# 2@Table\_locks\_immediate

- zhengdl126 - ITeye技术网站 http://zhengdl126.iteye.com/blog/2108618

Table\_locks\_immediate表示立即释放表锁数，Table\_locks\_waited表示需要等待的表锁数，  
如果Table\_locks\_immediate / Table\_locks\_waited > 5000，最好采用InnoDB引擎，  
因为InnoDB是行锁而MyISAM是表锁，对于高并发写入的应用InnoDB效果会好些。

5000次加锁请求中有1次需要等待

如果Table\_locks\_waited的值比较高，则说明存在着较严重的表级锁争用情况。

用状态变量 Table\_locks\_waited 和 Table\_locks\_immediate 来分析系统中的锁表争夺情况

# MySQL :: MySQL 5.7 Reference Manual :: 5.1.7 Server Status Variables

https://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html#statvar\_Table\_locks\_immediate

# 2@MySQL :: MySQL 5.7 Reference Manual :: 8.11.1 Internal Locking Methods

https://dev.mysql.com/doc/refman/5.7/en/internal-locking.html

This section discusses internal locking; that is, locking performed within the MySQL server itself to manage contention for table contents by multiple sessions. This type of locking is internal because it is performed entirely by the server and involves no other programs. For locking performed on MySQL files by other programs, see [Section 8.11.5, “External Locking”](https://dev.mysql.com/doc/refman/5.7/en/external-locking.html).

* [Row-Level Locking](https://dev.mysql.com/doc/refman/5.7/en/internal-locking.html#internal-row-level-locking)
* [Table-Level Locking](https://dev.mysql.com/doc/refman/5.7/en/internal-locking.html#internal-table-level-locking)
* [Choosing the Type of Locking](https://dev.mysql.com/doc/refman/5.7/en/internal-locking.html#internal-locking-choices)

Row-Level Locking

MySQL uses [row-level locking](https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_row_lock) for InnoDB tables to support simultaneous write access by multiple sessions, making them suitable for multi-user, highly concurrent, and OLTP applications.

To avoid [deadlocks](https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_deadlock) when performing multiple concurrent write operations on a single InnoDB table, acquire necessary locks at the start of the transaction by issuing a SELECT ... FOR UPDATE statement for each group of rows expected to be modified, even if the data change statements come later in the transaction. If transactions modify or lock more than one table, issue the applicable statements in the same order within each transaction. Deadlocks affect performance rather than representing a serious error, because InnoDB automatically [detects](https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_deadlock_detection) deadlock conditions and rolls back one of the affected transactions.

On high concurrency systems, deadlock detection can cause a slowdown when numerous threads wait for the same lock. At times, it may be more efficient to disable deadlock detection and rely on the [innodb\_lock\_wait\_timeout](https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_lock_wait_timeout) setting for transaction rollback when a deadlock occurs. Deadlock detection can be disabled using the [innodb\_deadlock\_detect](https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_deadlock_detect) configuration option.

Advantages of row-level locking:

* Fewer lock conflicts when different sessions access different rows.
* Fewer changes for rollbacks.
* Possible to lock a single row for a long time.

Table-Level Locking

MySQL uses [table-level locking](https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_table_lock) for MyISAM, MEMORY, and MERGE tables, permitting only one session to update those tables at a time. This locking level makes these storage engines more suitable for read-only, read-mostly, or single-user applications.

These storage engines avoid [deadlocks](https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_deadlock) by always requesting all needed locks at once at the beginning of a query and always locking the tables in the same order. The tradeoff is that this strategy reduces concurrency; other sessions that want to modify the table must wait until the current data change statement finishes.

Advantages of table-level locking:

* Relatively little memory required (row locking requires memory per row or group of rows locked)
* Fast when used on a large part of the table because only a single lock is involved.
* Fast if you often do GROUP BY operations on a large part of the data or must scan the entire table frequently.

MySQL grants table write locks as follows:

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

MySQL grants table read locks as follows:

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

Table updates are given higher priority than table retrievals. Therefore, when a lock is released, the lock is made available to the requests in the write lock queue and then to the requests in the read lock queue. This ensures that updates to a table are not“starved” even when there is heavy [SELECT](https://dev.mysql.com/doc/refman/5.7/en/select.html) activity for the table. However, if there are many updates for a table, [SELECT](https://dev.mysql.com/doc/refman/5.7/en/select.html) statements wait until there are no more updates.

For information on altering the priority of reads and writes, see [Section 8.11.2, “Table Locking Issues”](https://dev.mysql.com/doc/refman/5.7/en/table-locking.html).

You can analyze the table lock contention on your system by checking the [Table\_locks\_immediate](https://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html#statvar_Table_locks_immediate) and [Table\_locks\_waited](https://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html#statvar_Table_locks_waited) status variables, which indicate the number of times that requests for table locks could be granted immediately and the number that had to wait, respectively:

mysql> **SHOW STATUS LIKE 'Table%';**

+-----------------------+---------+

| Variable\_name | Value |

+-----------------------+---------+

| Table\_locks\_immediate | 1151552 |

| Table\_locks\_waited | 15324 |

+-----------------------+---------+

The Performance Schema lock tables also provide locking information. See [Section 25.11.12, “Performance Schema Lock Tables”](https://dev.mysql.com/doc/refman/5.7/en/performance-schema-lock-tables.html).

The MyISAM storage engine supports concurrent inserts to reduce contention between readers and writers for a given table: If aMyISAM table has no free blocks in the middle of the data file, rows are always inserted at the end of the data file. In this case, you can freely mix concurrent [INSERT](https://dev.mysql.com/doc/refman/5.7/en/insert.html) and [SELECT](https://dev.mysql.com/doc/refman/5.7/en/select.html) statements for a MyISAM table without locks. That is, you can insert rows into a MyISAMtable at the same time other clients are reading from it. Holes can result from rows having been deleted from or updated in the middle of the table. If there are holes, concurrent inserts are disabled but are enabled again automatically when all holes have been filled with new data. To control this behavior, use the [concurrent\_insert](https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html#sysvar_concurrent_insert) system variable. See [Section 8.11.3, “Concurrent Inserts”](https://dev.mysql.com/doc/refman/5.7/en/concurrent-inserts.html).

If you acquire a table lock explicitly with [LOCK TABLES](https://dev.mysql.com/doc/refman/5.7/en/lock-tables.html), you can request a READ LOCAL lock rather than a READ lock to enable other sessions to perform concurrent inserts while you have the table locked.

To perform many [INSERT](https://dev.mysql.com/doc/refman/5.7/en/insert.html) and [SELECT](https://dev.mysql.com/doc/refman/5.7/en/select.html) operations on a table t1 when concurrent inserts are not possible, you can insert rows into a temporary table temp\_t1 and update the real table with the rows from the temporary table:

mysql> **LOCK TABLES t1 WRITE, temp\_t1 WRITE;**

mysql> **INSERT INTO t1 SELECT \* FROM temp\_t1;**

mysql> **DELETE FROM temp\_t1;**

mysql> **UNLOCK TABLES;**

Choosing the Type of Locking

Generally, table locks are superior to row-level locks in the following cases:

* Most statements for the table are reads.
* Statements for the table are a mix of reads and writes, where writes are updates or deletes for a single row that can be fetched with one key read:
* UPDATE ***tbl\_name*** SET ***column***=***value*** WHERE ***unique\_key\_col***=***key\_value***;
* DELETE FROM ***tbl\_name*** WHERE ***unique\_key\_col***=***key\_value***;
* [SELECT](https://dev.mysql.com/doc/refman/5.7/en/select.html) combined with concurrent [INSERT](https://dev.mysql.com/doc/refman/5.7/en/insert.html) statements, and very few [UPDATE](https://dev.mysql.com/doc/refman/5.7/en/update.html) or [DELETE](https://dev.mysql.com/doc/refman/5.7/en/delete.html) statements.
* Many scans or GROUP BY operations on the entire table without any writers.

With higher-level locks, you can more easily tune applications by supporting locks of different types, because the lock overhead is less than for row-level locks.

Options other than row-level locking:

* Versioning (such as that used in MySQL for concurrent inserts) where it is possible to have one writer at the same time as many readers. This means that the database or table supports different views for the data depending on when access begins. Other common terms for this are “time travel,” “copy on write,” or “copy on demand.”
* Copy on demand is in many cases superior to row-level locking. However, in the worst case, it can use much more memory than using normal locks.
* Instead of using row-level locks, you can employ application-level locks, such as those provided by [GET\_LOCK()](https://dev.mysql.com/doc/refman/5.7/en/miscellaneous-functions.html#function_get-lock) and[RELEASE\_LOCK()](https://dev.mysql.com/doc/refman/5.7/en/miscellaneous-functions.html#function_release-lock) in MySQL. These are advisory locks, so they work only with applications that cooperate with each other. See[Section 12.20, “Miscellaneous Functions”](https://dev.mysql.com/doc/refman/5.7/en/miscellaneous-functions.html).

# InnoDB immediate table level locks | MySQL Tech

https://mysqltech.wordpress.com/2011/08/24/innodb-immediate-table-level-locks/

One of the best known features of InnoDB is row level locking. Each update and/or delete will only lock the records that are being modified so there should be very little in the way of lock contention for most applications. It is however, in some cases possible to get a table level lock in InnoDB (auto-inc for example) but it is very rare. This make it all the more interesting to monitor the global status variable that show how many table level locks were granted immediately ([*table\_locks\_immediate*](http://dev.mysql.com/doc/refman/5.1/en/server-status-variables.html#statvar_Table_locks_immediate)) and surprisingly enough, this counter goes up rather quickly during normal usage. In fact, even a very simple select will cause this counter to increment as shown below.

mysql> show global status like 'table\_locks\_immediate';

+-----------------------+-------+

| Variable\_name         | Value |

+-----------------------+-------+

| Table\_locks\_immediate | 250   |

+-----------------------+-------+

1 row in set (0.00 sec)

mysql> select count(\*) from some\_table;

+----------+

| count(\*) |

+----------+

|        2 |

+----------+

1 row in set (0.00 sec)

mysql> show global status like 'table\_locks\_immediate';

+-----------------------+-------+

| Variable\_name         | Value |

+-----------------------+-------+

| Table\_locks\_immediate | 251   |

+-----------------------+-------+

1 row in set (0.00 sec)

What’s happening here is that the separation between the database system and the storage engine is showing. The table\_locks\_immediate counter registers MySQL’s requests for table level locks and since MySQL is unaware of the locking capabilities of the storage engine of the table  it requests a table level lock for every select that it performs. In the case of InnoDB the storage engine then downgrades this to a row level lock but it does not seem to report this back to MySQL which still counts it as a table level lock.

In summary, don’t trust the table\_locks\_immediate counter to determine real table level locks. This counter seems to be most useful to compare with table\_locks\_waited to see if there is lock contention.

# How To Reduce table\_locks\_waited In MySQL/MyISAM

http://www.mysqlperformancetuning.com/how-to-reduce-table\_locks\_waited-in-mysql-myisam

By [Stephen Jayna](mailto:stephen@everita.com), 19th August 2009

The scourge of parallelism and scaling everywhere: locking. Or in MySQL/MyISAM — and to be more precise — table locks. Here's an overview of what to look out for and how one might go about reducing the frequency at which they occur.

Before you embark on this please read [How To Speed Up MySQL: An Introduction To Optimizing](http://www.everita.com/how-to-speed-up-mysql-by-optimization). It suggests some things you should consider first before getting down to the nitty gritty of reducing table\_locks\_waited. After all it's usually too many queries that are running too slowly that cause locked tables. Fix those first!

Know Your Enemy: show status like '%\\_locks\\_%';

mysql> show status like '%\\_locks\\_%';

+-----------------------+-------+

| Variable\_name | Value |

+-----------------------+-------+

| Table\_locks\_immediate | 53148 |

| Table\_locks\_waited | 17716 |

+-----------------------+-------+

2 rows in set (0.00 sec)

That's an upsetting ratio. For every query that had to twiddle its thumbs, waiting in anticipation for a shot at said table, only 3 flew through immediately. You could throw more hardware at it I suppose, it might help. The chances are however you can increase your throughput using one or all of the following methods without reaching for another CPU, more memory or even faster disks.

**Why Table Locking Occurs in MyISAM**

Locks exist, in a nutshell, to prevent queries from altering data while that data is being read by another process. Or vice-versa.

There are different types of locking in the database world. MyISAM happens to use table locks which are very fast. They are easier to implement when compared to the row-level locking employed by InnoDB and permit a higher query throughput. That assumes of course the number of writes that occur on your database is few. Or, and this is sometimes overlooked, that no one query takes more than an instant.

After all it's the being locked out that hurts — writes tend to go through pretty quickly — but if they have to wait for a query to complete...

Imagine The Scenario

1. 0.00 seconds A select query (ie. read-only) accesses the table, it will take around 2 seconds to complete.
2. 0.01 seconds Another select query accesses the table, it takes no time and will complete in an instant as it can run in parallel.
3. 0.02 seconds An insert, delete or update query (ie. write) attempts to write to that very same table before the first select has completed.
4. 0.03 seconds A select query comes in, again waiting for the first select to complete.
5. 0.04 seconds Another select query comes in, again waiting for the first select to complete.
6. 0.05 seconds Yet another select query comes in, again waiting for the first select to complete.
7. .
8. 1.99 seconds (~1000 queries later) Yet another select query comes in, again waiting for the first select to complete.

And so on and so forth. The queries in orange and red are blocked and have to wait for the first select to complete before they can be executed. **The selects in red are blocked by the write operation in orange which is in turn blocked by the long-running select in green.** This is what causes the table\_locks\_waited value to grow.

See what happens if you've a query that takes any more than an instant and another query comes along, needing to write to the table? A whopping great long queue that's what. And therefore a collapse in parallelism. You're not going to scale like this.

**How To Avoid Or Reduce Table Locking**

There's a comprehensive list of things you might consider here on [MySQL's](http://dev.mysql.com/doc/refman/5.0/en/table-locking.html) site. While they're all worthy of consideration I can't help feeling some of them will only serve to delay the inevitable.

Divide And Conquer: Your Queries

As described in [How To Speed Up MySQL: An Introduction To Optimizing](http://www.everita.com/how-to-speed-up-mysql-by-optimization) consider splitting up your queries. Ten queries that take an instant is preferable to one when locking is an issue. Make your application do the work: it'll scale more easily than MySQL.

Row-level Locking: InnoDB

InnoDB uses row-level locking. That is to say during normal operation it won't lock entire tables when only updating a single row. The locking mechanism itself is slower than MyISAM but if your table is heavily contended it might be that the table in question should be moved to InnoDB.

That has its downsides of course. Your indexes will grow in physical size, chances are you may need more memory to cope.

Divide And Conquer: Your Data

Divide your data. Shard. Partition. Ensure no one table is too big. What's too big? It depends. MySQL barely imposes any artificial limits on the size a table can grow to but you're going to have to eventually make a call to split the table into shards or chunks.

That's easier said than done of course. Most people shy away from sharding in MySQL because it's difficult. MySQL are beginning to [include partitioning](http://dev.mysql.com/doc/refman/5.1/en/partitioning-overview.html)themselves. It's the sort of thing that [Hadoop](http://hadoop.apache.org/hbase/) does very well though.

Alternatives To MySQL

There are a lot of non-traditional DBMS out there that can be superior to MySQL in some situations. Don't believe it if anyone suggests your data must reside in either MySQL, PostgreSQL, Oracle or perhaps Microsoft's SQL Server.

I'll write about [Hadoop](http://hadoop.apache.org/hbase/), [MonetDB](http://en.wikipedia.org/wiki/MonetDB) and friends sooner rather than later but I'm afraid they're more data warehouse than online transaction processing. Low latency they are not. [Not necessarily](http://www.monetdb.nl/projects/monetdb/SQL/Benchmark/TPCH/).

**Conclusion**

I hope we've given you some idea of how to start dealing with lock contended tables or at least food for thought as it were. Reducing table\_locks\_waited can tricky to get right and depends very much on your data and the nature of your queries. Make sure you do review [MySQL's page on the matter](http://dev.mysql.com/doc/refman/5.0/en/table-locking.html). A tweak to your configuration may be enough for your application in the short term.

.  
.  
ps. If you followed the link to the MonetDB benchmarks take them with a pinch of salt, I included it purely for fun. I wouldn't dare vouch for their validity!

# How To Speed Up MySQL: An Introduction To Optimizing

http://www.mysqlperformancetuning.com/how-to-speed-up-mysql-by-optimization

By [Stephen Jayna](mailto:stephen@everita.com), 3rd August 2009

Although there is nothing groundbreaking in this document consider it a bringing together of techniques for your first foray into optimization. We won't discuss the more esoteric methods of squeezing the very last millisecond out of MySQL. There are a myriad of parameters to tune: here's what you need to get right first.

That hackneyed phrase 'one size does [not] fit all' is worth bearing in mind too. Clearly the costs and benefits to your specific application must be carefully considered. Much of the advice within should be considered the lowest common denominator for any time sensitive MySQL based application whether it's web based or not.

**Low Hanging Fruit: Query Optimization**

First turn on your slow query log and make sure it's logging queries that don't use indexes.

In your my.cnf you need the following lines:

log\_slow\_queries = /var/log/mysql/mysql-slow.log

long\_query\_time = 1

log-queries-not-using-indexes

Unfortunately the stock MySQL distribution doesn't support logging queries that take less than a second. However [Percona](http://www.percona.com/percona-lab.html) have released a set of patches that allow you to do this. Certainly one second is fine to begin with; we're after the lowest hanging fruit after all.

If you're a web application and any given query is taking over a second to run you should do better, and chances are you can. If you're caching the query with something like [memcached](http://www.danga.com/memcached) perhaps it doesn't matter so much. Let's assume it does, after all some poor user is going to be stuck waiting for that query to complete, even if it's only once in a while.

cat /var/log/mysql/mysql-slow.log | more

# Time: 090804 12:45:44

# User@Host: stephen[stephen] @ localhost []

# **Query\_time: 11** Lock\_time: 0 Rows\_sent: 1 Rows\_examined: 91687

use stephen\_drupal;

SELECT uid,fid

FROM profile\_values

WHERE uid NOT IN

(SELECT uid FROM profile\_values WHERE fid = 13 AND value = 'no')

AND value REGEXP 'ert' AND uid != 4145

AND uid NOT IN

(SELECT uid FROM users WHERE status = 0)

ORDER BY RAND()

LIMIT 1;

This query was taken from a client's slow query log.

Explain Yourself

Firstly does your query make use of indexes? Here's how to find out how MySQL has decided to parse and execute your query:

**EXPLAIN EXTENDED**

SELECT uid,fid

FROM profile\_values

WHERE uid NOT IN

(SELECT uid FROM profile\_values WHERE fid = 13 AND value = 'no')

AND value REGEXP 'ert' AND uid != 4145

AND uid NOT IN

(SELECT uid FROM users WHERE status = 0)

ORDER BY RAND()

LIMIT 1

**\G;**

Use the parameter 'extended' to get any additional information MySQL has on the query. The '\G' will format the output into a more readable form.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: PRIMARY

table: profile\_values

type: range

possible\_keys: uid

key: uid

key\_len: 5

ref: NULL

rows: 91137

Extra: Using where; Using temporary; Using filesort

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 3

select\_type: DEPENDENT SUBQUERY

table: users

type: unique\_subquery

possible\_keys: PRIMARY

key: PRIMARY

key\_len: 4

ref: func

rows: 1

Extra: Using where; Full scan on NULL key

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 2

select\_type: DEPENDENT SUBQUERY

table: profile\_values

type: index\_subquery

possible\_keys: uid,fid

key: uid

key\_len: 5

ref: func

rows: 6

Extra: Using where; Full scan on NULL key

3 rows in set, 1 warning (0.00 sec)

**Indexes vs. Table Scanning**

Something to look out for is a high number in the rows column. What should be considered high depends on the size of your database and to some extent the nature of the query. If you're not using an index for whatever reason MySQL is scanning through each row and stress is being placed on your disks and I/O subsystem. It's true that sometimes it's more efficient for MySQL to scan a table rather than use an index, especially if the table is small, however you should be wary.

Memory

If your database fits neatly into memory or resides on a solid-state drive then the difference between using an index and a table scan may be less noticeable but it should still be fixed. Having plentiful amounts of memory is always a good thing but it can serve to mask serious underlying problems.

**Dependent Subqueries**

Performing a dependent subquery within an [NOT] IN clause is prone to trouble with MySQL. Generally speaking it's better if you steer clear of them, or at least be very wary of using them.

The query above can be re-written without using any subqueries making vast time savings. Don't be afraid to split your queries into two or more parts either. Especially if you're using MyISAM and are worrying about lock contention you'll find that two or three very fast queries are preferable to one long one.

**Splitting Queries: Transactional Safety And ACID Compliance**

Splitting your queries into parts has data consistency ramifications. On MyISAM this may have consequences you need to consider since transactions aren't available. With InnoDB, transactions, and row-level locking, this is less of an issue. More on this later however.

**Making Your Application Do The Work**

My philosophy has always been to ask the database to do as little as is reasonably possible and move processing into the application. The less work you give it the less resource it needs.

The fact of the matter is that it's almost always harder to scale your database server/system than to pop in another web server. That's if you're web serving of course, but I imagine it might apply to any application. Web servers are relatively independent: they don't rely on other web servers for their function in the way a database running across multiple servers does.

Neither do they get out of sync and have to catch up before they become available again. Or need battery-backed caches to remain consistent. Or highly skilled and expensive DBAs. They're a walk in the park compared to any traditional RDBMS frankly. And they can make much better use of all those CPUs than InnoDB or a lock-contended MyISAM can.

Okay, so it's true, this is a generalisation. It of course depends on your application. Perl, PHP and Python are just as competent at doing somersaults with your data as MySQL is, perhaps more so. Besides which you put yourself in control of potential bottlenecks and that's always worth something. You'd rather tinker with your code base than MySQL's surely?

**That Query: Optimized**

The initial query has been broken down into three. The first two queries are run and their results are fed into the third. Importantly we've changed REGEXP to a LIKE and in the second stage you'll see we remove the ORDER BY RAND() LIMIT 1 construct.

The SQL\_NO\_CACHE ensures the queries are run without the aid of the cache that will confuse our profiling. We acknowledge that the OS disk cache has been warmed up however.

That original query:

SELECT SQL\_NO\_CACHE uid,fid

FROM profile\_values

WHERE uid NOT IN

(SELECT uid FROM profile\_values WHERE fid = 13 AND value = 'no')

AND value REGEXP 'ert' AND uid != 4145

AND uid NOT IN

(SELECT uid FROM users WHERE status = 0)

ORDER BY RAND()

LIMIT 1;

**1 row in set (6.01 sec)**

The optimized query (Stage 1):

x =

SELECT SQL\_NO\_CACHE uid

FROM profile\_values

WHERE fid = 13 AND value = 'no';

**444 rows in set (0.00 sec)**

y =

SELECT SQL\_NO\_CACHE uid

FROM users WHERE status = 0;

**464 rows in set (0.00 sec)**

SELECT SQL\_NO\_CACHE uid,fid

FROM profile\_values

WHERE uid NOT IN (4145,{x},{y})

AND value LIKE '%ert%'

ORDER BY RAND()

LIMIT 1;

**1 row in set (0.52 sec)**

We can do better however. The construct "ORDER BY RAND() LIMIT 1" is less than optimal. There are a number of ways around this but I'm happy to break it into two queries. They will run in O(n) time rather than O(1) — that is to say the time taken is dependent on the size of the table — but in this case I happen to know the table size is always going to be very modest.

The optimized query (Stage 2):

rows\_in\_set =

SELECT SQL\_NO\_CACHE COUNT(uid)

FROM profile\_values

WHERE uid NOT IN (4145,{x},{y})

AND value LIKE '%ert%'

**1 row in set (0.13 sec)**

random\_number = a random number between 1 and rows\_in\_set

SELECT SQL\_NO\_CACHE uid,fid

FROM profile\_values

WHERE uid NOT IN (4145,{x},{y})

AND value LIKE '%ert%'

LIMIT {random\_number}, 1

**1 row in set (0.13 sec)**

The query went from 6.01 seconds to 0.52 seconds to finally a set of queries that took ~0.26 seconds. Admittedly there is some extra overhead with the additional bandwidth generated and processing the application must do to produce the third query from the first two but, in this case at least, it's negligible.

It might seem counterintuitive but decreasing the number of queries you send MySQL doesn't necessarily equate to a faster application. Once you've bought into the idea it's easier to scale web servers — and not MySQL — you might consider letting your application handle more of the work.

Even if we hadn't managed to decrease the overall computation time and had simply moved some of it from MySQL to your application it would still be a win. Your web server is much better at concurrency than MySQL is.

**Lock Contention: To MyISAM or InnoDB**

Most MySQL databases out there run on MyISAM. I don't have the figures and I wouldn't know how to go about acquiring them but since it's the default MySQL table type I'd wager that it is the case.

MyISAM is fine at many things and for many situations. You don't have to worry about queries that COUNT(\*) for example. It's a very useful query to perform but on InnoDB it isn't fast: you'll need to find an alternative. MyISAM is more efficient in terms of disk space, in part due to it making physically smaller indexes which gives you all the more chance of having them fit into memory. On databases where inserts and updates rarely feature it's faster. But you better make sure that's the case.

mysql> show status like 'Table\_locks\_waited';

+--------------------+-------+

| Variable\_name | Value |

+--------------------+-------+

| Table\_locks\_waited | 23412 |

+--------------------+-------+

A large or growing Table\_locks\_waited value is the sign of a serious concurrency bottleneck.

Any long running select will block an insert or update from happening. That will in turn prevent any further selects from occurring until after the queued insert or update is complete. Your computer — with all its wonderful parallelism — will suddenly begin working in serial, at least with respect to MySQL. It doesn't matter how many CPUs you throw at it, only one will be used. A performance nightmare. MyISAM is for the most part fantastic and probably powers much of the web but it — like everything else — has costs and benefits that must be weighed.

**Replication and Sharding**

If you're not suffering from lock contention, your queries are using the appropriate indexes (which you've made if they didn't exist) and you've optimised your queries then perhaps you need to consider replicating or sharding your data.

Replication

Replication is an excellent way to distribute your database to more than one location. It's used by some as a way to scale the number of reads that can be performed. However the number of writes the entire system can handle is limited to that of the weakest server. Replication is a very wasteful way to scale your application as exactly the same data appears on each server: hardly an efficient use of resources. However it's very simple to implement and thus remains a favoured method of scaling: in the first instance. Beyond that you'll have to think about sharding.

Sharding, Splintering or Partitioning Your Data

Sharding your data is a superior way to scale your application. It's efficient in the sense that the same data doesn't appear in more than one location. Rather you split your data into 'shards' which are placed on different disks or servers with individual resources dedicated to each shard.

In terms of implementation it is more difficult than replication, there's no question of that. It very much depends on your data and whether there is a way to divide it methodically. Ideally it's something that you would build into your application during its initial design. If you wish to scale your writes you must shard: replication will only take you so far.