index = 2 */
... WHERE index_part1='hello' AND index_part_3=5
    /* optimized like "index_part1='hello'" */
... WHERE index1=1 and index2=2 or index1=3 and index3=3;
    /* Can use index on index1 but not on index2 or index 3 */
```

These **WHERE** clauses do **NOT** use indexes:

```
... WHERE index_part2=1 AND index_part3=2 /* index_part_1 is not used */
... WHERE index=1 OR A=10                /* Index is not used in both AND parts */
... WHERE index_part1=1 OR index_part2=10 /* No index spans all rows */
```

Note that in some cases MySQL will not use an index, even if one would be available. Some of the cases where this happens are:

- If the use of the index would require MySQL to access more than 30 % of the rows in the table. (In this case a table scan is probably much faster, as this will require us to do much fewer seeks). Note that if such a query uses **LIMIT** to only retrieve part of the rows, MySQL will use an index anyway, as it can much more quickly find the few rows to return in the result.
- [Indexes](#): Column Indexes
- [Multiple-column indexes](#): Multiple-Column Indexes
- [Table cache](#): How MySQL Opens and Closes Tables

- [Creating many tables](#): Drawbacks to Creating Large Numbers of Tables in the Same Database
- [Open tables](#): Why So Many Open tables?

## 4.4 Column Indexes

All MySQL column types can be indexed. Use of indexes on the relevant columns is the best way to improve the performance of **SELECT** operations.

The maximum number of keys and the maximum index length is defined per table handler. You can with all table handlers have at least 16 keys and a total index length of at least 256 bytes.

For **CHAR** and **VARCHAR** columns, you can index a prefix of a column. This is much faster and requires less disk space than indexing the whole column. The syntax to use in the **CREATE TABLE** statement to index a column prefix looks like this:

```
KEY index_name (col_name(length))
```

The example below creates an index for the first 10 characters of the **name** column:

```
mysql> CREATE TABLE test (  
    name CHAR(200) NOT NULL,  
    KEY index_name (name(10)));
```

For **BLOB** and **TEXT** columns, you must index a prefix of the column. You cannot index the entire column.

In MySQL Version 3.23.23 or later, you can also create special **FULLTEXT** indexes. They are used for full-text search. Only the **MyISAM** table type supports **FULLTEXT** indexes. They can be created only from **VARCHAR** and **TEXT** columns. Indexing always happens over the entire column and partial indexing is not supported.

## 4.5 Multiple-Column Indexes

MySQL can create indexes on multiple columns. An index may consist of up to 15 columns. (On **CHAR** and **VARCHAR** columns you can also use a prefix of the column as a part of an index).

A multiple-column index can be considered a sorted array containing values that are created by concatenating the values of the indexed columns.

MySQL uses multiple-column indexes in such a way that queries are fast when you specify a known quantity for the first column of the index in a **WHERE** clause, even if you don't specify values for the other columns.

Suppose a table is created using the following specification:

```
mysql> CREATE TABLE test (  
    id INT NOT NULL,  
    last_name CHAR(30) NOT NULL,  
    first_name CHAR(30) NOT NULL,  
    PRIMARY KEY (id),  
    INDEX name (last_name,first_name));
```

Then the index **name** is an index over **last\_name** and **first\_name**. The index will be used for queries that specify values in a known range for **last\_name**, or for both **last\_name** and **first\_name**. Therefore, the **name** index will be used in the following queries:

```
mysql> SELECT * FROM test WHERE last_name="Widenius";
```

```
mysql> SELECT * FROM test WHERE last_name="Widenius"
      AND first_name="Michael";
```

```
mysql> SELECT * FROM test WHERE last_name="Widenius"
      AND (first_name="Michael" OR first_name="Monty");
```

```
mysql> SELECT * FROM test WHERE last_name="Widenius"
      AND first_name >="M" AND first_name < "N";
```

However, the `name` index will NOT be used in the following queries:

```
mysql> SELECT * FROM test WHERE first_name="Michael";
```

```
mysql> SELECT * FROM test WHERE last_name="Widenius"
      OR first_name="Michael";
```

For more information on the manner in which MySQL uses indexes to improve query performance.

## 4.6 How MySQL Opens and Closes Tables

`table_cache`, `max_connections`, and `max_tmp_tables` affect the maximum number of files the server keeps open. If you increase one or both of these values, you may run up against a limit imposed by your operating system on the per-process number of open file descriptors. However, you can increase the limit on many systems. Consult your OS documentation to find out how to do this, because the method for changing the limit varies widely from system to system.

`table_cache` is related to `max_connections`. For example, for 200 concurrent running connections, you should have a table cache of at least  $200 * n$ , where  $n$  is the maximum number of tables in a join. You also need to reserve some extra file descriptors for temporary tables and files.

The cache of open tables can grow to a maximum of `table_cache` (default 64; this can be changed with the `-O table_cache=#` option to `mysqld`). A table is never closed, except when the cache is full and another thread tries to open a table or if you use `mysqladmin refresh` or `mysqladmin flush-tables`.

When the table cache fills up, the server uses the following procedure to locate a cache entry to use:

- Tables that are not currently in use are released, in least-recently-used order.
- If the cache is full and no tables can be released, but a new table needs to be opened, the cache is temporarily extended as necessary.
- If the cache is in a temporarily-extended state and a table goes from in-use to not-in-use state, the table is closed and released from the cache.

A table is opened for each concurrent access. This means that if you have two threads accessing the same table or access the table twice in the same query (with `AS`) the table needs to be opened twice. The first open of any table takes two file descriptors; each additional use of the table takes only one file descriptor. The extra descriptor for the first open is used for the index file; this descriptor is shared among all threads.

You can check if your table cache is too small by checking the `mysqld` variable `opened_tables`. If this is quite big, even if you haven't done a lot of `FLUSH TABLES`, you should increase your table cache.

## 4.7 Drawbacks to Creating Large Numbers of Tables in the Same Database

If you have many files in a directory, open, close, and create operations will be slow. If you execute `SELECT` statements on many different tables, there will be a little overhead when the table cache is full, because for every table that has to be opened, another must be closed. You can reduce this overhead by making the table cache larger.

## 4.8 Why So Many Open tables?

When you run `mysqladmin status`, you'll see something like this:

Uptime: 426 Running threads: 1 Questions: 11082 Reloads: 1 Open tables: 12

This can be somewhat perplexing if you only have 6 tables.

MySQL is multithreaded, so it may have many queries on the same table simultaneously. To minimize the problem with two threads having different states on the same file, the table is opened independently by each concurrent thread. This takes some memory and one extra file descriptor for the data file. The index file descriptor is shared between all threads.

## 5 Optimizing the MySQL Server

- [System](#): System/Compile Time and Startup Parameter Tuning
- [Server parameters](#): Tuning Server Parameters
- [Compile and link options](#): How Compiling and Linking Affects the Speed of MySQL
- [Memory use](#): How MySQL Uses Memory
- [DNS](#): How MySQL uses DNS
- [SET OPTION](#): `SET` Syntax

### 5.1 System/Compile Time and Startup Parameter Tuning

We start with the system level things since some of these decisions have to be made very early. In other cases a fast look at this part may suffice because it not that important for the big gains. However, it is always nice to have a feeling about how much one could gain by changing things at this level.

The default OS to use is really important! To get the most use of multiple CPU machines one should use Solaris (because the threads works really nice) or Linux (because the 2.2 kernel has really good SMP support). Also on 32-bit machines Linux has a 2G file size limit by default. Hopefully this will be fixed soon when new filesystems are released (XFS/Reiserfs). If you have a desperate need for files bigger than 2G on Linux-intel 32 bit, you should get the LFS patch for the ext2 file system.

Because we have not run MySQL in production on that many platforms, we advice you to test your intended platform before choosing it, if possible.

Other tips:

- If you have enough RAM, you could remove all swap devices. Some operating systems will use a swap device in some contexts even if you have free memory.

- Use the `--skip-locking` MySQL option to avoid external locking. Note that this will not impact MySQL's functionality as long as you only run one server. Just remember to take down the server (or lock relevant parts) before you run `myisamchk`. On some system this switch is mandatory because the external locking does not work in any case. The `--skip-locking` option is on by default when compiling with MIT-pthreads, because `flock()` isn't fully supported by MIT-pthreads on all platforms. It's also on default for Linux as Linux file locking are not yet safe. The only case when you can't use `--skip-locking` is if you run multiple MySQL *servers* (not clients) on the same data, or run `myisamchk` on the table without first flushing and locking the `mysqld` server tables first. You can still use `LOCK TABLES/UNLOCK TABLES` even if you are using `--skip-locking`

## 5.2 Tuning Server Parameters

You can get the default buffer sizes used by the `mysqld` server with this command:

```
shell> mysqld --help
```

This command produces a list of all `mysqld` options and configurable variables. The output includes the default values and looks something like this:

Possible variables for option `--set-variable (-O)` are:

```
back_log          current value: 5
bdb_cache_size    current value: 1048540
binlog_cache_size current value: 32768
connect_timeout   current value: 5
delayed_insert_timeout current value: 300
delayed_insert_limit current value: 100
delayed_queue_size current value: 1000
flush_time        current value: 0
interactive_timeout current value: 28800
join_buffer_size  current value: 131072
key_buffer_size   current value: 1048540
lower_case_table_names current value: 0
long_query_time   current value: 10
max_allowed_packet current value: 1048576
max_binlog_cache_size current value: 4294967295
max_connections   current value: 100
max_connect_errors current value: 10
max_delayed_threads current value: 20
max_heap_table_size current value: 16777216
max_join_size     current value: 4294967295
max_sort_length   current value: 1024
max_tmp_tables    current value: 32
max_write_lock_count current value: 4294967295
myisam_sort_buffer_size current value: 8388608
net_buffer_length current value: 16384
net_retry_count   current value: 10
net_read_timeout  current value: 30
net_write_timeout current value: 60
query_buffer_size current value: 0
record_buffer     current value: 131072
record_rnd_buffer current value: 131072
slow_launch_time  current value: 2
sort_buffer       current value: 2097116
table_cache       current value: 64
```

```
thread_concurrency  current value: 10
tmp_table_size      current value: 1048576
thread_stack        current value: 131072
wait_timeout        current value: 28800
```

If there is a `mysqld` server currently running, you can see what values it actually is using for the variables by executing this command:

```
shell> mysqladmin variables
```

You can find a full description for all variables in the `SHOW VARIABLES` section in this manual.

You can also see some statistics from a running server by issuing the command `SHOW STATUS`.

MySQL uses algorithms that are very scalable, so you can usually run with very little memory. If you, however, give MySQL more memory, you will normally also get better performance.

When tuning a MySQL server, the two most important variables to use are `key_buffer_size` and `table_cache`. You should first feel confident that you have these right before trying to change any of the other variables.

If you have much memory ( $\geq 256\text{M}$ ) and many tables and want maximum performance with a moderate number of clients, you should use something like this:

```
shell> safe_mysqld -O key_buffer=64M -O table_cache=256 \
-O sort_buffer=4M -O record_buffer=1M &
```

If you have only 128M and only a few tables, but you still do a lot of sorting, you can use something like:

```
shell> safe_mysqld -O key_buffer=16M -O sort_buffer=1M
```

If you have little memory and lots of connections, use something like this:

```
shell> safe_mysqld -O key_buffer=512k -O sort_buffer=100k \
-O record_buffer=100k &
```

or even:

```
shell> safe_mysqld -O key_buffer=512k -O sort_buffer=16k \
-O table_cache=32 -O record_buffer=8k -O net_buffer=1K &
```

If you are doing a `GROUP BY` or `ORDER BY` on files that are much bigger than your available memory you should increase the value of `record_rnd_buffer` to speed up the reading of rows after the sorting is done.

When you have installed MySQL, the `support-files` directory will contain some different `my.cnf` example files, `my-huge.cnf`, `my-large.cnf`, `my-medium.cnf`, and `my-small.cnf`, you can use as a base to optimize your system.

If there are very many connections, "swapping problems" may occur unless `mysqld` has been configured to use very little memory for each connection. `mysqld` performs better if you have enough memory for all connections, of course.

Note that if you change an option to `mysqld`, it remains in effect only for that instance of the server.

To see the effects of a parameter change, do something like this:

```
shell> mysqld -O key_buffer=32m --help
```

Make sure that the `--help` option is last; otherwise, the effect of any options listed after it on the command line will not be reflected in the output.

### 5.3 How Compiling and Linking Affects the Speed of MySQL

Most of the following tests are done on Linux with the MySQL benchmarks, but they should give some indication for other operating systems and workloads.

You get the fastest executable when you link with `-static`.

On Linux, you will get the fastest code when compiling with `pgcc` and `-O3`. To compile `sql_yacc.cc` with these options, you need about 200M memory because `gcc/pgcc` needs a lot of memory to make all functions inline. You should also set `CXX=gcc` when configuring MySQL to avoid inclusion of the `libstdc++` library (it is not needed). Note that with some versions of `pgcc`, the resulting code will only run on true Pentium processors, even if you use the compiler option that you want the resulting code to be working on all x586 type processors (like AMD).

By just using a better compiler and/or better compiler options you can get a 10-30 % speed increase in your application. This is particularly important if you compile the SQL server yourself!

We have tested both the Cygnus CodeFusion and Fujitsu compilers, but when we tested them, neither was sufficiently bug free to allow MySQL to be compiled with optimizations on.

When you compile MySQL you should only include support for the character sets that you are going to use. (Option `--with-charset=xxx`). The standard MySQL binary distributions are compiled with support for all character sets.

Here is a list of some measurements that we have done:

- If you use `pgcc` and compile everything with `-O6`, the `mysqld` server is 1% faster than with `gcc 2.95.2`.
- If you link dynamically (without `-static`), the result is 13% slower on Linux. Note that you still can use a dynamic linked MySQL library. It is only the server that is critical for performance.
- If you strip your `mysqld` binary with `strip libexec/mysqld`, the resulting binary can be up to 4 % faster.
- If you connect using TCP/IP rather than Unix sockets, the result is 7.5% slower on the same computer. (If you are connection to `localhost`, MySQL will, by default, use sockets).
- If you connect using TCP/IP from another computer over a 100M Ethernet, things will be 8-11 % slower.
- If you compile with `--with-debug=full`, then you will loose 20 % for most queries, but some queries may take substantially longer (The MySQL benchmarks ran 35 % slower) If you use `--with-debug`, then you will only loose 15 %. By starting a `mysqld` version compiled with `--with-debug=full` with `--skip-safemalloc` the end result should be close to when configuring with `--with-debug`.
- On a Sun SPARCstation 20, SunPro C++ 4.2 is 5 % faster than `gcc 2.95.2`.
- Compiling with `gcc 2.95.2` for ultrasparc with the option `-mcpu=v8 -Wa,-xarch=v8plusa` gives 4 % more performance.

- On Solaris 2.5.1, MIT-pthreads is 8-12% slower than Solaris native threads on a single processor. With more load/CPU's the difference should get bigger.
- Running with `--log-bin` makes **[MySQL 1 % slower**.
- Compiling on Linux-x86 using gcc without frame pointers `-fomit-frame-pointer` or `-fomit-frame-pointer -ffixed-ebp` `mysqld` 1-4% faster.

The MySQL-Linux distribution provided by MySQL AB used to be compiled with `pgcc`, but we had to go back to regular gcc because of a bug in `pgcc` that would generate the code that does not run on AMD. We will continue using gcc until that bug is resolved. In the meantime, if you have a non-AMD machine, you can get a faster binary by compiling with `pgcc`. The standard MySQL Linux binary is linked statically to get it faster and more portable.

## 5.4 How MySQL Uses Memory

The list below indicates some of the ways that the `mysqld` server uses memory. Where applicable, the name of the server variable relevant to the memory use is given:

- The key buffer (variable `key_buffer_size`) is shared by all threads; Other buffers used by the server are allocated as needed.
- Each connection uses some thread-specific space: A stack (default 64K, variable `thread_stack`), a connection buffer (variable `net_buffer_length`), and a result buffer (variable `net_buffer_length`). The connection buffer and result buffer are dynamically enlarged up to `max_allowed_packet` when needed. When a query is running, a copy of the current query string is also allocated.
- All threads share the same base memory.
- Only the compressed ISAM / MyISAM tables are memory mapped. This is because the 32-bit memory space of 4GB is not large enough for most big tables. When systems with a 64-bit address space become more common we may add general support for memory mapping.
- Each request doing a sequential scan over a table allocates a read buffer (variable `record_buffer`).
- When reading rows in 'random' order (for example after a sort) a random-read buffer is allocated to avoid disk seeks. (variable `record_rnd_buffer`).
- All joins are done in one pass, and most joins can be done without even using a temporary table. Most temporary tables are memory-based (HEAP) tables. Temporary tables with a big record length (calculated as the sum of all column lengths) or that contain **BLOB** columns are stored on disk. One problem in MySQL versions before Version 3.23.2 is that if a HEAP table exceeds the size of `tmp_table_size`, you get the error **The table tbl\_name is full**. In newer versions this is handled by automatically changing the in-memory (HEAP) table to a disk-based (MyISAM) table as necessary. To work around this problem, you can increase the temporary table size by setting the `tmp_table_size` option to `mysqld`, or by setting the SQL option `SQL_BIG_TABLES` in the client program. In MySQL Version 3.20, the maximum size of the temporary table was `record_buffer*16`, so if you are using this version, you have to increase the value of `record_buffer`. You can also start `mysqld` with the `--big-tables` option to always store

- temporary tables on disk. However, this will affect the speed of many complicated queries.
- Most requests doing a sort allocates a sort buffer and 0-2 temporary files depending on the result set size.
  - Almost all parsing and calculating is done in a local memory store. No memory overhead is needed for small items and the normal slow memory allocation and freeing is avoided. Memory is allocated only for unexpectedly large strings (this is done with `malloc()` and `free()`).
  - Each index file is opened once and the data file is opened once for each concurrently running thread. For each concurrent thread, a table structure, column structures for each column, and a buffer of size  $3 * n$  is allocated (where  $n$  is the maximum row length, not counting `BLOB` columns). A `BLOB` uses 5 to 8 bytes plus the length of the `BLOB` data. The `ISAM/MyISAM` table handlers will use one extra row buffer for internal usage.
  - For each table having `BLOB` columns, a buffer is enlarged dynamically to read in larger `BLOB` values. If you scan a table, a buffer as large as the largest `BLOB` value is allocated.
  - Table handlers for all in-use tables are saved in a cache and managed as a FIFO. Normally the cache has 64 entries. If a table has been used by two running threads at the same time, the cache contains two entries for the table.
  - A `mysqladmin flush-tables` command closes all tables that are not in use and marks all in-use tables to be closed when the currently executing thread finishes. This will effectively free most in-use memory.

`ps` and other system status programs may report that `mysqld` uses a lot of memory. This may be caused by thread-stacks on different memory addresses. For example, the Solaris version of `ps` counts the unused memory between stacks as used memory. You can verify this by checking available swap with `swap -s`. We have tested `mysqld` with commercial memory-leakage detectors, so there should be no memory leaks.

## 5.5 How MySQL uses DNS

When a new thread connects to `mysqld`, `mysqld` will spawn a new thread to handle the request. This thread will first check if the hostname is in the hostname cache. If not the thread will call `gethostbyaddr_r()` and `gethostbyname_r()` to resolve the hostname.

If the operating system doesn't support the above thread-safe calls, the thread will lock a mutex and call `gethostbyaddr()` and `gethostbyname()` instead. Note that in this case no other thread can resolve other hostnames that is not in the hostname cache until the first thread is ready.

You can disable DNS host lookup by starting `mysqld` with `--skip-name-resolve`. In this case you can however only use IP names in the MySQL privilege tables.

If you have a very slow DNS and many hosts, you can get more performance by either disabling DNS lookup with `--skip-name-resolve` or by increasing the `HOST_CACHE_SIZE` define (default: 128) and recompile `mysqld`.

You can disable the hostname cache with `--skip-host-cache`. You can clear the hostname cache with `FLUSH HOSTS` or `mysqladmin flush-hosts`.

If you don't want to allow connections over **TCP/IP**, you can do this by starting **mysqld** with **--skip-networking**.

## 5.6 SET Syntax

**SET [OPTION] SQL\_VALUE\_OPTION= value, ...**

**SET OPTION** sets various options that affect the operation of the server or your client. Any option you set remains in effect until the current session ends, or until you set the option to a different value.

**CHARACTER SET character\_set\_name | DEFAULT**

This maps all strings from and to the client with the given mapping. Currently the only option for **character\_set\_name** is **cp1251\_koi8**, but you can easily add new mappings by editing the `sql/convert.cc` file in the MySQL source distribution. The default mapping can be restored by using a **character\_set\_name** value of **DEFAULT**. Note that the syntax for setting the **CHARACTER SET** option differs from the syntax for setting the other options.

**PASSWORD = PASSWORD('some password')**

Set the password for the current user. Any non-anonymous user can change his own password!

**PASSWORD FOR user = PASSWORD('some password')**

Set the password for a specific user on the current server host. Only a user with access to the **mysql** database can do this. The user should be given in **user@hostname** format, where **user** and **hostname** are exactly as they are listed in the **User** and **Host** columns of the **mysql.user** table entry. For example, if you had an entry with **User** and **Host** fields of **'bob'** and **'%.loc.gov'**, you would write:

```
mysql> SET PASSWORD FOR bob@"%.loc.gov" = PASSWORD("newpass");
```

or

```
mysql> UPDATE mysql.user SET password=PASSWORD("newpass") where user="bob'  
and host="%.loc.gov";
```

**SQL\_AUTO\_IS\_NULL = 0 | 1**

If set to **1** (default) then one can find the last inserted row for a table with an **auto\_increment** row with the following construct: **WHERE auto\_increment\_column IS NULL**. This is used by some ODBC programs like Access.

**AUTOCOMMIT= 0 | 1**

If set to **1** all changes to a table will be done at once. To start a multi-command transaction, you have to use the **BEGIN** statement. If set to **0** you have to use **COMMIT / ROLLBACK** to accept/revoke that transaction. Note that when you change from not **AUTOCOMMIT** mode to **AUTOCOMMIT** mode, MySQL will do an automatic **COMMIT** on any open transactions.

**SQL\_BIG\_TABLES = 0 | 1**

If set to **1**, all temporary tables are stored on disk rather than in memory. This will be a little slower, but you will not get the error **The table tbl\_name is full** for big **SELECT** operations that require a large temporary table. The default value for a new connection is **0** (that is, use in-memory temporary tables).

**SQL\_BIG\_SELECTS = 0 | 1**

If set to **0**, MySQL will abort if a **SELECT** is attempted that probably will take a very long time. This is useful when an inadvisable **WHERE** statement has been issued. A big query is defined as a **SELECT** that probably will have to examine more than **max\_join\_size** rows. The default value for a new connection is **1** (which will allow all **SELECT** statements).

**SQL\_BUFFER\_RESULT = 0 | 1**

- `SQL_BUFFER_RESULT` will force the result from `SELECT`'s to be put into a temporary table. This will help MySQL free the table locks early and will help in cases where it takes a long time to send the result set to the client.
- `SQL_LOW_PRIORITY_UPDATES = 0 | 1`  
If set to `1`, all `INSERT`, `UPDATE`, `DELETE`, and `LOCK TABLE WRITE` statements wait until there is no pending `SELECT` or `LOCK TABLE READ` on the affected table.
- `SQL_MAX_JOIN_SIZE = value | DEFAULT`  
Don't allow `SELECT`s that will probably need to examine more than `value` row combinations. By setting this value, you can catch `SELECT`s where keys are not used properly and that would probably take a long time. Setting this to a value other than `DEFAULT` will reset the `SQL_BIG_SELECTS` flag. If you set the `SQL_BIG_SELECTS` flag again, the `SQL_MAX_JOIN_SIZE` variable will be ignored. You can set a default value for this variable by starting `mysqld` with `-O max_join_size=#`.
- `SQL_SAFE_UPDATES = 0 | 1`  
If set to `1`, MySQL will abort if an `UPDATE` or `DELETE` is attempted that doesn't use a key or `LIMIT` in the `WHERE` clause. This makes it possible to catch wrong updates when creating SQL commands by hand.
- `SQL_SELECT_LIMIT = value | DEFAULT`  
The maximum number of records to return from `SELECT` statements. If a `SELECT` has a `LIMIT` clause, the `LIMIT` takes precedence over the value of `SQL_SELECT_LIMIT`. The default value for a new connection is ``unlimited.'' If you have changed the limit, the default value can be restored by using a `SQL_SELECT_LIMIT` value of `DEFAULT`.
- `SQL_LOG_OFF = 0 | 1`  
If set to `1`, no logging will be done to the standard log for this client, if the client has the `process` privilege. This does not affect the update log!
- `SQL_LOG_UPDATE = 0 | 1`  
If set to `0`, no logging will be done to the update log for the client, if the client has the `process` privilege. This does not affect the standard log!
- `SQL_QUOTE_SHOW_CREATE = 0 | 1`  
If set to `1`, `SHOW CREATE TABLE` will quote table and column names. This is `on` by default, for replication of tables with fancy column names to work. section [SHOW CREATE TABLE](#).
- `TIMESTAMP = timestamp_value | DEFAULT`  
Set the time for this client. This is used to get the original timestamp if you use the update log to restore rows. `timestamp_value` should be a UNIX Epoch timestamp, not a MySQL timestamp.
- `LAST_INSERT_ID = #`  
Set the value to be returned from `LAST_INSERT_ID()`. This is stored in the update log when you use `LAST_INSERT_ID()` in a command that updates a table.
- `INSERT_ID = #`  
Set the value to be used by the following `INSERT` or `ALTER TABLE` command when inserting an `AUTO_INCREMENT` value. This is mainly used with the update log.
- [SET TRANSACTION](#): `SET TRANSACTION` Syntax

## 6 Disk Issues

- As mentioned before, disks seeks are a big performance bottleneck. This problems gets more and more apparent when the data starts to grow so large that effective caching becomes impossible. For large databases, where you access data more or less randomly, you can be sure that you will need at least one disk seek to read and a couple of disk seeks to write things. To minimize this problem, use disks with low seek times.

- Increase the number of available disk spindles (and thereby reduce the seek overhead) by either symlink files to different disks or striping the disks.

#### Using symbolic links

This means that you symlink the index and/or data file(s) from the normal data directory to another disk (that may also be striped). This makes both the seek and read times better (if the disks are not used for other things).

#### Striping

Striping means that you have many disks and put the first block on the first disk, the second block on the second disk, and the Nth on the  $(N \bmod \text{number\_of\_disks})$  disk, and so on. This means if your normal data size is less than the stripe size (or perfectly aligned) you will get much better performance. Note that striping is very dependent on the OS and stripe-size. So benchmark your application with different stripe-sizes. Note that the speed difference for striping is **very** dependent on the parameters. Depending on how you set the striping parameters and number of disks you may get a difference in orders of magnitude. Note that you have to choose to optimize for random or sequential access.

- For reliability you may want to use RAID 0+1 (striping + mirroring), but in this case you will need  $2*N$  drives to hold  $N$  drives of data. This is probably the best option if you have the money for it! You may, however, also have to invest in some volume-management software to handle it efficiently.
- A good option is to have semi-important data (that can be regenerated) on RAID 0 disk while storing really important data (like host information and logs) on a RAID 0+1 or RAID N disk. RAID N can be a problem if you have many writes because of the time to update the parity bits.
- You may also set the parameters for the file system that the database uses. One easy change is to mount the file system with the noatime option. That makes it skip the updating of the last access time in the inode and by this will avoid some disk seeks.
- On Linux, you can get much more performance (up to 100 % under load is not uncommon) by using hdparm to configure your disk's interface! The following should be quite good hdparm options for MySQL (and probably many other applications):
  - `hdparm -m 16 -d 1`

Note that the performance/reliability when using the above depends on your hardware, so we strongly suggest that you test your system thoroughly after using `hdparm`! Please consult the `hdparm` man page for more information! If `hdparm` is not used wisely, filesystem corruption may result. Backup everything before experimenting!
  - On many operating systems you can mount the disks with the 'async' flag to set the file system to be updated asynchronously. If your computer is reasonable stable, this should give you more performance without sacrificing too much reliability. (This flag is on by default on Linux.)
  - If you don't need to know when a file was last accessed (which is not really useful on a database server), you can mount your file systems with the noatime flag.
  - [Symbolic links](#): Using Symbolic Links

## 6.1 Using Symbolic Links

You can move tables and databases from the database directory to other locations and replace them with symbolic links to the new locations. You might want to do this, for example, to move a database to a file system with more free space or increase the speed of your system by spreading your tables to different disk.

The recommended way to do this, is to just symlink databases to different disk and only symlink tables as a last resort.

- [Symbolic links to databases](#): Using Symbolic Links for Databases
- [Symbolic links to tables](#): Using Symbolic Links for Tables

### 6.1.1 Using Symbolic Links for Databases

The way to symlink a database is to first create a directory on some disk where you have free space and then create a symlink to it from the MySQL database directory.

```
shell> mkdir /dr1/databases/test
shell> ln -s /dr1/databases/test mysqld-datadir
```

MySQL doesn't support that you link one directory to multiple databases. Replacing a database directory with a symbolic link will work fine as long as you don't make a symbolic link between databases. Suppose you have a database `db1` under the MySQL data directory, and then make a symlink `db2` that points to `db1`:

```
shell> cd /path/to/datadir
shell> ln -s db1 db2
```

Now, for any table `tbl_a` in `db1`, there also appears to be a table `tbl_a` in `db2`. If one thread updates `db1.tbl_a` and another thread updates `db2.tbl_a`, there will be problems.

If you really need this, you must change the following code in ``mysys/mf_format.c``:

```
if (flag & 32 || (!lstat(to,&stat_buff) && S_ISLNK(stat_buff.st_mode)))
```

to

```
if (1)
```

On Windows you can use internal symbolic links to directories by compiling MySQL with `-DUSE_SYMDIR`. This allows you to put different databases on different disks.

### 6.1.2 Using Symbolic Links for Tables

Before MySQL 4.0 you should not symlink tables, if you are not very carefully with them. The problem is that if you run `ALTER TABLE`, `REPAIR TABLE` or `OPTIMIZE TABLE` on a symlinked table, the symlinks will be removed and replaced by the original files. This happens because the above command works by creating a temporary file in the database directory and when the command is complete, replace the original file with the temporary file.

You should not symlink tables on system that doesn't have a fully working `realpath()` call. (At least Linux and Solaris support `realpath()`)

In MySQL 4.0 symlinks is only fully supported for **MyISAM** tables. For other table types you will probably get strange problems when doing any of the above mentioned commands.

The handling of symbolic links in MySQL 4.0 works the following way (this is mostly relevant only for **MyISAM** tables).

- In the data directory you will always have the table definition file and the data/index files.
- You can symlink the index file and the data file to different directories independent of the other.
- The symlinking can be done from the operating system (if **mysqld** is not running) or with the **INDEX/DATA DIRECTORY="path-to-dir"** command in **CREATE TABLE**.
- **myisamchk** will not replace a symlink with the index/file but work directly on the files the symlinks points to. Any temporary files will be created in the same directory where the data/index file is.
- When you drop a table that is using symlinks, both the symlink and the file the symlink points to is dropped. This is a good reason to why you should NOT run **mysqld** as root and not allow persons to have write access to the MySQL database directories.
- If you rename a table with **ALTER TABLE RENAME** and you don't change database, the symlink in the database directory will be renamed to the new name and the data/index file will be renamed accordingly.
- If you use **ALTER TABLE RENAME** to move a table to another database, then the table will be moved to the other database directory and the old symlinks and the files they pointed to will be deleted.
- If you are not using symlinks you should use the **--skip-symlink** option to **mysqld** to ensure that no one can drop or rename a file outside of the **mysqld** data directory.

Things that are not yet supported:

- **ALTER TABLE** ignores all **INDEX/DATA DIRECTORY="path"** options.
- **CREATE TABLE** doesn't report if the table has symbolic links.
- **mysqldump** doesn't include the symbolic links information in the output.
- **BACKUP TABLE** and **RESTORE TABLE** don't respect symbolic links.

## [7 How To Optimize Our Application](#)

To get our application *really* database-independent, we need to define an easy extendable interface through which we manipulate our data. If high performance is more important than exactness, as in some web applications, it is possible to create an application layer that caches all results to give you even higher performance. By letting old results 'expire' after a while, we can keep the cache reasonably fresh. This provides a method to handle high load spikes, in which case we can dynamically increase the cache and set the expire timeout higher until things get back to normal. In this case the table creation information should contain information of the initial size of the cache and how often the table should normally be refreshed. Indexes are one of the

keys to speed in large databases. No matter how simple your table, a 500,000-row table scan will never be fast. If you have a site with a 500,000-row table, you should really spend time analyzing possible indexes and possibly consider rewriting queries to optimize your application.

### Other Optimization Steps

This section lists a number of miscellaneous tips for improving query processing speed:

- Use persistent connections to the database to avoid connection overhead. If you can't use persistent connections and you are initiating many new connections to the database, you may want to change the value of the `thread_cache_size` variable..
- Always check whether all your queries really use the indexes you have created in the tables. In MySQL, you can do this with the `EXPLAIN` statement..
- Try to avoid complex `SELECT` queries on `MyISAM` tables that are updated frequently, to avoid problems with table locking that occur due to contention between readers and writers.
- Columns with identical information in different tables should be declared to have identical data types. Before MySQL 3.23, you get slow joins otherwise. Try to keep column names simple. For example, in a table named `customer`, use a column name of `name` instead of `customer_name`. To make your names portable to other SQL servers, you should keep them shorter than 18 characters.
- With `MyISAM` tables that have no deleted rows, you can insert rows at the end at the same time that another query is reading from the table. If this is important for you, you should consider using the table in ways that avoid deleting rows. Another possibility is to run `OPTIMIZE TABLE` after you have deleted a lot of rows.
- If you very often need to calculate results such as counts based on information from a lot of rows, it's probably much better to introduce a new table and update the counter in real time. An update of the following form is very fast:
- `UPDATE tbl_name SET count_col=count_col+1 WHERE key_col=constant;`
- This is really important when you use MySQL storage engines such as `MyISAM` and `ISAM` that have only table-level locking (multiple readers / single writers). This will also give better performance with most databases, because the row locking manager in this case will have less to do. If you need to collect statistics from large log tables, use summary tables instead of scanning the entire log table. Maintaining the summaries should be much faster than trying to calculate statistics ``live." It's much faster to regenerate new summary tables from the logs when things change (depending on business decisions) than to have to change the running application!
- When using a normal Web server setup, images should be stored as files. That is, store only a file reference in the database. The main reason for this is that a normal Web server is much better at caching files than database contents, so it's much easier to get a fast system if you are using files.
- Use in-memory tables for non-critical data that is accessed often, such as information about the last displayed banner for users who don't have cookies enabled in their Web browser.
- If you need really high speed, you should take a look at the low-level interfaces for data storage that the different SQL servers support! For example, by accessing the MySQL

- MyISAM** storage engine directly, you could get a speed increase of two to five times compared to using the SQL interface. To be able to do this, the data must be on the same server as the application, and usually it should only be accessed by one process (because external file locking is really slow). One could eliminate these problems by introducing low-level **MyISAM** commands in the MySQL server (this could be one easy way to get more performance if needed). By carefully designing the database interface, it should be quite easy to support this types of optimization.
- If you are using numerical data, it's faster in many cases to access information from a database (using a live connection) than to access a text file. Information in the database is likely to be stored in a more compact format than in the text file, so accessing it will involve fewer disk accesses. You will also save code in your application because you don't have to parse your text files to find line and column boundaries.
  - Replication can provide a performance benefit for some operations. You can distribute client retrievals among replication servers to split up the load. To avoid slowing down the master while making backups, you can make backups using a slave server.
  - Declaring a **MyISAM** table with the **DELAY\_KEY\_WRITE=1** table option makes index updates faster because they are not flushed to disk until the table is closed. The downside is that if something kills the server while such a table is open, you should ensure that they are okay by running the server with the **--myisam-recover** option, or by running **myisamchk** before restarting the server. (However, even in this case, you should not lose anything by using **DELAY\_KEY\_WRITE**, because the key information can always be generated from the data rows.)